Rule-Assisted Prefetching in Web-Server Caching *

Bin Lan, Stephane Bressan, Beng Chin Ooi, Kian-Lee Tan
Department of Computer Science, National University of Singapore
3 Science Drive 2, Singapore 117543
{lanb,steph,ooibc,tankl}@comp.nus.edu.sg

ABSTRACT

Web servers manage large number of documents of widely variable sizes. Moreover, the access patterns on the documents may also change over time. While some documents are highly popular over a prolonged period of time, we expect newly added documents to increase in popularity while demand for most older documents decreases. It is therefore important to design effective caching strategy at the web server. In this paper, we present our approach to the problem. Our main contribution lies in the design of a novel prefetching strategy, called RAP. RAP identifies a set of association rules from the Web server's access log. Unlike existing mining strategy, RAP's miner values recently added log records more than earlier log records. Based on the rules, RAP predicts and prefetches documents from users initial requests. We conducted extensive study to evaluate RAP. The results show that RAP significantly outperforms existing schemes. We also show that the mining and caching cost is relatively low.

Categories and Subject Descriptors

H.3.5 [**Information Systems**]: Information storage and Retrieval—*Online Information Services* [Web-based services]; I.5.2 [**Computing Methodologies**]: Pattern Recognition—*Design Methodology* [Pattern analysis]

General Terms

Management Performance

Keywords

Web Server, Caching, Prefetching, Pattern Analysis

1. INTRODUCTION

*The full version of the paper can be downloaded from http://www.comp.nus.edu.sg/~lanb/cikm00.ps

CIKM 2000, McLean, VA USA

The research on Web-caching (caching of Web-documents) has mainly concentrated on secondary storage caching at the proxy server side or, more recently at the client side. Cache management at the server side, i.e. placement (fetching and prefetching) and replacement policies of documents in the server's main memory has been relatively neglected. One argument for this lack of research is that the requirements are not significantly different from those of traditional file systems or even database systems, and therefore standard techniques can be used. However, Web-server caching has unique characteristics and requirements.

First, as opposed to the typical access patterns in file systems and database systems, from the point of view of the Web-server, the same user almost never reads the same document twice in a short period time (typically one day). This indeed is due to the availability of caches at the client (and proxy) side. Multiple requests for the same document only occur when the remote caches are full *or* when the user explicitly requests a refreshed (up-to-date) document. Finally, because of the proxy server's cache, a fresh request for a document may never reach the Web-server. Incidentally, this suggests (as we will show empirically) that the hyper-link structure of the HTML document may not be as relevant as the access log for learning the server's access patterns.

Second, as opposed to traditional database objects, web documents have a coarser granularity than the page (or block). Nevertheless, as opposed to general file systems, for a specific Web-site, although anything but uniform, the document size distribution can be characterized [3] or relatively accurately guessed. Intuitively, it also seems to be the case that, unlike files in general, Web-documents being in a common pool may be accessed with very similar patterns by most users. Finally, it is clear that main memory caching on Web-servers cannot be directly compared to client and proxy's secondary storage caching. The scarcity of the main memory calls for carefully designed memory management strategies.

In this paper, we present our approach to the web-caching problem. We integrated three aspects of web-caching into a single framework: initial loading of the cache, prefetching of potentially useful documents and cache replacement policies. In particular, our main contribution lies in the design of a novel prefetching strategy, called RAP. RAP identifies a set of association rules from the Web server's access log. Unlike existing mining strategy, RAP's miner values recently added log records more than earlier log records. Based on the rules, RAP predicts and prefetches documents from users' initial requests. We conducted extensive study to evaluate RAP

(coupled with the initial placement and replacement strategies) on both static and dynamic workloads. The results show that RAP significantly outperforms existing schemes. We also show that the mining and caching cost is relatively low. To our knowledge, this is the first of its kind to study all three aspects in designing a web-caching strategy.

2. RELATED WORK

The access log has been shown to reflect the actual requests reaching the server behind the clients' and proxies' caches. Arlitt et al [3] studied six representative access logs; two of which, the NASA and ClarkNet access log files are used in our study. The authors identified ten different invariants, and hypothesized that these invariants exist in most access logs. In particular, they showed that, in their benchmarks, the file size distribution is not uniform and the daily patterns are representative. Generally, the daily patterns seem to be a reliable hypothesis. This is confirmed by the results of Tatarinov et al [11], who proposed to prefetch documents that are the most frequently accessed according to the previous day's access log. We will refer to this latter method as *STATIC*.

