Chapter VIII

Cache Management for Web-Powered Databases

Dimitrios Katsaros and Yannis Manolopoulos
Aristotle University of Thessaloniki, Greece

ABSTRACT

The Web has become the primary means for information dissemination. It is ideal for publishing data residing in a variety of repositories, such as databases. In such a multi-tier system (client - Web server - underlying database), where the Web page content is dynamically derived from the database (Web-powered database), cache management is very important in making efficient distribution of the published information. Issues related to cache management are the cache admission/replacement policy, the cache coherency and the prefetching, which acts complementary to caching. The present chapter discusses the issues, which make the Web cache management radically different than the cache management in databases or operating systems. We present a taxonomy and the main algorithms proposed for cache replacement and coherence maintenance. We present three families of predictive prefetching algorithms for the Web and characterize them as Markov predictors. Finally, we give examples of how some popular commercial products deal with the issues regarding the cache management for Web-powered databases.

INTRODUCTION

In the recent years the World Wide Web or simply the Web (Berners-Lee, Caililiau, Luotnen, Nielsen & Livny, 1994) has become the primary means for information dissemination. It is a hypertext-based application and uses the *HTTP* protocol for file transfers. What started as a medium to serve the needs of a specific

scientific community (that of Particle Physics), has now become the most popular application running on the Internet. Today it is being used for many purposes, ranging from pure educational to entertainment and lately for conducting business. Applications such as digital libraries, video-on-demand, distance learning and virtual stores, that allow for buying cars, books, computers etc. are some of the services running on the Web. The advent of the *XML* language and its adoption from the World Wide Web Council as a standard for document exchange has enlarged many old and fueled new applications on it.

During its first years the Web consisted of static *HTML* pages stored on the file system of the connected machines. When new needs arose, such as the E-Commerce or the need to publish data residing in other systems, e.g., databases, it was realized that we could not afford in terms of storage to replicate the original data in the Web server's disk in the form of *HTML* pages. Moreover, it would make no sense to replicate data that would never be requested. So, instead of static pages, an application program should run on the Web server to receive the requests from clients, retrieve the relevant data from the source and then pack them into *HTML* or *XML* format. Even the emerged "*semistructured" XML databases*, which store data directly into the *XML* format, need an application program which will connect to the DMBS and retrieve the *XML* file (or fragment). Thus, a new kind of pages, dynamically generated and a new architecture were born. We have no more the traditional couple of a Web client and a Web server, but a third part is added, the application program, running on the Web server and serving data from an underlying repository, in most of the cases being a database. This scheme is sometimes referred to as *Web-powered database* and the Web site, which provides access to a large number of pages whose content is extracted from databases, is called *data intensive Web site* (Atzeni, Mecca & Merialdo, 1998; Yagoub, Florescu, Issarny & Valduriez, 2000). The typical architecture for such a scenario is depicted in Figure 1. In this scheme there are three tiers, the database back-end, the Web/application server and the Web client. In order to generate dynamic content, Web servers must execute a program (e.g., *server-side scripting mechanism*). This program (script) connects to the DBMS, executes the client query, gets the results and packs them in *HTML/XML* form in order to return them to the user. Quite a lot of server-side scripting mechanisms have been proposed in the literature (Greenspun, 1999; Malaika, 1998). An alternative to having a program that generates *HTML* is the several forms of *annotated HTML*. The a*nnotated HTML,* such as *PHP*, *Active Server Pages*, *Java Server Pages,* embeds scripting commands in an *HTML* document.

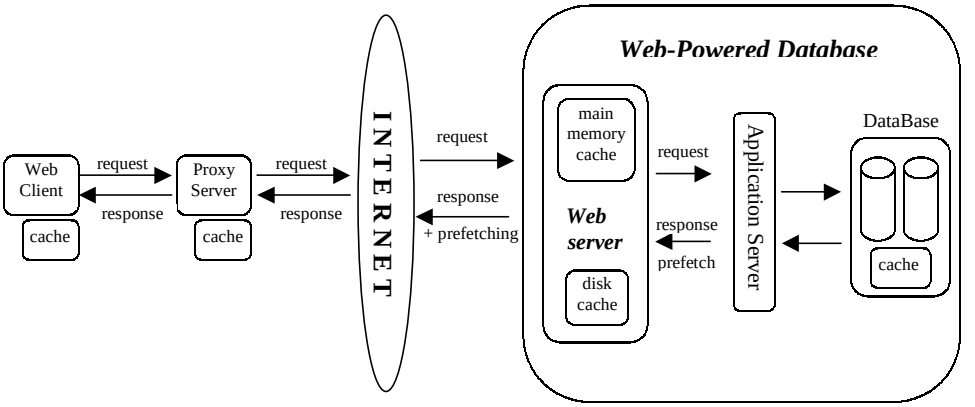
The popularity of the Web resulted in heavy traffic in the Internet and heavy load on Web the servers. For Web-powered databases the situation is worsened by the fact that the application program must interact with the underlying database to retrieve the data. So, the net effect of this situation is network congestion, high client perceived latency, Web server overload and slow response times for Web servers. Fortunately the situation is not incurable due to the existence of *reference locality* in Web request streams. The *principle of locality* (Denning & Schwartz, 1972)

asserts that: (a) correlation between immediate past and immediate future references tends to be high, and (b) correlation between disjoint reference patterns tends to zero as the distance between them tends to infinity. Existence of reference locality is indicated by several studies (Almeida et al., 1996; Breslau, Cao, Fan, Phillips & Shenker, 1999).

There are two types of reference locality, namely *temporal* and *spatial* locality (Almeida et al., 1996). Temporal locality can be described using the *stack distance model*, as introduced in (Mattson, Gecsei, Slutz & Traiger, 1970). Existence of high temporal locality in a request stream results in a relatively small average stack distance and implies that recently accessed data are more likely to be referenced in the near future. Consider for example the following reference streams: AABCBCD and ABCDABC. They both have the same *popularity profile*[1] for each item. Evidently, the stack distance for the first stream is smaller than for the second stream. This can be deduced from the fact that the number of intervening references between any two references for the same item in the first stream is smaller than for the second stream. Thus, the first stream exhibits higher temporal locality than the second. Spatial locality on the other hand, characterizes correlated references for different data. Spatial locality in a stream can be established by comparing the total number of unique subsequences of the stream with the total number of subsequences that would be found in a random permutation of that stream. Existence of spatial locality in a stream implies that the number of such unique subsequences is smaller that the respective number of subsequences in a random permutation of the stream. Consider for example the following reference streams: ABCABC and ACBCAB. They both have the same *popularity profile* for each item. We can observe in the first stream that a reference to item B always follows a reference to item A and is followed by a reference to item C. This is not the case in the second stream and we cannot observe a similar rule for any other sequence of items.

Due to the existence of temporal locality we can exploit the technique of *caching*, that is, temporal storage of data closer to the consumer. Caching can save resources, i.e., network bandwidth, since fewer packets travel in the network, and

Figure 1: Architecture of a typical web-powered database



time, since we have faster response times. Caching can be implemented at various points along the path[2] of the flow of data from the repository to the final consumer. So, we may have caching at the DBMS itself, the Web server's memory or disk, at various points in the network (proxy caches (Luotonen & Altis, 1994)) or at the consumer's endpoint. Web proxies may cooperate so as to have several proxies to serve each other's misses (Rodriguez, Spanner & Biersack, 2001). All the caches present at various points comprise a *memory hierarchy*. The most important part of a cache is the mechanism that determines which data will be accommodated in the cache space and is referred to as the cache *admission/replacement policy*.

Caching introduces a complication: how to maintain cache contents fresh, that is, consistent with the original data residing in the repository. The issue of cache consistency is of particular interest for Web-powered databases, because their data are frequently updated by other applications running on top of the DBMS and thus the cached copies must be invalidated or refreshed.

Obviously, requests for "first-time accessed" data and "non-cacheable" data (containing personalized, authentication information, etc.) cannot benefit from caching. In these cases, due to the existence of spatial locality in request streams, we can exploit the technique of *preloading* or *prefetching,* which acts complementary to caching. Prefetching deduces future requests for data and brings that data in cache before an explicit request is made for them. Prefetching may increase the amount of traveling data, but on the other hand can significantly reduce the latency associated with every request.

Contributions

This chapter will provide information concerning the management of Web caches. It intends by no means to provide a survey of Web caching. Such a survey, although from a very different perspective, can be found in (Wang, 1999). The target of the chapter is *twofold*. Firstly, it intends to clarify the *particularities* of the Web environment that call for different solutions regarding the replacement policies, cache coherence and prefetching in the context of the *Web-powered databases.* It will demonstrate how these particularities made the old solutions (adopted in traditional database and operating systems) inadequate for the Web and how they motivated the evolution of new methods. Examples of this evolution regard the replacement, coherence and prefetching techniques for the Web. The second objective of the chapter is to present a taxonomy of the techniques proposed so far and to sketch the most important algorithms belonging to each category. Through this taxonomy, which goes from the simplest to the most sophisticated algorithms, the chapter intends to clearly demonstrate the *tradeoffs* involved and show how each category deals with them. The demonstration of some popular representative algorithms of each category intends to show how the tradeoffs affect the complexity in implementation of the algorithms and how the ease of implementation can be compromised with the performance. Finally, another target of the present chapter is to present the practical issues of these topics through a description of how some popular commercial products deal with them.

