An Agent Based Method for Web Page Prediction

Debajyoti Mukhopadhyay, Priyanka Mishra, and Dwaipayan Saha

Web Intelligence & Distributed Computing Research Lab,
Department of Computer Science & Engineering, Techno India,
West Bengal University of Technology,
EM 4/1, Sector V, Salt Lake, Calcutta 700091, India
{debajyoti.mukhopadhyay, priyanka147, dwaipayansaha}@gmail.com

Abstract. Studies have been conducted on pre-fetching models based on decision trees, Markov chains, and path analysis. However, the increased uses of dynamic pages, frequent changes in site structure and user access patterns have limited the efficacy of these static techniques. One of the techniques that are used for improving user latency is Caching and another is Web pre-fetching. Approaches that bank solely on caching offer limited performance improvement because it is difficult for caching to handle the large number of increasingly diverse files. An agent based method is proposed here to cluster related pages into different categories based on the access patterns. Additionally page ranking is used to build up the prediction model at the initial stages when users are yet to invoke any page.

1 Introduction

The exponential proliferation of Web usage has dramatically increased the volume of Internet traffic and has caused serious performance degradation in terms of user latency and bandwidth on the Internet. The use of the World Wide Web has become indispensable in everybody's life which has also made it critical to look for ways to accommodate increasing number of users while preventing excessive delays and congestion. Studies have been conducted on pre-fetching models based on decision trees, Markov chains, and path analysis [1] [2] [4]. There are several factors that contribute to the Web access latencies such as: server configuration, server load, client configuration, document to be transferred, network characteristics.

Web Caching is a technique that made efforts to solve the problem of these access latencies. Specially, global caching methods that straddle across users work quite well. However, the increasing trend of generating dynamic pages in response to HTTP requests from users has rendered them quite ineffective. The following can be seen as the major reasons for the increased use of dynamic Web pages:

(1) For user customized Web pages, the content of which depends on the users' interests. Such personalized pages allow the user to reach the information they want in much lesser time. (2) For pages that need frequent updating, it is irrational to make those changes on the static Web pages. Maintaining a database and generating the content of the Web pages from the database is a much cheaper alternative. Pages displaying sports updates, stock updates, weather information etc. which involve a lot

N.T. Nguyen et al. (Eds.): KES-AMSTA 2007, LNAI 4496, pp. 219–228, 2007.

of variables are generated dynamically. (3) Pages that need a user authentication before displaying their content are also generated dynamically, as separate pages are generated as per the user information for each user. This trend is increasing rapidly. (4) All response pages on a secure connection are generated dynamically as per the password and other security features such as encryption keys. These pages expire immediately by resetting the Expire field and/or by the Pragma directive of 'nocache' in the HTTP header of the server response, to prevent them from being misused in a Replay attack.

As the Internet grows and becomes a primary means of communication in business as well as the day-to-day life, the majority of Web pages will tend to be dynamic. In such a situation, traditional caching methods will be rendered obsolete. The dynamic pages need a substantial amount of processing on the server side, after receiving the request from the client and hence contribute to the increase in the access latency further. An important pre-fetching task is to build an effective prediction model and data-structure for predicting the future requests of the user and then sending those predicted requests to the user before he/she actually makes the request.

2 Proposed Method

2.1 Prediction Model

Links are made by Web designers based on relevance of content and certain interests of their own. In our method, we classify Web pages based on hyperlink relations and the site structure. We use this concept to build a category based dynamic prediction model. For example in a general portal www.abc.com all pages under the movies section fall under a single unique class. We assume that a user will preferably visit the next page, which belongs to the same class as that of the current page. To apply this concept we consider a set of dominant links that point to pages that define a particular category. All the pages followed by that particular link remain in the same class. The pages are categorized further into levels according to the page rank in the initial period and later, the users' access frequency [6] [7] [8].

The major problem in this field is that, the prediction models have been dependent on history data or logs [5]. They were unable to make predictions in the initial stages [3]. We present the structure of our agent based prediction model in Figure 1. Our method is not dependent on log data, rather it is built up using ranking of pages and updated dynamically as HTTP requests from the users arrive [6]. To begin with, HTTP requests arrive at the Predictor Agent. The Predictor Agent uses the data from the data-structure for prediction, and after predicting the forthcoming requests, passes the requested URL to the Update Agent to update the data-structure.

