OLAP on Search Logs: An Infrastructure Supporting Data-Driven Applications in Search Engines

Bin Zhou[†][*]   Daxin Jiang[‡]   Jian Pei[†]   Hang Li[‡]

[†]Simon Fraser University   [‡]Microsoft Research Asia

[†]{bzhou,jpei}@cs.sfu.ca   [‡]{djiang, hangli}@microsoft.com

ABSTRACT

Search logs, which contain rich and up-to-date information about users' needs and preferences, have become a critical data source for search engines. Recently, more and more data-driven applications are being developed in search engines based on search logs, such as query suggestion, keyword bidding, and dissatisfactory query analysis. In this paper, by observing that many data-driven applications in search engines highly rely on online mining of search logs, we develop an OLAP system on search logs which serves as an infrastructure supporting various data-driven applications. An empirical study using real data of over two billion query sessions demonstrates the usefulness and feasibility of our design.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications—*Data Mining*

General Terms

Algorithms, Experimentation

Keywords

OLAP, search log, query session, suffix tree

1. INTRODUCTION

Search logs, which record users' search behavior, contain rich and up-to-date information about users' needs and preferences. While search engines retrieve information from the Web, users implicitly vote for or against the retrieved information as well as the services using their clicks. Moreover, search logs contain crowd intelligence accumulated from millions of users, which may be leveraged in social computing, customer relationship management, and many other areas.

---

[*]This work was done when Bin Zhou was an intern at Microsoft Research Asia.

In recent years, search logs have become a more and more important data source for a wide scope of applications.

Using search logs, we may develop a good variety of data-driven applications in a search engine, which are truly useful for the users and highly profitable for the search engine. For example, by examining the queries frequently asked by users after the query *"KDD 2009"*, a search engine can suggest queries such as *"Paris hotel"* which may improve users' search experience (e.g., [3, 12]). As another example, by analyzing query sequences and click-through information in search logs, a search engine can help an advertiser to bid for relevant keywords (e.g, [9]). Some further examples include using search logs to improve the web search ranking (e.g., [1, 13]), personalize web search results (e.g., [6, 18]), correct search query spellings (e.g., [14, 15]), and monitor and evaluate the performance of search engines and other web services (e.g., [2, 7]).

Given that there are many different data-driven applications in search engines, how many specific search log analysis tools do we have to build? Obviously, building one specific search log analysis tool per application is neither effective nor efficient. Can we develop a search log analysis infrastructure supporting the essential needs of many different data-driven applications? There are some interesting opportunities and challenges.

First, although different data-driven applications carry different purposes and technical demands, by a careful survey of a variety of real applications, we find that many analysis tasks can be supported by a small number of search log pattern mining functions, as exemplified in Section 3.2. This observation makes the idea of building a central search log analysis infrastructure feasible and practical. Building a search log analysis infrastructure presents a new opportunity for developing more effective and powerful search engines.

Second, although there are numerous previous studies on extracting interesting patterns from search logs, those methods cannot be directly applied to building the search log analysis infrastructure. The target infrastructure should be able to serve multiple applications and answer general requests on search log pattern mining. However, almost every previous work is designed for a specific task – the algorithms and data structures are customized for specific tasks. Therefore, those methods cannot meet the diverse requirements in a search log analysis infrastructure.

Third, search logs are often huge and keep growing rapidly over time. Moreover, many data-driven applications require online response to the pattern mining queries. Therefore, a search log analysis infrastructure has to be constructed in a distributed computation environment. Organizing search
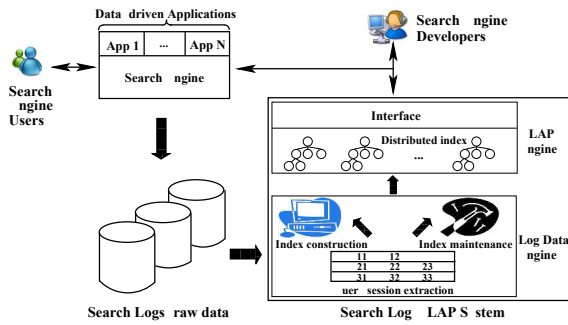
Figure 1: The framework of the OLAP system.

logs and supporting online mining in a distributed environment present great challenges.

In this paper, we present an initiative towards building an *online analytic processing* (OLAP for short) system on search logs as the search log analysis infrastructure. The central idea is that we build a simple yet scalable and distributable index on search logs so that sequential patterns in search logs can be mined online to support many real applications. The role of the OLAP system is shown in Figure 1. As shown in Section 4, the system not only can support online applications in search engines, but also can facilitate online analysis of search logs by search engine developers. We make the following contributions.

First, we argue that many traditional aggregate functions, such as *sum*, *min* and *max*, may not be suitable to support OLAP on search logs. Instead, we identify three novel mining functions, namely *forward search*, *backward search*, and *session retrieval*, which are particularly useful for data analysis on search logs. We further illustrate how to integrate those novel functions with OLAP operations, such as roll-up and drill-down, on search logs.

Second, we develop a simple yet scalable and distributable index structure to support OLAP on search logs using the three mining functions. We do not follow the materialization approach widely adopted in building data warehouses on relational data, since those approaches cannot be straightforwardly applied to sequential search logs. Instead, we use a suffix-tree index. Moreover, we develop a distributed suffix tree construction and maintenance method under the *MapReduce* programming model [5].

Last, we demonstrate the feasibility and potential of a search log analysis infrastructure through a systematic empirical study. We extracted from a major commercial search engine a search log containing more than 4.96 billion searches, 2.48 billion sessions, and 1.48 billion unique queries. The experimental results indicate that our OLAP system on search logs is effective to support various data-driven applications, and our design is efficient in practice.

The rest of the paper is organized as follows. Section 2 briefly reviews the related work. Section 3 identifies several OLAP functions and operations on search logs. In Section 4, we propose our approach to scalable OLAP on search logs. A systematic empirical study conducted on a large real data set is reported in Section 5. Section 6 concludes the paper.