The rest of the chapter is organized as follows. The Section "Background" provides some necessary background on the aforementioned topics and presents the peculiarities of the Web that make Web cache management vastly different from cache management in operating systems and centralized databases. Moreover, it will give a formulation for the Web caching problem as a combinatorial optimization problem and will define the performance measures used to characterize the cache efficiency in the Web. The Section "Replacement Policies" will present a taxonomy along with the most popular and efficient cache replacement policies proposed in the literature. The Section "Cache Coherence" will elaborate on the topic of how to maintain the cache contents consistent with the original data in the source and the Section "Prefetching" will deal with the issue of how to improve cache performance through the mechanism of prefetching. For the above topics, we will not elaborate on details of how these algorithms can be implemented in a real system, since each system provides its own interface and extensibility mechanisms. Our concern is to provide only a description of the algorithms and the tradeoffs involved in their operation. The Section "Web Caches in Commercial Products" will describe how two commercial products, a proxy cache and a Web-powered database, cope with cache replacement and coherency. Finally, the Section "Emerging and Future Trends" will provide a general description of the emerging *Content Distribution Networks* and will highlight some directions for future research.

BACKGROUND

The presence of caches in specific positions of a three-tier (or multi-tier, in general) architecture, like that presented earlier, can significantly improve the performance of the whole system. For example, a cache in the application server, which stores the "hot" data of the DBMS, can avoid the costly interaction with it. Similarly, data that do not change very frequently can be stored closer to the consumer e.g., in a proxy cache. But in order for a cache to be effective, it must be tuned so that it meets the requirements imposed by the specialized characteristics of the application it serves. The primary means for tuning a cache is the admission/replacement policy. This mechanism decides which data will enter the cache when a client requests them and which data already in cache will be purged out in order to make space for the incoming data when the available space is not sufficient. Sometimes these two policies are integrated and are simply called the *replacement policy*. The identification of the cached pages is based on the URL of the cached page (with any additional data following it, e.g., query string, the POST body of the documents)[3].

For database-backed Web applications, the issue of cache consistency is of crucial importance, especially for applications that must always serve fresh data (e.g., providers of stock prices, sports scores). Due to the requirements of *data freshness* we would expect that all dynamically generated pages (or at least, all pages with frequently changing data) be not cached at all. Indeed, this is the most

simple and effective approach to guarantee data freshness. But, it is far enough from being the most efficient. Due to the existence of temporal locality, some data get repeatedly requested and consequently arises the issue of redundant computation for their extraction from the database and their formatting in *HTML* or *XML*. We would like to avoid such repeated computation, because it places unnecessary load on the DBMS and the application server and does not use efficiently the existing resources, i.e., cache space in the Web/application server (or proxy, client). Moreover, the update frequency of some data may not be very high and consequently we can tolerate a not very frequent interaction with the DBMS to refresh them. In addition, recent studies show that even in the presence of high update frequency, materializing some pages into the Web server leads to better system performance (Labrinidis & Roussopoulos, 2000). The main issue in cache consistency is how to identify which Web pages are affected by changes to base data and consequently propagate the relevant changes. This issue concerns caches both inside the Web-powered databases and outside it (client, proxy). For caches inside the Web-powered database we have a second issue, the order in which the affected pages must be recomputed. Consider for example two dynamically generated pages. The first one is not very popular and is costly to refresh (due to the expensive predicate involved or the large volume of generated data) whereas the second one is popular and very fast to refresh. Obviously, recomputing the first page before the second one is not a wise choice, since it may compromise the accuracy of the published data or contribute to the access latency. In general, we need efficient and effective mechanisms to invalidate the contents of the cache or to refresh them.

The issues pertaining to the management of a cache (admission/replacement, coherency) are not new and there exists a rich literature in the field. They were examined in the context of operating systems (Tanenbaum, 1992) and databases (Korth, Silberschatz & Sudarshan, 1998), as well. But, the Web introduces some peculiarities not present in either the operating systems or databases.

1. First, the data in the Web are mostly for read-only purposes. Rarely, we have transactions in the Web that need to write the data back to the repository.
2. Second, we have variable costs associated with the retrieval of the data. Consider, for example, the case of a client who receives data from different Web servers. It is obvious that depending on the load of the servers and the network congestion it will take different time to download the data. As a second example, consider a client retrieving data from the same Web server but it requires different processing to generate them.
3. Third, the Web data are of varying sizes whereas in the databases and operating systems the blocks that move through the levels of memory hierarchy are of constant size, which equals the size of a disk block.
4. Fourth, the access streams seen by a Web server are the union of the requests coming sometimes from a few thousands Web clients and not from a few programmed sources as happens in the case of virtual memory paging systems.

These characteristics call for different solutions concerning Web cache management. Consequently, an effective caching scheme should take into account

the aforementioned factors, that is, recency of reference, frequency of reference, size, freshness, downloading time. Depending on the point in the *data flow path* where caching is implemented, some of these factors may be more important than the others or may not be present at all.

The benefits reaped due to caching can be limited (Kroeger, Long & Mogul, 1997). Obviously, caching is worthless for first-time accessed data. Also, caching is useless for data that are invalidated very frequently due to changes in the repository and there is a need for immediate propagation of the updates that take place in the repository and affect cached data. Moreover, for Web-powered databases which export base data in the form of Web pages (called *WebViews* in (Labrinidis & Roussopoulos, 2000)), it is preferable to "materialize" the WebViews at the Web server and constantly update them in the background when changes to base data occur (Labrinidis & Roussopoulos, 2000).

In general, the drawbacks of caching originate from caching's *reactive* nature. Caching attempts to cure a "pathological" situation, in our case the performance inefficiency, after it has been seen for the first time. Thus, in situations when caching is of limited usefulness, a *proactive* mechanism is needed, a mechanism that will take some actions in advance in order to prevent system's performance deterioration. This mechanism is prefetching, which is the process of deducing future accesses for data and bringing those data into the cache in advance, before an explicit request is made for that data.

Prefetching has also been extensively examined in the context of operating systems and databases. In general, there exist two prefetching approaches. Either the client will inform the system about its future requirements (Patterson, Gibson, Ginting, Stodolsky & Zelenka, 1995; Cao, Felten, Karlin & Li, 1996) or, in a more automated manner and transparently to the client, the system will make predictions based on the sequence of the client's past references (Curewitz, Krishnan & Vitter, 1993). The first approach is characterized as *informed* prefetching, where the application discloses its exact future requests and the system is responsible for bringing the respective objects into the cache. The second approach is characterized as *predictive* prefetching, where the system makes predictions based on the history of requests for data. Finally, there is another technique, termed *source anticipation*, that is, the prefetching of all (or some part thereof) of the embedded links of the document.

In the design of a prefetching scheme for the Web, its specialties must be taken into account. Two characteristics seem to heavily affect such a design: a) the client server paradigm of computing the Web implements, b) its hypertextual nature. Therefore, informed prefetching seems inapplicable in the Web, since a user does not know in advance its future requirements, due to the "navigation" from page to page by following the hypertext links. *Source anticipation* (Klemm, 1999), may work in some cases, but seems inappropriate in general, because there is no apriori information about which of a large set of embedded links the client is likely to request. On the other hand, predictive prefetching seems more viable, especially under the

assumption that there is sufficient spatial locality in client requests, because such a prefetching method could use the history of requests to make predictions.

Problem Formulation

Let us now provide a formal statement for the problem of cache management for a Web-powered database. Let O be the set of all data objects that may be requested in any instance of the time during cache operation. For each object $d \in O$ there a positive size s_d and a positive cost c_d associated with it. The cost c_d is a function F of the following parameters: size s_d, recency of reference r_d, frequency of reference f_d and freshness a_d, that is, $F=F(s_d, r_d, f_d, a_d)$. A request sequence is a function $\Sigma:[1..m] \rightarrow O$ and will be denoted as $\sigma=\sigma_1, \sigma_2, \ldots, \sigma_m$. The set of all possible request sequences will be denoted as R. When no cache is used, the cost in servicing sequence σ is $C(\sigma)=\sum_{k=1}^{m} c_{\sigma k}$. Let the cache size be X. We will assume that $s_d \leq X$, $\forall\ 1 \leq d \leq m$, that is, no data object is larger than the cache. We define the *cache state* S_k at time k to be the set of objects contained in the cache at that time.

Definition 1 (The Generalized Caching Problem (Hosseini-Khayat, 2000)) Let $\sigma=\sigma_1, \sigma_2, \ldots, \sigma_m$ be a sequence of requests resulting in a sequence of cache states $S_0, S_1, \ldots, S_m$ such that, for all S_k, $k=1,2,\ldots,m$

$$S_k = \begin{cases} (S_{k-1} - E_k) \cup \{\sigma_k\} & \text{if } \sigma_k \notin S_{k-1} \\ S_{k-1} & \text{if } \sigma_k \in S_{k-1} \end{cases} \quad (1)$$

where $E_k \subset \Sigma_{k-1}$ denotes the set of items purged out of the cache. Find among all state sequences satisfying Equation (1), a state sequence that minimizes the cost function

$$F((S_k), \sigma) = \sum_{k=1}^{m} \delta_k c_{\sigma k}$$

where

$$\delta'_k = \begin{cases} 0 & \text{if } \sigma_k \in S_{k-1} \\ 1 & \text{if } \sigma_k \notin S_{k-1} \end{cases}$$

and (S_k) denotes the state sequence.

This problem can be viewed both as off-line and on-line depending on how requests are presented to the algorithm. When the request sequence is apriori known then the problem is off-line, otherwise it is on-line. The off-line case of this problem with equals costs and sizes for each object has an optimal solution due to Belady (Belady, 1966) and is the *LFD* [4] algorithm, which evicts from cache the page whose next reference is furthest in the future. But, non-uniformity in costs and sizes introduces complications and *LFD* is not optimal any more as shown in the next two examples.

Example 1 (Varying Costs). Let us consider the following four unit size

objects {1,2,3,4} with associated costs {1,3,10,2}. Consider the request sequence σ = 1,2,3,4,2,3,4 and a cache with size X = 2. Then, *LFD* produces the following states: ø, {1}, {1,2}, {3,2}, {4,2}, {4,2}, {4,3}, {4,3} and incurs a cost of 26. Another algorithm, called *CIFD* [5] which evicts the page with the smallest value of $\frac{Cost}{ForwardDistance}$ produces the following states ø, {1}, {1,2}, {3,2}, {3,4}, {2,3}, {2,3}, {4,3} and incurs a cost of 21.

