World Wide Web: Internet and Web Information Systems, 5, 67–88, 2002
© 2002 Kluwer Academic Publishers. Manufactured in The Netherlands.

Prediction of Web Page Accesses by Proxy Server Log

YI-HUNG WU and ARBEE L. P. CHEN yihwu@mx.nthu.edu.tw, alpchen@cs.nthu.edu.tw
Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan 300, R.O.C.

Abstract

As the population of web users grows, the variety of user behaviors on accessing information also grows, which has a great impact on the network utilization. Recently, many efforts have been made to analyze user behaviors on the WWW. In this paper, we represent user behaviors by sequences of consecutive web page accesses, derived from the access log of a proxy server. Moreover, the frequent sequences are discovered and organized as an index. Based on the index, we propose a scheme for predicting user requests and a proxy-based framework for prefetching web pages. We perform experiments on real data. The results show that our approach makes the predictions with a high degree of accuracy with little overhead. In the experiments, the best hit ratio of the prediction achieves 75.69%, while the longest time to make a prediction only requires 2.3 ms.

Keywords: WWW, data mining, user behavior, prediction, suffix tree, proxy server log

1. Introduction

Due to the great popularity of the WWW, huge amounts of web pages have spread over various web sites and the population of web users grows rapidly. The proliferation of web users gives rise to a variety of behaviors on accessing information. User behavior has a great impact on network utilization. Owing to a fine perspective of business and industry, numerous analyses of user behavior on the WWW have been made for discovering marketing intelligence [7]. Within the WWW environment, the information about user behavior is often kept in three ways, i.e., the *history record* of a browser, the *access log* of a proxy server, and the *request log* of a web server. In this paper, our approach is based on the access log.

In our study, there are three major topics in this field. The first topic is the clustering of user behavior. Several approaches are proposed to measure the similarity between two browses and divide them into clusters. Yan et al. [26] apply the analytical results to user clustering and hyperlinks generation. In this approach, sequences of user requests are collected by the *session identification* technique, which distinguishes the requests for the same web page in different browses. A sequence is represented as a numeric vector that keeps the numbers of requests for the web pages involved. By the similarity measure, all the sequences are divided into clusters and so are the web pages. After that, a user is assigned to a cluster based on the web pages they request recently. Finally, the web pages that are frequently requested in the cluster will be recommended to the user as the extra hyperlinks.

In view of the browser's functions, e.g., the local cache and bookmarks, the data logged by the server do not necessarily imply a correspondence to the user behavior. To remedy this situation, Shahabi et al. [20] design the *profiler* to collect the user behavior. The profiler attaches a Java applet program, which keeps track of the user actions on the browser, to each web page. The details of user behavior, such as the duration of user browsing, are completely logged. In addition to the profiler design, they also consider clustering the sequences of user requests. Compared with Yan's approach, Shahabi et al. further take into account the order of user requests for the similarity measure. Furthermore, Zarkesh et al. [30] propose an approach to evaluate the quality of a web site. They assume that a good web site leads to good clusters. Therefore, their approach estimates the goodness of a web site by comparing its clusters with a predefined set of clusters.

The *WebWatcher* designed by Joachims et al. [13] provides another solution. Basically, the WebWatcher acts as a *tour guide*, which advises the user regarding which hyperlinks to follow and learns by observing the user actions. The user-specified interests and the texts around the hyperlinks clicked by the user are gathered. As a result, the WebWatcher highlights the hyperlinks in the web pages to be read, which are considered relevant to the user. However, the frequent modifications of web pages for the individual users will bring the system heavy loads. Perkowitz and Etzioni [19] introduce a scheme of composing web pages for the individual users dynamically. The goal is to create a web site that automatically adapts its organization and presentation to user behavior. However, due to the complexity of the automation for building a web site, only the dynamic generation of index pages for a web site is studied in their work.

The second topic is the discovery of significant behavior. For transaction databases, Agrawal and Srikant [2] present an algorithm for discovering frequent buying sequences for most customers. They define the field named *sequential pattern discovery* and call the derived sequences the *sequential patterns*. The problem of discovering significant behavior from log data is similar to the one of mining sequential patterns from transaction data. Therefore, some apply the techniques for sequential pattern discovery to this topic. For example, Spiliopoulou [21] proposes a process for mining navigational patterns to evaluate the usage of web sites. In addition, Mobasher et al. [17] apply data mining techniques to usage pattern discovery in order to create decision rules for customizing a web site.

In our previous work [28], an efficient algorithm for mining frequent buying sequences from transaction databases is presented. In [25], we adopt this algorithm to derive the *hot sequences*, which stand for the general behavior in browsing. Chen et al. [8] apply the mining algorithms for transaction data [18] to find popular sequences from log data, which are called the *traversal patterns*. In this work, they assume that a web server always logs the user requests with contextual information.

Zaïane et al. [29] also perform data mining on log data and develop a knowledge discovery tool *WebLogMiner*. They focus on the discovery of time-series patterns by using the OLAP techniques and a multidimensional cube. Their experiences show that the data preparation step is crucial and time-consuming. In addition, Masseglia et al. [16] design the *WebTool* for applying the data mining techniques to the log data analysis. In this work, they illustrate how to customize the hyperlink structure of a web site by the derived patterns. Berkhin et al. [5] implement the interactive tools to explore the navigation activities of a web site. They focus on the path analysis and provide user-configurable facilities for path extraction and filtering. Srivastava et al. [22] provide a detailed taxonomy of the related works in this area.

Third, some seek a deep understanding of user behavior and build the behavioral models. Almeida et al. [3] derive the regularity of user behavior based on the temporal and spatial locality. On the other hand, Crovella and Bestavros [10] employ the notion of *self-similarity* to characterize the user behavior.

Cunha and Jaccoud [11] further consider the prefetching of web pages. They develop two user models to help predicting the user requests. Kraiss and Weikum [15] introduce a probabilistic model, based on the *Markov chain theory*, to simulate the arrivals of user requests. They design a storage hierarchy for caching data and build a general framework for prefetching. In addition, Griffioen and Appleton [12] propose the concept of *probability graph* to predict future file system needs based on the past file activities. In this paper, we also consider the prefetching of web pages but focus on the behavioral discovery and the data indexing.

On the WWW, the client–server framework often incurs a great risk that the server is a bottleneck. To preserve availability and efficiency, caching techniques are commonly used. Caching serves to reduce the latency time by bringing web pages as close to users as possible. Vakali [23] presents an overview of cache replacement algorithms for a proxy server based on the idea of preserving a history record for cached web pages. On the other hand, Belloum and Hertzberger [4] propose a collaborative caching system that takes the dynamics of Internet traffic into consideration.

Prefetching serves to further reduce the latency time by caching web pages as soon as possible. For a prefetching service, the web pages to be prefetched are identified by the prediction of user requests. Therefore, the accuracy of prediction has a direct impact on the quality of service. Both the prefetching and caching strategies rely on the locality of reference, which assumes that recently accessed data have larger probabilities to be accessed in the near future. Moreover, the popularity of web pages is also useful for prediction. Yang and Zhang [27] present an integrated architecture, in which a certain amount of caching space is reserved for prefetching.

