World Wide Web: Internet and Web Information Systems, 6, 259–279, 2003

Optimal Algorithms for Finding User Access Sessions from Very Large Web Logs *

ZHIXIANG CHEN chen@cs.panam.edu
Department of Computer Science, University of Texas-Pan American, USA

ADA WAI-CHEE FU adafu@cse.cuhk.edu.hk
Department of Computer Science, Chinese University of Hong Kong, Hong Kong

FRANK CHI-HUNG TONG ftong@eti.hku.hk
Department of Computer Science and Information Systems, The University of Hong Kong, Hong Kong

Abstract

Although efficient identification of user access sessions from very large web logs is an unavoidable data preparation task for the success of higher level web log mining, little attention has been paid to algorithmic study of this problem. In this paper we consider two types of user access sessions, *interval sessions* and *gap sessions*. We design two efficient algorithms for finding respectively those two types of sessions with the help of some proposed structures. We present theoretical analysis of the algorithms and prove that both algorithms have optimal time complexity and certain error-tolerant properties as well. We conduct empirical performance analysis of the algorithms with web logs ranging from 100 megabytes to 500 megabytes. The empirical analysis shows that the algorithms just take several seconds more than the baseline time, i.e., the time needed for reading the web log once sequentially from disk to RAM, testing whether each user access record is valid or not, and writing each valid user access record back to disk. The empirical analysis also shows that our algorithms are substantially faster than the sorting based session finding algorithms. Finally, optimal algorithms for finding user access sessions from distributed web logs are also presented.

Keywords: Web log mining, data preparation, user access sessions, data structures, time complexity

1. Introduction

Web log mining has been receiving extensive attention recently (e.g., [2,4–9,14,17,19–21]) because of its significant theoretical challenges and great application and commercial potentials. The goal of web log mining is to discover user access behaviors and interests that are buried in vast web logs that are being accumulated every day, every minute and every second. The discovered knowledge of user access behaviors and interests will certainly help the construction and maintenance of real-time intelligent web servers that are able to dynamically tailor their designs to satisfy users' needs [11,15,16]. The discovered knowledge will also help the administrative personnel to predict the trends of the users' needs

* The extended abstract of this paper was published in *Advances in Knowledge Discovery and Data Mining (Proceedings of PAKDD'02), Lecture Notes in Computer Science 2336*, pages 290–296, Springer, 2002.

so that they can adjust their products to attract more users (and customers) now and in the future [6].

One unavoidable data preparation task for the success of web log mining is efficient identification of user access sessions. *The definition of* **session** *or* **visit** *is the group of activities performed by a user from the moment she enters a server site to the moment she leaves the site* [18]. A user access session determines the set of all the web pages a user has accessed from the beginning of the session to the end. This kind of understanding is similar to the "*market basket*" concept in data mining and knowledge discovery [1,8], the set of all items in the market basket a customer buys at a retail store. As such, the existing data mining techniques, such as the popular association rule mining [1], may be applied to web log mining [21]. Some researchers [10] have suggested that user access sessions are too coarse grained for mining tasks such as the discovery of association rules so that it is necessary to refine single user access sessions into smaller transactions. But such refining process is also based on identification of user access sessions.

It must be pointed out that the market basket concept is "*deterministic*", i.e., what are in a market basket is very clear and can be automatically recorded to some market basket log file system when the customer checks her shopping out at the retail store cashier. On the other hand, it is not easy to find sessions from the web log, because it is not clear when a session starts or when it ends. Some users access the web page after page at a very fast pace. Other users access the web at a very slow pace, visiting one for a while, doing something, and then visiting another. Other biggest impediments to collecting reliable web usage data include local cashing and proxy servers. The difficulty of finding users and user access sessions from web logs have been addressed in [7,10,12,13]. Researchers have proposed cut-off thresholds to approximate the start and the end of a session (e.g., [10]). One may also propose a 30-minutes gap between any two pages accessed by the user. If the gap limit is exceeded, then a session boundary should be inserted between the two pages [3,15,16].

There is little algorithmic study of efficient identification of user access sessions in the literature. Web logs are typically large, from hundreds of megabytes to tens of gigabytes of user access records. The logs are usually stored in hard disks. Certainly, after eliminating invalid or irrelevant records from the raw web log, sorting algorithms can be applied to group the same user's access records together. After that a one-pass sequential reading process can be used to cut each group of the same user's access records into sessions according to some threshold. This sorting approach may be efficient for smaller web logs, but it is not practically efficient for finding sessions from very large web logs, say, logs with hundreds of megabytes or tens of gigabytes of records. Sorting a web log of size n alone will take $O(n \log n)$ time in theory. The actual time needed by sorting will be very large due to content swaps between RAM and hard disk. When real-time applications based on dynamic discovery of the users' hidden access patterns is concerned, faster algorithms that have the ability to find user access sessions in several seconds or several minutes is highly demanded. Without such faster algorithms, any higher level mining tasks will be slowed down.

In this paper we will settle the problem of efficient identification of user access sessions from web logs. We consider two types of user access sessions: the interval sessions such

that each session consists of pages accessed by the same user within a time limit, and the gap sessions such that each session consists of pages accessed by the same user with pairwise page access time gaps below a threshold. Those two types of sessions have been studied by other researchers (e.g., [2,10]). We design two efficient algorithms with the help of some proposed data structures. We present theoretical analysis of the algorithms, and prove that both algorithms have optimal time complexity, i.e., the time complexity of both algorithms is respectively linear in the size of the web log. We show that the actual time needed by each of the two algorithms is no more than twice the baseline time – the time needed for reading the web log once sequentially from disk to RAM, testing whether each user access record is valid or not, and writing each valid user access record back to disk. We also prove that the algorithms have certain error-tolerant properties. That is, when some users' access records are missing from the web log, our algorithms are still able to find the correct sessions of all the other users and preserve much of the information for the users with the missing records. These error-tolerant properties enable the algorithms to overcome difficulties in session identification when local cashing or proxy servers are used. We conduct empirical performance analysis of the algorithms with web logs ranging from 100 to 500 megabytes. The empirical analysis shows that the performances of the algorithms are just several seconds more than the baseline time. The empirical analysis also shows that our algorithms are substantially faster than the sorting based session finding algorithms. In reality, heavily accessed web servers may have some distributed file systems to store the user access records. In such a distributed case, records in the same user access session may be scattered into several logs. We will design two optimal algorithms for finding user access sessions from distributed logs.

The rest of the paper is organized as follows. In Section 2, we give formal definitions of α-interval sessions and β-gap sessions. In Section 3, we design data structures and two session finding algorithms. In Section 4, we prove that our two algorithms are correct and have optimal time complexity. We also prove that both algorithms have certain error-tolerant properties. In Section 5, we report our empirical performance analysis of the two algorithms in comparison with the baseline performance and the performance of the sorting based session finding algorithms. In Section 6, two optimal algorithms for finding respectively interval sessions and gap sessions from distributed web logs are designed. In Section 7, we consider the use of hashing to improve the performance of our algorithms further. We conclude the paper in Section 8.