The granularity of the caching is also an interesting issue. Traditional cache management considers pages. Web servers need to fetch entire documents. The impact of the document size in the placement and replacement policies in the context of Web-caching has received some attention(e.g., [1]).

The idea of learning access patterns to determine a prefetching strategy for databases, file systems or Web caches is quite natural. For instance FIDO [10] is a database cache "that learns to fetch". In [7], a predictive approach was proposed for file systems caches. Several authors have proposed to use statistical information for Web prefetching between client and proxies[6, 5, 4, 9]. In the different context of server to client and server to proxy pushing (where the cache space is not a sparse resource but where the bandwidth is), we ourselves have shown [8] that the use of association rules mined from the access log can significantly improve the hit rate.

Data mining and association rule[2] have been extensively studied in the literature. Besides Web-caching, many applications can benefit by the mining of access patterns and discovery of Web usage.

3. A FRAMEWORK FOR WEB-CACHING

3.1 System Architecture

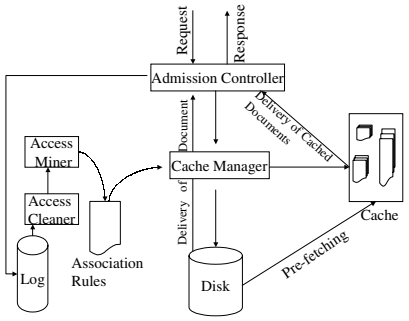


Figure 1: Buffer Management Based on *RAP*

Figure 1 shows the basic architecture for supporting the RAP strategy. RAP requires components capable of learning the association rules from the access log: the *Access Cleaner* and the *Access Miner*, as well as a component controlling the placement, prefetching of documents according to the mined rules, and replacement of documents when the cache is full: the *Cache Manager*. The *Access Cleaner* cleanses the data and filters out irrelevant information (such as dynamically generated documents , "Not Found" documents, etc.) Then the *Access Miner* analyses and discovers access patterns in the form of Association Rules. It forwards these rules to the Cache Manager. The *Cache Manager* implements the actual RAP strategy, which we shall describe in more detail below.

In practice, in addition to the above mentioned components, an *Admission Controller* determines if a request can be processed or should be queued or rejected according to the availability of various system resources. In this paper, we shall assume that all requests are admitted.

3.2 The Access Miner

After Access Cleaner's data cleansing process, the access log contains a chronological list of entries recording which document (identified by its URL and characterized by its size) has been requested by which user (identified by the IP address of the client machine . We call a *transaction* the chronological list of entries for a given user over a fixed period of time. We shall refer to the fixed period of time as the *transaction's window*. The transaction window is a parameter of the Access Miner. Looking at all the transactions in this time interval, we construct rules of the form $A \rightarrow B$, where A and B are documents (local URL). The intuitive interpretation of such rules is that after document A has been requested by a user, document B is likely to be soon requested by the same user, since it is usually the case according to the access log.

Our approach to mining association rules differs from the traditional approach [2] in that only consecutive documents in the transactions contribute to the rules. Moreover, as a first cut, we only focus on rules whose itemset size is 1, i.e., the *antecedent* and *consequent* of the rule has only one document. These restrictions significantly reduce the complexity of the mining process.

3.3 The Cache Manager

We have identified three main issues that the cache manager should address. First, at the beginning of the day, the cache has to be loaded. In this paper, we adopt a day-to-day cache management strategy. In other words, at the beginning of each day, the cache is initially loaded with popular documents. Here, it is important to determine the right set of documents to load. Second, useful documents should be prefetched based on users' earlier requests. Again, here, it is important to predict accurately the documents that a user may want from his/her earlier requests. Third, stale or unpopular documents have to be identified for replacement. This is especially the case when the cache is full and the system attempts to fetch a new document. Thus, a good replacement policy is needed to select a victim among the documents in the cache.

Clearly, there can be many different Cache Managers based on solutions on each of the issues. In fact, each of the issues has been separately addressed in the literature. We, how-

ever, feel that the various aspects interact with one another and should be integrated into a single framework. Our approach to these issues are as follows: First,*Loading the cache at the beginning of the day.* Second,*Prefetch potentially useful documents based on users' requests.* Last,*Identify stale documents that should be replaced.* In this paper, we will examine the following policies: LRU,LRU-MIN,LRU-2,LFU-MIN and OPT.