Klemm [14] proposes a client-side Java-based prefetching agent *WebCompanion*, which is based on the *estimated round-trip time* of each web page. Given a web page, the WebCompanion extracts the embedded hyperlinks and estimates their round-trip times based on the server statistics. By the strategies of selective prefetching and session control, it claims to achieve on average 50 percent speedup of the latency time. Bestavros [6] also implements a system for speculating user requests and disseminating data. The experimental results show that both the server load and the latency time are reduced. The goal of web page prefetching is to reduce the latency time. However, the erroneous retrieval of web pages will bring extra costs to the network traffic. As pronounced by Crovella and Barford [9], prefetching of web pages is counter-productive as a result of a straightforward approach.

Seeing that the accuracy of the prediction has a great impact on the performance of the prefetching service, we concentrate on how to make the prediction accurate. In our approach, the user behavior, represented by a sequence of consecutive web page accesses, is derived from the access log of a proxy server. All the sequences are stored in disk as a tree structure to provide the ability of incremental update. Based on the data mining concepts, the sequences that appear frequently are identified. An index structure is employed to cache the frequent sequences in memory and to make predictions. The proposed structure guarantees that at most two block accesses are required for one prediction. Finally, the corresponding web pages are prefetched by a ranking method.

To alleviate the workload of a single proxy server for prediction, in this paper, the derived sequences are divided into groups based on the web sites they belong to and the users who issue the requests. In this way, the workload can be easily distributed among a number of proxy severs. Moreover, different user communities can establish a variety of prefetching services on their own proxy servers. As a result, a distributed cooperative environment that coordinates the prefetching services for different user communities will be built. In this paper, we only focus on the case of a single proxy server to develop the techniques for predicting user requests.

The rest of this paper is organized as follows. Section 2 introduces our framework for prefetching web pages. Our approach is described in Section 3. Section 4 illustrates the experimental results. In Section 5, we conclude with our contributions and future works.

2. Overview

On the WWW, it is time consuming to issue a number of requests frequently for a prefetching service. Moreover, it is costly to cache the prefetched web pages. An effective mechanism equipped with both facilities is required. In our approach, the proxy server is chosen as the foundation of the prefetching service. The log data is extracted and reorganized into sequences of web page accesses (called the *paths*). After that, the paths are put into the mining process to discover frequent sequences (called the *patterns*). In this way, the user request will be predicted when the sequence of the recent requests matches up to one of the patterns. Finally, the proxy server prefetches the related web pages and stores them in the proxy cache. The proxy-based prefetching framework is shown in Figure 1.

For efficiency and flexibility, we organize the paths as a tree (called the *path tree*). The proposed structure is adapted from Wang's approach in [24] by incorporating a tree structure into each node. The tree structure (called the $PVB + tree$) provides the ability of incremental update. For the mining process, we apply the data mining concepts to pattern discovery. Compared with the problem of mining association rules [1], a path is like a transaction and its subpaths are the candidates to be frequent itemsets. In Section 3.3, we will introduce the measures of *support* and *confidence* for pattern discovery. To reduce the overhead of prediction, we organize the patterns as an index (called the *pattern tree*).

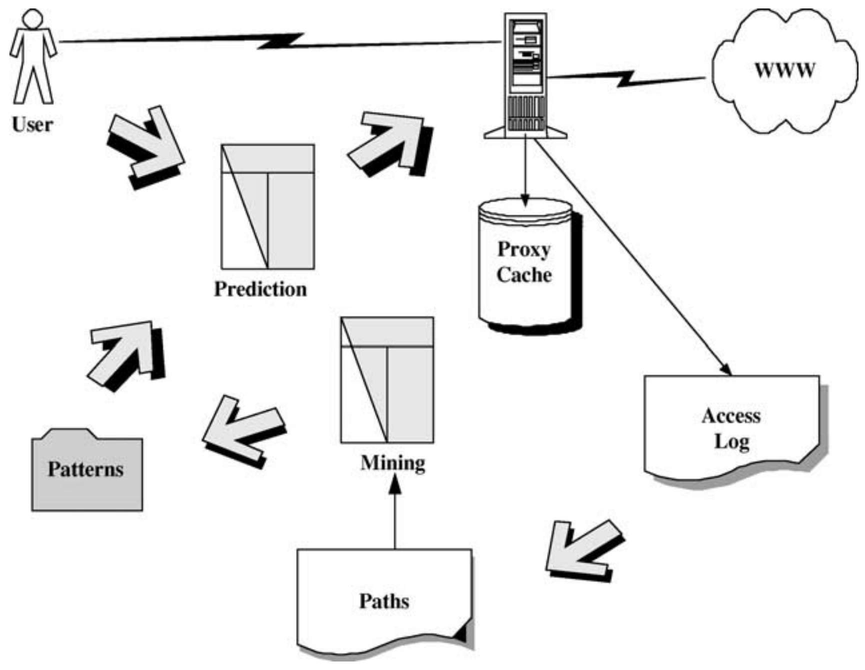

Figure 1. The proxy-based prefetching framework.

We divide our approach into four stages: *collection*, *storage*, *mining*, and *prediction*. The flowchart of our approach is shown in Figure 2, where rectangles stand for process units and ellipses refer to storage units. The area surrounded by a dotted rectangle refers to the parts that require online processing. The collection stage (i.e., *path collector*) extracts the log data and derives a series of paths. The storage stage (i.e., *storage manager*) builds and maintains a set of path trees. Periodically, the mining stage (i.e., *index generator*) discovers the patterns and organized them as a set of pattern trees. The prediction stage (i.e., *user monitor*) keeps track of the user requests and makes the predictions accordingly. At last, the prediction results are kept in the *user log* for each user and the *request dispatcher* integrates all of them into a sorted list for prefetching.

3. Our approach

3.1. Collection

Whenever the proxy server responds to the user with a web page, it adds a new record to the access log. The access log keeps all the records in temporal order. Moreover, the requests from different users (and different browses of the same users) are mixed together. Therefore, an effective way to collect meaningful information from the access log is required.

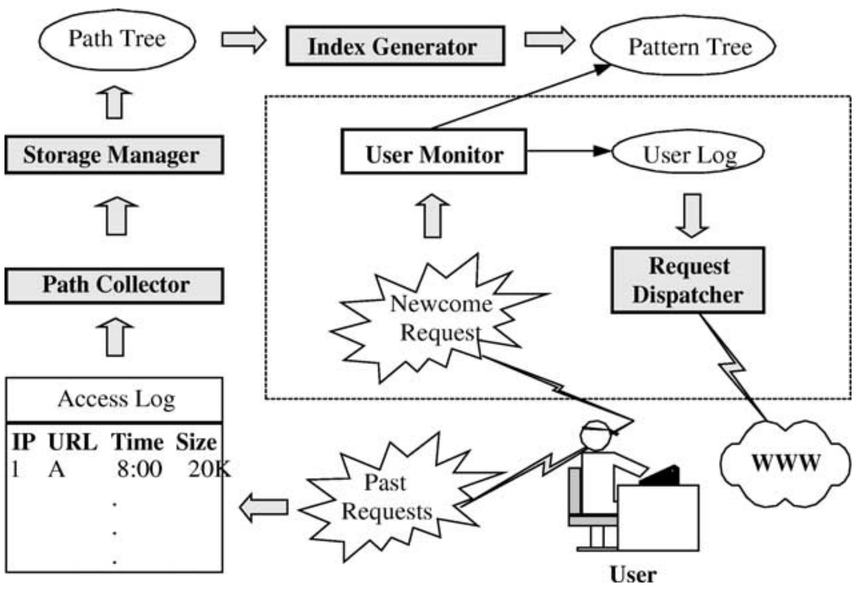

Figure 2. The flowchart of our approach.