2. Web logs and user access sessions

The web log of any given web server is a sequential file with one user access record per line. Each user access record consists of the following fields: (1) user's IP address or host name, (2) access time, (3) request method (e.g., "GET", "POST"), (4) URL of the page accessed, (5) protocol (e.g., HTTP/1.0), (6) return code, and (7) number of bytes transmitted. Other fields may also be included when the server is specially configured. Examples of user access records are given as follows:

```
rios.cs.panam.edu __ [08/JAN/2001 : 16 : 10 : 15 −0500] "GET/Welcome.htmlHTTP/1.0"
200 2645
trix.cs.uoregon.edu __ [13/FEB/2001 : 15 : 45 : 47 −0500] "GET/people.htmlHTTP/1.0"
200 2388
206.254.0.168 __ [20/MAY/2001 : 18 : 59 : 43 −0500] "GET/Ugradcourses.htmlHTTP/1.0"
404 281
```

One should note that user access records in a web log are ordered according to record access time with the oldest access record placed on the top and the newest at the bottom. We state this property in the following proposition. Given any user access record r, let $t(r)$ be the number of seconds starting at `01/Jan/1900:00:00:00` and ending at the time when r was performed.

Proposition 1. Let $\mathcal{L} = \{r_1, \ldots, r_n\}$ be any given web log $\mathcal{L} = \{r_1, \ldots, r_n\}$, where r_i denotes the ith user access record in the log. Then, for any $1 \leqslant i \leqslant j \leqslant n$, we have $t(r_i) \leqslant t(r_j)$.

The behaviors of web users can be understood with the concept of user access sessions. Informally, a user access session is the set, ordered or not, of all the web pages from the time the user starts to search for something to the time she quits the search. E.g., when a person in the US wants to find a tour map of Hong Kong over the web, she starts at a time to search `www.yahoo.com`. She spends 30 minutes to visit many web pages in order to find a map. The access session for this person is the set of all the web pages she has visited during her 30 minutes endeavor. Unfortunately, due to the vast diversity of web users it is difficult to *precisely* define what is a user access session. In this paper, we propose two ways to define user access sessions in the following. Similar definitions can be found in [3,10,15,16].

1. α-interval sessions: the duration of a session may not exceed a threshold of α.

   Let $\mathcal{L}$ be a web log and α a positive value. Given any user u, let $r_1, \ldots, r_m$ be the ordered list of u's access records in $\mathcal{L}$. The α-interval sessions for u is a list of subsets $s_1 = \{r_{i_0+1}, \ldots, r_{i_1}\}, s_2 = \{r_{i_1+1}, \ldots, r_{i_2}\}, \ldots, s_k = \{r_{i_{k-1}+1}, \ldots, r_{i_k}\}$ such that $s_1 \cup s_2 \cup \cdots \cup s_k = \{r_1, \ldots, r_m\}$, and for $1 \leqslant j \leqslant k$ we have

   $$0 \leqslant t(r_{i_j}) - t(r_{i_{j-1}+1}) \leqslant \alpha,$$
   $$t(r_{i_j+1}) - t(r_{i_{j-1}+1}) > \alpha,$$

   where $r_{i_0+1} = r_1$ and $r_{i_k} = r_m$.

2. β-gap sessions: the time between any two consecutively assessed pages may not exceed a threshold of β.

   Let $\mathcal{L}$ be a web log and β a positive value. Given any user u, let $r_1, \ldots, r_m$ be the ordered list of u's access records in $\mathcal{L}$. The β-gap sessions for u is a list of subsets $s_1 = \{r_{i_0+1}, \ldots, r_{i_1}\}, s_2 = \{r_{i_1+1}, \ldots, r_{i_2}\}, \ldots, s_k = \{r_{i_{k-1}+1}, \ldots, r_{i_k}\}$ such that $s_1 \cup s_2 \cup \cdots \cup s_k = \{r_1, \ldots, r_m\}$, also for $1 \leqslant j \leqslant k$ and for $i_{j-1} + 1 \leqslant l < i_j$ we have

$$0 \leqslant t(r_{l+1}) - t(r_l) \leqslant \beta,$$
$$t(r_{i_j+1}) - t(r_{i_j}) > \beta,$$

where $r_{i_0+1} = r_1$ and $r_{i_k} = r_m$.

In the α-interval session definition, we assume that a user should not spend too much time for each session. So we give a value α to limit the time a user can have for a session. Usually, we may set α to 30 minutes.

In the β-gap session definition, we assume that a user should not be "idle" for too much time between any two consecutive accesses in any session. So we give a value β to limit the time a user can be "idle" between two consecutive accesses in a session. Usually, we may set β to 30 minutes.

Certainly, there are exceptions to the α and β limits. For example, a person may need to spend a few hours to find a right tour map of Hong Kong over the web, and so the session exceeds the α limit. However, recent studies show that these two kinds of limits are quite accurate in web usage analysis [3].

3. Data structures and algorithms

Here we propose two efficient algorithms for finding interval sessions and gap sessions respectively. For each new user we maintain a "*user node*". Each "*user node*" will have a URL linked list to store the web pages the user has accessed. We cannot allow user nodes and URL linked lists to exceed the limit of RAM. We overcome the difficulty with the help of Proposition 1, the sorted property of the web log. When an access record of a new user arrives, we create a user node for her; when a new access record of some existing user arrives, we check the time of this new record and start time (or the current time) of the user node to determine whether the α threshold is exceeded for interval sessions (or the β threshold is exceeded for gap sessions), so that we know how to control the size of the URL linked list for the user. After we have processed certain number of valid user access records, we purge all those user nodes whose start (or current) times are beyond the α (or β) threshold in comparison with the time of the most recent record processed, thus the number of user nodes is under control.

3.1. Data structures

Our algorithms maintain a data structure as shown in Figure 1. We define a URL node structure to store the URL and the access time of a user access record and a pointer to point to the next URL node. We define a user node to store the following information for a user: the user ID that is usually the hostname or IP address in the access record; the start time that is the time of the first access record entered into the user node; the current time that is the time of last access record entered into the user node; the URL node pointer that points to the linked list of URL nodes; the counter to record the total number of URL nodes added; and two user node pointers to respectively point to the previous user node and

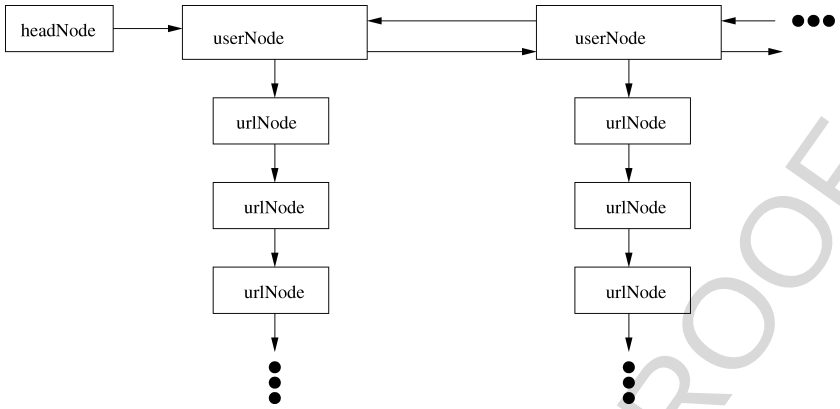


Figure 1. Data structures.

```
struct urlNodeType{
    urlNodeType *next;
    char *urlName;
    long int accessTime;
}
struct headNodeType{
    userNodeType *head;
    int counter;
}
```

```
struct userNodeType{
    userNodeType *prevUserNodePtr;
    userNodeType *nextUserNodePtr;
    char* usrID;
    long int sessionStartTime;
    long int currentAccessTime;
    int urlNumber;
    urlNodeType *listPtr;
}
```

Figure 2. Data structure definitions.