2. RELATED WORK

Most of the OLAP methods focus on relational data. Recently, OLAP has been extended to other data. In particu-

lar, Lo *et al.* [16] extended OLAP to S-OLAP on sequence data, which is highly related to our study. In S-OLAP, a user defines a pattern template of interest. For example, template $\langle XYYX \rangle$ captures a round trip by travelers where X and Y are two subway stations. Aggregates such as `count()` are computed on a sequence database for each instance of the pattern template such as $\langle$"Downtown" "Waterfront" "Waterfront" "Downtown"$\rangle$. An inverted index and a join-based algorithm are proposed to support efficient computation.

S-OLAP and the OLAP on search logs in this paper extend OLAP to sequence data from different angles. First, S-OLAP assumes that a user specifies pattern templates. However, in data-driven applications in search engines, pattern templates are often unknown. Instead, specific query sequences are used as constraints in search. Second, S-OLAP is inefficient to support online response to several sequence mining functions, such as forward search, backward search, and session retrieval, since the results have to be computed by joining the inverted lists on the fly. As shown in Section 3, these functions are commonly required by many data-driven applications in search engines. Finally, the join-based algorithm in S-OLAP cannot be scaled up to distributed computation environment. Our empirical study shows that the join-based algorithm may cause heavy network traffic and thus severely degrade the system performance when the inverted lists are stored in different machines. Thus, S-OLAP cannot handle a huge amount of session data generated by commercial search engines.

Many previous studies were conducted on analyzing search logs for various application scenarios. Several studies modeled the click-through information in search logs as implicit relevance feedback, which can be used to improve ranking algorithms. For instance, Joachims [13] analyzed click-through data and developed a ranking method. Agichtein *et al.* [1] proposed a robust interactive model to improve ranking results. Various approaches using search logs were also proposed for query suggestion (e.g. [3, 12]), expansion (e.g., [4, 8]), and substitution (e.g., [14, 15]). For example, Cui *et al.* [4] extracted probabilistic correlations between query terms and document terms by analyzing search logs and used the correlations to select high-quality expansion terms for new queries. Jones *et al.* [14] identified typical substitutions web searchers made to their queries from search logs, and leveraged the information to improve the quality of user queries. Some other studies include using search logs to improve personalized search (e.g., [6, 18]), generate advertisement keywords (e.g. [9]), predict clicks on sponsored search (e.g., [19]), summarize web pages (e.g., [21]), organize search results (e.g., [22]), evaluate the performance of search engines (e.g., [2, 7]), and so on. However, there is little work on building a general log analysis infrastructure to support various online data-driven applications in search engines.

3. OLAP ON SEARCH LOGS

In this section, we first briefly review how to extract query sessions from search logs. Then, we present three basic sequence mining functions on query sessions, and show that those functions can be used to conduct OLAP on search logs.

3.1 Query Session Extraction

Conceptually, a search log is a sequence of queries and click events. Since a search log often contains the information from multiple users over a long period, we can divide a search log into sessions.

In practice, we can extract sessions in two steps, as described in [3]. First, for each user, we extract the queries by the user from the search log as a stream. Then, we segment each user's stream into sessions based on a widely adopted rule [23]: two queries are split into two sessions if the time interval between them exceeds 30 minutes.

Formally, let Q be the set of unique queries in a search log. A *query sequence* $s = \langle q_1 \cdots q_n \rangle$ is an ordered list of queries, where $q_i \in Q$ $(1 \leq i \leq n)$. n is the *length* of s, denoted by $|s| = n$. A *subsequence* of sequence $s = \langle q_1 \cdots q_n \rangle$ is a sequence $s' = \langle q_{i+1} \cdots q_{i+m} \rangle$ where $m \geq 1$, $i \geq 0$, and $i + m \leq n$, denoted by $s' \sqsubseteq s$. In particular, s' is a *prefix* of s if $i = 0$. s' is a *suffix* of s if $i = n - m$. The *concatenation* of two sequences $s_1 = \langle q_1 \cdots q_{n_1} \rangle$ and $s_2 = \langle q'_1 \cdots q'_{n_2} \rangle$ is $s_1 \circ s_2 = \langle q_1 \cdots q_{n_1} q'_1 \cdots q'_{n_2} \rangle$.

In many search engine applications, frequency is often used as the measure in analysis. Given a set of query sessions $D = \{s_1, s_2, \ldots, s_N\}$, the *frequency* of a query sequence s is $freq(s) = |\{s_i | s \sqsubseteq s_i\}|$ $(s_i \in D)$. The *session frequency* of s is $sfreq(s) = |\{s_i | s = s_i\}|$.

3.2 Session Sequence Mining Functions

To build a search log analysis infrastructure, we collect different requirements on mining search logs at a major commercial search engine. We find that the major needs can be summarized into three basic session sequence mining functions, namely forward search, backward search and session retrieval. Importantly, many data-driven applications are based on mining frequent sequences. Moreover, different from traditional sequential pattern mining [20] which finds the complete set of sequential patterns, a data-driven application typically relies on the top-k query sequences related to a given query sequence s.

DEFINITION 1 (FORWARD SEARCH). *In a set of sessions, given a query sequence s and a search result size k, the **forward search** finds k sequences $s_1, \ldots, s_k$ such that $s \circ s_i$ $(1 \leq i \leq k)$ is among the top-k most frequent sequences that have s as the prefix.* ∎

EXAMPLE 1 (FORWARD SEARCH). *A user's search experience can be improved substantially if a search engine can predict the user's search intent and suggest some highly relevant queries. This is the central idea behind the query suggestion application. For example, a user planning to buy a car may browse different brands of cars. After the user conducts a sequence of queries $s = \langle$ "Honda" "Ford"$\rangle$, a search engine may use a forward search to find the top-k query sequences $s \circ q$, and suggest the queries q to the user. Such queries may be about some other brands like "Toyota", or about comparisons and reviews like "car comparison".* ∎

A forward search only considers sequences s_i that are consecutive to query sequence s in sessions. This is because non-consecutive queries may not be closely related in semantics. For example, a user may raise a sequence of queries $\langle$"*Beijing Olympics*" "*US basketball team 2008*" "*Kobe Bryant*" "*LA Lakers*"$\rangle$. Although each pair of consecutive queries are closely related, the relationship between "*Beijing Olympics*" and "*LA Lakers*" is relatively weak. Thus, we only consider consecutive query sequences in our mining functions.