Due to the local cache and utilities provided by the browser, it is impractical for the proxy server to keep all the actions of user browsing. For example, if a user uses the backward and forward commands for browsing (when the local cache is not full), the requests will not be sent to the proxy server. In this case, the access log will miss this part of user behavior. From the viewpoint of a proxy server, the access log captures only part of the user behavior. We call it the *access behavior*. We build a web service on the proxy server by considering the access behavior.

We adopt sequences of requests from individual users to represent the access behavior. Given a user (or an IP address), a series of the corresponding records (i.e., requests) constitutes the access behavior of the user. The access log usually keeps a complete description for each request. The following is an example:

```
890984441.324   0   140.117.11.12 UDP_MISS/000   76   ICP_QUERY
http://www.yam.org.tw/b5/yam/...
```

For simplicity, we consider only part of the description, i.e., the arrival time of a request (e.g., 890984441.324), the IP address of the user's computer (e.g., 140.117.11.12), and the URL (e.g., `http://www.yam.org.tw/b5/yam/`). Every record collected from the access log is reduced to a *triplet*. Moreover, all the triplets with the same IP address are concatenated as a long sequence in temporal order. Such a sequence of triplets completely specifies the access behavior of a user (IP address).

A sequence of triplets often spans a long time and implies the access behavior that results from more than one browse. Note that a browse means a series of navigation for a specific goal. Therefore, we have to find the places to cut the sequence into *segments*,

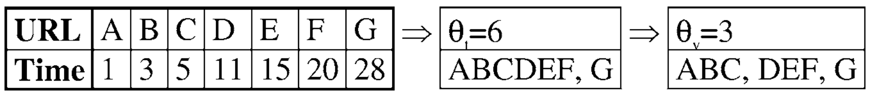

Figure 3. The time-based heuristic.

which correspond to the browses. In this paper, we adopt a time-based heuristic to identify the browses (i.e., segments), which is based on the following two assumptions:

1. When the interval between two consecutive triplets is very long, a new browse begins after this interval.
2. When the interval between two consecutive triplets is very different from the average of the previous ones, a new browse begins after this interval.

By the arrival times, we compute the interval between every two consecutive triplets and cut the sequence when the interval is longer than an upper bound θ_t. Further, by the second assumption, we cut the sequence when the difference between the interval and the average of the previous ones is larger than a variation limit θ_v. For simplicity, we set θ_t and θ_v to be constant.

Example. Consider the sequence ABCDEFG in Figure 3 as an example. The arrival times are set to be the values in the second row. We compute the six intervals (i.e., from A$\rightarrow$B to F$\rightarrow$G) as 2, 2, 6, 4, 5, and 8. Let θ_t and θ_v be set to 6 and 3, respectively. The last link F$\rightarrow$G is cut because its interval is longer than θ_t. Further, the third link C$\rightarrow$D is also cut because the difference between its interval and the average of ABC (i.e., $6 - 2$) is larger than θ_v. Finally, the sequence is cut into three segments ABC, DEF, and G.

For saving the storage space, we cut each segment into pieces, named the *access path*, such that the URLs on an access path belong to the same web site. An access path is represented as five fields of data. They are *hostname* (the web site), *path length* (the number of requests), *path time* (represented by the arrival time of the last request), *URL list* (the list of URLs), and *out-link flag* (a flag indicating whether there is a link to another access path). For the prefetching service, we only collect the access paths with at least two requests.

3.2. Storage

In the collection stage, the access paths are accumulated day by day. The storage space will be exhausted fast if we store them without organization. Therefore, an efficient way to store the access paths is required. We organize the access paths as a tree structure to enable data sharing and incremental update.

A URL is composed of two parts: *hostname* (the address of the web site) and *pathname* (the file location). We build a B+ tree (called the *host tree*) to keep the hostnames. For each hostname, we store the corresponding access paths together as a tree, named the *path tree*. As a result, the access paths are distributed over a set of path trees.

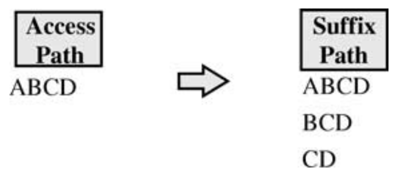

Figure 4. The enumeration of the suffix paths.

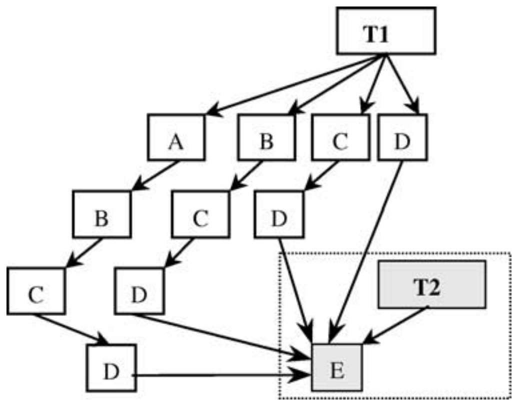

Figure 5. The path tree.

Due to the variety of user browsing, it is difficult to set up proper time limits to guarantee that every access path corresponds to a single browse. In other words, it is possible for each request on the access path to be the starting point of a browse. To handle such uncertainty, we replace every access path with all its suffixes (called the *suffix paths*). A suffix path is part of an access path by removing some prior requests.

Figure 4 shows the enumeration of suffix paths for the access path ABCD. Note that the suffix paths with only one request (e.g., D) are ignored during the enumeration. That is, every request except the last one on the access path is taken as the starting point of a suffix path. Although the use of suffix paths brings extra cost, it guarantees that the entire access behavior is captured.

A path tree consists of *nodes* and *pointers*. A root keeps the identifier of the path tree, while the other nodes refer to the individual requests on the suffix paths. Every suffix path corresponds to a path from the root to one of the other nodes. A pointer indicates the order of two requests on the suffix paths. In addition, the links between any two access paths are kept at the leaf nodes by some extra pointers (called the *out-links*). For brevity, we will use *path* to represent suffix path or path on the path tree in the rest of this paper (unless explicitly specified otherwise).

Consider Figure 5, there are two path trees T1 and T2. Assume the pathnames A, B, C, and D belong to the same web site and E belongs to another web site. Four pointers are used to keep the out-links (i.e., the pointers from D to E). On a path tree, each pointer is associated with four types of information. They are *position* (the disk location of the next node along the same path), *URL* (the pathname of the next node along the same path), *timestamp* (the time when the last modification of the next node along the same path occurs), and *count* (the number of paths that use this pointer). In this way, the paths with the same prefix will share their common nodes and pointers.

Owing to the variety of paths, a node on the path tree can be associated with a large number of pointers. Therefore, for each node, we use a *PV-B+ tree* to organize the set of pointers. A PV-B+ tree behaves like a B+ tree in the following way:

1. A PV-B+ tree consists of two types of nodes. The leaf nodes are physical nodes (abbreviated as p-nodes), while the other nodes are virtual nodes (abbreviated as v-nodes). A p-node keeps the four types of information (i.e., position, URL, timestamp, and count), while a v-node keeps URLs for traversing the tree.
2. For search and update, we use the URLs on v-nodes to traverse a PV-B+ tree and regard the contents on p-nodes as our targets.

Example. In the left-hand side of Figure 6, we show two snapshots of a path tree. The first one is a tree with only two paths ABC and ACD. After inserting the paths ACF, ADE, and AE, we obtain a larger tree. For each snapshot, we detail the corresponding PV-B+ trees at the right-hand side. For instance, the area surrounded by the dotted lines corresponds to the PV-B+ tree for node A. The one for the smaller tree has only one p-node with two pointers for B and C. We assume that E and F belong to the other path trees. In the following, we illustrate what happens to the PV-B+ tree when the three paths are inserted one by one.