Table 1. Thresholds used in session finding algorithms

Threshold	Purpose
α	Threshold for defining interval sessions
β	Threshold for defining gap sessions
γ	Threshold for removing old user nodes

the next. Finally, we define a head node structure with two elements, one pointing to the beginning of the two-way linked list of user nodes and the other storing the total number of valid records that have been processed so far. The actual definitions are given in Figure 2.

We shall use the following symbols to denote thresholds throughout the rest of the paper.

3.2. *Finding interval sessions*

We now present our algorithm for finding α-interval sessions. This algorithm maintains a two-way linked list of user nodes as defined in the previous subsection. At a higher level,

Algorithm ISessionizer (Interval Sessionizer):

```
    input:
        infile: input web log file
        outfile: output user access session file
        α > 0: threshold for defining interval sessions
        γ > 0: threshold for removing old user nodes
    begin
1.      open infile and outfile
2.      createHeadNode(S); S.head = null; S.counter = 0
3.      while (infile is not empty)
4.              readRecord(infile, r)
5.              if (isRecordValid(r))
6.                      n = findRecord(S, r)
7.                      if (n is null)
8.                              addRecord(S, r)
9.                      else if (t(r) − n.startTime ⩽ α)
10.                             n.currentTime = t(r)
11.                             addURL(n.urlListPtr, r)
12.                     else
13.                             writeSessionAndReset(outfile, n, r)
14.             S.counter = S.counter + 1
15.             if (S.counter > γ)
16.                     purgeNodes(outfile, S, t(r)); S.counter = 1
17.     cleanList(outfile, S);
18.     close infile and outfile
    end
```

Figure 3. Algorithm ISessionizer.

the algorithm repeats the following tasks until the end of the web log. (1) Read a user access record r from disk to RAM. (2) Test whether r is valid or not. (3) If r is valid then add r to its corresponding user node D in the user node linked list. (4) Compare the start time of D with the time $t(r)$ to see whether a session is found for the user. If yes then output the session and reset D with information in r, otherwise add the URL and the access time of r to the URL linked list of D. Finally, (5) check if sufficient valid records have been processed so far. If yes, then purge all the old user nodes and output their sessions. The algorithm, called ISessionizer, is given in Figure 3. We now explain in details the functions we used in algorithm ISessionizer.

`CreateHeadNode`(S) creates a record S of *headNodeType*.

`readRecord`(*infile*, r) reads a user access record from the input web log file *infile* and stores the record at r.

`isRecordValid`(r) checks whether the user access record is valid or not. The purpose of this function is to clean the raw web log to eliminate those outliers or irrelevant

records. Depending on the particular application requirements, there are different criteria for judging whether a record is valid or not. E.g., any access record with a return code 400 or above may be considered invalid; accesses to image or audio files may be considered invalid; and accesses performed by automatic web indexers may also be considered invalid.

`findRecord`(S, r) searches the linked list *S.head* of user nodes. If it finds a node with the same user ID as the record r, then it returns the node, otherwise it returns *null*.

`addRecord`(S, r) creates a new node of userNodeType with information of the access record r and adds the new node to the linked list *S.head*.

`addURL`(*n.listPtr*, r) creates a new node of *urlNodeType* with information from r and adds the new node to the URL linked list *n.listPtr*.

`writeSessionAndReset`(*outfile*, n, r) first outputs the session consisting of all the URLs in the URL linked list *n.listPtr* to the output file *outfile*. It then deallocates memory of the URL linked list *n.listPtr*. It finally resets the user node n with information of r, i.e., setting

$$n.startTime = t(r),\ n.currentTime = t(r),\ \text{and}\ n.urlNumber = 1,$$

and creating a new *urlNodeType* node with the URL and the access time of r and letting *n.listPtr* point to the new node.

`purgeNodes`(*outfile*, $S, t(r)$) searches every user node n in the linked list *S.head*. If it finds that $t(r) - n.startTime > \alpha$, then it outputs the session consisting of the URLs in *n.listPtr* to *outfile*, and then deletes the URL linked list *n.listPtr* and the user node n via memory deallocation.

`clean`(*outfile*, S) is called when the end of the log is reached. This function outputs, for every user node, the session consisting all URLs in the URL linked list of the user node. It then deallocates memory for URL linked lists and the user linked list.

3.3. Finding gap sessions

The gap session finding algorithm, called GSessionizer, is similar to algorithm ISessionizer. They differ at steps 9 and 16. At step 9 GSessionizer checks, in order to decide whether a session is found or not, whether the "time gap" between the current record and the last record of the same user is beyond the threshold β or not. At step 16, algorithm GSessionizer will call function *purge2Nodes*(*outfile*, $S, t(r)$) to check, for every user node in the user node linked list *S.head*, whether the "time gap" between the current record and the last record of the user node is beyond the threshold β or not. If so, it purges the user node in a similar manner as algorithm ISessionizer does. The description of GSessionizer is given in Figure 4.

Remark 1. We should point out that different thresholds α, β and γ may be needed for different applications. Usually, we may choose $\alpha = \beta = 30$ minutes. One may choose $\gamma = 500$ or a very large value if large RAM is available.

Algorithm GSessionizer (Gap Sessionizer):

input:

infile: input web log file

outfile: output user access session file

$\beta > 0$: threshold for defining gap sessions

$\gamma > 0$: threshold for removing old user nodes

This algorithm is the same as algorithm ISessionier except that Line 9 is replaced by

9. else if $(t(r) - n.currentTime \leqslant \beta)$

and Line 16 is replaced by

16. purge2Nodes(*outfile*, $S, t(r)$); $S.counter = 1$

Figure 4. Algorithm GSessionizer.

4. Theoretical analysis

In this section we give theoretical analysis of algorithms ISessionizer and GSessionizer. We will prove that both algorithms are correct and have optimal time complexity. We will also prove that both algorithms have certain error-tolerant properties, i.e., when some user access records are missing from the web log they are still able to find correct sessions for all the users whose records are not missing.

4.1. Theoretical analysis of algorithm ISessionizer

We first prove the correctness of algorithm ISessionizer.

Theorem 1. Given $\alpha > 0$, let $\mathcal{S}$ denote the set of all the α-interval sessions in $\mathcal{L}$. Then, algorithm ISessionizer finds exactly the set $\mathcal{S}$.