In our prediction model shown in Figure 2, we categorize the users on the basis of the related pages they access. Our model is divided into levels based on the popularity of the pages. Each level is a collection of disjoint classes and each class contains related pages. Each page placed in higher levels has higher probability of being predicted.

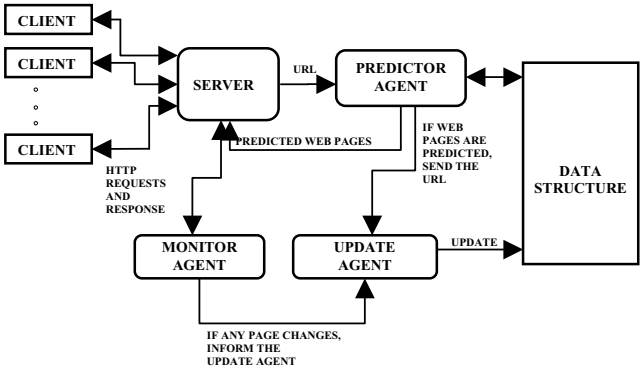

Fig. 1. Proposed Structure of the Agent Based Prediction Model

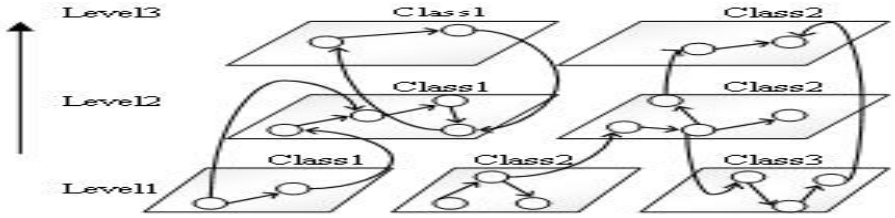

Fig. 2. The Prediction Model 'T'

Mathematically the model may be represented as:

Let T is the Prediction Model which is a logical reconstruction of the site graph. T is composed of discrete classes each containing some URLs,

Let $C = \{C_1, C_2, C_3, \ldots, C_n\}$ is the set of classes, where n= Number of classes.

For every element C_i in C there exists,

$CU_i = \{U_1, U_2, U_3, \ldots, U_m\}$ which is a set of URLs in plane C_i

And $i = 1,2,\ldots,n$, and $k \neq j$ for all U_k, U_j

belonging to C where $k, j = 1,2,\ldots,m$

Also,

$$\bigcap_{P=1}^{P=n} C_p = \{\ \}$$

$$\bigcup_{P=1}^{P=n} C_p = T$$

Each C_i in C has its own level number.

The connotation of 'class' can be defined as follows, when a user requests for a particular URL the next few pages have higher probability to have the same 'class' in

which the previous page belongs. So before starting the predictions the URLs must be assigned to their 'class' values.

For constructing the initial model, we define a subset of the set of total pages in the site as dominant pages. We assume these pages are direct descendant of the home page of the site. Based on these dominant pages, classification of the pages in the site is done. For example, in a general portal www.abc.com, sports.html may be a dominant page which is the base page for the 'sports' class. The candidates for dominant pages may be chosen manually by the site administrator or all the links in the home page may be considered as dominant pages before the server is started. The algorithm to create the initial model is as follows:

```
Let S= (U, L) is the digraph representing the website where U is the set
of URLs and L is set of links. The set of dominant-pages D is the subset
of U.

Clearly number of dominant pages = number of distinct classes.

Input: The site graph S.
       and the set of dominant pages D.
Output: The prediction model T.

An empty array called common-page is used for holding pages which are
linked from more than one dominant page.
Initially stack1, stack2 are empty.

1: Based on the page ranks, assign level numbers to pages.
2: Put all the elements of D in stack1, and assign them a unique class
   number.
   3: while(stack1 and stack2 are not empty),
      If(stack1 is empty)
       Pop all the pages from stack2 and push back to stack1.
      end if
   pop the first element from Stack1, name it P.
    for(all pages pointed by P)
      if(any page is already assigned a class number) then
         if(class number is same as P) then
         do nothing
      else
        add that page to common-page.
      end else
     else
       a. assign the class number of P as the class number of that page.
       b. push the page to stack2
     end else.
    end for
   end while.
4: for(each page in common-page)
     Reassign the class number same as that of the class having maximum
     number of links pointing to it. For any further conflict (i.e., if
     two or more classes have same number of links to this page) choose
     any random one.
```

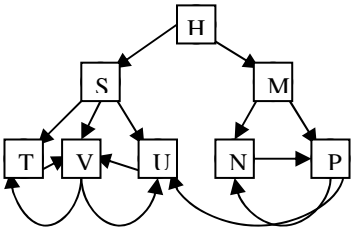

Fig. 3. Flow Diagram of the Algorithm

The above algorithm can be depicted as shown in Figure 3. Here, H is the home page. and two dominant pages are S and M. Now S is marked with class number CS and M is marked with class number CM where CS ≠ CM.