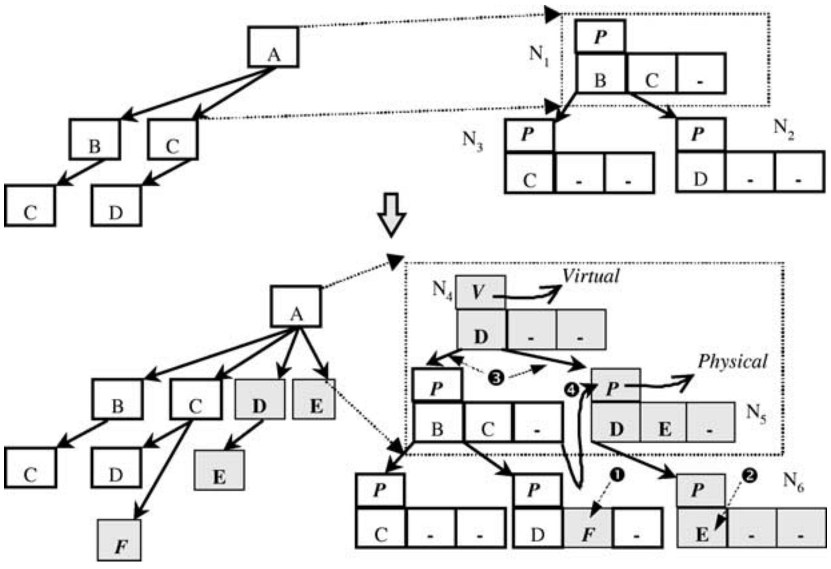

Figure 6. The construction of the path tree.

1. Insert ACF: Follow the pointer from N_1 to N_2 (i.e., path AC) and then insert F to N_2 with the out-link to the other path tree as ❶ depicts.
2. Insert ADE: Create a new PV-B+ tree with only one p-node (i.e., N_6) and then insert E to N_6 with the out-link to the other path tree as denoted by ❷. In addition, a new pointer from N_1 to N_6 (i.e., path AD) is also built.
3. Insert AE: N_1 splits because it is full at this moment. As a result, the PV-B+ tree for node A is composed of one v-node and two p-nodes. Four URLs are equally distributed into N_1 and the new p-node N_5. Furthermore, the v-node N_4 chooses D as the key for comparison and two pointers for N_1 and N_5 are created respectively as ❸ indicates.

For efficiency, all the p-nodes in the same PV-B+ tree are linked in sequence. Taking Figure 6 as an example, N_1 and N_5 are linked together by ❹. In this way, the p-nodes in the same PV-B+ tree can be accessed sequentially and efficiently.

3.3. Mining

In practice, not all paths are useful guides for prediction. In addition, the increasing number of paths makes the path trees large. Therefore, we further apply the data mining concepts to the discovery of important access behavior. In the previous stages, we find some features to specify the importance of a path. For a prefetching service, we simplify them as follows:

- *Hostname*. The paths with different hostnames are equally treated.
- *Path length*. The impact of the path length is ignored.
- *Path time*. Only the last time that a path appears is recorded. Therefore, we define a constraint called the *expired time* to select the paths that appear recently.

As a result, a path is called an *important path* if it contains at least two requests and does not violate the expired time constraint. Moreover, we employ the *path frequency* of a path μ (denoted by C_μ), indicating how many times μ appears, to quantify the importance of a path. The count kept by the ending node corresponds to the path frequency of a path. Because we have to equally treat the paths with different hostnames, we also consider the *total frequenc*y of a path tree T (denoted by C_{T}), indicating how many times the paths in T appear. As a result, we estimate the importance of μ, named the *support* of μ (denoted by Sup_μ), as follows:

$$\mathrm{Sup}_\mu = \frac{C_\mu}{C_{\mathrm{T}}}, \tag{1}$$

where T is a path tree, and $\mu \in \mathrm{T}$.

From the viewpoint of a small granularity, in a PV-B+ tree, the Count field on a p-node corresponds to one of the path frequencies. For the PV-B+ tree of the root, all the Count fields on p-nodes constitute the total frequency of the path tree. Based on the support measure, the degrees of importance for the paths are estimated. What we discover is the popular access behavior for the proxy server. At this moment, the popular web pages are also identified.

Furthermore, given a path, the dependence of a request on this path is also utilized for prediction. Let $\mu\nu$ denote a path. The probability that a request ν appears immediately after μ is equal to the conditional probability that $\mu\nu$ will appear if μ appears. Given the supports of two paths μ and $\mu\nu$, we estimate the dependence of ν on μ, which is named the *confidence* of $\mu\nu$ (denoted by $\mathrm{Con}_{\mu\nu}$), as follows:

$$\mathrm{Con}_{\mu\nu} = \frac{\mathrm{Sup}_{\mu\nu}}{\mathrm{Sup}_{\mu}} = \frac{C_{\mu\nu}}{C_{\mu}}, \tag{2}$$

where both μ and $\mu\nu$ are important paths.

It is easy to extend the definition to cover general cases, e.g., the dependence of one path on another. However, it will lead to a high overhead if we predict a sequence of requests for each user at one time. Given the expired time, we select a set of important paths from the path tree and compute their supports and confidences. Moreover, we assume that the paths with high supports and confidences are helpful in making good predictions. Therefore, we set up two thresholds *minimum support* (denoted by α) and *minimum confidence* (denoted by β) to filter out the paths with low supports or confidences. Finally, all the qualified paths, called the *access patterns*, will be used for making predictions.

On the WWW, a web page may contain not only text but also other media such as images, sounds, and videos. As far as the proxy server is concerned, the media embedded in a web page are considered as independent requests. In this case, a series of requests appears close and arrives in the same order. This phenomenon will bring extra high confidence measures. Similarly, the frames in a web page also result in the same situation. To speed up prefetching, we introduce a measure for a sequence of requests μ, called the *grouping confidence* of μ(denoted by G-Con_{μ}):

$$\text{G-Con}_{\mu} = \frac{C_{\mathrm{B}}}{C_{\mathrm{A}}}, \tag{3}$$

where μ is a sequence of requests from A to B.

Note that the sequence μ starts at any node on the path tree. If G-Con_{μ} is high enough, the requests in μ will form a *group* that will be prefetched at the same time. We also set up a threshold *minimum grouping confidence* (denoted by γ) to limit the number of groups.

Example. Take the path tree T in Figure 7 as an example. In T, each node is associated with an alphabet and a number, which denote the URL and the count, respectively. We assume that both the timestamps associated with ❸ and ❹ are expired. Therefore, only the paths across ❶ or ❷ are important paths. Moreover, C_{T} is the sum of the counts associated with ❶ and ❷ (i.e., 200). The following applies the three formulae to the paths across ❶ and derives the access patterns based on the thresholds α, β, and γ.

1. By formula (1), both $\mathrm{Sup}_{\mathrm{AB}}$ and $\mathrm{Sup}_{\mathrm{ABC}}$ are 15%, and $\mathrm{Sup}_{\mathrm{ABCD}}$ equals 7.5%.
2. By formula (2), $\mathrm{Con}_{\mathrm{AB}}$ is 20%, $\mathrm{Con}_{\mathrm{ABC}}$ is 100%, and $\mathrm{Con}_{\mathrm{ABCD}}$ is 50%. Based on the thresholds α and β, we conclude that the access patterns are AB and ABC.