4. RULE-ASSISTED PREFETCHING

4.1 Mining Prefetching Rules

Our mining algorithm is different from existing algorithms. First, the proposed algorithm is a near-line algorithm that incrementally refines the rules as transactions are added into the log. Second, earlier logs can influence the choice of the rules. Third, recently added logs are given greater values than earlier logs. Finally, the algorithm operates on a window of logs that stretches for 2 days[1]. Because the generation of the rules from the large itemset is straightforward, we just show how to generate the large itemset in the following. We first give an overview of the approach before discussing the algorithm in detail.

We shall use the following notation in our discussion. Let L_i denote the large itemset for a database D_i. We denote L_{ij} as the large itemset for $D_i \cup D_{i+1} \ldots \cup D_j$. Let C_{ij} be a set of candidate itemset for $D_i \cup D_{i+1} \ldots \cup D_j$. Let R_{ij} be the rules extracted for $D_i \cup D_{i+1} \ldots \cup D_j$. Finally, we denote $supp(l|D_i \cup D_{i+1} \ldots \cup D_j)$ as the support of l in $D_i \cup D_{i+1} \ldots \cup D_j$, and $supp(l|D_i)$ be the support of l in D_i.

The algorithm organizes a day (i.e., 24 hours) into k equal time segments, $[t_1,\ t_2), [t_2,\ t3), \ldots, [t_k, t_{k+1})$ where $t_{i+1} - t_i = t_i - t_{i-1}$ for $k \geq i > 1$. t_1 is the beginning of each day, and t_{k+1} is the close of the day. At t_1, the algorithm gets the large itemset L_0 from the log records of the last 24 hours . Let this database be denoted as D_0. During $[t_i, t_{i=1})$, new access records are added to the access logs, and we denote these datasets as D_i, i.e., at t_2, we will generate the large itemset L_{01} from the logs of D_1 and L_0; at t_3, we will generate the large itemset L_{02} from the logs of D_2 and L_{01}; and so on till at t_{k+1}, we will generate the large itemset L_{0k} from the logs of D_k and $L_{0(k-1)}$. These are the 2 days' log records that are examined during the day from t_1 to t_{k+1}. The next day will begin with $D = \cup_{i=1}^{k} D_i$, and a whole new set of logs for that day. This cycle is repeated with the algorithm looking at 2 days' log only.

Our approach comprises the following steps:

1. At time t_1 (which is the beginning of a new day), we generate L_0 (also denoted as L_{00}) from D_0.
2. In general, at time $t_i (i > 1)$, we will generate the large itemset L_{0i} from D_i and the large itemset $L_{0(i-1)}$.

We are now ready to look at the algorithm of generating the L_{0i} from D_i and $L_{0(i-1)}$. The algorithm proceeds as follows.

1. Generate a candidate set of items, C. This is done as follows.
   (a) Generate L_i from D_i.
   (b) Prune $L_{0(i-1)}$.
   Remove items from $L_{0(i-1)}$ when they do not appear in D_i. In fact, such items will only appear again during the course of the day if they become large item in any of the access logs. Otherwise, since they are not frequently accessed, it makes sense not to prefetch based on them. We denote the pruned large itemsets as $L'_{0(i-1)}$.
   (c) $C = L_i \cup L'_{0(i-1)}$
2. Generate the large itemset L_{0i}
   (a) Recompute the support $supp(c| \cup_{j=0}^{i} D_j$ for each $c \in C$ as follows:
   $$w * supp(c| \cup_{j=0}^{i-1} D_j) + (1 - w) * supp(l|D_i)$$
   where w is a weighting factor that measures the relative importance of the large itemsets $L_{0(i-1)}$ in $\cup_{j=0}^{i} D_j$.
   (b) Generate L_{0i} from C, which includes those items with supports greater than some predetermined threshold, minSupport.

It is worth noting that (1)We just need to scan $D_i (i > 0)$ to get L_{0i} instead of scanning $\cup_{j=0}^{i} D_j$. In other words, it costs less and therefore make it practical . (2)We generate L_i from D_i using traditional mining algorithms. (3)For simplicity, we have set the minSupport to be the same for the generation of L_i and L_{0i}.

4.2 Prefetching Scheme

We can generate the rules from corresponding large itemset. Given that we have a set of rules, the prefetching scheme works as follows.