Symmetrically, we have backward search.

DEFINITION 2 (BACKWARD SEARCH). *In a set of sessions, given a query sequence s and a search result size k,* the **backward search** *finds k sequences $s_1, \ldots, s_k$ such that $s_i \circ s$ $(1 \leq i \leq k)$ is among the top-k most frequent sequences that have s as the suffix.* ∎

EXAMPLE 2 (BACKWARD SEARCH). *Keyword bidding is an important service in sponsored search. A search engine may provide a keyword generation application to help a customer to select keywords to bid.*

A small electronic store may find keyword "digital camcorder" expensive. By a backward search, the search engine can find the query subsequences that often appear immediately before query "digital camcorder" in query sessions. For example, some users may raise queries "digital video recorder", "DV", or "DC" before "digital camcorder" in sessions, since they may not get the term "camcorder" at the first place. The small electronic store may bid for those keywords which are cheaper than "digital camcorder" but carry similar search intent. ∎

Forward search and backward search focus on finding subsequences. In some situations, the whole sessions may need to be retrieved as a pattern.

DEFINITION 3 (SESSION RETRIEVAL). *In a set of sessions, given a query sequence s, the **session retrieval** finds the top-k query sessions $s_1, \ldots, s_k$ in session frequency ($sfreq$) that contain s.* ∎

EXAMPLE 3 (SESSION RETRIEVAL). *Search logs can be analyzed by search engine developers to monitor the search quality and diagnose the causes of user dissatisfactory queries. For example, suppose the click-through rate of query "Obama" is high in the past, but drops dramatically recently. To investigate the causes, a dissatisfactory query diagnosis (DSAT) application can find the top-k sessions containing "Obama" using a session retrieval function. By analyzing those sessions, if a search engine developer finds that the sessions containing query "election" have high click-through rate, while the recent sessions containing query "inauguration" have low click-through rate, the reason for the decrease of the click-through rate may be that the search engine does not provide enough fresh results about Obama's inauguration.* ∎

3.3 OLAP on Session Data

OLAP is well defined on relational data. For example, consider a transaction table on sales of electronic goods in Canada $T(tid, prod, cust, agent, amount)$ where the attributes tid, $prod$, $cust$, $agent$, and $amount$ are transaction-id, product name, customer name, agent name, and sales amount, respectively. A user may want to analyze how the sales amount is related to various factors such as product category, customer group, agent group, as well as their combinations. OLAP queries retrieve group-by aggregates on $amount$ using various combinations of dimension values, such as the sum of amount for all cameras sold by the Vancouver agents. Using OLAP, a user can roll up or drill down along different group-by levels, such as comparing the sales amount of cameras by agents in Vancouver and that by agents in Canada.

Sessions are query sequences. To conduct OLAP on session data, a user can specify a query sequence s. Analogous to the relational case, each query in s can be considered as a dimension, while the frequency of s can be considered as the measure. The three basic session sequence mining functions in Section 3.2 are then regarded as the aggregate functions.

The *sequential drill-down operations* (drill-down for short) on s can be defined as aggregations by either prepending a sequence s_1 at the head of s or appending s_1 at the tail of s. In other words, the sequential drill-down operations perform on either sequence $s_1 \circ s$ or sequence $s \circ s_1$. Reversely, the *sequential roll-up operations* (roll-up for short) on s can be defined as aggregations by removing a subsequence of s either at the head or tail.

EXAMPLE 4 (OLAP OPERATIONS ON SESSION DATA). *In query suggestion application, the search engine provides suggestions each time the user raises a query. For example, when a user raises a query "Honda", the search engine can apply a forward search function on $s_1 = \langle$ "Honda"$\rangle$ and get the top-k queries as the candidates for query suggestion. Suppose the user raises a second query "Ford", the search engine can drill down using the forward search function to find out the top-k queries following sequence $s_2 = \langle$ "Honda" "Ford"$\rangle$ as the candidates for query suggestion.*

In keyword bidding application, suppose a user applies the backward search function on sequence $s_1 = \langle$ "digital camcorder"$\rangle$ and finds the sequence $s_2 = \langle$ "DV" "digital camcorder"$\rangle$ interesting. The user may further roll up using the forward search function and find out the top-k queries following $s_3 = \langle$ "DV"$\rangle$. Those top-k queries may also be good candidates to bid. ∎

To implement OLAP on relational data effectively and efficiently, materialization is often used. That is, all aggregates are pre-computed and indexed so that whenever a query comes, the result can be retrieved promptly. One challenge in OLAP on session data using the three basic sequence mining functions is that the dimensions are not explicitly pre-defined. There are a huge number of possible query sequences. Quantitatively, if a length up to l is considered, then the total number of possible query sequences is $\sum_{i=1}^{l} |Q|^l$, where Q is the set of unique queries. In a search engine, there can be billions of unique queries, which make the pre-computation for even short query sequences (e.g., $l = 3$) infeasible.

4. SCALABLE OLAP ON SEARCH LOGS

The size of search logs in search engines is usually very large. It is infeasible to scan a search log on the fly to conduct online sequence mining. Thus, we need an effective index on search logs. Moreover, even an index on search logs is often too large to be computed and stored in a single machine. Index construction has to be distributed.

The framework of our OLAP system on search logs, as shown in Figure 1, consists of a *log data engine* and an *OLAP engine*. The log data engine extracts query sessions from search logs as described in Section 3.1, constructs distributed indexes from query sessions, and maintains the indexes when new logs arrive. We introduce the index structure in Section 4.1, and describe in Section 4.2 how to construct and maintain the index structures in a distributed manner.

The OLAP engine delegates online mining requests from data-driven applications to corresponding index servers, and integrates results from index servers. We describe online mining in Section 4.3.