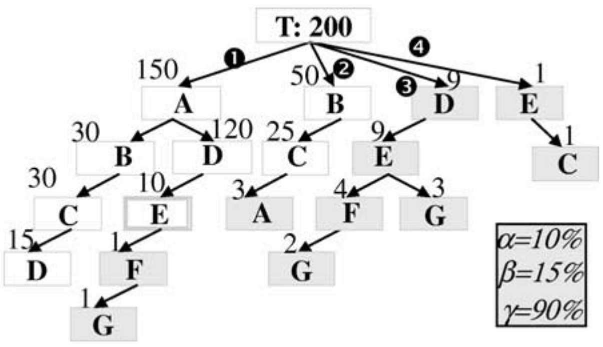

Figure 7. A path tree and the three thresholds.

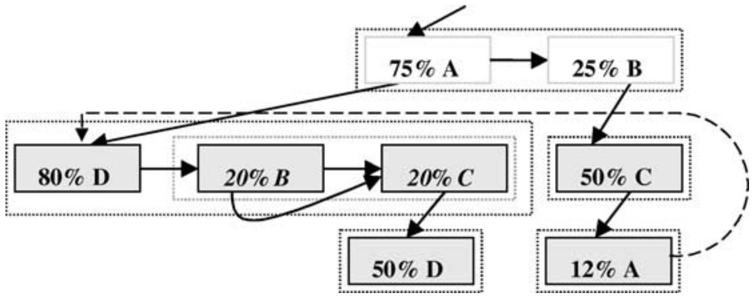

Figure 8. The pattern tree.

3. By formula (3), G-Con$_{\mathrm{AB}}$ is 20% and G-Con$_{\mathrm{BC}}$ equals 100%. According to the threshold γ, we find that B and C will form a group. Note that we do not have to compute G-Con$_{\mathrm{ABC}}$ because G-Con$_{\mathrm{AB}}$ is lower than the threshold γ.

On an access pattern, each request except the first one is called a *candidate* for the requests before it. That is, given an access pattern $\mu\nu$, the request ν is a candidate for μ. Given a sequence of requests, we select the access patterns that are prefixed by the sequence and extract the corresponding set of candidates from them. The access patterns derived from a path tree are organized into the *pattern tree*. Conceptually, every access pattern corresponds to a path on the pattern tree. For brevity, we use the *pattern* to mean a path on the pattern tree in the rest of this paper (unless explicitly specified otherwise).

A pattern tree is composed of *buckets* and *pointers*, where a bucket keeps the candidates for a specific pattern and the pointers maintain the order on patterns. For clarity, we call the pointer pointing to a bucket the *b-pointer*. The number of candidates in a bucket is varied in reality. Each candidate is represented as three fields, including its *URL*, the *confidence* of the corresponding pattern, and a *b-pointer* pointing to the next bucket.

Example. Consider the pattern tree in Figure 8 as an example. The solid rectangles refer to candidates and the dotted rectangles stand for buckets. In this pattern tree, there are five buckets in total. The root bucket is only used as the starting point for prediction and a reference for other buckets. Recall that B and C on the pattern ABC will form a group. Therefore, the bucket pointed to by A contains three candidates (i.e., D, B, C). Moreover, the confidence of C (20%) is computed by multiplying the original confidence (i.e., $\text{Con}_{\text{ABC}} = 100\%$) with the confidences of all the candidates in the same group (i.e., $\text{Con}_{\text{AB}} = 20\%$). The following illustrates how to utilize the pattern tree for prediction.

1. When a user issues a request for A, through the b-pointer of A, we will reach the bucket that contains three candidates (i.e., D, B, C). All of them will be the web pages to be prefetched and their confidences will contribute to the ranking in the next stage. After that, the current bucket will be stored into the user log.
2. If the user issues another request for C immediately after the prediction, we continue the prediction from the user log and follow the b-pointer of C to reach the bucket that contains only one candidate D (with confidence 50%). The case that the new request is included in the user log is called a *success*.
3. If the new request does not exist in the user log (e.g., A), we restart the prediction from the root bucket. This case is called a *failure*. In this way, our approach guarantees that at most two block accesses are required to make a prediction.
4. If the user has not issued any request for a long time, we discard the user log. This case is called *timeout*.

3.4. Prediction

The final stage uses the index built in the mining stage to predict the user requests. The requests that appear in the access log are periodically collected to start the prediction. In other words, all the requests that arrive within a time period are collected and served at the same time. Moreover, the time-based heuristic introduced in Section 3.1 is employed to determine whether a request belongs to the previous browse or starts a new browse.

The prediction for a user often generates more than one candidate. Furthermore, a web page can be the candidate for more than one user. Therefore, how to integrate the candidates to provide an effective schedule for prefetching is another issue we have to handle. In this paper, we assume that only one proxy server is utilized for prefetching.

Intuitively, the confidence of a candidate is a good clue for ranking. Consider the case of a single user. The candidate with a high confidence is more likely to generate a success than the one with a low confidence. Therefore, we apply the confidence measure to ranking of the candidates for each user. On the other hand, for a candidate, the number of the users involved is also a good clue for ranking. In general, the candidate that involves more users is likely to benefit more users after it is prefetched. Therefore, we combine the influences from both the candidates themselves and the users involved in our ranking methods. The following formula is proposed to estimate the *static rank* of a candidate i (denoted by SR_i), where U denotes the set of all the users served and j is a user in U. Note that Conf_{ij} is set to zero if user j does not own candidate i:

$$SR_i = \frac{1}{|U|}\sum_{j\in U}\mathrm{Conf}_{ij}, \tag{4}$$

where Conf_{ij} is the confidence of candidate i for user j.

In the static ranking method, all the users involved have the same right to be served by the prefetching service. That is, all the users have the same influences on the ranking. However, in reality, not every user deserves to be served, such as naïve users and the users who often behave irregularly. Therefore, we propose another method to take into account how the users behave in the previous service.

At the start of a new browse, the user is given a constant right to request a prefetching service. We quantify this right and call it the *weight*. Moreover, the weight is dynamically adjusted by the following two alternatives:

1. When the new request leads to a success, the weight increases.
2. When the new request leads to a failure, the weight decreases.

The user will not be served for a while after the weight is down to zero. Furthermore, the volume, each time the weight increases or decreases, is called the *quota* (denoted by δ). Intuitively, the larger δ is, the smaller the number of users to be served will be. In other words, δ adapts to the change of server load. For simplicity, δ is set to a constant value in this paper. In the following, we reformulate the computation of the rank, which is called the *dynamic rank* of a candidate i (denoted by DR_i), where $\underline{U}$ denotes the set of users who have nonzero weights.

$$DR_i = \frac{1}{|\underline{U}|}\sum_{j\in\underline{U}}(\mathrm{Conf}_{ij} \times \mathrm{Weight}_j), \tag{5}$$

where Weight_j is the weight of user j.

Compared with formula (4), formula (5) takes the weights of the individual users into consideration and the users whose weights are down to zero are not served. Therefore, the static ranking method is viewed as a special case of the dynamic ranking method, where the weights are equal and constant.

Example. Take Figure 9 as an example. Two predictions are made at 10 : 00 and 14 : 00. We obtain the confidences of candidates from Figure 8. The following illustrates what happens as we adopt the above two methods to rank the candidates, respectively.