1. Let the request be for document A.
2. Scan the rule database[2] for rules of the form $A \rightarrow X$ for some document X.
3. Order these rules in descending order of their confidence values.
4. Find the first rule (in descending order) whose X is not in the cache. Read X into the cache.

As can be seen, the scheme prefetches at most one document each time.

5. EXPERIMENTAL STUDY

5.1 Experimental Setup and Performance Metrics

We generated synthetic datasets for our experimental study. To ensure that the datasets are as realistic as possible, we examine the traces from three publicly available access logs – NASA [3], ClarkNet [3], and SF100 (http://www.science.nus.edu.sg/Academic/SFM/index.html) – and generated our datasets according to the distribution found in these logs.

It is interesting that all three data sets exhibit similar characteristics: in terms of reference frequency, between 80%

[1]We used *day* as the unit here. The algorithm can be generalized for other unit of time.

[2]The rule database can be organized using some indexing scheme, e.g., as a hash table.

and 95% of the requests are addressed at 10% of the documents(we call it *10-90 Rule*); and repetitive bursty pattern that occurs daily. Because of the bursty daily pattern, we decide to define the transactions' window to be one day. Moreover, we also noted the following second order transaction-based characteristics on three workloads. With respect to the *length of the transaction*, we found that more than 90% of users access less than 50 objects on the server in one transaction. In terms of *inter access time in the transaction*, (i.e., the average time between the consecutive requests in the transaction), we found that the distribution follows the Pareto distribution. Moreover, the *inter-user arrival time* (i.e., time between two consecutive users "arrival" at the server) follows an exponential distribution. Table 1 summarizes the characteristic related to the transaction.

Based on the characteristics of the real workloads, we generated two synthetic datasets:

- *Static data set.* The number of Internet users is growing at an unprecedented rate and Web-servers service an ever-increasing number of requests to larger and larger collections of documents. To take into account this trend, we used the three core data sets to model much bigger volume of transactions that exhibit similar access patterns. We artificially duplicated the documents and their structures 128 times as different documents, and increased the total traces with reference to these objects by the same magnitude. In these workloads, popular documents become unpopular at a very slow rate. Thus, we shall refer to these workloads as *static* workload.

- *Dynamic data set.* More and more commercial activities happen on the Web, and they constantly change the content and flavor to meet the customers requirement for fresh things. In addition, many portal sites even refresh their contents automatically every day.[3] For example, Yahoo generates 50 GB data per day. Table 2 summarizes the settings for the dynamic synthesis workload used in our study.

  In order to quantify the *dynamic*, we assume that 10% of the "hot" files in previous day will be "cold" in the next day. The total entries number in DB is about $6,550,000$.

The objective of our study is to maximize the rate at which the Web-server can send the responses to the clients' requests. The more requested objects found in the cache, the higher the rate. We do not wish to complicate the management of our cache by allowing splitting of files into pages and the retaining portions of files. Although files are transferred as units to the clients, they are nevertheless of variable size and are transferred as blocks from the disk into main memory. The *file hit rate* is defined as the ratio between the number of files requested and found in the cache (hit) to the total number of requests (hit and miss). The *block hit rate* is defined as the ratio of the sum of the sizes of the files requested and found in the cache to the sum of the sizes of all the files requested.

5.2 Experiments on Static Workload

Since the workload is static, the popular objects remain largely popular. As such, we adopt a simplified version of RAP where the rules are obtained from the logs the day before. In other words, we have only one log per day, and there is no smaller incremental logs.

The default settings for the various parameters are as follows. The transaction window is one day according to the invariants observed for the three benchmarks. The minimum support and minimum confidence thresholds chosen for each of the three data sets have been respectively set to 1% and 10% for all three access logs for the sake of uniformity. We discuss the tuning of the thresholds in section *5.2.3*. For these thresholds, the popular documents and the mined rules are few compared to the total number of documents. Consequently few association rules are mined for each popular document. For all the experiments, we consider a block size of 4096 bytes(4KB).

5.2.1 Comparing the RAP Variants

In this section we compare the different variants of RAP obtained by integrating RAP and a replacement policies. We evaluate the schemes for four buffer sizes, 1, 4, 16 and 64 MB. Figures 2 to 4 report the file hit rate and block hit rate for each of the three access logs. As expected, all strategies improve as the buffer size increases. Naturally, the differences between the strategies are attenuated as the buffer size increases. We also see that no significant difference in the relative performance and the trend appears between the file and block hit rate measurements. This is due to the skewness of the file size distribution.