Example 2 (Varying sizes). Let us consider the following four unit cost objects {1,2,3,4} with associated sizes {1,1,2,2}. Consider the request sequence σ = 1,2,3,4,3,1,2,3,4,2 and a cache with size X = 4. Then, *LFD* produces the following states: ø, {1}, {1,2}, {1,2,3}, {4,3}, {4,3}, {1,3}, {1,2,3}, {1,2,4}, {1,2,4} and incurs a cost of 7. Another algorithm, called *Size* [6], which evicts the largest size page produces the following states ø, {1}, {1,2}, {1,2,3}, {1,2,4}, {1,2,3}, {1,2,3}, {1,2,3}, {1,2,3}, {1,2,4} and incurs a cost of 6.

The optimal solution for the generalized caching problem was presented in (Hosseini-Khayat, 2000) which is proven to be in *NP*. Any solution for the on-line version of the problem must rely on past information in order to make replacement decisions. If some probabilistic information regarding the requests is known, then we can derive some optimal solutions for the on-line problem, as well (see (Hosseini-Khayat, 1997)). But, in practice any solution for the problem (optimal or suboptimal) must be *practical*, that is, it must perform acceptably well, it must be easy to implement and should run fast, without using sophisticated data structures and in addition should take into account the peculiarities of the Web.

Performance Measures

Depending on the specific factors that we want to consider in the design of a caching policy, (recency, frequency of reference, consistency, size, downloading latency, etc.) we can modify the above definition appropriately and express it as an optimization problem.

The most commonly used performance measures used to characterize the efficiency of a caching algorithm are the hit ratio, byte hit ratio and delay savings ratio (Shim, Scheuermann & Vingralek, 1999).

Let D be the set of objects in a cache in a time instance. Let r_d be the total references for object d, cr_d the number of references for object d satisfied by the cache. Let s_d be the size of the object d and g_d be the average delay incurred while obtaining it.

Definition 2 (Hit Ratio). **The hit ratio of a cache is the percentage of requests satisfied by the cache:**

$$\sum_{d \in D} cr_d \Big/ \sum_{d \in D} r_d$$

In essence, improving the hit ratio is equivalent to reducing the average latency seen by a client.

Definition 3 (Byte Hit Ratio). The byte hit ratio (or weighted hit ratio) of a cache is the percentage of bytes satisfied by the cache. That is,

$$\sum_{d \in D} s_d * cr_d \Big/ \sum_{d \in D} s_d * r_d$$

In essence, improving the byte hit ratio is equivalent to reducing the average traffic of data from the source to the consumer.

Definition 4 (Delay Savings Ratio). The delay savings ratio determines the fraction of communication delays which is saved by satisfying requests from cache. That is,

$$\sum_{d \in D} g_d * cr_d \Big/ \sum_{d \in D} g_d * r_d$$

DSR is closely related to *BHR*. The latter can be seen as an approximation for the former, where the delay to obtain an object is approximated by its size.

REPLACEMENT POLICIES

Replacement algorithms deal with the problem of the limited cache space. They try to keep in cache the most "valuable" data. The "value" of a datum is usually a function of several parameters, say recency, access frequency, size, retrieval cost, frequency of update etc. The replacement policy makes replacement decisions based on this value. A good replacement strategy should be able to achieve a good balance among all of them and at the same time to "weigh" differently the most important of them. An additional requirement for the policy is the ability to easily implement it without the need of maintaining sophisticated data structures.

It is almost impossible to categorize all the Web cache replacement algorithms proposed so far into categories with distinguishable boundaries. Nevertheless, following the proposal of (Aggrawal, Wolf & Yu, 1999), we will catagorize the algorithms into three main categories, namely a) *traditional policies and direct extensions of them*, b) *key-based* policies and finally c) *function-based* policies. Moreover, for the last category, which includes the majority of the proposed algorithms, we will further discriminate them based on whether they are based on *LRU* or *LFU* or incorporate both.

Traditional policies and direct extensions of them. The algorithms belonging to the first category comprise direct application of policies proposed in the context of operating systems and databases or modifications of them to take into account the variable size of documents. *LRU* replaces the object, which was least

recently referenced. This is the most popular algorithm used today and capitalizes on temporal locality. Its simplicity stems from the fact that in order to make replacement decisions it only needs to maintain a heap with the IDs of the cached objects. Its overhead is $O(n)$ in space (n is the number of cached objects) and $O(1)$ time per access. This is the main reason for its use by many commercial products. An extension to *LRU* is *LRU-K* (O'Neil, O'Neil & Weikum, 1993), which replaces the document whose k-th reference is furthest in the past. *FIFO* replaces the objects, which entered first in the cache. *LFU* replaces the object with the least number of references. A variant of the *LFU*, the *LFU-Aging* policy (Robinson & Devarakoda, 1990) considers both the object's access frequency and its age in the cache (the recency of last access). Both *LRU* and *LFU* optimize the byte-hit ratio.

These policies were all examined in a recent study (Abrams et al., 1996) and they were found to be inadequate for Web caching. The primary reason is that they fail to take into account the variable size of Web objects. Object's size can have a dramatic effect on cache's performance, as it has already been shown its effect on the Belady's optimal algorithm in Example 2. In alleviating this drawback for the *LRU*, *LRU-THOLD* was proposed. *LRU-THOLD* is a variant of *LRU* that avoids the situation in which a document that is large compared to the cache size causes the replacement of a large number of smaller documents. This policy is identical to *LRU*, except that no document larger than a threshold size is cached. (Even if the cache has room, a document whose size is larger than the threshold is never cached.) A policy tailored for Web objects is the *SIZE* policy (Abrams et al., 1996), which replaces the largest object in the cache. *SIZE* aims at improving the hit ratio, since it favors small objects.

Key-based policies. All the above policies suffer from the drawback that they use only a simple characteristic of the cached objects in order to make replacement decisions e.g., *LRU* uses recency, *SIZE* uses size, etc. In alleviating this drawback, *key-based* policies use a couple or more "keys" to obtain the objects in sorted order of their "utility". The recency of reference, the size, the frequency of reference, etc. can be used as keys by these policies. One of them is selected as primary key, another as secondary key, etc. As replacement victim is selected the object with the least (greatest) value of the primary key. Ties are broken using a secondary key, then using a tertiary key and so on.

A representative algorithm of this category is the *LRU-MIN* (Abrams et al., 1996), which is a variant of *LRU* that tries to minimize the number of documents replaced. Let s_d be the size of the incoming object d, which does not fit in the cache. If there are objects in the cache with size at least s_d, then *LRU-MIN* removes the least recently used such object. If there are no such objects, then starts removing objects in LRU order of size $s_d/2$, then objects of size $s_d/4$, and so on until enough free space has been created. $LOG_2(SIZE)$ (Abrams et al., 1996) is another key-based policy which uses the $\log_2(size)$ as primary key and time since last access as secondary key. *Hyper-G* (Abrams et al., 1996) is another algorithm, which uses frequency of reference as primary key, recency of reference as secondary key and object's size as tertiary key.

It is evident that key-based policies prioritize some factors over others. This may not be always correct. In essence, traditional policies and key-based policies fail to integrate all the relevant factors into a single value that characterizes the utility of keeping an object into the cache. For example, with *LRU*, the cache can be populated with objects referenced only once purging out documents with higher probability of reference. Consequently, *LFU* would seem more appropriate, but *LFU* prevents popular (in the past) "dead"[7] documents from being evicted from cache and needs an "aging" mechanism to avoid "cache pollution". Such a mechanism requires fine-tuning of several parameters and thus it is difficult to implement. *SIZE* performs well with respect to hit ratio, but it is the worst policy when optimizing byte hit ratio (Abrams et al., 1996) for which *LFU* is the best policy. This is exactly what *function-based* policies do. They assign a utility value to each cached object, which is a function of various factors, such as recency, size, retrieval cost, etc.

Function-based policies. Function-based policies assign to every object in the cache a value, "utility value", which characterizes the benefit of retaining this object in the cache. This "utility value" is a function of several parameters, such as recency of reference, frequency of reference, size, retrieval cost, etc. Replacement decisions are made using this "utility value".[8]

It is not possible to partition these policies into disjoint groups, because they incorporate into their "utility value" different subsets of the set of parameters mentioned above. Nevertheless, we choose to categorize them into three groups, based on whether they capture temporal locality (recency of reference), popularity (frequency of reference) or both. Thus in the first category we have the *function-based policies extending LRU*, in the second category the *function-based policies extending LFU* and in the third, the *function-based policies integrating LRU-LFU*.

Function-based policies extending LRU. The common characteristic of this family of algorithms is that they extend the traditional *LRU* policy with size and retrieval cost considerations. Their target is to enhance the popular *LRU* algorithm with factors that account for the special features of the Web. The most important of them are the *GreedyDual-Size* (Cao & Irani, 1997) and *Size-Adjusted LRU (SLRU)* (Aggrawal, Wolf & Yu, 1999). Both can be used for browser, proxy or Web server caching, as well.

GreedyDual-Size (GD-Size) (Cao & Irani, 1997). The *GreedyDual-Size* is an elegant algorithm based on the *GreedyDual* (Young, 1994) that combines gracefully recency of reference with retrieval cost c_d and size s_d of an object. The "utility value" associated with an object that enters the cache is:

$$\mathrm{UV}_{\text{GD-Size}} = \frac{c_d}{s_d}.$$

When replacement is needed, the object with the lowest $UV_{GD\text{-}Size}$ is replaced. Upon replacement, the $UV_{GD\text{-}Size}$ values of all objects are decreased at an amount equal to the $UV_{GD\text{-}Size}$ value of the replacement victim. Upon re-reference of an object d its $UV_{GD\text{-}Size}$ value is restored to c_d/s_d. Thus, the $UV_{GD\text{-}Size}$ for a cached object

grows and reduces dynamically upon re-references of the object and evictions of other objects.