1. Initially push M, S into the stack1.
   stack1: M, S. stack2: void
2. Pop S and assign T, U and V the same class number as S i.e., CS and push them in stack2.
   stack1: M. stack2: T, U, V.
3. Pop M and assign N, P the class number CM. Push N, P in stack2.
   stack1: void. stack2: T, U, V, N, P.
4. Pop all the elements from stack2 and push back to stack1.
   stack1: P, N, V, U, T. stack2: void.
5. Since all the neighbors of T, U, V and N are already marked with class number and no conflict arises, they are popped one by one.
   stack1: P, stack2: void.
6. Pop P from stack1. The neighbor of P is N and U. But the 'class' number of P is equal to that of N, hence no action is taken. Since U is already marked with a 'class' number and that class number is not equal to the 'class' number of P, there is a conflict. So U is put in the common-page array.
   stack1: void. stack2: void.
7. Since stack1 and stack2 are both empty, we are entering step 4 of the algorithm. The common-page contains U. Since from CS class, it has two back links (from S and V) and from CM class it has only one back link (from P only), U will retain its class number CS.

The disjoint classes signify the categorization of the pages accessed by the users. Each level signifies the possibility of the page to be accessed in the future. Higher the level higher is the possibility to be accessed. The pages in the various classes are promoted to the higher levels based on the number of accesses to that page by the user. The next request for a page is predicted according to its presence in a higher level than the current page that points to it. More than one page is predicted and sent to the user's cache depending upon the presence of links in the higher levels.

After calculating page ranks, a normalized value in the range of 1 to p is assigned to each page where p is the number of pages in the site. For storage reasons, the number of levels is restricted to a predefined constant value L, where typically $L=\lceil\sqrt{p}\rceil$. We further divide the p pages into L sets. For each set, classes are formed

depending upon the actual links present between them. Thus pages are categorized into disjoint classes "C." Each level and class is assigned a distinct number. In order to search for the presence of a page, the URL name is used as a key to the hash table data- structure.

Since we are working with a range of values for a level, we assign a counter to all the pages except those already in the uppermost level. For each request the counter is incremented, and when it reaches L, the page is promoted to the next higher level. Pages may traverse between levels when any of the following conditions occur: a) the page is demoted to a lower level when the time stamp value assigned to it expires; b) the page is promoted to a higher level if it has been modified recently. This is discussed further in Sub-section 2.3.

2.2 Predictor Agent

All required information about the pages of the Website is indexed using their URLs in a hash table where a URL acts as the key. When a request is received, a search on the hash table is conducted and the information thus obtained is analyzed in the following manner:

1. Get the level and class number of the requested URL.
2. Get the links associated with the page and also fetch their respective levels and class numbers.
3. Determine Prediction-Value (P-value) pairs for the entire candidate URLs, where a P-value pair is defined as [Level, Rank].
4. Sort the links of the requested URL according to the four types of precedence relations between two P-value pairs (Li, Ri) and (Lj, Rj): (Li, Ri) $<\cdot$(Lj, Rj); (Li, Ri) $\cdot>$ (Lj, Rj); (Li, Ri) $\|$ (Lj, Rj); (Li, Ri) $\approx$ (Lj, Rj).
5. Compare the links' level number with the URLs' level number.
6. Compare the class numbers of the links with that of the requested URL. The link having the same class number will get preference.
7. The links in the higher levels are the predicted links to be sent to the users' cache.

2.3 Update Agent

In the updating process we adjust the counter value and decide whether the page should go to a higher level, class numbers are assigned at the initial stage and remain static. The process may be described as follows:

```
1. Check the local counter associated with the requested URL.
2. If the counter value is less than (L-1) then increment the
   counter
      Else
      Fetch the current level number of the URL. Let it
      be L.
      Increment L.
3. Reset the counter and time stamp.
```

The Update Agent also checks periodically for a page that is present in a higher level and has not been accessed for a long duration to relegate it to a lower level according to a predetermined threshold value. This periodic process compares the timestamp of all the pages with this threshold value and demotes those pages which

exceed this threshold. There's another periodic check that checks the last date of modification of the page. If there is recent modification then the page is raised to a higher level. This is done as a recently modified page always has higher probability of being accessed by the user.