Proof: For any given user u, suppose that all the α-interval sessions performed by u are $s_i = \{r(i)_1, \ldots, r(i)_{j_i}\}$, $i = 1, \ldots, m$. Without loss of generality, we assume that s_i are sorted in increasing order based on the access time $t(r(i)_1)$. In order to prove the theorem, it is sufficient to prove that the algorithm finds exactly all the interval sessions s_i for the user u. Since the algorithm reads user access records one at a time sequentially from the log $\mathcal{L}$, the algorithm will reads $r(1)_1$ at some time T. Because $r(1)_1$ is the very first valid access record performed by the user u, there is no user node for u in the user linked list *S.head* at the time T. Hence, at this point, the algorithm will create a new user node D for u and adds it to the linked list *S.head*. For any k with $1 \leqslant k \leqslant j_1$, the algorithm will continue to read $r(1)_k$. Because s_1 is an α-interval session for u, we have

$$t\big(r(1)_k\big) - t\big(r(1)_1\big) \leqslant \alpha.$$

This implies that the algorithm will add $r(1)_k$ into the URL linked list $D.listPtr$. Moreover, the above property implies that even if a *purgeNodes*( ) function call occurred before

or at the time the algorithm added the access record $r(1)_k$, the algorithm will not purge the user node D, since the algorithm only purges those user nodes whose start times are α below the access time of the current access record being processed.

After the algorithm has read $r(1)_1, \ldots, r(1)_{j_1}$ and added them into the URL linked list *D.listPtr*, we have to consider the following cases.

CASE 1: The algorithm made some calls of *purgeNodes*( ) after it read $r(1)_{j_1}$ but before it reads $r(2)_1$. This means that the algorithm read some other user's access record r with

$$t\big(r(1)_{j_1}\big) < t(r) < t\big(r(2)_1\big)$$

and called *purgeNodes*( ) based on the time $t(r)$. We have two subcases to consider. In the first subcase, we have $t(r) - t(r(1)_1) > \alpha$. This means that the algorithm would purge the user node D, and thus output the session s_1. After the purging, the linked list *S.head* has no node for the user u. Hence, when the algorithm read the user u's access record $r(2)_1$, it would create a new user node for u and add it into the linked list *S.head*.

In the second subcase we have $t(r) - t(r(1)_1) \leqslant \alpha$. This means that the user node D was not qualified to be purged so it would stay in the user node linked list *S.head*. Let us consider how the algorithm works when it reads the user u's access record $r(2)_1$. Because both s_1 and s_2 are α-interval sessions and, they are sorted in increasing order of their start times, we have

$$t\big(r(2)_1\big) - t\big(r(1)_1\big) > \alpha.$$

Hence, at this point the algorithm will call the function *writeSessionAndReset*( ) to output the session s_1 consisting of all the URLs in the URL linked list *D.listPtr*, and reset the user node D with information of $r(2)_1$.

CASE 2: The algorithm has not made any calls of the function *purgeNodes*( ) after reading $r(1)_{j_1}$ but before reading $r(2)_1$. This simply implies that the user node D remains in the linked list *S.head*. As in the second subcase above, when the algorithm reads the user u's access record $r(2)_1$, it calls the function *writeSessionAndReset*( ) to output the session s_1 consisting of all the URLs in the URL linked list *D.listPtr* and to reset D with information of $r(2)_1$.

In either case, before or when the algorithm reads the user u's access record $r(2)_1$, the algorithm finds and outputs the session s_1 and, the user node D will be created (or reset) with information of $r(2)_1$ in the user linked list *S.head*. Following the similar process, the algorithm finds sessions s_i before or at the point it reads u's access record $r(i+1)_1$, $2 \leqslant i \leqslant m-1$. After the algorithm reads $r(m)_1$, in the similar way to the process of s_1, the algorithm reads $r(m)_k$ for $1 \leqslant k \leqslant j_m$ and adds it into the URL linked list *D.listPtr*. Since there are no more u's access records in the log, the algorithm outputs the session s_m either in some later call of the function *purgeNodes*( ) or the function *cleanList*(*outfile,S*) when it reaches the end of the log. □

We introduce several notations in Table 2.

Theorem 2. Assume that the web log $\mathcal{L}$ have N user access records with M valid ones. Then, the time complexity of the α-interval session finding algorithm ISessionizer with a

Table 2. Definitions of four time values

Notation	Definition
T_1	Time to read one record from the web log $\mathcal{L}$
T_2	Time to check whether a record is valid or not
T_3	Time to write the URL and the access time of a valid record from RAM to an output file
T_4	Time to search once the user linked list *S.head* and the URL linked list at each user node

γ-threshold for purging user nodes is $\theta(N)$. Furthermore, let T_1, T_2, T_3 and T_4 be defined as in Table 2, then the actual time needed by algorithm ISessionizer for finding sessions from $\mathcal{L}$ is at most

$$(T_1 + T_2)N + T_3M + T_4\left(1 + \frac{1}{\gamma}\right)M.$$

Proof: The α-interval session finding algorithm ISessionizer has a main loop controlled by the number of user access records in the web log $\mathcal{L}$. For each loop iteration, the algorithm performs four tasks as follows. (1) It reads a user access record from $\mathcal{L}$. (2) It checks whether a user access record is valid or not. (3) For each valid record, it searches the user linked list to add a new user node, or to add a new URL node, or to output a session and to reset a user node. (4) For every sequence of γ many valid records, searches the user linked list once to purge all those user nodes that are too old to admit new access records. When the two parameters α and γ are given, any of those four tasks can be done in constant time. Hence, the time complexity of algorithm ISessionizer has an upper bound $O(N)$. Obviously, in order to find sessions from $\mathcal{L}$, any algorithm must process every user access record in $\mathcal{L}$. Thus, the time complexity of algorithm ISessionizer has a lower bound $\Omega(N)$. Therefore, its time complexity is $\theta(N)$.

We can also estimate the actual time needed by algorithms ISessionizer. The actual time needed for performing task (1) is T_1N, for task (2) is T_2N, and for tasks (3) and (4) combined is at most $(T_3 + T_4(1 + 1/\gamma))M$. Therefore, the total actual time needed by the algorithm is at most $(T_1 + T_2)N + T_3M + T_4(1 + 1/\gamma)M$. □

It is easy to see that T_1 and T_2 are two constants determined by the speed of the computer. For any particular application, T_3 is also a constant. However, we have some remarks for T_4:

Remark 2. T_4 is determined by the size of the algorithm's data structure, i.e., the number of user nodes and the total number of URL's nodes at any given time. This size is determined by the number of users using the web server and the number of accesses performed by those users within the session threshold α. When $\alpha = 30$ minutes, for a not heavily accessed web server, the size varies from a few hundreds to a few thousands. For a heavily accessed web server, the size may be as large as hundreds of thousands. But, the size is still bounded by some constant. Most importantly, all computing of task (4) is carried out in RAM. Hence, T_4 may be a relatively smaller constant than T_1 and T_3.

Remark 3. The baseline time needed by any session finding algorithm is at least $(T_1 + T_2)N + T_3M$, because any session finding algorithm needs to read all user access record once from disk to RAM, to check whether any record is valid or not, and to write URLs and access times of valid records from RAM back to disk.

Remark 4. Algorithm ISessionizer is optimal, because it has linear time complexity. Moreover, its actual time is just $T_4(1 + 1/\gamma)M$ more than the baseline time of session finding.

In the following we will show that our algorithm is error-tolerant in the sense that when some user access records are missing from the web log, the algorithm is still able to find all the correct sessions for all users whose records are not missing. Due to wide applications of local cashing or proxy servers, the real session of a user may not be saved in the web log. E.g., the real session of p_1, p_2, p_1, p_3 may be saved as p_1, p_2, p_3, because the second access p_1 may not be logged due to local cashing. Hence having the ability to tolerate missing records is a necessary quality for a session finding algorithm.