4.1 Suffix Trees and Reversed Suffix Trees

A core task in the three basic sequence mining functions is *subsequence matching*: given a set of sequences D and a

SID	Query sequences	SID	Query sequences
s_1	$\langle q_1 q_2 q_3 q_4 \rangle$	s_5	$\langle q_6 q_1 q_2 q_5 \rangle$
s_2	$\langle q_1 q_2 q_4 q_5 \rangle$	s_6	$\langle q_1 q_2 q_3 q_5 \rangle$
s_3	$\langle q_6 q_1 q_2 q_5 \rangle$	s_7	$\langle q_1 q_2 q_3 q_6 \rangle$
s_4	$\langle q_1 q_2 q_3 q_4 \rangle$	s_8	$\langle q_6 q_1 q_2 q_5 \rangle$

Table 1: A running example of 8 query sessions.
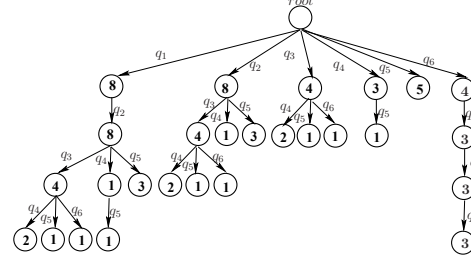

Figure 2: A suffix tree for Table 1.

query sequence s, find the sequences in D of which s is a subsequence. Suffix trees and their compacted forms [11] are effective data structures to solve the problem, since a subsequence s' of a sequence s must be a prefix of a suffix of s.

A suffix tree organizes all suffixes of a given sequence into a prefix sharing tree such that each suffix corresponds to a path from the root node to a leaf node in the tree. By organizing all the suffixes of s into a tree structure, to check whether sequence s' is a subsequence of s, we can simply examine whether there is a path corresponding to s' from the root of the suffix tree.

Most of the existing methods [10] construct a suffix tree for indexing one (long) sequence. However, in the case of query session mining, the average sequence length is short, but the number of sequences is huge. We extend the suffix tree structure for indexing query sessions straightforwardly. Figure 2 shows a suffix tree for the query sessions in Table 1. In the tree, each edge is labeled by a query and each node except for the root corresponds to the query sequence constituted by the labels along the path from the root to that node. Moreover, each node is associated with a value representing the frequency of the corresponding query sequence in the search log.

To serve backward search, we also build a reversed suffix tree. For each query session $s = \langle q_1 q_2 \ldots q_n \rangle$, we obtain a reversed query sequence $s' = \langle q_n q_{n-1} \ldots q_1 \rangle$ and insert all suffixes of s' into the reversed suffix tree. Figure 3 shows the reversed suffix tree for the query sessions in Table 1.

4.2 Distributed Suffix Tree Construction

A search log may contain billions of query sessions. The resulting suffix tree and reversed suffix tree cannot be held into the main memory or even the disk of one machine. The existing studies on disk-based suffix tree construction (e.g., [17]) target at indexing one long sequence. To the best of our knowledge, there is no existing work on efficient construction of a suffix tree for a large number of sequences on multiple computers.

We develop a distributed suffix tree construction method under the MapReduce programming model [5], a general architecture for distributed processing on large computer clus-

Figure 3: A reversed suffix tree for Table 1.

Figure 4: Distributed index construction in 3 steps.

ters. The central idea of MapReduce is to divide a large task into chunks so that they can be processed in parallel. At the beginning, the whole data set is stored distributively in the cluster; each computer possesses a subset of data. In the map phase, each computer processes its local subset of data and emits pieces of intermediate results, where each piece is associated with a key. In the reduce phase, all pieces of intermediate results carrying the same key are collected and processed on the same computer.

Figure 4 shows the major steps of our distributed index construction method. In the first step, we compute all suffixes and the corresponding frequencies using the MapReduce model. In the second step, we partition the entire set of suffixes into several parts such that each part can be held in the main memory of one index server. The first two steps are conducted under the MapReduce model. In the last step, we construct the local suffix trees and reversed suffix trees on each index server. As to be explained shortly, the partitioning method in the second step guarantees that the local (reversed) suffix trees are subtrees of the global (reversed) suffix trees.

In the map phase of Step 1, each computer processes a subset of query sessions. For each query session s, the computer emits an intermediate key-value pair $(s', 1)$ for every suffix s' of s, where the value 1 here is the contribution to frequency of suffix s' from s. In the reduce phase, all intermediate key-value pairs having suffix s' as the key are processed on the same computer. The computer simply outputs a final pair $(s', freq(s'))$ where $freq(s')$ is the number of intermediate pairs carrying key s'.

The above MapReduce method returns all suffixes of sessions and their frequencies. We need to organize them into a suffix tree. Ideally, we want that the suffix tree can be held in main memory so that online mining can be conducted quickly. However, since the number of all suffixes is usually very large, the whole suffix tree cannot fit in one machine. To tackle the problem, we partition a suffix tree into subtrees so that each subtree can be held into the main memory of an index server. Moreover, we require all subtrees are exclusive from each other so that there are no identical paths on two subtrees. Finally, we try to make sure that the sizes of the subtrees do not vary dramatically in the hope that the online mining workload can be distributed relatively evenly on the index servers.

One challenge is that it is hard to estimate the size of a subtree using only the suffixes in the subtree, since the suffixes may share common prefixes. For example, a subtree with two suffixes $s_1 = \langle q_1 q_2 q_3 \rangle$ and $s_2 = \langle q_1 q_2 q_4 \rangle$ has only 4 nodes since the two suffixes share a prefix $\langle q_1 q_2 \rangle$.

Given a set of suffix sequences, a simple yet reachable upper bound of the size of the suffix tree constructed from the suffix sequences is the total number of query instances in the suffix sequences. For example, the upper bound of the size of the suffix tree constructed from $s_1 = \langle q_1 q_2 q_3 \rangle$ and $s_2 = \langle q_1 q_2 q_4 \rangle$ is 6. Using this upper bound in space allocation is conservative. The advantage is that we reserve sufficient space for the growth of the tree when new search logs are added.