Upon an eviction, the algorithm requires as many subtractions as is the number of objects in the cache. In order to avoid this, we can maintain an "inflation" value *L*, which is set to the "utility value" of the evicted object. Upon re-reference of an object d, instead of restoring its $UV_{GD\text{-}Size}$ to c_d/s_d, we offset this by *L*. Below we present the algorithmic form of *GreedyDual-Size* with the above modification.

Algorithm 1 (GreedyDual-Size (GD-Size) (Cao & Irani, 1997))
Initialize L ← 0
Process each request in turn. The current request is for document *d*.
(1). if *d* in cache
(2). UV(d) ← $L + c_d/s_d$.
(3). if *d* not in cache
(4). while there is not enough cache space for *d*
(5). Let $L \leftarrow min_{q \hat{I} cache}\ UV(q)$
(6). Evict *q* such that *UV(q) = L*.
(7). Bring *d* into cache and set $UV(d) = L+c_d/s_d$.

Depending on the cost measure we want to optimize, i.e., hit ratio, byte hit ratio we can set the retrieval cost c_d appropriately.

A nice characteristic of this algorithm is its on-line optimality. It has been proved that *GreedyDual-Size* is *k*-competitive, where *k* is the ratio of the cache size to the size of the smallest document. This means that for any sequence of accesses to documents with arbitrary costs and arbitrary sizes, the cost of cache misses under *GreedyDual-Size* is at most *k* times that under the offline optimal replacement algorithm. This ratio is the lowest achievable by any online replacement algorithm.

Size-Adjusted LRU (SLRU) (Aggrawal et al., 1999). SLRU (and its approximation, the *Pyramidal Selection Scheme (PSS)*) strives to incorporate size and cost considerations into *LRU* along with cache consistency issues. Usually, objects have an associated *Time-To-Live (TTL)* tag, attached by the generating source (e.g., Web server), or an *Expires* tag that defines the lifetime of the object. We can exploit this tag, when present, in order to decide the freshness of an object and incorporate this factor into the replacement policy.

Let us define the *dynamic frequency* of an object *d* to be $1/"T_{dk}$, where $"T_{dk}$ is the number of references that intervene between the last reference for the object *d* and the current *k*-th reference. Let α_d be the time between when the object was last accessed and the time it first entered the cache. Also, let β_d be the difference between the time the object expires and the time of its last access. Then, the *refresh overhead factor* r_d is defined as $min(1, \alpha_d/\beta_d)$. If the object has not been accessed before, then $r_d=1$. If we are not aware of object's expiration time then $r_d=0$. The algorithm assigns to each object in the cache a "utility value", which is equal to:

$$UV_{SLRU} = \frac{c_d * (1 - r_d)}{s_d * \Delta T_{dk}}.$$

Its goal it to minimize the sum of the "utility value" of the evicted objects. Let S_k be the cache content at the k-th reference. Let R be the amount of additional memory required to store an incoming object and consider the decision variable y_d which is 0 for an object, if we wish to retain it into the cache and 1 otherwise. Then, the problem of selecting which objects to evict from cache can be expressed as:

$$\text{Minimize } \sum_{d \in S_k} (y_d * c_d * (1 - r_d)) / \Delta T_{dk}$$

such that

$$\sum_{d \in Sk} (s_d * y_d) \geq R \qquad \text{and} \qquad y_d \{0,1\}.$$

This is a version of the well-known *knapsack problem* where the items that we wish to store in the knapsack are those with the least "utility value". This problem is known to be in *NP* (Garey & Johnson, 1979). In practice, there exist fast and efficient heuristics. A greedy solution is to sort the objects in non-decreasing order of $(s_d * \Delta T_{dk})/(c_d * (1-r_d))$ and keep purging from cache the objects with the highest index in this sorting.

Function-based policies extending LFU. It is well known (Coffman & Denning, 1973) that when[9] a) the requests are independent and have a fixed probability, and b) the pages have the same size, then the optimal replacement policy is to keep in cache those pages with the highest probability of reference. In other words, the best online algorithm under the Independent Reference Model is the *LFU.* Based on this and on the observation (Abrams et al., 1996) that frequency–based policies achieve very high byte hit rates, the *function-based policies extending LFU* enhance the traditional *LFU* algorithm with size and/or retrieval cost considerations. The *LFU with Dynamic Aging (LFU-DA)* (Dilley &Arlitt, 1999) extends the *LFU-Aging* with cost considerations, whereas the *HYBRID* algorithm incorporates size and cost considerations. It is interesting to present how *HYBRID* computes the "utility value" of each object in the cache, since it was the first that incorporated the factor of the downloading delay into the replacement decision.

HYBRID (Wooster & Abrams, 1997). Let a document d located at server s be of size s_d and has been requested f_d times since it entered the cache. Let the bandwidth to server s be b_s and the connection delay to s be c_s. Then, the utility value of each document in the cache is computed as follows:

$$UV_{HYBRID} = \frac{(c_s + W_b / b_s) * f_d^{W_f}}{s_d} \text{ , } W_f \text{ and } W_b \text{ are tunable constants.}$$

The replacement victim is the object with the lowest UV_{HYBRID} value.

Obviously this algorithm is highly parameterized. Constants W_b and W_f weigh the bandwidth and frequency respectively, whereas c_s and b_s can be computed from the time it took to connect to server s in the recent past. *HYBRID* may not be used only for proxy caches, but for caches inside the Web-powered database as well. For such caches the first factor of the nominator can be replaced by a factor determining the cost to generate a Web page.

This family of function-based policies does not include many members, since it does not appear to be a wise choice to ignore the temporal locality in the design of a replacement policy. So the later efforts concentrated in extending other policies (like *GreedyDual-Size*) with frequency considerations.

Function-based policies integrating LRU-LFU. This category integrates into the replacement policy both recency and frequency of reference. As expected, these algorithms are more sophisticated than all the previous. The integration of recency and frequency with size and cost considerations results on the one hand in improved performance and on the other hand in having many tunable parameters. As example algorithms of this category we present two algorithms, namely the *Lowest Relative Value (LRV)* and the *Least Normalized Cost Replacement for the Web with Updates (LNC-R-W3-U)*.

Lowest Relative Value (LRV) (Rizzo & Vicisano, 2000). A replacement algorithm tailored for proxy caches and evaluating statistical information is the *Least Relative Value (LRV).* Statistical analysis of several traces showed that the probability of access for an object increases with the number of previous accesses for it and also, that the time since the last access is a very important factor and so is the size of an object. All, these parameters have been incorporated into the design of *LRV*, which strives to estimate the probability of reaccessing an object as a function of the time of its last access, its size and the number of past references for it.

The utility value of each cached document which has been accessed *i* times in the past is computed as follows:

$$\mathrm{UV}_{\mathrm{LRV}} = \frac{c_d}{s_d} * P_r(i_d, t_d, s_d)$$

where P_r is the probability of re-reference for a document *d*, given that it has already been referenced *i* times, evaluated at the time of its last access *t*. The dependence of $P_r(i_d, t_d, s_d)$ on these three parameters has been verified with statistical analysis of several real Web traces. We explain in the next paragraph how P_r can be derived.

Let $\varphi(t)$ and $\Phi(t)$ denote the probability density function and the cumulative distribution function of the time between consecutive requests to the same document. Statistical information obtained from real Web traces shows that $\varphi(t)$ is *approximately independent* from the number of previous accesses to a document. $\varphi(t)$ is also the pdf of the next access time conditioned to the fact that the document gets requested again. Let *Z* denote the probability that a document gets re-referenced, evaluated at the time of the previous access. Assuming the independence stated above, then: a) $Z=Z(i_d, s_d)$ and b) the pdf of the next access time can be expressed as $Z*\varphi(t)$, and thus P_r can be computed as:

$$P_r = \int_{\omega=t}^{\infty} Z * \varphi(\omega) * d\omega = Z * (1 - \Phi(t)) .$$

Now, the function $Z=Z(i_d, s_d)$ which describes the dependence of P_r on the number of references to a document and on its size s_d must be computed. *LRV*

neglects the dependence of Z on s_d for documents that have been referenced more than once. Thus:

$$Z(i_d, s_d) = \begin{cases} M(i_d, s_d) & \text{if } i_d = 1, \\ N(i_d) & \text{otherwise.} \end{cases}$$

The function $N(i_d)$ can be computed as:

$N(i_d) = \dfrac{\| D_{i+1} \|}{\| D_i \|}$, where $\|D_i\|$ is the number of documents accessed i times.

The function $M(i_d, s_d)$ can be computed from statistical information gathered from request streams. $M(i_d, s_d)$ is not estimated for every distinct value of s_d, but for groups[10] of values instead.

Now we turn to the function $\Phi(t)$. The function $\Phi(t)$ cannot be computed analytically. Thus we must rely on statistical information available on request streams in order to derive an approximation $\tilde{\Phi}(t)$ for it. Such an approximation is the following:

$$\tilde{\Phi}(t) = c * \log\left(\frac{f(t) + \gamma_1}{\gamma_1}\right), \quad \text{where} \quad f(t) = \gamma_2 * (1 - e^{\frac{t}{\gamma_2}}) \cdot$$

where γ_1, γ_2 are constants in the range 10..100 and $>5*10^5$, respectively.[11]

Thus, the probability P_r of re-reference for an object can be estimated as:

$$P_r(i_d, t_d, s_d) = \begin{cases} M(1, s_d)(1 - \tilde{\Phi}(t)) & \text{if } i = 1, \\ N(i_d)(1 - \tilde{\Phi}(t)) & \text{otherwise.} \end{cases}$$

Apparently, *LRV* has many parameters that need to be tuned appropriately. This means additional cost and complexity, but *LRV* can make more clever replacement decisions since it considers more information regarding requests streams. This is a fundamental tradeoff: the more information we use, the more efficient our processing is, but this efficiency comes at increased computation cost.