More importantly we see that RAP-LFU-MIN performs best among all variants. One natural question to ask is whether the combination of RAP with a replacement strategy is an orthogonal issue to the performance improvement brought by the association rule-based prefetching. For this we have tested the performance of each individual replacement policy in the absence of prefetching. In these tests, the cache behaves as a file system cache: documents are brought into the cache when they are requested and are replaced according to the replacement policy.

Figure 5 shows the results for the variants of RAP with a buffer size of 16 MB as well as the results for the individual replacement policies. We see that the performance of the replacement policies and the performance of the corresponding combinations of RAP with the replacement policies are ordered in the same way. Moreover, we observe that the differences between the RAP-based schemes is relatively small compared to non-RAP-based schemes. This shows RAP's ability to prefetch documents that are indeed useful, making replacement policies less of an issue for a fixed memory size. Clearly, RAP-OPT performs the best, and provides an upper bound for the RAP strategies. From the results, we also note that RAP-LFU-MIN is the best strategy among the RAP strategies for the implementation of a practical system. It is the one we use in the sequel unless otherwise specified.

5.2.2 Comparison with Alternative Pre-fetching Strategies

We now compare RAP with representative alternative strategies. We use two strategies based on the knowledge of the hyper-structure of the documents. The first strategy, Hyper-link, prefetches all the documents referenced by the document currently being prefetched. Such a strategy is known to be very effective on the client side (although its

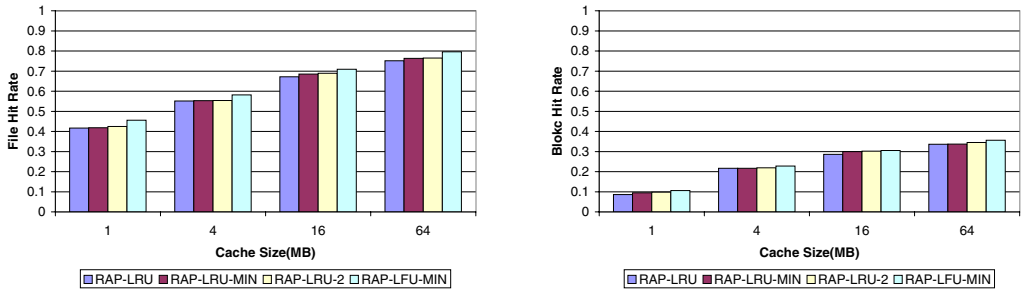
[3] We call these kinds of sites *dynamic servers*.

transaction length	90% of the transaction has less than 50 objects
inter access time in the transaction	Pareto Distribution($\alpha = 1, \beta = 0.15 \sim 0.27$)
inter transaction time	exponential distribution

Table 1: Summary of Characteristics Related to the Transaction

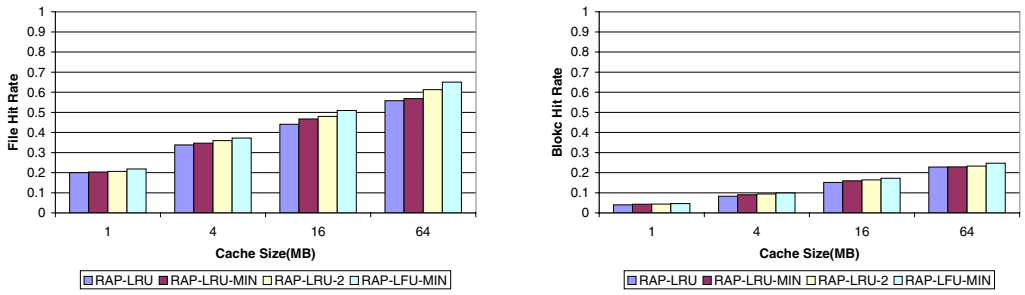
total number of files	5000
file size	Pareto Distribution with $\alpha = 1024, \beta = 0,50$
transaction length	50
inter access time in the transaction	Pareto Distribution($\alpha = 1, \beta = 0.20$)
inter transaction time	exponential distribution with $\beta = 20$
Day Number	30(one month)