To partition the suffix tree, for each query $q \in Q$, we again apply the MapReduce approach and compute the upper bound of the subtree rooted at q. In the map phase, each suffix sequence s generates an intermediate key-value pair $(q_1, |s|-1)$, where q_1 is the first query in s, and $|s|-1$ is the number of queries in s except for q_1. In the reduce phase, all intermediate key-value pairs carrying the same key, say q_1, are processed by the same computer. The computer outputs a final pair $(q_1, size)$ where $size$ is the sum of values in all intermediate key-value pairs with key q_1. Clearly, $size$ is the upper bound of the size of the subtree rooted at query q_1. If $size$ is less than the size limit of an index server, the whole subtree rooted at q_1 can be held in the index server. In this case, we assign all the suffixes whose first query is q_1 to the same index server. Otherwise, we can further divide the subtree rooted at q_1 recursively and assign the suffixes accordingly. In this way, we can guarantee that the local suffix trees on different index servers are exclusive.

New search log sessions keep arriving incrementally. To incrementally maintain the suffix tree, when a new batch of query sessions arrive, we only process the new batch using the MapReduce process similar to that in Step 1 and compute the frequency $freq(s)$ of each suffix s within the new batch. Then, the new set of suffixes are assigned to the index server according to the subtrees that they should be hosted. If a subtree in an index server exceeds the memory capacity after the new suffixes are inserted, the subtree is partitioned recursively as in Step 2 and more index servers are used. Finally, each local suffix tree is updated to incorporate the new suffixes.

The reversed suffix trees can be constructed and maintained in a similar way.

4.3 Online Mining

Using a suffix tree and a reversed suffix tree, we can support the three basic sequence mining functions online.

A forward search can be implemented using a suffix tree. Let s be a query sequence. We can search the suffix tree and find a path from the root to a node v that matches s. If such a path does not exist, then no query session contains s and thus nothing can be returned by the forward search. If such a path is found, then we only need to search the subtree rooted at v, and find the top-k nodes in the subtree with the largest frequencies. The paths from v to those nodes are the answers.

A thorough search of the subtree rooted at v can be costly if the subtree is large. It is easy to see that in a suffix tree, the frequency at a node v is always larger than or equal to that at any descendant of the node. Thus, we can conduct a best-first search to find the top-k answers.

EXAMPLE 5 (FORWARD SEARCH). *Consider the sessions in Table 1 and the corresponding suffix tree in Figure 2. Suppose the query sequence is $s = \langle q_1 q_2 \rangle$. A forward search starts from the root node and finds a path matching s (the left most path in the figure).*

The node v corresponding to s has 3 child nodes. We can follow the labels on the edges from v and form a candidate answer set Cand. During the forward search process, Cand is maintained as a priority queue in frequency descending order. Therefore, $Cand = \{q_3, q_5, q_4\}$ at the beginning. If the user is interested in top-2 answers, we first pick the head element q_3 from Cand. As Cand is maintained as a priority queue, q_3 has the largest frequency and can be safely placed into the final answer set R. This is due to a good property of the suffix tree: any descendant node v' cannot have a frequency higher than that in any of its ancestor nodes v. In the next step, all the sequences corresponding to the child node of v are inserted into Cand. The priority queue now becomes $Cand = \{q_5, q_3 q_4, q_4, q_3 q_5, q_3 q_6\}$. Again, we pick the head element q_5 from Cand and place it in R. Therefore, the top-2 answers are $R = \{q_3, q_5\}$. If the user is interested in top-3 answers, the queue is updated to $Cand = \{q_3 q_4, q_4, q_3 q_5, q_3 q_6\}$ since q_5 does not have a child. The top-3 answers are $R = \{q_3, q_5, q_3 q_4\}$. ∎

A suffix tree may be distributed in multiple index servers. When the search involves multiple index servers, each index server looks up the local subtree and returns the local top-k results to the OLAP server. Since the local subtrees are exclusive, the global top-k results must be among the local top-k results. Therefore, the OLAP server only needs to examine all the local top-k results and select the most frequent ones as the global top-k results.

Similarly, a backward search can be conducted using a reversed suffix tree.

To conduct session retrieval online, we pre-compute the frequencies of sessions in a session table by a MapReduce process. Specifically, in the map phase, each session s generates a key value pair $(s, 1)$. In the reduce phase, all identical sessions are assigned to the same computer and thus the frequency of the session can be computed.

We enhance the suffix tree by adding the session information. Each leaf node of the suffix tree is attached a list of the IDs of the sessions containing the suffix. This can be easily computed as a byproduct in the suffix tree construction. Moreover, all session IDs in the list are sorted in the frequency descending order. Figure 5 shows the enhanced suffix tree of our running example.

Session retrieval with a query sequence s can be conducted online using the enhanced suffix tree. We first find the node
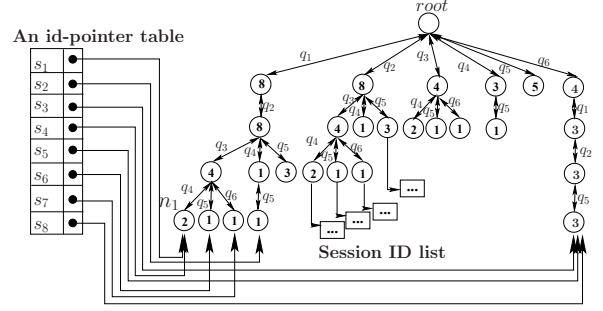

Figure 5: An enhanced suffix tree.

v such that the path from the root of the suffix tree to v matches s. We then search the leaf nodes in the subtree rooted at v and find the IDs of the top-k frequent sessions. Once we have the IDs of the top-k sessions, the last step is to get the query sequences of the corresponding sessions. To avoid physically storing the query sequences, which could be expensive in space, we reuse the query sequences on the suffix tree. That is, instead of storing the actual query sequence in the mapping table, for each session ID, we store a pointer to the leaf node corresponding to the query session in the suffix tree. We call this mapping table *id-pointer table*. As an example, in Figure 5, the entry for session s_1 in the id-pointer table points to leaf node n_1. To find out the sequence of s_1, we only need to trace the path from the leaf node n_1 back to the root and then reverse the order of the labels on the path. In this example, the path from n_1 to the root is $\langle q_4 q_3 q_2 q_1 \rangle$, and thus $s_1 = \langle q_1 q_2 q_3 q_4 \rangle$.