Least Normalized Cost Replacement for the Web with Updates (*LNC-R-W3-U*) (Shim, Scheuermann & Vingralek, 1999). The *LNC-R-W3-U* algorithm takes into account, in addition to other factors, the cost to validate an expired object. Its target is to minimize response time rather that the hit ratio and consequently it attempts to minimize the delay savings ratio in which it incorporates the cost to validate an "expired" object.

Let r_d be the average reference rate for object d, g_d the mean delay to fetch it into the cache, u_d the average validation rate and vc_d the average delay to perform a validation check. Then, the "utility value" for object d is defined as:

$$UV_{LNC\text{-}R\text{-}W3\text{-}U} = \frac{r_d * g_d - u_d * vc_d}{s_d}.$$

Using a greedy heuristic, as in the case of *SLRU*, we can select replacement victims. The major issue for this algorithm is that it has many parameters r_d, g_d, u_d, vc_d which are difficult to compute. We can compute g_d and vc_d using a weighted sum of their latest values and their past values as follows:

$$vc_d^{new} = (1-\mu) * vc_d^{old} + \mu * vc_{sample}$$

$$g_{di}^{new} = (1-\mu) * g_d^{old} + \mu * g_{sample}$$

where vc_{sample} and g_{sample} are the most recently measured values of the respective delays and ¼ is a constant that "weighs" the recent with the old measurements.

The mean reference rate and mean validation rate can be computed using a sliding window of K most recent reference times as:

$$r_d = \frac{K}{t - t_K}$$

where t is the current time and t_K is the time of the oldest reference in the sliding window.

The mean validation rate can be computed from a sliding window of last K distinct Last-Modified timestamps[12] as:

$$u_d = \frac{K}{t_r - t_{uK}}$$

where t_r is the time when the latest version of object d was received by the cache and tu_K is the K-th most recent distinct Last-Modified timestamp of object d (i.e., the oldest available distinct Last-Modified). If fewer than K samples are available, then K is set to the maximal number of available samples.

Apart from *LRV* and *LNC-R-W3-U,* there exist quite a lot of algorithms in this family. Some of them belong to the family of *GreedyDual-Size*, in the sense that they incorporate frequency of reference considerations into the original algorithm (see Dilley & Arlitt, 1999; Jin & Bestavros, 2001). Some others are more adhoc in the sense that the "utility value" they define for a document is a ratio relating frequency, recency, cost, in a non uniform manner (e.g., exponential) (Niclausse, Liu & Nain, 1998).

Discussion

In this section, we have presented a categorization of cache replacement algorithms for Web objects. We have also presented the most important algorithms belonging to each category. Some of them are relatively simple, whereas some others are more sophisticated. There is no clear "champion" algorithm, which performs best in all cases. As we will see in the section, which discusses replacement issues for some major commercial products, the tendency is to make replacement decisions considering only expiration times and recency of reference. This is because these factors are easy to handle and do not impose high load on the system in order to make replacement decisions. Moreover, they do not require complicated data structures for the maintenance of the metadata associated with cached objects.

CACHE COHERENCE

Web-powered databases generate dynamic data with sometimes high update frequencies. This makes the issue of cache consistency (or cache coherence) critical. The purpose of a cache consistency mechanism is to ensure that cached data are eventually updated to reflect the changes to the original data. Caches can provide either *strong* or *weak* consistency. The former form of consistency guarantees that stale data are never returned to the clients. Thus, the behavior of the whole system (Web-powered database and client applications) is equivalent to there being only a single (uncached) copy of the data, except from the performance benefits of the cache. Weak consistency allows served copies to diverge temporarily from the copy in the origin server. In other words, caches providing weak consistency may not return to the client the result of the last "write" operation to a datum.

The coherency requirements associated with Web objects depend in general on the nature of the objects and the client's tolerance. As a concrete example consider a cache that serves stock prices, sports and weather information. This particular cache will usually be forced to provide strong consistency for stock prices because of the stringent client requirements, whereas it may provide weak consistency for sports and weather data.

Cache consistency can be achieved through either *client-driven* or *server-driven* protocols.[13] In the former, the cache is responsible for contacting the source of original data in order to check the consistency of its cached data. In the latter, the data source is responsible for notifying the caches, which store its data, for any committed "writes". These two options are the two extremes in the spectrum of possible alternatives. They represent the fundamental trade-off in cache coherency: client caches know when their data are requested, but they do not know when they are modified[14]. On the other hand, servers have complete knowledge of "writes" on data, but they do not know which clients that they have ever requested any of their data, are still interested in them.

Consequently, these two approaches differ in the burden they impose on the server and the network, and on the "read" and "write" performance. The former approach may impose heavy or unnecessary load on servers due to many validating messages for unchanged resources. The latter requires the server to maintain client state, keeping information about which client caches which objects (Cao & Liu, 1998). Moreover, these approaches have different resilience to failures. Network partition, for example, can prevent the client from receiving invalidation information and thus to use stale data. Another complication for server-driven protocols is how to identify their clients. This can be solved with the use of tokens, called "*cookies*". Cookies are encoded strings of data generated by the Web server and stored on the client. There are two types of cookies: persistent cookies and session cookies. Persistent cookies are usually stored in a text file on the client and may live indefinitely. Session cookies usually reside in the memory space of the client and typically are set to expire after a period of time determined by the Web server. For the "cookie-phobic" clients, who do not permit cookies to be stored on their machine,

many Web sites simply embed cookies as parameters in the URLs. This is typically done by inserting (or appending) a unique sequential number, called session ID, into all the <a href=…> links in the site's *HTML* code[15].

An amalgamation of these two extreme approaches, called *Leases* (Gray & Cheriton, 1989), is to have the server inform the client about "writes" for a specified period of time, and after that time, the clients are responsible for validating their data. This approach tries to combine the advantages of the aforementioned extreme solutions by fine-tuning the period for which the server notifies about changes in data.

Cache consistency has been studied extensively in computer architecture, distributed shared memory, network and distributed file systems (Howard et al., 1988; Nelson, Welch & Ousterhout, 1988) and distributed database systems (Franklin, Carey & Livny, 1997). In the Web, data are mostly for read only purposes, so many of the approaches proposed in the context of distributed databases and distributed shared memory systems cannot be applied in the Web. The most relative fields are that of network and distributed file systems, but the challenge in the Web environment is to make the solutions proposed in these fields to scale to the large number of clients, the low bandwidth of the Internet, the frequent failures (client, server) and the network partitions. In Subsection "Cache coherence maintenance", we will present the different approaches for cache consistency maintenance, categorized as either client or server-driven.

For Web-powered databases, since served objects (*HTML-XML* fragments (Challenger, Iyengar & Dantzig, 1999; Yagoub et al., 2000), *HMTL-XML* pages) are derived from the data residing in the underlying database, arises the need to maintain the dependencies between base data and derived data, so as to be able to identify stale objects when changes to base data take place. Thus, the issue of *object change detection* is also very important. In Subsection "Object change detection" we will present an approach for the detection of which objects are affected by changes to base data.

Cache Coherence Maintenance

Client-driven protocols. Client-driven protocols rely on the client to validate the contents of its cache before returning them as a hit. The simplest approach, called *poll-each-read* (Yin, Alvisi, Dahlin & Lin, 1999; Cao & Liu, 1998), is to send a validating message to the respective server every time the data are requested, in order to confirm that the data are fresh or in the opposite case, to retrieve the modified ones. The primary advantage of this approach is that it never returns stale data to clients without the client knowing it.[16] But this protocol imposes high load on servers due to the very many validating messages received by clients, high network traffic due to many control messages traveling in the network and unnecessary client latency, in case the data are not modified, since every "read" request must be preceded by a contact to the server. Consequently this policy suffers from poor "read" performance, but on the other hand, is very efficient for "writes" since they proceed immediately, without any invalidation

procedure from the server to the clients. In other words, this policy is ideal when there are very few "reads" and many "writes".

Trying to reduce the communication messages for data validation and read latency, another policy based on *poll-each-read*, namely *poll*, assumes that the data remain valid for a specific period of time after their latest validation. If this time interval is assumed to be zero, then *poll* reduces to *poll-each-read*. This policy cannot guarantee strong consistency. The challenge is to determine an appropriate value for this time interval. By choosing a small value for this interval, we can reduce the percentage of stale objects returned to the clients, but the number of validating messages to the server increases significantly. The server can associate with every datum a value for this interval, *Time-To-Live (TTL)* (Cate, 1992) or *Expires*, or the client can calculate a value by its own for every data it caches (*adaptive-TTL*) (Cao & Liu, 1998).

Server-driven protocols. On the other extreme of spectrum, lies the protocol that has the servers to notify the clients for any changes on data. This policy is called *Invalidation* or *Callback* (Howard et al., 1988; Nelson et al., 1988) and clearly, it requires the servers to maintain some information recording which clients cache which data. Before any "write" occurs, the server must send out *invalidation* messages to all clients caching these data. This policy guarantees strong consistency, within the time it is required for a message to be delivered to all clients. On the other hand, the maintained "state" information for clients can grow to unmanageable amount, when there are a lot of clients and a lot of requested resources. Moreover, it causes bursts of server activity for sending out the invalidation messages to all clients caching a copy of the modified object. Obviously, the "read" performance of this policy is very good, since cached data are guaranteed to be always fresh, but the "write" performance may become very bad in cases of network partitions, client failures and large number of clients, due to the communication overhead for delivering the invalidation messages. To state it simply, this policy is ideal in environments where there are a lot of reads and a few writes. Nevertheless, server-driven consistency has been shown to be very efficient in keeping the clients and the server *synchronized* and this efficiency comes at relatively little cost (Cao & Liu, 1998; Yin, Alvisi, Dahlin & Iyengar, 2001).