Theorem 3. Given any user access session $s = \{r_1, \ldots, r_m\}$ of user u, assume that a record r_k with $1 \leqslant k \leqslant m$ is missing from the web server log. If $1 < k \leqslant m$, then algorithm ISessionizer finds all the sessions in the log except s. For the session s, algorithm ISessionizer finds $s' = \{r_1, \ldots, r_{k-1}, r_{k+1}, \ldots, r_m\}$. If $k = 1$, then algorithm ISessionizer finds all the sessions of all the users other than u, and all u's sessions performed before the time $t(r_1)$, however it may not find u's sessions performed after the time $t(r_1)$.

Proof: Since missing r_k from the log will not affect the sessions of all the other users, the algorithm finds all the sessions for all the other users. We consider the case of $1 < k \leqslant m$. For the user u, missing r_k will not affect u's sessions performed before the time $t(r_1)$, and the algorithm finds all those sessions. When the algorithm reads the record r_1, it will either create a new user node D for u (or reset an existing user node D of u) with information of r. The node D has $t(r_1)$ as its start time. The algorithm then reads and stores the records $r_2, \ldots, r_{k-1}, r_{k+1}, \ldots, r_m$. If the user does not have any access past r_m, then it finds the session s' when a later call of *purgeNodes* is made or when the end of the log is reached. If the user u has sessions past r_m, then consider the session $p = \{p_1, \ldots, p_l\}$ that is closest to r_m among those past r_m. Because s and p are both sessions, we have $t(p_1) - t(r_1) > \alpha$. Hence, at the time the algorithm reads p_1, it either has found the session s' before or at this moment. In the first case, at this moment the user linked list *S.head* of the algorithm does not have a user node for u, so it creates a new user node for u with information of p_1 and adds it into the linked list. In the second case, it outputs the session s' and resets u's existing user node with information of p_1. In either case, at this point the algorithm has a user node for u with the start time of $t(p_1)$. From this point on, the algorithm finds the rest of u's sessions.

When $k = 1$, i.e., r_1 is missing from the web log, as above the algorithm finds all sessions for users other than u, and all u's sessions performed before the time $t(r_1)$. In the following we show by a counterexample that the algorithm may not find u's sessions

performed after the time $t(r_1)$. Let $s = \{r(0)_1, \ldots, r(0)_{j_0}\} = \{r_1, \ldots, r_m\}$. Consider that the user has a sequence of sessions $\{r(i)_1, \ldots, r(i)_{j_i}\}$, $1 \leqslant i \leqslant m$, that have been performed after the time $t(r_1)$. Assume for $i = 0, \ldots, m-1$,

$$t\big(r(i+1)_1\big) - t\big(r(i)_2\big) \leqslant \alpha \quad \text{and} \quad t\big(r(i+1)_2\big) - t\big(r(i)_2\big) > \alpha.$$

Because $r(0)_1 = r_1$ is missing from the web log, when $r(0)_2 = r_2$ is read from the log the algorithm will use the access record $r(0)_2$ to create or reset a user node for u. That is, the start time of the user node of u is $t(r(0)_2)$. Thus, by the above assumption, the algorithm finds the following incorrect sessions for u: $\{r(0)_2, \ldots, r(0)_{j_0}, r(1)_1\}, \{r(1)_2, \ldots, r(1)_{j_1}, r(2)_1\}, \ldots, \{r(m-1)_2, \ldots, r(m-1)_{j_{m-1}}, r(m)_1\}, \{r(m)_2, \ldots, r(m)_{j_m}\}$. □

4.2. *Theoretical analysis of algorithm GSessionizer*

Algorithm GSessionizer is almost the same as algorithm ISessionizer. They differ only at steps 9 and 16. At step 9, in order to decide whether a session is found or not, algorithm GSessionizer checks whether the "time gap" between the current record and the last record of the same user is beyond the threshold β or not, while algorithm ISessionizer checks whether the "time gap" between the current record and the first record is beyond the threshold α or not. At step 16, algorithm GSessionizer checks for every user node in the user linked list *S.head* whether the "time gap" between the current record and the last record of the user node is beyond the threshold β or not and if so it purges the user node, while algorithm ISessionizer checks whether the "time gap" between the current record and the first record in *S.head* is beyond the threshold α or not and if so it purges the user node. Because of the above similarity, theoretical analysis for algorithm GSessionizer can be done in the same way as for algorithm ISessionizer. So we just state the following theorems for algorithm GSessionizer but leave the proofs for the interested readers.

Theorem 4. Given $\beta > 0$, let $\mathcal{S}$ denote the set of all the β-gap sessions in $\mathcal{L}$. Then, algorithm GSessionizer finds exactly the set $\mathcal{S}$.

Theorem 5. Assume that the web log $\mathcal{L}$ have N user access records with M valid ones. Then, the time complexity of the β-gap session finding algorithm GSessionizer with a γ-threshold for purging user nodes is $\theta(N)$. Furthermore, the actual time needed by algorithm GSessionizer for finding sessions from $\mathcal{L}$ is at most

$$(T_1 + T_2)N + T_3 M + T_4\left(1 + \frac{1}{\gamma}\right)M,$$

where T_i, $1 \leqslant i \leqslant 4$, are defined in Table 2.

Theorem 6. Given any user access session $s = \{r_1, \ldots, r_m\}$ of the user u, assume that a record r_k with $1 \leqslant k \leqslant m$ is missing from the web server log. Then, algorithm GSes-

sionizer finds all the sessions in the log except s. For the session s, the algorithm finds $s' = \{r_1, \ldots, r_{k-1}, r_{k+1}, \ldots, r_m\}$.

We should point out that Theorem 6 is a little bit stronger than Theorem 3. In Theorem 6, even if the first record of some user's access session s is missing from the log, algorithm GSessionizer can still find all the correct sessions other than s for the user. This is true, because missing the first record of s will not decrease the "*time gap*" between the last record of s and the first record of the user's session that is past but closest to s. We sketch the proof for Theorem 6 as follows. For any given session $s = \{r_1, \ldots, r_m\}$ of the user u, consider any u's session $q_1 = \{r'_1, \ldots, r'_u\}$ before s and any u's session $q_2 = \{r''_1, \ldots, r''_v\}$ after s. According to the β-gap session definition, we have $t(r_1) - t(r'_u) \geqslant \beta$ and $t(r''_1) - t(r_m) \geqslant \beta$. Thus, by Proposition 1, we have for any j with $1 \leqslant j \leqslant m$, $t(r_j) - t(r'_u) \geqslant \beta$ and $t(r''_1) - t(r_j) \geqslant \beta$. When r_k is missing from the session s, the time gap between the last record r'_u in the session q_1 and any remaining record in s is at least β. If algorithm GSessionizer has not identified the session q_1 before reading the first remaining record r_j in s, then when the first remaining record r_j in s is read, algorithm GSessionizer will use the access time $t(r_j)$ to identify the session q_1. Moreover, since $t(r''_1) - t(r_j) \geqslant \beta$ for $1 \leqslant j \leqslant m$, by the β-gap session definition, r''_1 does not belong to s or s', no matter whether r_k is missing from s or not. Hence, algorithm GSessionizer finds the session s' when r''_1 is read, and the information of r''_1 is used to reset the user node for u for finding the session q_2. Note that the nice property in Theorem 6 also holds if any subset of s is missing.