Table 2: Synthetic Dynamic Data Set Setting



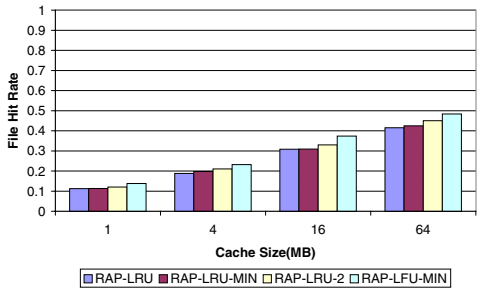
(a) File Hit Rate (b) Block Hit Rate

Figure 2: File Hit Rate and Block Hit Rate for SF100

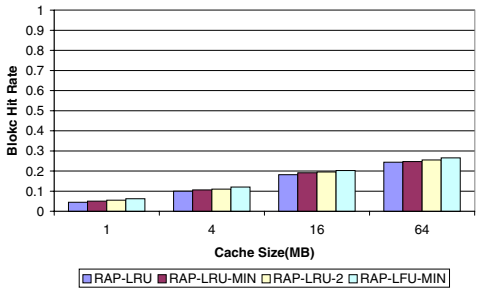


(a) File Hit Rate (b) Block Hit Rate

Figure 3: File Hit Rate and Block Hit Rate for NASA



(a) File Hit Rate



(b) Block Hit Rate

Figure 4: File Hit Rate and Block Hit Rate for ClarkNet

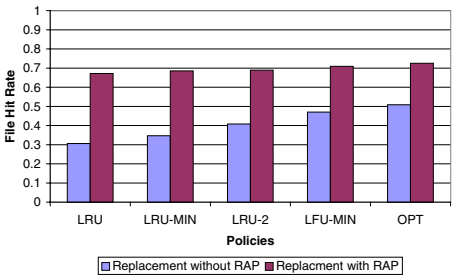


Figure 5: Comparison of Replacement Policies with and without RAP for SF100 (Cache Size = 16MB)

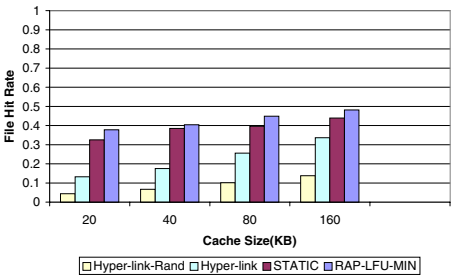


Figure 6: Comparing Alternative Prefetching Schemes for Chapter 3 of SF100

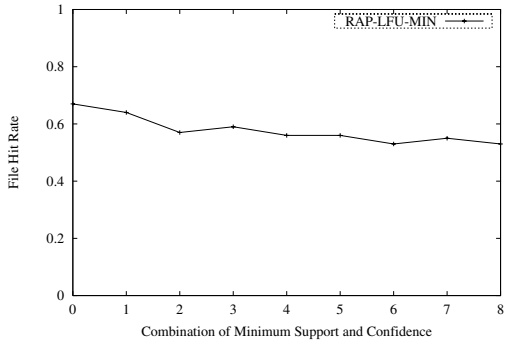
efficiency depends on the network's and servers' latency). The amount of prefetching this strategy incurs may be large and may trigger series of replacements. A simplified version of Hyper-link, Hyper-link-Rand, prefetches one of the referenced documents selected randomly. Another strategy we consider is the STATIC strategy proposed in [11]. The most popular files according to the previous day's log are statically loaded into the cache (as far as the cache size allows it) at the beginning of the day. No replacement takes place.

Since we analyzed the structure of only a subset of SF100, the results presented are concerned with much smaller logs and smaller document sets. We chose chapter 3 of SF100, which contains 403 distinct files of total size $645KB$. For these reasons, we have also reduced the cache size. The performance results for cache sizes 20, 40, 80, and 160KB are presented in Figure 6. As shown, both hyper-structure-based schemes do not yield a very good performance, and prefetching based on hyper-structures does not appear to be a good idea. While Hyper-link prefetches too many documents, Hyper-link-Rand prefetches documents that may not be relevant. The STATIC method is quite efficient despite its simplicity. It is obvious that RAP-LFU-MIN remains the more superior algorithm, demonstrating that prefetching can be more effective than simply loading the cache with the popular documents of the last 24 hours.