2.4 Monitor Agent

This module continuously monitors the server for any modifications in any of the pages and informs the update engine to make necessary changes in the Data Structure. For example if a page has been recently modified, it's DM (date of modification) should also be modified in the data-structure. Update Agent takes necessary actions to change the page's date of modification and promote it to one level higher. This is done as a recently modified page always has higher probability of being accessed by the user.

3 Experimental Setup

The prediction model is implemented using a link data-structure which is shown in Figure 4:

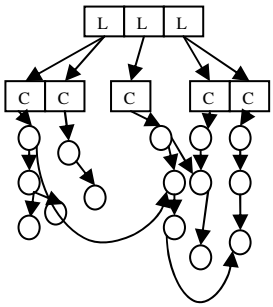

Fig. 4. Data-Structure Representing the Prediction Model

This data-structure represents the categorization of the URLs where the levels L_1, L_2,..Ln acts as index of the respective classes C_1, C2,….,C_n. Each Li where i = 1,…, n is the root for its classes and each class is the root for the respective URL trees. This data-structure is implemented in the form of a hash table with URLs being used as the key. Table 1 shows the implementation of the above data-structure.

Following are the brief description of each of the labels used in the data-structure: 'Key' represents the key of the current row in the hash table; 'URL' represents the Web address of the page; 'LC' represents the local counter associated with the page which represents the number accesses made to the page in a particular level. When this value reaches (L–1) the page is promoted to a higher level and this counter is reset; 'L#' is the level number and 'C#' is the class number; 'TS' is the timestamp associated with the URL that represents the duration for which the page has been in a

particular level; 'DM' represents the last date of modification of the page; 'Links' represents the list of links to which the current URL points.

When a request is received the row for the requested URL is fetched from the hash table using URL as the key. The level and class numbers are obtained. The links corresponding to this URL are obtained from the 'links' field of the hash table. The class and level numbers of these links is obtained from their respective rows in the hash table. Thus we can make predictions on the basis of level, rank and class values of the linked URLs.

Table 1. Tabular Representation of the Data-Structure

Key	URL	LC	L#	C#	TS	DM	Links
A1	A	2	1	2	Xx	Yy	A4,..
A2	B	0	1	2	Xx	Yy	A8,..
A3	C	2	1	3	Xx	Yy	A1,..
A4	D	1	2	3	Xx	Yy	A5,..
A5	E	3	2	4	Xx	Yy	A7,..
A6	F	1	2	4	Xx	Yy	A5,...

To implement the prediction algorithm, a highly parameterized prediction system is implemented in JAVA code. In order to get the real time HTTP request from the client, we developed a simple HTTP server. The HTTP server only supports the 'GET' operation and a very limited range of hard-coded MIME type. We hosted the web-server in our research lab and processed the requests. Finally, for the purpose of these tests, we focused on potential predictive performance (e.g., accuracy) and ignore certain aspects of implementation efficiency.

We evaluated the usefulness of our pre-fetching scheme using a simulator. This simulator has a GUI (Graphical User Interface). As described in the Section 1, we recorded the real time HTTP requests from the user and then showed the results of the prediction technique by means of a simulation environment. We coded a number of JAVA classes to implement the simulation environment.

A step by step execution of the simulator is also given below:

- It scans all the HTML pages and evaluates all the hyperlinks present in each of the pages.
- Based on these links it creates the site graph.
- It then applies the algorithm for creating the discrete classes onto the site graph.
- The simulator assigns unique numbers to the pages from the same classes in the Prediction Model.
- Now based on the site structure, it evaluates the rank of each HTML page.
- Based on these rank values the simulator now assigns the level value to each page. Higher rank implies higher level.
- After the creation of the initial model, the simulator starts the server.

For each request, the simulator updates the data-structure and displays the results in the graphical users interface.

4 Experimental Results and Evaluations

We examined the hit percentage vs. user session as per the prediction window size. Figure 5 is a snapshot of the server's session interface showing the hit/miss that occurred after each user request.