Remark 5. When a record r_k is missing from a user's access session $s = \{r_1, \ldots, r_m\}$ due to local caching, the best that a session finding algorithm can do is to find $s' = \{r_1, \ldots, r_{k-1}, r_{k+1}, \ldots, r_m\}$ correctly. Hence, the error-tolerant property exhibited in Theorem 6 guarantees that algorithm GSessionizer finds correctly all β-gap sessions for all users, no matter whether local caching is used at client computers or not.

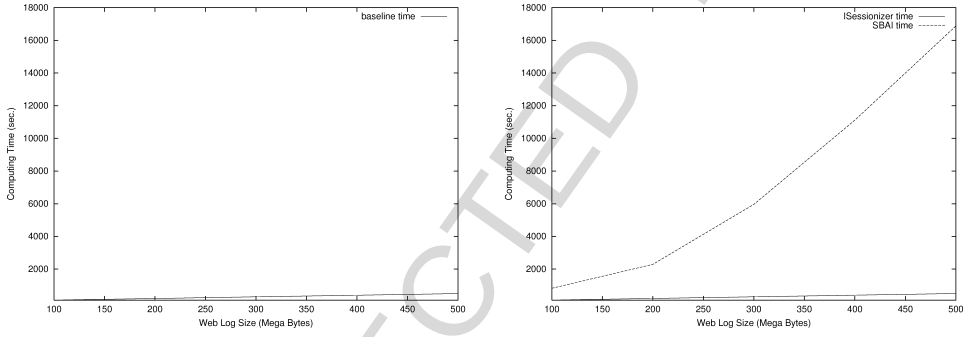
5. Empirical analysis

In this section we report empirical performance analysis of algorithms ISessionizer and GSessionizer. Although web log mining has been studied extensively recently (e.g., [2,4–9,14,17,21]), yet, to our best knowledge, no algorithmic study of efficient finding of user access sessions has been reported in the literature. The obvious session finding algorithm would be the one that sorts the log to list each individual user's access records together as a group and then to cut each group into sessions. In the following we shall give two separate sorting based algorithms for finding respectively α-interval sessions and β-gap sessions. We shall compare the performance of our algorithms ISessionizer and GSessionizer with that of those two sorting based algorithms.

We use five web logs *LOG100MB, LOG200MB, LOG300MB, LOG400MB* and *LOG500MB* with respectively 100, 200, 300, 400, and 500 megabytes of user access records that were collected from the web server of the Department of Computer Science,

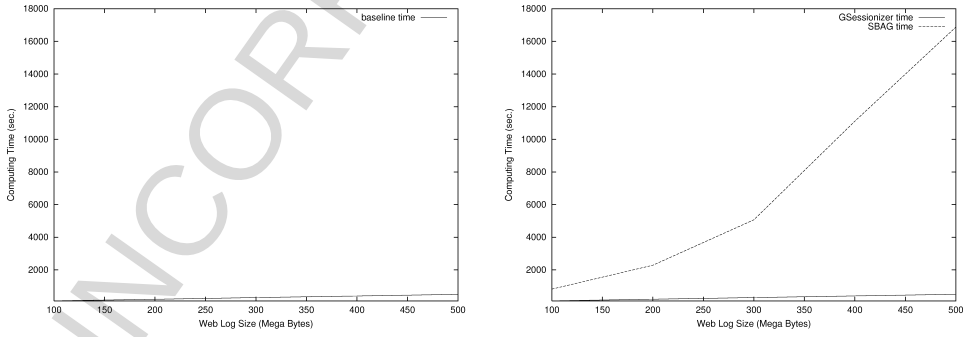
the University of Texas-Pan American. The computing environment is a Dell OptiPlex GX1 PC with 512 megabytes of RAM and 8 gigabytes of hard disk. The baseline time is the time needed for reading a web log once sequentially from disk to RAM, testing whether each user access record is valid or not, and writing each valid user access record back to disk.

The sorting based algorithm, denoted by *SBAI*, for finding α-interval sessions is the one that sorts the web log to group the same user's access records together and then reads the sorted log sequentially once from disk to RAM so that the algorithm divides the valid access records in each group into α-interval sessions. The sorting based algorithm, denoted by *SBAG*, for finding β-gap sessions works similarly as algorithm SBAI, but it divides the valid access records in each group into β-sessions. We utilize the sort command in the Unix environment, which supports a standard external file sorting. We choose $\beta = \alpha = 30$ minutes and $\gamma = 500$. We use the same *isRecordValid*( ) function for the four algorithms and the baseline time testing as well. Performance comparisons are illustrated in Figures 5 and 6.



(a) Baseline Performance (b) ISessionizer vs. SBAI

Figure 5. Performances of algorithm ISessionizer vs. SBAI vs. baseline time.



(a) Baseline Performance (b) GSessionizer vs. SBAG

Figure 6. Performances of algorithm GSessionizer vs. SBAG vs. baseline time.

VTEX(JK) PIPS No:5139849 artty:res (Kluwer BO v.2003/05/06)
j5139849.tex; 4/06/2003; 9:46; p. 15

From Figures 5 and 6, our empirical analysis shows that both algorithms ISessionizer and GSessionizer have almost the same performance that is just several seconds more than their baseline time performance, and they are much faster than the sorting based algorithms SBAI and SBAG. E.g., to find sessions from a web log of 500 megabytes of user access records, the baseline time is 501 seconds, algorithm ISessionizer needs 508 seconds in comparison with 16880 seconds needed by the sorting based algorithm SBAI; and algorithm GSessionizer needs 504 seconds in comparison with 16877 seconds needed by the sorting based algorithm SBAG. In this case, both algorithms ISessionizer and GSessionizer are more than 32 times faster than the sorting based algorithms!

6. Finding user access sessions from distributed web logs

In a heavily accessed web server, the logging process of user access records may be distributed for many good reasons. One is to reduce the workload of the server. Another is that the architecture of the server itself may be distributed: the main server has several distributed proxy servers so that when a user access the main server, the user's request may be automatically dispatched to one of the proxy servers. Thus, in this case, users' access records are logged in several distributed files.

Let $\mathcal{L}_1, \ldots, \mathcal{L}_d$ be d distributed web logs. We design two optimal algorithms to identify user access sessions from the union of those distributed logs $\mathcal{L}_1 \cup \cdots \cup \mathcal{L}_d$. We know that the sorted property stated in Proposition 1 still holds for each log $\mathcal{L}_i, 1 \leqslant i \leqslant d$. The difficulty we are facing now is that records in a given user access session may be scattered among those d logs. Our new approach is as follows:

Read one user access record r_i from log $\mathcal{L}_i$ for $1 \leqslant i \leqslant d$.
Repeat
    Select the oldest record r_j from $r_1, \ldots, r_d$.
    Input r_j to algorithm ISessionizer (or GSessionizer).
    Read the next record r_j from log $\mathcal{L}_j$.
until all logs are empty

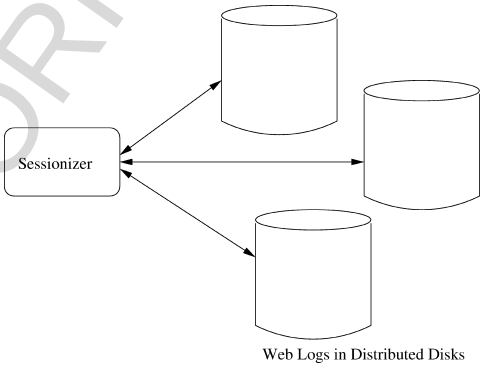


Figure 7. Distributed web logs.