In between these two extremes lies the *Leases* protocol (Gray & Cheriton, 1989; Yin et al., 1999). A cache using *Leases* requires a valid lease, in addition to holding the datum, before returning that datum in response to a read request. A client holding a valid lease on an object is sure that no "write" on this object will proceed before it is being notified about it (or take his permission). When an object is fetched from the origin server, the server returns a lease guaranteeing that the object will not be modified during the lease *term*. After the lease expires, a read of the object requires that the cache extends first the lease on the object, updating the cache if the object has been modified since the lease expiration.

When a "write" to an object must take place and an unreachable client (due to crash or network partition) holds a lease on this object, the server needs only to wait for the expiration of this lease before proceeding into the write. So, the *Leases*

protocol limits the starvation of "writes". Similarly, when the server crashes, it can restore the information on the leases it has granted if its has saved them on secondary storage or alternatively (due to the high I/O cost the former approach incurs) it can only wait for a period of time equal to the longest lease it has granted[17] before commit any writes. *Leases* present a trade-off similar to *Poll*; long lease terms reduce the cost of reads, amortizing lease renewal over many reads, but on the other hand, delay the "writes". Short lease terms present several advantages. First, they minimize the delay due to client or server failures. Second they reduce the storage requirements on the server and third they minimize the false-sharing (Gray & Cheriton, 1989) that occurs. False sharing occurs when the server needs to modify an object, for which a client not currently accessing that object holds a lease. Longer-term leases are more efficient for "hot" objects with relatively little write-sharing. The determination of the optimal *lease term* depends on a number of factors, such as object popularity, state-space overhead, control messages overhead, read/write frequency. Some analytical models addressing the issue of optimal lease duration were presented in (Duvvuri, Shenoy & Tewari, 2000).

Leases are efficient when the cost of the lease renewal is amortized over many reads. For the Web though, the interaccess time for an object may span several minutes degrading thus the performance benefits of the original *Leases*. So, *Volume Leases* were proposed in (Yin et al., 1999) to amortize the lease renewal cost over reads to many objects. Objects at the server are grouped into volumes. Usually a volume groups together related objects (e.g., objects that tend to be accessed together). A client can access an object in its cache when it holds a valid lease on both the object (*Object Lease*) and the volume to which this object belongs (*Volume Lease*). The server can modify an object as soon as either lease (*Object* or *Volume*) expires.

With a lease term equal to zero, the *Leases* is equivalent to *Poll-each-read*. *Leases* bears some similarity with the *TTL* approach, but the difference being that the former guarantees strong consistency.

Summary. In Table 1 we present the consistency maintenance protocols categorized along two dimensions; the first being the form of consistency they provide and the second being the part that initiates the consistency check.[18]

Object Change Detection

When caching dynamically generated objects, a key problem is to determine which objects become obsolete, when base data change. This is important for the scheduling of the invalidation messages that must be sent to the caches (or in general

Table 1: Categorization of consistency maintenance protocols

Protocol / Consistency	**Weak**	**Strong**
Client-driven	Poll, (adaptive-)TTL	Poll-each-read
Server-driven		Invalidation, Leases

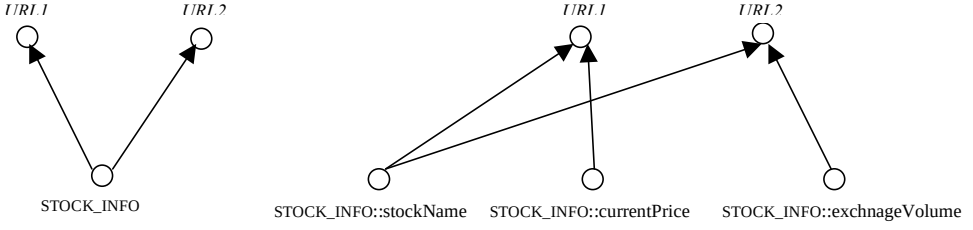
to invalidate a cached object) in order to maintain strong consistency. For the identification of the objects affected by changes to base data, caches must maintain information about the *dependencies* of cached objects on base data.

For this purpose, the *Object Dependence Graph (ODG)* and *Data Update Propagation (DUP)* were introduced in (Iyengar & Challenger, 1998; Challenger et al., 1999). *DUP* maintains the correspondence between *objects* (defined to be cacheable items) and *underlying data*, which change and affect the values of *objects*. Dependencies between objects and data are recorded in the *ODG.* A vertex in this graph corresponds to an object or a datum. An edge from a vertex *d* to a vertex *o* indicates that a change to *d* affects *o*. By transitivity, "hidden" dependencies exist in *ODG* as well. Using graph traversal algorithms (depth-first, breadth-first) we can determine the objects affected by changes to base data.

As a simple concrete example, consider a site serving stock related information, which uses a relational database consisting of a table *STOCK_INFO(stockName, currentPrice, exchangeVolume)* and two dynamically generated *HTML* pages (URLs), *URL1* publishing the pairs of *stockName* and *currentPrice* and a *URL2* publishing the pairs of *stockName* and *exchangeVolume*. We can represent the dependencies of these URLs on base data by an *ODG* as in Figure 2(a) or Figure 2(b). In the left part of the figure, there is only one datum (the base table) and two objects (*URL1* and *URL2*), whereas in the right part, there exist the same URLs, but three data (the three columns) of the base table. By appropriately choosing the data, we can control the coarseness of the dependencies. Based on the dependencies described in the left part of the figure, any modification on the table results in an invalidation of both objects, whereas the dependencies expressed in the right part of the figure allow us to prevent invalidation of *URL1* when occur changes only to the *exchangeVolume*.

Cache managers maintain directories containing information about cached objects. This information may include the *ODG* as well. Upon receiving a notification for a modified datum they can invalidate the objects depended on it. Obviously, the object dependencies are communicated to caches by the application programs that generate the objects. The invalidation events can be generated by trigger mecha-

Figure 2: Examples of the object dependence graph representing different coarseness dependencies between objects and base data



(a) (b)

nisms or specially crafted application scripts (Challenger et al., 1999). Of course, this presents a complication, since the DBMS, the application servers and caches may be independent components, but the success of the system in (Challenger et al., 1999), which served as the Web site for the Winter Olympic Games of 1998, demonstrates the feasibility of deploying similar solutions.

PREFETCHING

We pointed out earlier (see Section "Background") that caching has limitations. It is not useful for first-time referenced objects, and its usefulness decreases when objects tend to change frequently. To cure caching's reactive nature, *prefetching* has been proposed in order to enhance its content. Prefetching is the process of deducing future requests for objects and bringing those objects into the cache before an explicit request is made for them. Prefetching presents a fundamental tradeoff; the more objects are brought into the cache the greater is the probability of a cache hit, but so is the generated traffic. An effective and efficient prefetching scheme should maximize the number of hits due to its action and at the same time minimize the incurred cost due to the prefetched objects. This cost may represent cache space, bandwidth, etc.

The core issue in employing a prefetching scheme is the deduction of future requests. What is needed is a mechanism that will "suggest" objects to be prefetched from their origin location, the "server". In general, there exist two possibilities for the deduction of future references. Either there is complete knowledge about them or they must be predicted. The former is called *informed* and the latter *predictive* prefetching.

Informed prefetching occurs in cases where the client (e.g., an application program) knows exactly the resources is going to request in the near future and reveals them into the cache (Patterson et al., 1995; Cao et al., 1996). The latter is responsible for programming its caching and prefetching decisions in order to increase its performance. *Informed* prefetching is actually a scheduling policy subject to a set of constraints regarding cache space, timeliness of prefetching and available bandwidth. This model requires the communication bandwidth between the applications and the cache to be stable and thus can be implemented only in cases where the cache is somehow embedded into the application, e.g., databases, operating systems, etc.

Informed prefetching is completely exonerated from the burden of guessing which objects will be requested. Currently, such a situation where an application knows exactly its future requests is not frequent in the Web, because requests are not generated by a few programmed sources, as happens in the case of operating or database systems, but originate directly from Web clients (usually humans). Moreover, in the Web we cannot assume fixed bandwidth between the cache and the origin location of the data. We cannot assume this even for caches inside the Web-powered database, because their load depends on the external Web client requests.

Predictive Prefetching for the Web

In the Web, we need an alternative mechanism for deducing future references. The only possibility is to take advantage of the *spatial locality* present in Web request streams. Spatial locality captures the co-reference of some resources and it is revealed as "regularity patterns" in the request streams. Studies examining request streams in proxy and Web servers (Almeida et al., 1996) confirmed the existence of spatial locality. The remaining issue is to "quantify" spatial locality, that is, to discover the dependencies between references for different data. Such dependencies can be discovered from past requests for resources[19] and used for making predictions about future requests. The dependencies, which can be expressed as rules, drive prefetching decisions. In other words, they select which objects will be prefetched.

In what follows, we will describe the general form of a Markov predictor. In the sequel, we will present the three families into which the existing prefetching algorithms can be categorized and will also explain how they can be interpreted as Markov predictors. Their interpretation as Markov predictor is important in order to understand their differences and shortcomings.

Markov Predictors

Let $T_r = <\mathrm{tr}_1, \ldots, tr_k>$ be a sequence of consecutive requests for documents (called *transaction*) made by a client. Let also, $S=<\mathrm{d}_1, \ldots, d_n>$, $n \leq k$, be a sequence of accesses, which is a subsequence[20] of this transaction. Given a collection of client transactions, the probability $P(S)$ of an access sequence S is the normalized number of occurrences of S inside the collection of transactions. Thus, if the total number of transactions is nTr, and the sequence S appears $fr(S)$ times inside this collection, then $P(S)=fr(S)/nTR$.