1. At 10:00 (denoted by ❶), IP1 has three candidates D, B, and C, while IP2 has only one candidate C. For dynamic ranking, the weights of IP1 and IP2 are set to 0.6 and 1.6, respectively.
   - By formula (4), $SR_{\mathrm{D}} = (80\% + 0)/2 = 40\%$, $SR_{\mathrm{B}} = (20\% + 0)/2 = 10\%$, and $SR_{\mathrm{C}} = (20\% + 50\%)/2 = 35\%$. Therefore, a schedule based on the static ranking method is D→C→B.
   - $DR_{\mathrm{D}} = (80\% \cdot 0.6 + 0)/2 = 24\%$, $DR_{\mathrm{B}} = (20\% \cdot 0.6 + 0)/2 = 6\%$, and $DR_{\mathrm{C}} = (20\% \cdot 0.6 + 50\% \cdot 1.6)/2 = 46\%$ by formula (5). Therefore, a schedule based on the dynamic ranking method is C→D→B.

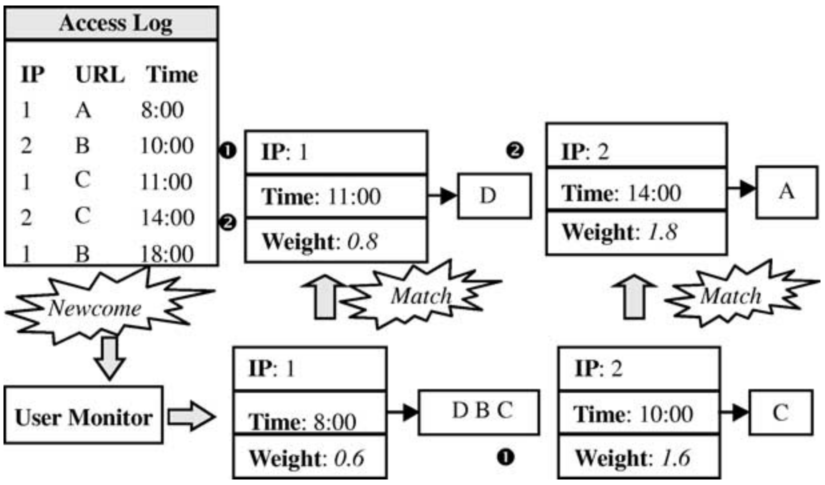

Figure 9. An example for making predictions.

2. At 14 : 00 (as denoted by ❷). IP1 has one candidate D, while IP2 also has one candidate A. For dynamic ranking, the weights of IP1 and IP2 become 0.8 and 1.8 (δ is set to 0.2), respectively.

- By formula (4), $SR_{\mathrm{D}} = (50\% + 0)/2 = 25\%$, and $SR_{\mathrm{A}} = (0 + 12\%)/2 = 6\%$. Therefore, a schedule based on the static ranking method is D→A.
- $DR_{\mathrm{D}} = (50\% \cdot 0.8 + 0)/2 = 20\%$, and $DR_{\mathrm{A}} = (0 + 12\% \cdot 1.8)/2 = 10.8\%$ by formula (5). Therefore, a schedule based on the dynamic ranking method is D→A.

4. Performance evaluation

To evaluate the effectiveness and the efficiency, we implement all the four stages of our approach. After that, we make a set of experiments on the log data provided by the National Center of High-Performance Computing in Taiwan. We build the index for prediction based on the *training set*. For evaluation, we use the *testing set* to simulate the process of prediction. At last, we compare the simulation results with the real scenarios to evaluate the effectiveness of our approach. On the other hand, we observe the overheads due to prediction.

We make the experiments on the training sets with various sizes, which refer to the time window (denoted by Σ) for selecting the records from the log data. For simplicity, in this paper the time window is set to one hour. Given a training set, we collect the testing set in two ways:

1. The requests that appear in the next hour after the training set (denoted by TEST = H).
2. The requests that appear in the same hour of the next day after the training set (denoted by TEST = D).

Table 1. Experimental parameters

Symbol	Definition	Value
TEST	The way to collect the testing set	H, D
Σ	The size of the training set	1 (hour)
α	The minimum support	10%, 20%, 30%, ..., 90%
β	The minimum confidence	10%, 20%, 30%, ..., 90%
γ	The grouping confidence	100%

Table 2. Performance metrics

Symbol	Definition	Computation
λ	The hit ratio	Number of successes $\div$ Number of predictions
τ	The service rate	Number of predictions $\div$ Number of requests
Γ	The contribution	Number of successes $\div$ Number of requests
ω_n	The number of index	The number of pattern trees
ω_t	The prediction time	The processing time to make a prediction

Table 3. Parameter settings for the experiment on α

Parameters			
Constants	β	γ	Σ
	50%	100%	1
Variables	α	10%, 20%, 30%, ..., 90%	
	TEST	D, H	
Metrics	τ, λ, Γ, ω_n, ω_t		

Table 1 lists all the parameters with their values used in the experiments. We perform the experiments to examine the parameter settings. In this paper, we focus on the set of metrics for performance evaluation as shown in Table 2. The experimental results are obtained by averaging the measurements from a series of tests. The thresholds α and β have great impacts on the number of access patterns, and the effectiveness and efficiency of our approach. Therefore, we focus on the influences of α and β on the performance of our approach.

4.1. Experiment on α

The settings of parameters and metrics are listed in Table 3. In this experiment, we consider one-hour log data as the training set (Σ). Furthermore, the minimum confidence (β) and the grouping confidence (γ) are set to 50% and 100%, respectively. All these parameters are controlled and fixed to minimize their impacts on the experimental results. As for the parameter to be analyzed – the minimum support (α), we give a series of tests on the values ranging from 10% to 90% and estimate the set of metrics, including λ, Γ, τ, ω_n, and ω_t. In addition, we also make the experiments on different ways to collect the testing set (TEST).

Based on the curves in Figures 10 and 11, we observe the influence of α on the hit ratio and the service rate, respectively. For both curves in Figure 10, the hit ratio reaches 60% or above when α exceeds 70%. Moreover, the testing set that comes from the next

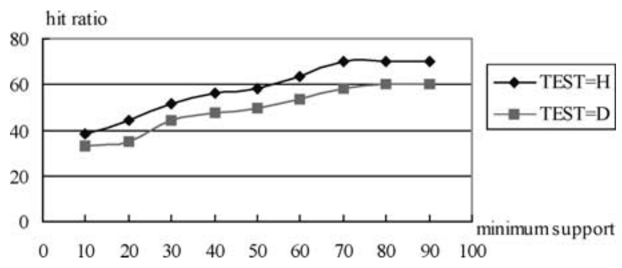

Figure 10. The hit ratio (λ%) vs. minimum support (α%).

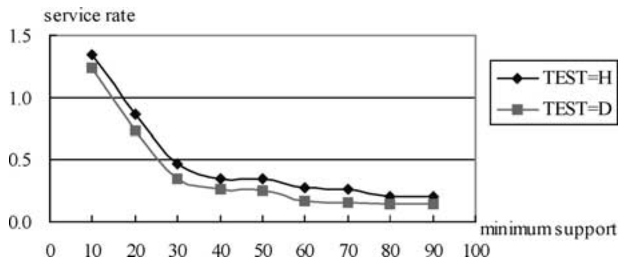

Figure 11. The service rate (τ%) vs. minimum support (α%).