In Figures 8 and 9, We present our algorithms DISessionizer and DGSessionizer for finding α-interval sessions and β-gap sessions from distributed logs, respectively.

One should notice that the kernels of algorithms DISessionizer and DGSessionizer are respectively algorithms ISessionizer and GSessionizer. The extra function *findOldestRecord*( ) finds which of the d records is the oldest and returns the index of that record to the integer variable j. The function *readRecord*($\mathcal{L}_j, r_j$) reads the next record from log $\mathcal{L}_j$ and stores it at r_j if $\mathcal{L}_j$ is not empty, otherwise it returns *null* to r_j. *purge2Nodes*(*outfile*, $S, t(r)$) checks, for every user node in the user node linked list $S.head$, whether the "time gap" between the current record and the last record of the user node is beyond the threshold β or not. If so, it purges the user node in a similar manner as algorithm ISessionizer does. Because the sorted property of Proposition 1 holds for each log $\mathcal{L}_i$, $1 \leqslant i \leqslant d$, the correctness of both algorithms follows in the same ways as algorithms ISessionizer and GSessionizer. One can also show that the time complexity of each

Algorithm DISessionizer (Distributed Interval Sessionizer):

input:
$\mathcal{L}_1, \ldots, \mathcal{L}_d$: input distributed web logs
$\mathcal{F}$: output user access session file
$\alpha > 0$: threshold for defining interval sessions
$\gamma > 0$: threshold for removing old user nodes

```
begin
1.    open L_1, ..., L_d and F
2     readRecord(r_i, L_i)
3.    createHeadNode(S); S.head = null; S.counter = 0
4.    while (L_1, ..., L_d are not all empty)
5.          j = findOldestRecord(r_1, ..., r_d)
6.          if (isRecordValid(r_j))
7.                n = findRecord(S, r_j)
8.                if (n is null)
9.                      addRecord(S, r_j)
10.               else if (t(r_j) − n.startTime ⩽ α)
11.                     n.currentTime = t(r_j)
12.                     addURL(n.urlListPtr, r_j)
13.               else
14.                     writeSessionAndReset(F, n, r_j)
15.         S.counter = S.counter + 1
16.         if (S.counter > γ)
17.               purgeNodes(F, S, t(r_j)); S.counter = 1
18.         readRecord(L_j, r_j)
19.   cleanList(S, F)
20.   close F, L_1, ..., L_d
end
```

Figure 8. Algorithms DISessionizer.

Algorithm DGSessionizer (Distributed Gap Sessionizer):
input:
$\mathcal{L}_1, \ldots, \mathcal{L}_d$: input distributed web logs
$\mathcal{F}$: output user access session file
$\beta > 0$: threshold for defining gap sessions
$\gamma > 0$: threshold for removing old user nodes
This algorithm is the same as DISessionizer except
Line 10 is replaced by
10. else if $(t(r_j) - n.currentTime \leqslant \beta)$
and Line 17 is replaced by
17. purge2Nodes$(\mathcal{F}, S, t(r_j))$; $S.counter = 1$

Figure 9. Algorithm DGSessionizer.

of algorithms DISessionizer and DGSessionizer is optimally $\theta(N)$, where N is the total number of records in the union of the d logs. Furthermore, the actual time needed by each of algorithms DISessionizer and DGSessionizer for finding user access sessions for the d logs is at most $(T_1' + T_2')N + T_3'M + T_4'(1 + 1/\gamma)M$, where T_1' is the time needed to read one record from $\mathcal{L}_i$, $1 \leqslant i \leqslant d$, T_2' is the time needed to find the oldest record r_j from d records $r_1, \ldots, r_d$ and to check whether the oldest record r_j is valid or not, T_3' is the time needed to write the URL and the access time of a valid record to the output file, T_4' is the time needed to search once the user linked list $S.head$ and the URL linked list at each user node, and finally M is the total of valid records in the union of $\mathcal{L}_1 \cup \cdots \cup \mathcal{L}_d$. Similarly, one can also show that algorithms DISessionizer and DGSessionizer have the similar error tolerant property as algorithms ISessionizer and GSessionizer, i.e., when records of some users are missing from the logs, the algorithms find all the correct sessions for all the other users.

Remark 6. Algorithms DISessionizer and DGSessionizer find user access sessions from distributed web logs such as logs at several distributed proxy web servers of the main web server. The error-tolerant property exhibited in Theorem 6 guarantees that algorithm DGSessionizer finds correctly all β-gap sessions for all users, no mater whether proxy servers have been used for the main web server or not, or no matter whether local cashing has been used at client computers or not.

7. Hashing vs. linear search

In our implementation of algorithms ISessionizer and GSessionizer, the functions *findRecord*(S, r), *addRecord*(S, r) and *addURL*$(n.listPtr, r)$ are based on linear search. Those three functions can be replaced by other more efficient methods such as hashing. In this section, we consider the use of hashing. The hashing we apply is given by $H(x) = f(x) \ldots key$ where x is a string of characters, f is function to convert a string of characters to an integer, and key is the prime modulus to do the mod operation.

Table 3. Performance differences of hashing vs. linear search

Hashing function	100 MB log	200 MB log	300 MB log	400 MB log	500 MB log
H_1 with $key = 3373$	−0.14	−0.221	−0.091	−0.26	+0.025
H_2 with $key = 29$	−0.431	−0.391	−0.381	−1.102	−0.271
H_3 with $key = 203$	−0.411	−0.471	−0.171	−0.771	−0.441
H_4 with $key = 5003$	+0.601	+0.804	+1.982	+1.352	+2.984

We has experimented with four different hashing functions:

H_1 = hashing the full hostname (i.e., the user ID);

H_2 = hashing the first character of the hostname;

H_3 = hashing the first two characters of the hostname; and

H_4 = hashing the polynomial value of the full hostname.

More precisely, we consider each character as an integer given by its ASCII code value. Let $x = x_1x_1 \cdots x_n$ be a string of n characters in ASCII codes. We have

$$H_1(x) = (x_1 + x_2 + \cdots + x_n) \ldots 3373,$$
$$H_2(x) = x_1 \ldots 29,$$
$$H_3(x) = (x_1 + x_2) \ldots 203,$$
$$H_4(x) = (x_1 10^{n-1} + x_2 10^{n-2} + \cdots + x_n 10^{n-n}) \ldots 5003.$$

We report the performance comparisons of hashing vs. linear search in Table 3. In the table, the first column gives the hashing functions used, and the other columns give the time differences of algorithm ISessionizer and the modified algorithm ISessionizer with the given hashing functions. A negative value at an entry means that the modified algorithm is slower than the original algorithm ISessionizer, while a positive value means that the modified algorithm is faster. Values in all entries are measured in seconds. It is easy to see that hashing functions cannot effectively improve the performance of the original algorithm ISessionizer, and sometimes can be even slower. Similar empirical results can be obtained for algorithm GSessionizer. However, we should point out that when the number of users using the web server and the number of accesses performed by users within a given session threshold are sufficiently large, then hashing functions can effectively improve the performances of algorithms ISessionizer and GSessionizer.