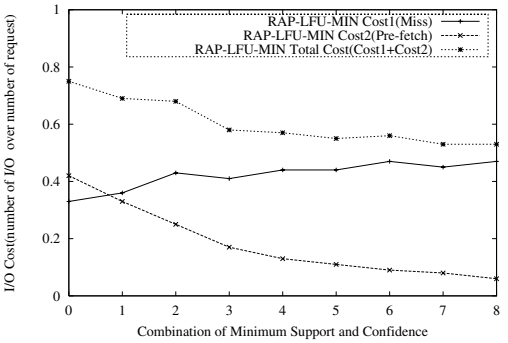
5.2.3 Effect of the Support and Confidence Thresholds

Given the simplicity and effectiveness of STATIC, we need to underline that the main advantage of RAP is that it can be tuned by choosing different support and confidence thresholds. These thresholds allow us to control the mining granularity and therefore the responsiveness of the system to diluted patterns. The following experiment is using RAP-LFU-MIN with SF100. We chose 9 different combinations of the minimum support and minimum confidence thresholds. Figure 7 shows the performance of RAP-LFU-MIN for the 9 combinations. Figure 7(a) confirms that the hit rate degrades as the number of rules mined decreases. However, by observing 7(a), it may seem unusual that the hit rate is not badly affected by the drastic decrease in the number of association rules. The results are logical as accesses to Web pages are highly skewed. From the access log, one is able to observe that 10% of all the files at any particular instance account for 90% of the Web accesses. This property of the three test-beds used explains the relative success of STATIC. However, RAP can perform better than STATIC as well as adapt to different distributions.

Given the actual inter-request rate, we can safely consider that the I/Os necessary for the prefetching can be interleaved with the normal execution without impacting nega-



(a)



(b)

Figure 7: Effects of Pre-fetching(SF100, Cache Size=16MB)(Combination of Minimum Support and Confidence is sorted by the descending number of association rules)

tively on the response time. If the service load grows, it is always possible to design a system architecture in which prefetching is done asynchronously thus limiting the negative interaction between service and prefetching. Nevertheless, we study the two different I/O cost for the RAP strategy in order to understand the consequences of the choice of the thresholds:

- The I/Os caused by the fetching of documents not in the cache (missed-documents), which is the main cost from the user's response time point;
- The I/Os caused by the prefetching of documents.

Figure 7(b) shows the percentage of I/Os incurred for the fetching of documents not in the cache (missed-I/Os), for the prefetching itself (prefetch-I/Os), and the total number of I/Os. Naturally, the prefetching cost increases with the number of association rules generated. This increase dominates the performance gain for the response. It is therefore important to set the threshold parameters high enough to allow prefetching to be interleaved with the requests or performed asynchronously.

An important observation from this experiment is that the total I/O cost fluctuates within a small range, providing us room to fine-tune the minimum support and confidence levels to yield a higher hit rate while keeping the cost low.

5.3 Experiments on Dynamic Workload

For the dynamic workload, popular documents become unpopular and unpopular documents become popular at a much faster rate. Here, we evaluate RAP with four incremental databases, i.e. a day is split into four equal time periods. As default, we also set the weight, w, to 0.45 to favor the incremental databases slightly. This is also the optimal setting for our data set. We compare RAP against two other schemes:

- Mine the rules from yesterday's log only. This is essentially the RAP strategy used for the static workload. We shall refer to this as RAP_1.
- Mine the rules from all past logs. This method is referred to as RAP_{ALL}.

Before looking at the experimental results, we shall first look at how the algorithms perform in terms of their abilities to produce good rules. We observe that two consecutive sets of logs, R_1 and R_2, can be viewed as follows: the earlier set, R_1, is essentially a training data set used to predict the rules that may appear in the later set, R_2. In other words, we have two measures:

$$rightness = \frac{number\ of\ rules\ r\ where\ r \in R_1\ and\ r \in R_2}{number\ of\ rules\ r\ where\ r \in R_1}$$

This reflects how many rules in R_2 are predicted correctly by mining R_1.

$$wrongness = \frac{number\ of\ rule\ r\ where\ r \notin R_1\ and\ r \in R_2}{number\ of\ rule\ r\ where\ r \in R_1}$$

This indicates how many rules may be wrongly predicted.

Clearly, a good algorithm should lead to high rightness value, and low wrongness value.

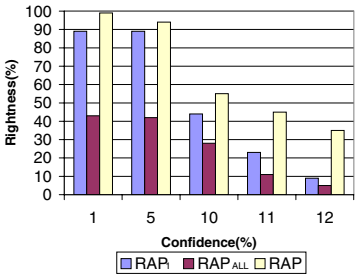
In the experiments, we set the minSupport=1%. Because at this minSupport, we can get the "hot" files, which accounts for 10% of the DB size, which agrees with the *10-90 rule*. We vary the minConfidence from 1% to 12%.