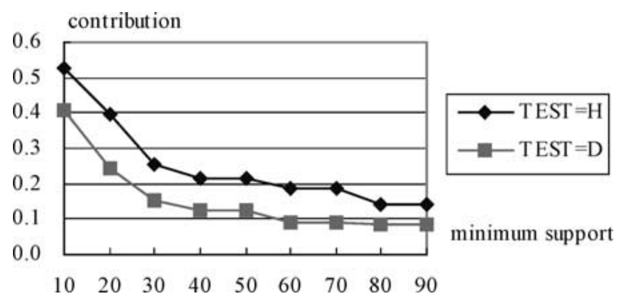

Figure 12. The contribution (Γ%) vs. minimum support (α%).

hour (TEST = H) always achieves a hit ratio better than the other one. The best hit ratio obtained in the series of tests is 74.29%. On the other hand, the service rate will be under 0.5% when α exceeds 30%. In other words, a majority of requests cannot be predicted because they do not appear in the pattern trees. Similarly, the testing set that comes from the next hour (TEST = H) achieves a service rate better than the other one. The best service rate obtained in the series of tests is only 1.78%. To illustrate the contribution of prediction to the proxy server, we combine the effects of hit ratio and service rate into the two curves in Figure 12. In general, both curves drop quickly in the beginnings and then slow down when α exceeds 30%.

The larger α is, the fewer paths that appear in the pattern trees will be. Therewith, the number of pattern trees also decreases. As shown in Figure 13, the declined curve supports

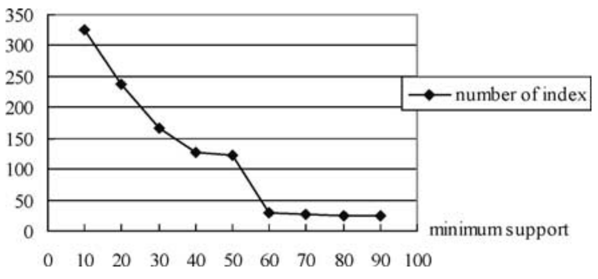

Figure 13. The number of index (ω_n) vs. minimum support (α%).

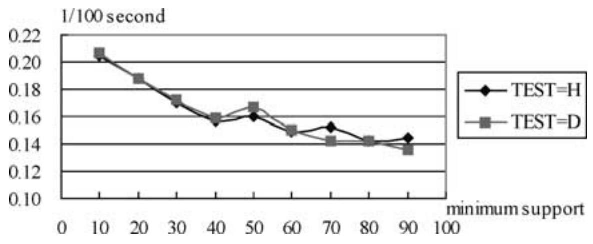

Figure 14. The prediction time (ω_t) vs. minimum support (α%).

Table 4. Parameter settings for the experiment on β

Parameters			
Constants	α	γ	Σ
	50%	100%	1
Variables	β	10%, 20%, 30%, . . . , 90%	
	TEST	D, H	
Metrics	τ, λ, Γ, ω_n, ω_t		

such an argument. As α increases, the time to rebuild the index is reduced because of fewer patterns. This experiment confirms that the longest time to rebuild the index (about 510 s) is measured as α is set to 10%. That means it takes no more than nine minutes to rebuild the index for one-hour log data. Because both the path trees and the pattern trees are built offline, what we really concern is the time to make a prediction. Similarly, the prediction time also decreases as α increases. In Figure 14, the time measure on the y-axis indicates the average time to make a prediction for a single request. From the series of tests, the longest time to make a prediction is 2.3 ms.

4.2. *Experiment on β*

The settings of parameters and metrics are listed in Table 4. As the previous experiment, only one-hour log data is taken as the training set (Σ). Furthermore, the minimum support (α) and the grouping confidence (γ) are set to 50% and 100%, respectively. All these parameters are controlled and fixed to minimize their impacts on the experimental results.

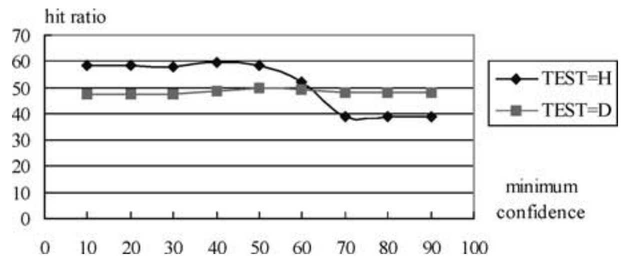

Figure 15. The hit ratio (λ%) vs. minimum confidence (β%).

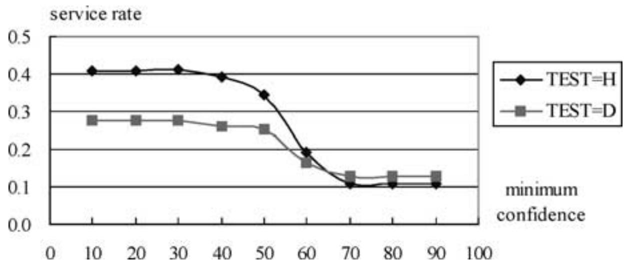

Figure 16. The service rate (τ%) vs. minimum confidence (β%).

As for the parameter to be analyzed – the minimum confidence (β), we give a series of tests on the values ranging from 10% to 90% and estimate the set of metrics, including λ, τ, Γ, ω_n, and ω_t. In addition, we also make the experiments on different ways to collect the testing set (TEST).

From Figure 15, we observe the influence of β on the hit ratio. Notice that the two kinds of testing sets lead to different curves. For the testing set that comes from the next hour (TEST = H), the hit ratio slowly decreases as β increases. On the other hand, if we adopt the testing set that comes from the same hour of the next day (TEST = D), the hit ratio almost keeps constant as β increases. The best hit ratio obtained in the series of tests is 75.69%. On the other hand, for both kinds of testing sets, the service rate decreases as β increases. As Figure 16 shows, the decrease is sharper if the testing set comes from the next hour (TEST = H). When β exceeds 60%, the service rate will be under 0.2%. The best service rate obtained in the series of tests is only 0.77%. Similarly, we combine the effects of hit ratio and service rate into the two curves in Figure 17. It verifies that the two kinds of test sets achieve almost the same contribution when β exceeds 60%.

As described previously, the increase of β has a great impact on the number of pattern trees. As shown in Figure 18, the number of pattern trees decreases sharply as β exceeds 50%. Intuitively, the prediction time should decrease as β increases because of fewer candidates. However, Figure 19 indicates that β has no obvious impact on the prediction time. From the series of tests, the longest time to make a prediction is 1.9 ms.

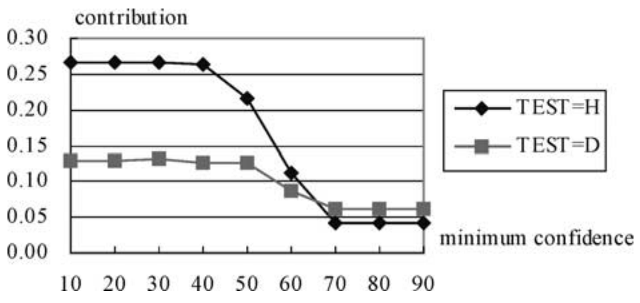

Figure 17. The contribution (Γ%) vs. minimum support (β%).

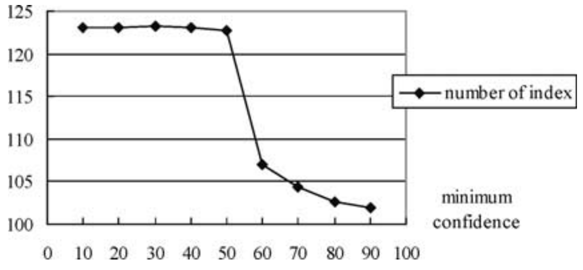

Figure 18. The number of index (ω_n) vs. minimum confidence (β%).