To speed up the search, we can further store at each internal node v in the suffix tree a list of k_0 sessions that are most frequent in the subtree of v, where k_0 is a number so that most of the session retrieval requests ask for less than k_0 results. In practice, k_0 is often a small number like 10. Once the list is stored, the session retrieval operations requesting less than k_0 results obtain the IDs of the top-k sessions directly from the node and thus do not need to search the leaf nodes in the subtree. Only when a session retrieval requests more than k_0 results we need to search the subtree.

5. AN EMPIRICAL STUDY

In this section, we report a systematic evaluation of our OLAP system using a real query log data set extracted from a major commercial search engine. The data set contains $4,963,601,307$ searches and $1,481,946,526$ unique queries. We apply the session segmentation method discussed in Section 3.1 and derive $2,488,594,484$ sessions.

We first examine the characteristics of session data. Figure 6(a) shows the distribution of the length of sessions. For those sessions of lengths greater than 1, the length of sessions follows a power-law distribution. It indicates that the majority of sessions in the search log are not long. As introduced in Section 4, the level of a suffix tree is at most the length of the longest query sequence. Therefore, the suffix tree for the session data is not high, which suggests that the query answering process could be very efficient since it is not necessary to traverse deep in the suffix tree index.

We also plot the number of unique sessions with respect to their frequencies in Figure 6(b). The figure shows that the frequency of unique query sessions also follows a power-

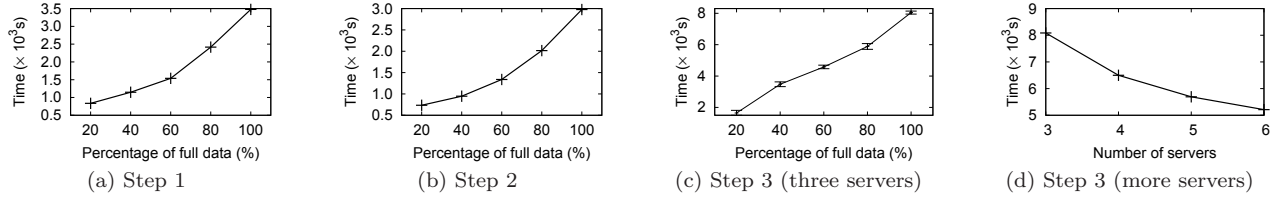(a) Step 1 (b) Step 2 (c) Step 3 (three servers) (d) Step 3 (more servers)

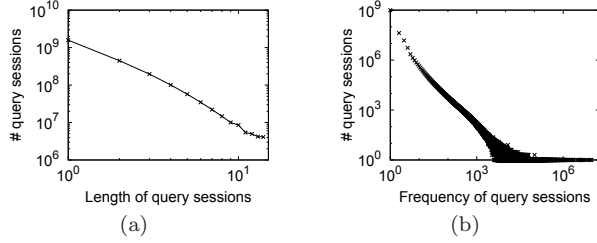Figure 7: The index construction time for each step.

Figure 6: The distributions of (a) session lengths and (b) session frequencies.

law distribution. This indicates that the suffix tree index can greatly compress the data since many sessions have identical query sequences.

5.1 Index Construction

We build the suffix tree index and the reversed suffix tree index using the methods developed in Section 4. In Section 4.3, we presented several strategies to achieve the sessional retrieval function. Based on the analysis, the strategy in which each leaf node is associated with a list of session IDs can balance the storage size and the query answering response time. By default, the index built in the following subsections refers to that strategy.

Our index construction methods follow a distributed implementation. We use a MapReduce cluster consisting of 20 computers. The map phase and the reduce phase are automatically scheduled by the system. Moreover, up to 6 computers are used as index servers. All the 26 computers run the 64 bit Microsoft Windows Server 2003 operating system, each with an Intel Xeon 2.33 GHz CPU and 8 GB main memory.

To examine the scalability of our index construction algorithm, we measure the construction time with respect to uniform samples of the whole search log data with different sizes. As elaborated in Section 4.2, the index construction consists of three steps. The first step extracts all suffixes and their frequencies using a MapReduce approach. The second step partitions the suffixes and allocates them to individual index servers. These two steps are carried out in the MapReduce cluster. The last step constructs the local suffix tree as well as the reversed suffix tree on each index server. We first use three computers as index servers. The run time for each step is shown in Figures 7(a), 7(b) and 7(c), respectively.

For the first two steps, the runtime increases over-linearly when data size increases. By monitoring the CPU usage and I/O status of the computers in the MapReduce system, we find that the overall time of each step is dominated by writing intermediate and final results to disks. In the first step,

both intermediate and final results are (suffix, frequency) pairs. Since the total number of intermediate and final pairs increases over-linearly with respect to the size of sessions, so does the total runtime of the first step. In the second step, both intermediate and final results of the MapReduce procedure are (prefix, size) pairs, where prefix corresponds to a subtree and size is the estimated number of nodes in the subtree. To reduce the chances of recursive partition of the suffix and reverse suffix trees, in our implementation, we choose the length of prefixes to be 2. Consequently, the overall time for the second step is over-linear since the total numbers of intermediate and final (prefix, size) pairs increase over-linearly with respect to the size of sessions.

Although the first two steps do not scale linearly, the overall time is acceptable. In particular, even on the whole data set containing about 2.5 billion sessions, the first step and the second step still finish within one hour, respectively. This verifies that our algorithms under the MapReduce programming model are efficient to handle large data sets.

In Figure 7(c), the curve shows the average runtime, while the deviation bars indicate the variance of the runtime among the three index servers. We can see the average run time scales well with respect to the size of data. Moreover, the variation of runtime is very small, which indicates that the suffixes are partitioned evenly and the load of the index servers is well balanced.

To measure the effectiveness of distributed indexing, we use more machines as index servers. The runtime for the first two steps is similar to that in Figures 7(a) and 7(b), since we only add index servers without changing the MapReduce cluster. Figure 7(d) shows the average runtime for the third step using four, five, and six machines, respectively. All machines have the same configuration. We can see that the average runtime per index server drops dramatically when more machines are used. Although the decrease is not strictly linear, the average runtime when six machines are used is only 64.4% of that when three machines are used.