Let $S = <d_1, \ldots, d_n>$ be a sequence of accesses. Then the conditional probability that the next accesses will be to $d_{n+1}, \ldots, d_{n+m}$ is $P(d_{n+1}, \ldots, d_{n+m} \mid d_1, \ldots, d_n)$. This probability is equal to:

$$P(d_{n+1}, \ldots, d_{n+m} \mid d_1, \ldots, d_n) = \frac{P(d_1, \ldots, d_n, d_{n+1}, \ldots, d_{n+m})}{P(d_1, \ldots, d_n)}$$

Therefore, given a collection of client transactions, rules of the form

$$d_1, \ldots, d_n \Rightarrow d_{n+1}, \ldots, d_{n+m} \qquad (2)$$

can be derived, where $P(d_{n+1}, \ldots, d_{n+m} \mid d_1, \ldots, d_n)$ is equal to or larger than a user-defined cut-off value T_c. $P(d_{n+1}, \ldots, d_{n+m} \mid d_1, \ldots, d_n)$ is the *confidence* of the rule. The left part of the rule is called the *head* and the right part is called the *body* of the rule.

The dependency of forthcoming accesses on past accesses defines a *Markov chain*. The number of past accesses considered in each rule for the calculation of the corresponding conditional probability is called the *order* of the rule. For instance, the order of the rule $A,B \Rightarrow C$ is 2.

Definition 5 (*n-m* Markov predictor). An *n-m* Markov predictor calculates conditional probabilities $P(d_{n+1}, \ldots, d_{n+m} \mid d_1, \ldots, d_n)$ between document accesses and

discovers rules of the form (2). The head of each rule has size equal to *n* and the body has maximum size equal to *m*.

A *predictive prefetching algorithm* can be defined as a collection of *1-m, 2-m,..., n-m* Markov predictors.

Below we present the three families into which existing predictive prefetching mechanisms for the Web can be categorized. The first two are adopted from the context of operating and database systems, whereas the third is particularly suited for the Web environment.

Dependency Graph (DG)

The algorithms belonging to this family are based on the concept of the D*ependency Graph (DG).* This concept was originally proposed in the context of operating systems (Griffioen & Appleton, 1994) and was later adopted in the Web (Padmanabhan & Mogul, 1996, Cohen, Krishnamurthy & Rexford, 1999; Jiang & Kleinrock, 1998).

The *DG* depicts the pattern of access to different objects. The graph has a node for each object that has ever been accessed. There is an arc from node X to node Y, if and only if at some point in time, Y was accessed within *w* accesses after X and both accesses were done by the same client[21]. The user-specified parameter *w* is called the *lookahead window*. We maintain the following information in the graph: a) the number of accesses to each node X and b) the number of transitions from node X to node Y. The confidence of a rule, say XÞY, is the ratio of the number of transitions from X to Y to the total number of access to X itself. Figure 3(a) depicts a *DG* constructed from two request sequences ABCACBD and CCABCBCA. The numbers on the arcs denote the number of transitions from the source to the target of the arc, whereas the numbers next to each node denote the number of accesses to each node.

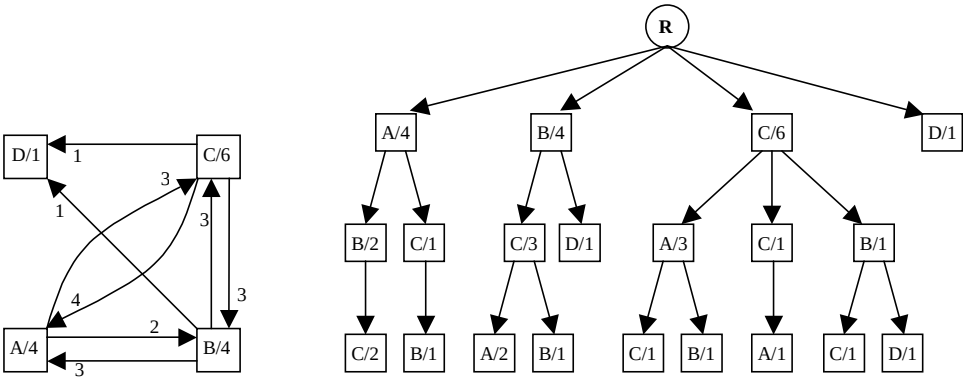
The *Dependency Graph (DG)* algorithm uses a *1-1* Markov predictor. It calculates conditional probabilities $P(d_i \mid d_j)$ for all d_i, d_j belonging to a transaction, provided that the number of intervening requests between d_i, and d_j does not exceed *w*. It maintains a set of rules of the form $d_i \Rightarrow d_j$. For a user who has accessed the sequence of documents $T_r = <tr_1, ..., tr_n>$, *DG* searches all rules with head tr_n and prefetches all documents *d* for which $tr_n \Rightarrow d$ is a rule.

Prediction by Partial Match (PPM)

The work by Curewitz, Krishnan and Vitter (Curewitz et al., 1993) identified the relation between compression schemes and prediction. Thus, they proposed the adoption of a well-known text compressor, namely the *Predictor by Partial Match*, in order to carry out predictive prefetching in object databases. Its usefulness was later investigated in the Web environment (Fan, Cao, Lin & Jacobson, 1999; Palpanas & Mendelzon, 1999; Chen, Park & Yu, 1998; Deshpande & Karypis, 2001).

The *PPM* scheme is based on the notion of a *k*-order *PPM* predictor. A *k*-order *PPM* predictor maintains *j-1 Markov* predictors, for all $1 \leq j \leq k$. It employs a Markov

Figure 3: (a) Dependence graph (lookahead window 2) and (b) Predictor by partial match for two request streams ABCACBD and CCABCBCA



(a) (b)

predictor, which has the constraint that the preceding *j* "events" must be consecutive in the request stream. A *PPM* predictor is depicted as a tree where each path or subpath that emanates from the root corresponds to a distinct sequence of consecutive requests. A *k*-order *PPM* can be constructed from a collection of transactions as follows: we move a "sliding window" of size *j* (for all *j=1,2,..,k+1*) over each transaction. For every sequence of accesses of length *j*, we either create a new path of length j in the tree initializing its associated counter to 1, or (in case such a path already exists) we simply increment by one the counter of the path. Figure 3(b) illustrates a 2nd order *PPM* predictor, where paths emanate from the tree root with maximum length equal to *k*+1=2+1 (=3). The counter associated with each node depicts the number of times this node was requested, after all nodes before it in the path, were requested. The counter for a child of the root depicts the total number of appearances of this node in the transactions.

The *k*-order *PPM* algorithm uses a collection of *1-1, 2-1, ..., k-1 Markov* predictors (*k* is a user-specified constant) with the additional constraint that accesses are consecutive. These Markov predictors calculate conditional probabilities of the form $P(d_{n+1} \mid d_n)$, $P(d_{n+1} \mid d_{n-1}, d_n)$, …, $P(d_{n+1} \mid d_{n-k+1}, ..., dp_n)$ and determine the corresponding rules, which have head sizes equal to 1,2,…,*k*, respectively.

In summary, we see that the following facts hold, see also (Nanopoulos, Katsaros & Manolopoulos, 2002):

- *DG* considers dependencies between pairs of references only (*first-order dependencies*). The considered references need not be consecutive.
- *PPM* considers dependencies not only between pairs of references (*higher-order dependencies*). The considered references must be consecutive.

These two facts highlight the inefficiency of *DG* and *PPM* schemes to address the requirements in the Web environment. Due to the hypertextual nature of the Web, the Web workloads exhibit *higher order dependencies* between references.

Higher order dependencies describe the fact that a future reference may depend not only on one specific reference made in the past, but also on a "longer history". Moreover, due to the navigational nature of information seeking in the Web, correlated references may not be consecutive. Thus, an effective predictive prefetching scheme should address these requirements.

Prefetching Based on Association Rules Mining–The WM_o *Algorithm*

In (Nanopoulos, Katsaros & Manolopoulos, 2001; Nanopoulos et al., 2002) we developed the algorithm WM_o to address the aforementioned requirements. WM_o is based on the association rules mining paradigm (Agrawal & Srikant, 1994).

The algorithms[22] for association rules discovery (e.g., the *Apriori* (Agrawal & Srikant, 1994)) process transactional databases and derive rules of the form (2). They work in several phases. In each phase, they make a pass over the database of transactions. In the *k*-th pass, they determine the *frequent*[23] *k*-itemsets and create the set of candidate (*k+1*)-itemsets. In the next pass, they determine the frequent *(k+1)*-itemsets, and so on. The frequent 1-itemsets are the frequent items appearing in the transactions database. After the discovery of all the frequent itemsets, they make one pass over the database in order to determine the association rules.

Apriori-like algorithms are not appropriate for deriving prefetching rules for the Web. Their shortcoming is that they do not take into account the ordering of the items inside a transaction. In (Nanopoulos et al., 2002) the WM_o algorithm was proposed and showed that generalizes the existing prefetching algorithms (the algorithms belonging to the family of *DG* and *PPM*). WM_o works like the standard *Apriori* algorithm, but has a different candidate generation procedure in order to address the particularities of the Web mentioned earlier.

In WM_o, unlike the standard *Apriori*, an itemset[24] is "supported" by a transaction if the itemset is a subsequence of the transaction. Recall that in the *Apriori*, an itemset is "supported" by a transaction if it is a subset of the transaction. Thus, the WM_o takes into account the ordering between the accesses in a transaction. This feature is very important for the purposes of Web prefetching, because a rule like $A \triangleright B$ is apparently different than a rule $B \triangleright A$. The WM_o algorithm is able to produce both rules (if they exist), whereas the *Apriori* would have produced only one of them. WM_o achieves this by adopting a different candidate generation procedure which works as follows: let two frequent (*k*-1) itemsets be $L_1=<p_1,\ldots,p_{k-1}>$ and $L_2=<q_1,\ldots,q_{k-1}>$. If $p_i=q_i$ for all $i=1,2\ldots,k-2$, then WM_o produces both candidates $C_1=<p_1,\ldots,p_{k-1},q_{k-1}>$ and $C_2=<q_1,\ldots,q_{k-1},p_{k-1}>$ Whereas the *Apriori* would have produced only the first of them.