8. Conclusions

User access sessions are buried in very large web server logs with hundreds of megabytes or tens of gigabytes of user access records. For the success of higher level web log mining processes, identification of user access sessions is a necessary and unavoidable data preparation task, and efficient algorithms are needed to accomplish this task. However, there is little algorithmic study in the literature about how user access sessions may be identified

from very large web logs. We study in the paper the efficient identification of user access sessions from very large web logs. We consider two types of user access sessions, the α-interval sessions under the assumption that a user should not spend too much time for each sessions and the β-gap sessions under the assumption that a user should not be "*idle*" too much between two consecutive page accesses. Those two types of sessions have been used by other researchers (e.g., [2,9]). We realize that there are exceptions to those two assumptions, but recent studies show that those measures are quite accurate in web usage analysis [3].

We propose two algorithms for finding respectively α-interval sessions and β-gap sessions. We prove the correctness of the two algorithms and show that they have optimal time complexity. We also prove that when access records of some users are missing from the web log, the two algorithms can still find the correct sessions for all the other users and preserves much of the information for the users with missing records. We conduct empirical performance analysis with five logs of 100 to 500 megabytes of user access records. Our empirical results show that the performances of the two algorithms are just several seconds more than the baseline time, i.e., the time needed for reading the web log once sequentially from disk to RAM, testing whether each user access record is valid or not, and writing each valid user access record back to disk. The empirical results also shows that our algorithms are substantially faster than the sorting based session finding algorithms. In the case of the log of 500 megabytes, our algorithms are more than 33 times faster than the sorting based algorithms! We also design optimal algorithms for finding user access sessions from distributed web logs.

The approach of using *interval* and *gap thresholds* to divide the access records into sessions is an approximation to the *real sessions* the user has performed. It is worthwhile to study how to find more accurate approaches to real sessions of the users. One should realize that due to the protection of user privacy one should not reply on the applications of *cookies* or the related techniques to get answers from the users. Since the real sessions are buried in the vast web server log, it might be possible to use some unsupervised machine learning methods.

Recall that the actual time needed by our algorithms is propositional to T_4 (the time needed to search the linked list of the user nodes). The value of T_4 is linear in the total number L of user nodes that the algorithms may have at any moment. It is easy to understand that L solely depends on how may different users may access the web server within α or β amount of time. For a web server that is not heavily accessed, such as the web server of the Computer Science Department at some university, L is not very big. According to our statistics about the web server of the Computer Science Department at University of Texas-Pan American, L is just a few hundreds with an average value of 187 when α (or β) is set to 30 minutes. For a heavily accessed web server such as a popular search engine, L may be ranging from several thousands to tens of thousands. In such cases, smarter search algorithms should be adopted in our algorithms to search the data structures.

VTEX(JK) PIPS No:5139849 artty:res (Kluwer BO v.2003/05/06)
j5139849.tex; 4/06/2003; 9:46; p. 20

Acknowledgment

The authors thank Mr. Ho Pong Leung for implementing hash functions to replace the linear search functions in our session finding algorithms. This research is supported by the Hong Kong RGC Earmarked Grant UGC REF.CUHK 4179/01E.

References

[1] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. Inkeri Verkamo, "Fast discovery of association rules," *Advances in Knowledge Discovery and Data Mining*, 1996, 307–328.
[2] B. Berendt and M. Spiliopoulou, "Analysis of navigation behavior in web sites integrating multiple information systems," *The VLDB Journal* 9, 2000, 56–75.
[3] B. Berendt, B. Mobasher, M. Spiliopoulou, and J. Wiltshire, "Measuring the accuracy of sessionizers for web usage analysis," in *Proceedings of the Workshop on Web Mining at the First SIAM International Conference on Data Mining*, April 2001, 7–14.
[4] J. Borges and M. Levene, "Data mining of user navigation patterns," *MS99*, 1999.
[5] A. G. Buchner, M. Baumgarten, and S. S. Anand, "Navigation pattern discovery from internet data," *MS99*, 1999.
[6] A. G. Buchner and M. D. Mulvenna, "Discovering internet marketing intelligence through online analytical web usage mining," *ACM SIGMOD RECORD*, Dec. 1998, 54–61.
[7] L. Catledge and J. Pitkow, "Characterizing browsing behaviors on the world wide web," *Computer Networks and ISDN Systems* 27, 1995.
[8] M. S. Chen, J. S. Park, and P. S. Yu, "Efficient data mining for path traversal patterns," *IEEE Transactions on Knowledge and Data Engineering* 10(2), 1998, 209–221.
[9] R. Cooley, B. Mobasher, and J. Srivastava, "Web mining: Information and pattern discovery on the world wide web," in *Proc. IEEE Intl. Conference Tools with AI*, 1997.
[10] R. Cooley, B. Mobasher, and J. Srivastava, "Data preparation for mining world wide web browsing patterns," *Journal of Knowledge and Information Systems* 1(1), 1999.
[11] M. Perkowitz and O. Etzioni, "Adaptive web pages: Automatically synthesizing web pages," in *Proceedings of AAAI/IAAI'98* 1998, 727–732.
[12] J. Pitkow, "In search of reliable usage data on the WWW," in *Proceedings of the Sixth World Wide Web Conference*, Santa Clara, CA, 1997, 451–463.
[13] P. Pirolli, J. Pitkow, and R. Rao, "Silk from sow's ear: Extracting usable structures from the Web," in *Proceedings of the 1996 Conference on Human Factors in Computing Systems (CHI'96)*, Vancouver, British Columbia, Canada, 1996.
[14] C. Shababi, A. M. Zarkesh, J. Abidi, and V. Shah, "Knowledge discovery from user's web page navigation," in *Proceedings of the Seventh IEEE Intl. Workshop on Research Issues in Data Engineering (RIDE)*, 1997, 20–29.
[15] M. Spiliopoulou and L. C. Faulstich, "Wum: A tool for web utilization analysis," in *Proceedings of EDBT Workshop WebDB'98*, LNCS1590, Springer Verlag, 1999, 184–203.
[16] M. Spiliopoulou, C. Pohle, and L. C. Faulstich, "Improving the effectiveness of a web site with web usage mining," in *KDD'99 Workshop on Web Usage Analysis and User Profiling WEBKDD'99*, Aug. 1999.
[17] L. Tauscher and S. Greenberg, "Revisitation patterns in world wide web navigation," in *Proceedings of Int. Conf. CHI'97*, 1997.
[18] W3C. World wide web committee web usage characterization activity, *W3C Working Draft: Web Characterization Terminology and Definitions Sheet*, pages www.w3.org/1999/05/WCA-terms/, 1999.
[19] Y.-H. Wu and A. L. P. Chen, "Prediction of Web page accesses by proxy server log," *World Wide Web* 5(1), 2002, 67–88.
[20] Q. Yang and H. H. Zhang, "Integrating Web prefetching and caching using prediction models," *World Wide Web* 4(4), 2001, 299–321.
[21] O. Zaïane, M. Xin, and J. Han, "Discovering web access patterns and trends by applying olap and data mining technology on web logs," in *Advances in Digital Libraries*, April, 1998, 19–29.