We also measure the size of index storage. Figure 8(a) shows the memory usage of the index structure using three servers. Clearly, the index size is roughly linear with respect to the data set size. Moreover, the deviation bars show that the index servers have balanced load. Last, as shown in Figure 8(b), the memory usage per index server drops to nearly half when the number of index servers increases from three to six.

5.2 OLAP Performance

To test the efficiency of our system, we randomly extract a sample set of 1,000 query sequences of length varying between 1 and 4 from the original search log. We evaluate the response time on the three OLAP functions, i.e., forward search (FS), backward search (BS), and session re-
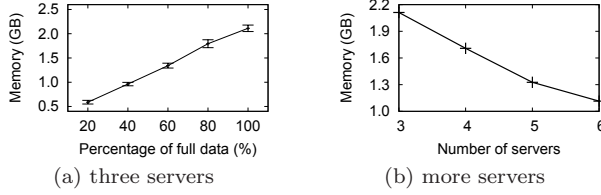
Figure 8: Memory usage of index servers.

Figure 9: The effect of l and k on response time.

trieval (SR). Three index servers are used to hold the index and run the mining tasks jointly.

Two factors may affect the response time: the length l of the input query sequences and the answer set size k. In the following experiments, we set $l = 2$ and $k = 10$ by default.

In the experiments, the average response time for a forward/backward search is 0.19 second and the average response time for a session retrieval is 0.72 second. The results verify the feasibility of our approach for online data-driven applications in search engines. Obviously, the response time can be further decreased by caching technologies and more advanced hardware.

We further investigate the effect of the factors, l and k, respectively, by varying one factor but keeping the other factor fixed at the default value. The effect of the length parameter l is shown in Figure 9(a). When l increases, the response time for all three OLAP functions decreases. This is because the longer the query sequence, the more specific the sequence is, and thus the smaller the search space that the query answering algorithms need to traverse.

The effect of the size parameter k is shown in Figure 9(b). When k increases, the response time of all the three functions increases. Generating a larger result set requires visiting more candidate nodes during the search procedure.

5.3 Comparison with S-OLAP

For comparison, we implement the S-OLAP system as described in [16] and extend the pattern manipulation operations in [16] to answer the forward/backward search and session retrieval. These extended solutions are referred to as the *inverted list* approach. The main extension is as follows. For each query sequence of length 1, the inverted list method constructs an inverted list in which the IDs of the sessions containing that query sequence are kept. For each query sequence of length greater than one, the inverted list can be derived by joining the two inverted lists for two subsequences. Since the join operation does not guarantee the consecutiveness of subsequences, the joined inverted list has be to scanned once to remove invalid sessions. The join operation in the inverted list approach can be carried out either offline or on the fly. To make a tradeoff between the index storage size and the response time, only the inverted lists for short query sequences, e.g., 1 and 2, are pre-computed offline, while the inverted lists for long sequences are computed on the fly.

The inverted list approach cannot be extended to a distributed computation environment straightforwardly. First, during the query answering procedure, the inverted lists for long sequences have to be computed on the fly. In a distributed environment, the inverted lists to be joined may be stored in different machines. Consequently, the join operation may make the network traffic a bottleneck of the system. Moreover, the computation of the inverted lists for
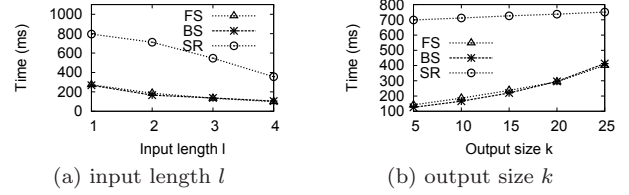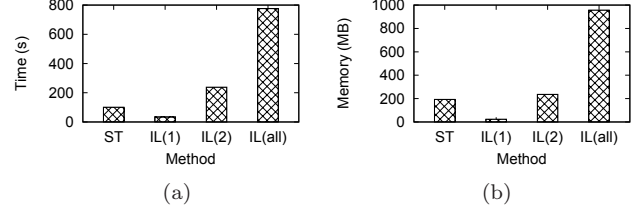

Figure 10: Comparison between ST and IL on (a) index construction time and (b) memory usage.

long sequences may need multiple rounds of join operation, where each round involves the combination of subsequences. It is a costly to maintain the intermediate joining results in a distributed way.

Since the inverted list approach cannot be easily extended to a distributed environment, we use a small uniform sample data set whose inverted lists can be held in the main memory of a single machine. This small set contains $27,091,874$ searches, $13,587,124$ sessions, and $6,579,346$ unique queries. To make a fair comparison, our method also uses only one index server. Moreover, for a thorough comparison, we consider different options to construct the inverted lists. Specifically, **IL(1)** refers to the option that only the inverted lists for sequences of length 1 are pre-computed, **IL(2)** refers to the option that the inverted lists for sequences of lengths 1 and 2 are pre-computed, and **IL(all)** refers to that the complete set of inverted lists for all sequences are pre-computed. Finally, we use **ST** to refer to our suffix tree method.

We first examine the index construction time for the suffix tree approach and the inverted list approach. For the suffix tree approach, the index construction includes building both the suffix tree and the reversed suffix tree. The results are shown in Figure 10(a). When the **IL(1)** option is adopted, only the inverted lists for the sequences of length 1 are pre-computed. Therefore, the construction time for the **IL(1)** option is the smallest. However, when the inverted lists for longer sequences are built, the index construction time increases dramatically. The runtime for the **IL(all)** option is the most costly, almost 7 times longer than that of the suffix tree approach.

Figure 10(b) shows the actual memory usage of the indexes in different approaches. The results have a trend similar to that in Figure 10(a). The **IL(1)** option has the smallest index size, while the inverted lists for option **IL(all)** are the largest. The index of the suffix tree approach is slightly smaller than that of option **IL(2)**.