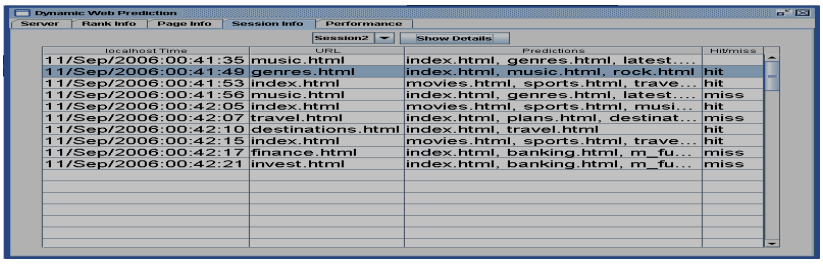

Fig. 5. Session Interface

The hit percentage remained consistent throughout the testing period including the initial stages. The size of the prediction window was taken as two and three considering the number of pages in our test environment. Size of a prediction window indicates the number of Web-Pages sent to the Client-cache by the Web-Server while predicting the pages. The average hit percentage was found to be around 35% with a prediction window size of 2 and 51% with a prediction window size of 3, an improvement of around more than 15%. In Figure 6, sessions recorded during the testing period at different intervals with variable prediction window sizes are plotted.

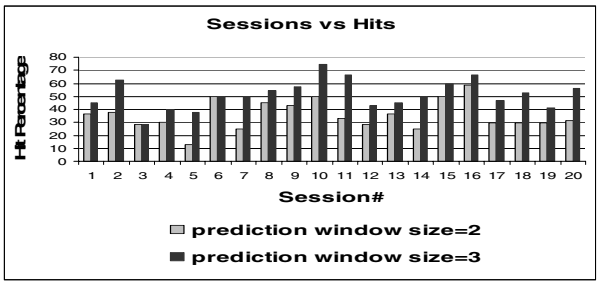

Fig. 6. Chart showing comparative hit ratio with different prediction window sizes

5 Conclusions

In most of the cases prediction of Web pages is done using logs and history data requiring a huge amount of memory to implement. Another problem found is the

inability to build up the prediction model in the initial stages when no log or history data is available. The use of page ranking in our method enables to build our prediction model in the initial stages and make predictions right away. Henceforth our model updates itself as per the access patterns of users. Categorizing the users into different classes also help as we don't have to keep track of each user as all access patterns are maintained in the form of sessions. Updating the model dynamically according to access patterns of users as well as changes in the content of the Website is computationally cheaper as it doesn't put extra load on the Web traffic for requesting or maintaining extra information.

Acknowledgements

We extend our sincere thanks to the anonymous reviewers for their valuable comments, which helped improve the paper. This work was supported by Techno India (West Bengal University of Technology), Calcutta, India.

References

1. Chen, X., Zhang, X.: Popularity-Based Prediction Model for Web Prefetching. IEEE Computer. Vol.36, No.3 (2003) 63-70
2. Palpanas, T., Mendelzon, A.: Web Prefetching using Partial Match Prediction. Department of Computer Science, University of Toronto. Technical Report CSRG-376 (1998) 1-21
3. Davison, B.D.: Learning Web Request Patterns. Web Dynamics: Adapting to Change in Content, Size, Topology and Use. Springer-Verlag, Berlin Heidelberg New York (2004) 435-460
4. Bonino, D., Corno, F., Squillero, G.: An Evolutionary Approach to Web Request Prediction. 12th International WWW Conference, Budapest, Hungary (2003) poster S2
5. Su, Z., Yang, Q., Lu, Y., Zhang, H.: WhatNext: A Prediction System for Web Requests using N-gram Sequence Models. 1st Int. Conf. on Web Information System and Engineering (2000) 200-207
6. Brin, S., Page, L.: The Anatomy of a Large-scale Hypertextual Web Search Engine. 7th WWW Int. Conf., Brisbane, Australia (1998) 107-117
7. Kleinberg, J.: Authoritative sources in a hyperlinked environment. 9th ACM-SIAM Symposium on Discrete Algorithms, ACM Press (1998) 668-677
8. Mukhopadhyay, D., Biswas, P.: FlexiRank: An Algorithm Offering Flexibility and Accuracy for Ranking the Web Pages. Lecture Notes in Computer Science, Vol. 3816. Springer-Verlag, Berlin Heidelberg New York (2005) 308 – 313