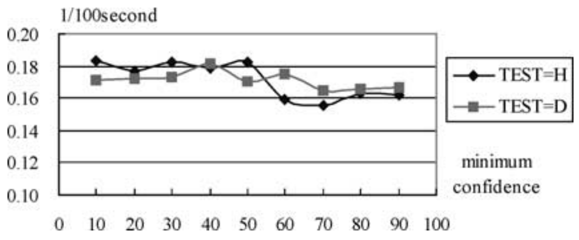

Figure 19. The prediction time (ω_t) vs. minimum confidence (β%).

5. Conclusion

In this paper, we propose a new approach for predicting user requests based on the recent behavior of the individual users. To provide a systematic analysis of user behavior, we present a procedure that consists of four stages, i.e., collection, storage, mining, and prediction. Our approach derives the patterns based on the access log of a proxy server. Moreover, the patterns are organized as a compact index. By the index, we provide an effective way to make the prediction.

In the experiments, the best hit ratio of the prediction achieves 75.69%, while the longest time to make a prediction only requires 1.9 ms. However, our experiments show that the average service rate is very low. In the future, we will continue our work to promote the service rate. For example, we will consider partial matching of the pathnames instead of exact matching of the entire URLs.

The other problem is the setting of the three thresholds used in the mining stage. These thresholds have great impacts on the construction of the pattern trees. The use of minimum support and minimum confidence is to prune the useless paths. Obviously, some information may be lost if the pruning effects are overestimated. On the other hand, the grouping confidence is only useful for the strongly related web pages due to some editorial techniques, such as the embedded images and the frames.

The goal of this research is to provide a prefetching service with good performance. The related issues also include the dynamic control of prefetching activities and the enhancement of cache management. Currently, we have been working on the cache enhancement based on the prediction results.

References

[1] R. Agrawal and R. Srikant, “Fast algorithms for mining association rules,” in *Proceedings of VLDB Conference*, 1994, pp. 487–499.

[2] R. Agrawal and R. Srikant, “Mining sequential patterns,” in *Proceedings of IEEE Conference on Data Engineering*, 1995, pp. 3–14.

[3] V. Almeida, A. Bestavros, M. Crovella, and A. Oliveira, “Characterizing reference locality in the WWW,” in *Proceedings of IEEE Conference on Parallel and Distributed Information Systems*, 1996, pp. 92–103.

[4] A. Belloum and L. O. Hertzberger, “Scalable federation of Web cache servers,” *World Wide Web* 4, 2001, 255–275.

[5] P. Berkhin, J. D. Becher, and D. J. Randall, “Interactive path analysis of Web site traffic,” in *Proceedings of ACM SIGKDD Conference*, 2001, pp. 414–419.

[6] A. Bestavros, “Speculative data dissemination and service to reduce server load, network traffic and service time for distributed information systems,” in *Proceedings of IEEE Conference on Data Engineering*, 1996, pp. 180–187.

[7] A. Büchner and M. D. Mulvenna, “Discovering Internet marketing intelligence through online analytical Web usage mining,” in *ACM SIGMOD Record* 27(4), December 1998, 54–61.

[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), March/April 1998, 209–220.

[9] M. Crovella and P. Barford, “The network effects of prefetching,” in *Proceedings of IEEE INFOCOM Conference*, 1998.

[10] M. Crovella and A. Bestavros, “Self-similarity in World Wide Web traffic: Evidence and possible causes,” in *Proceedings of ACM SIGMETRICS Conference*, May 1996.

[11] C. R. Cunha and C. F. B. Jaccoud, “Determining WWW user’s next access and its applications to prefetching,” in *Proceedings of IEEE International Symposium on Computers and Communications*, July 1997, pp. 1–3.

[12] J. Griffioen and R. Appleton, “Automatic prefetching in a WAN,” in *Proceedings of IEEE Workshop on Advances in Parallel and Distributed Systems*, 1993.

[13] T. Joachims, D. Freitag, and T. Mitchell, “WebWatcher: A tour guide for the World Wide Web,” in *Proceedings of International Joint Conference on Artificial Intelligence*, August 1997.

[14] R. P. Klemm, “WebCompanion: A friendly client-side Web prefetching agent,” *IEEE Transactions on Knowledge and Data Engineering* 11(4), July/August 1999, 577–594.

[15] A. Kraiss and G. Weikum, “Integrated document caching and prefetching in storage hierarchies based on Markov-chain predictions,” *VLDB Journal* 7, 1998, 141–162.

[16] F. Masseglia, P. Poncelet, and M. Teisseire, “Using data mining techniques on Web access logs to dynamically improve hypertext structure,” *ACM SIGWEB Newsletter* 8(3), October 1999, 13–19.

[17] B. Mobasher, R. Cooley, and J. Srivastava, “Automatic personalization based on Web usage mining,” *Communications of the ACM* 43(8), August 2000, 142–151.

[18] J. S. Park, M. S. Chen, and P. S. Yu, “An effective hash based algorithm for mining association rules,” in *Proceedings of ACM SIGMOD Conference*, 1995, pp. 175–186.

[19] M. Perkowitz and O. Etzioni, “Adaptive Web sites,” *Communications of the ACM* 43(8), August 2000, 152–158.

[20] C. Shahabi, A. M. Zarkesh, J. Adibi, and V. Shah, “Knowledge discovery from user Web-page navigation,” in *Proceedings of Workshop on Research Issues in Data Engineering*, 1997, pp. 20–29.

[21] M. Spiliopoulou, “Web usage mining for Web site evaluation,” *Communications of the ACM* 43(8), August 2000, 127–134.

[22] J. Srivastava, R. Cooley, M. Deshpande, and P. N. Tan, “Web usage mining: Discovery and applications of usage patterns from Web data,” *SIGKDD Explorations* 1(2), 2000, 12–23.

[23] A. Vakali, “Proxy cache replacement algorithms: A history-based approach,” *World Wide Web* 4, 2001, 277–297.

[24] K. Wang, ”Discovering patterns from large and dynamic sequential,” *Journal of Intelligent Information Systems* 9, 1997, 33–56.

[25] Y. H. Wu, Y. H. Chen, and A. L. P. Chen, “Querying and browsing the resources in Internet,” in *Proceedings of International Computer Symposium*, 1996, pp. 9–16.

[26] T. W. Yan, M. Jacobsen, H. Garcia-Molina, and U. Dayal, “From user access patterns to dynamic hypertext linking,” in *Proceedings of International WWW Conference*, May 1996.

[27] Q. Yang and H. H. Zhang, “Integrating Web prefetching and caching using prediction models,” *World Wide Web* 4, 2001, 299–321.

[28] S. J. Yen and A. L. P. Chen, “An efficient approach to discovering knowledge from large databases,” in *Proceedings of International Conference on Parallel and Distributed Information Systems*, 1995, pp. 8–18.

[29] O. R. Zaïane, M. Xin, and J. W. Han, “Discovering Web access patterns and trends by applying OLAP and data mining technology on Web logs,” in *Proceedings of IEEE Conference on Advances in Digital Libraries*, 1998, pp. 19–29.

[30] A. M. Zarkesh, J. Adibi, C. Shahabi et al., “Analysis and design of server informative WWW-sites,” in *Proceedings of ACM Conference on Information and Knowledge Management*, 1997, pp. 254–261.