We further analyze the query response time of our approach and the inverted list approach. Similar to the evaluation in Section 5.2, we consider the effect of the length l of the input query sequences and the size k of answer sets. By default, $l = 2$ and $k = 10$.
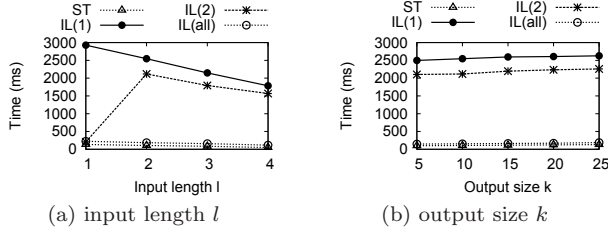
Figure 11: Comparison on forward search.

Figure 12: Comparison on session retrieval.

Figure 11(a) shows the query response time when the length of input query sequences varies from 1 to 4. Our suffix tree approach has the shortest response time, and the **IL(all)** option has comparable performance. However, the **IL(1)** option is very slow for all lengths of input query sequences, while the response time for option **IL(2)** increases dramatically when the length of input sequences is greater than 1. In general, to conduct the forward search for a query sequence of length l, the inverted list approach needs to look up the inverted lists for sequences of length $l + 1$. If the inverted lists are not pre-computed, they have to be generated by expensive join operations on the fly. That is the reason why option **IL(1)** is slow for all lengths and option **IL(2)** becomes dramatically more costly at length 2.

Figure 11(b) shows the query response time when the size k changes. For all the four curves, the response time increases when k increases because the larger the size k, the more candidates need to be processed. The suffix tree approach and the **IL(all)** option are much faster than options **IL(1)** and **IL(2)** because by default $l = 2$. We omit the results on backward search here, since the results are similar to those on forward search.

We finally examine the response time for session retrieval. Figure 12(a) shows the results when the length l of the input query sequences varies. In general, to retrieve sessions using query sequence of length l, the inverted list approach needs to look up the inverted lists for sequences of length l. Consequently, the response time for option **IL(1)** increases dramatically when the length is larger than 1, since the answer to longer sequences requires more join operations on the fly. Similarly, the response time for option **IL(2)** increases dramatically when the length is over 2. The suffix tree approach and the **IL(all)** option have better performance, and the response time is much shorter than that of the cases when the online join operations are executed. Figure 12(b) shows the results when the size k varies. The **IL(1)** option is the slowest, since $l = 2$.

The suffix tree approach is more suitable than the inverted list approach to serve as the index structure of a log analysis infrastructure. First, compared with the options such as **IL(1)** and **IL(2)**, where only partial inverted lists are pre-computed, the suffix tree approach is more scalable to the length of input sequences and the efficiency is comparable to that of the **IL(all)** option. Second, compared with the **IL(all)** option, where all inverted lists are pre-computed, the suffix tree approach has much smaller construction time and memory usage. Finally and most importantly, the suffix tree approach can be scaled up to a distributed environment and handle a huge amount of log data in real applications.

6. CONCLUSIONS

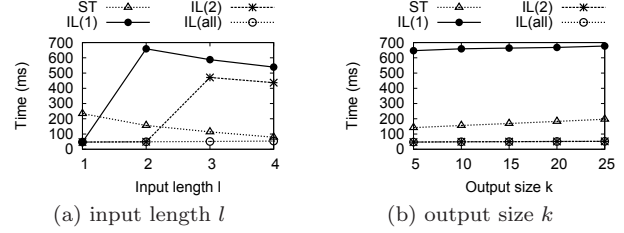In this paper, we proposed an OLAP system on search logs. It can support many data-driven applications in search engines, such as query suggestion and keyword bidding. We formalized three OLAP functions on search logs, and developed a simple yet effective suffix-tree index to support online mining of query sessions. We implemented our methods in a prototype system and conducted a systematic empirical study to verify the usefulness and feasibility of our design.

7. REFERENCES

[1] E. Agichtein *et al*. Improving web search ranking by incorporating user behavior information. In *SIGIR'06*.

[2] E. Agichtein *et al*. Learning user interaction models for predicting web search result preferences. In *SIGIR'06*.

[3] H. Cao *et al*. Context-aware query suggestion by mining click-through and session data. In *KDD'08*.

[4] H. Cui *et al*. Probabilistic query expansion using query logs. In *WWW'02*.

[5] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI'04*.

[6] Z. Dou *et al*. A large-scale evaluation and analysis of personalized search strategies. In *WWW'07*.

[7] G. E. Dupret and B. Piwowarski. A user browsing model to predict search engine click data from past observations. In *SIGIR'08*.

[8] B. M. Fonseca *et al*. Concept-based interactive query expansion. In *CIKM'05*.

[9] A. Fuxman *et al*. Using the wisdom of the crowds for keyword generation. In *WWW'08*.

[10] R. Giegerich and S. Kurtz. From ukkonen to mccreight and weiner: A unifying view of linear time suffix tree construction. *Algorithmica*, 19:331–353, 1997.

[11] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[12] C.-K. Huang *et al*. Relevant term suggestion in interactive web search based on contextual information in query session logs. *J. Am. Soc. Inf. Sci. Technol.*, 54(7):638–649, 2003.

[13] T. Joachims. Optimizing search engines using clickthrough data. In *KDD'02*.

[14] R. Jones *et al*. Generating query substitutions. In *WWW'06*.

[15] M. Li *et al*. Exploring distributional similarity based models for query spelling correction. In *ACL'06*.

[16] E. Lo *et al*. Olap on sequence data. In *SIGMOD'08*.

[17] B. Phoophakdee and M. J. Zaki. Genome-scale disk-based suffix tree indexing. In *SIGMOD'07*.

[18] F. Qiu and J. Cho. Automatic identification of user interest for personalized search. In *WWW'06*.

[19] M. Richardson *et al*. Predicting clicks: estimating the click-through rate for new ads. In *WWW'07*.

[20] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *EDBT'96*.

[21] J.-T. Sun *et al*. Web-page summarization using clickthrough data. In *SIGIR'05*.

[22] X. Wang and C. Zhai. Learn from web search logs to organize search results. In *SIGIR'07*.

[23] R. W. White *et al*. Studying the use of popular destinations to enhance web search interaction. In *SIGIR'07*.