Figure 8 shows the goodness measures. Clearly, RAP outperforms the other RAP_1 and RAP_{ALL} by having the largest number of rules predicted correctly, and the fewest number of rules predicted wrongly. RAP_{ALL} performs the worst because it depends on too much historical information. In particular, because of the dynamic workload, older logs no longer provide a good indication of the access pattern and thus introduces noise into the mining strategy.

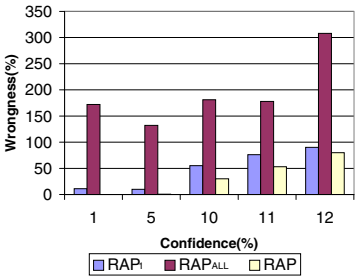
We also examine the time to mine the rules. It costs 217109,469907 and 21246(millisecond) for RAP_1,RAP_{ALL} and RAP,respectively ,which clearly demonstrates that RAP is not only effective but efficient as compared to the other two schemes. Since RAP_{ALL} is both ineffective and inefficient, we shall not study it further.

Figure 9 shows the hit rate result of RAP and RAP_1 as the cache size varies. As expected, RAP is superior by virtue of the fact that it looks at more recent logs in determining the prefetching rules.

We also study the effect of weights and the number of the incremental log on the performance RAP and present the results in the full paper.



(a) Rightness



(b) Wrongness

Figure 8: Comparison of Goodness Measures

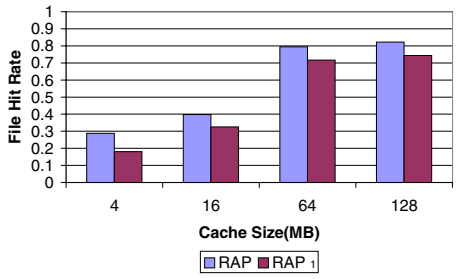


Figure 9: RAP **vs** RAP_1

6. CONCLUSION

We have discussed a new approach to prefetch documents in Web-server caches. The new cache management strategy we propose, called RAP, is based on simple association rules mined from the server's access log. We evaluated RAP on both static and dynamic workloads. Our performance study shows that RAP coupled with the LFU-MIN replacement policy performs the best overall. Moreover, RAP-LFU-MIN outperforms other alternative schemes that have been proposed in the literature. We are currently studying the strategy using theoretical access patterns. At the same time we are implementing a complete Web-server using RAP-LFU-MIN. Our implementation uses the extensible Web server Jigsaw distributed by the World Wide Web Consortium.

Acknowledgment

We like to thank Arlitt and the Computer Center of National University of Singapore, for respectively making the NASA and ClarkNet, and SF100 data sets available to us.

7. REFERENCES

[1] C. Aggarwal et al. Caching on the World Wide Web. *IEEE Transactions on Knowledge and Data Engineering*, 11(1):94–107, Jan. 1999.

[2] R. Agrawal et al. Mining association rules between sets of items in large databases. In *SIGMOD Conference*, pages 207–216, 1993.

[3] M. F. Arlitt et al. Web server workload characterization:the search for invariants. In *SIGMETRICS*, pages 126–137, 1996.

[4] A. Bestavros. Using speculation to reduce server load and service time on the WWW. In *CIKM*, 95.

[5] K. M. Curewitz et al. Practical prefetching via data compression. In *SIGMOD Conference*, 1993.

[6] L. Fan et al. Web prefetching between low-bandwidth clients and proxies:potential and performance. In *SIGMETRICS*, 1999.

[7] J. Griffioen et al. Reducing file system latency using a predictive approach. In *USENIX*, 1994.

[8] B. Lan et al. Making Web servers pushier. In *WEBKDD'99*, 1999.

[9] V. N. Padmanabhan et al. Using predictive prefetching to improve World Wide Web latency. *ACM SIGCOMM Computer Communication Review*, 26(3):22–36, 1996.

[10] M. Palmer et al. Fido: A cache that learns to fetch. In *VLDB*, pages 255–264, 1991.

[11] I. Tatarinov et al. Static caching in Web servers. In *Proceedings of 6th International Conference on Computer Communication and Networks*, 1997.