Due to its *Apriori*-like nature, the WM_o algorithm is able to produce rules of the form (2). It is obvious also, that WM_o uses a collection of *1-m, 2-m, …, k-m Markov* predictors (*k* is determined by the data and not prespecified as in *PPM*) without the constraint that accesses should be consecutive. Thus, WM_o addresses the requirements of the Web environment. In (Nanopoulos et al. 2001; Nanopoulos et al. 2002), extensive experiments are presented that confirm the superiority of WM_o, which combines the virtues of *PPM* and *DG*.

Discussion

All three schemes presented earlier, address the question of *what* to prefetch. But, in designing an efficient and effective prefetching scheme two more questions must be answered. *When* to prefetch and *where* to place prefetched data. Prefetching must be timely. If a prefetching is issued too early, it might displace useful data from cache. If it is issued too late, the prefetched data may arrive late and thus do not contribute in latency reduction. The question of when to prefetch has not been addressed in the Web yet, since the Internet bandwidth varies from time to time and the load on Web/application servers may experience high peaks. The question of where to place prefetched data has not been addressed either. There are many alternatives, as many as the locations where a cache can be placed in the *data flow path.* Each such choice provides diverse opportunities for improving the performance and varying complications in its deployment.

In this section we have presented three methods for deriving predictions, which can be used by a predictive prefetching scheme for the Web. Although, prediction is the core issue in such a scheme, a lot of work must be done before prefetching can be efficiently employed in the Web environment.

WEB CACHES IN COMMERCIAL PRODUCTS

In this section we will present how cache consistency and replacement is managed in a commercial proxy server, the *Squid* proxy cache, and in a high performance Web-powered database, the Oracle9*i* Application Server (Oracle9*i*AS).[25] Moreover, we will briefly comment on some efforts in augmenting the functionality of commercial products with prefetching capabilities.

Cache Management in Proxy Caches

The Squid proxy. One of the most popular proxy servers used today is the *Squid* proxy (Squid, 2001). *Squid* implements both disk-resident and main memory caching of objects retrieved from Web servers. The default replacement policy is the *list-based LRU*[26]. Moreover, it implements a "watermarking" policy to reclaim cache space. It periodically runs an algorithm to evict objects from cache when its utilization exceeds a watermark level. There are two watermark levels, a "low-water mark" and a "high-water" level. Replacement begins when the swap (disk) usage is above the low-water mark and attempts to maintain the utilization near the low-water mark. If the utilization is close to the low-water mark, less replacement is done each time. As the swap utilization gets close to the high-water mark, the object eviction becomes more aggressive. Finally, *Squid* implements a *size-based object admission policy* enabling the determination of the objects that will enter the cache. A minimum and a maximum value for the size of the objects can be defined.

Squid switched from a *TTL-based* expiration model to a *Refresh-Rate* model. Objects are no longer purged from the cache when they expire. Instead of assigning *TTL*'s when the objects enter the cache, freshness requirements are now checked

when the objects are requested. If an object is "fresh", it is given directly to the client. If it is "stale", then an *If-Modified-Since* request is made for it. When checking the object freshness, the *Squid* calculates the following values:

- *AGE*, is how much the object has aged since it was retrieved, that is:
  $$AGE = NOW - OBJECT\_AGE.$$
- *LM_AGE*, is how old the object was when it was retrieved, that is:
  $$LM\_AGE = OBJECT\_DATE - LAST\_MODIFIED\_TIME.$$
- *LM_FACTOR*, is the ratio of *AGE* to *LM_AGE*, that is:
  $$LM\_FACTOR = AGE \ / \ LM\_AGE.$$
- *CLIENT_MAX_AGE*, is the (optional) maximum object age that the client will accept, as taken from the HTTP/1.1 *Cache-Control* request header.
- *EXPIRES*, is the (optional) expiry time from the server reply headers.

These values are compared with the parameters of the "refresh pattern" rules. The refresh parameters are the following: *MIN_AGE, PERCENT, MAX_AGE*.

The following algorithm is applied for determining if an object is fresh or stale:

Algorithm 2 (Squid's Cache Refresh Model (Squid, 2001))

```
(1). BEGIN
(2). if( exists(CLIENT_MAX_AGE) )
(3). if( AGE > CLIENT_MAX_AGE )  return STALE
(4). if (AGE ≤ MIN_AGE)  return FRESH
(5). if ( exists(EXPIRES) )
(6). if( EXPIRES ≤ NOW )  return STALE
(7). else  return FRESH
(8). if( AGE > MAX_AGE )  return STALE
(9). if( LM_FACTOR < PERCENT )  return FRESH
(10). return STALE
(11). END
```

Other proxies. The *Apache*(Apache, 2001) proxy cache uses *TTL*-based consistency as well. The object's lifetime is computed from the server-supplied *Expires* response header when it is present; otherwise it is computed as a portion from the last modification time (using the *Last-Modified* response header) of the object. So, *TTL_Apache = weight_factor**(*NOW- Last-Modified*). The *Jigsaw* (Jigsaw, 2001) proxy server is *TTL*-based as well. *TTL*'s for cached objects are set from the *Expires* response headers. If they are not present, *TTL* is default set to 24 hours.

Cache Management in Oracle9*i*AS

Before describing how Oracle9*i*AS deals with dynamic data caching, we will briefly present the performance problems encountered by high-load Web-powered databases and will also present a couple of architecture alternatives employing caches to improve their scalability and performance.

Many database-backed Web sites receive millions of requests per day. Such Web sites are those offering news on a "hot" event in progress (e.g., a major sports

event) or search engines' sites. Since the time required to serve a dynamic page can be orders of magnitude larger than the respective time for a static page, it is clear that the Web server may experience a very high load during peak times. This is also true for the application server that creates the Web pages and the underlying database from where it retrieves the relevant data. In order to increase the scalability of the whole system and at the same time improve efficiency, many Web sites increase the number of Web servers, so as to reduce the per-server load, thus resulting in a cluster of Web servers (*Web server farm*). In most of the times, this scheme has a front-end load distributor[27], which is responsible for distributing more evenly the load among the machines of the farm. Figure 4 depicts this architecture.[28]

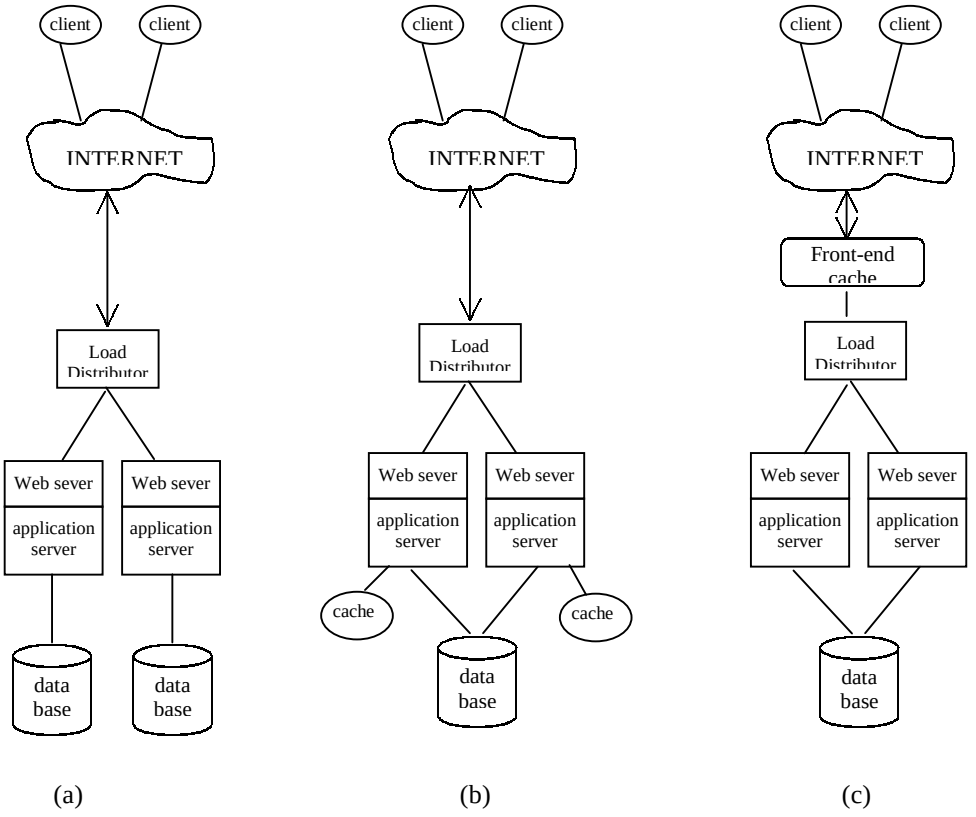
The standard configuration in such a cluster of Web servers is to replicate the database itself (Figure 4(a)). This scheme is very simple, but it does not cache dynamically generated pages and moreover it is very difficult to keep the database replicas synchronized. An alternative is to avoid replicating the database, but provide instead a *middle-tier cache* (depicted in Figure 4(b)) in order to reduce the database load. This scheme cannot avoid the redundancy of computation at the Web and application servers and it requires synchronization between middle-tier caches and the database. Finally, another scheme is to provide the cluster with a front-end cache (depicted in Figure 4(c)). This cache is capable of caching dynamic content forwarding any requests resulting in a cache miss into the servers of the cluster. This architecture provides a separation between *content publication*, handled by the front-end cache, and *content generation*, handled by the Web and application servers and the underlying database. It avoids database replication and multiple caches, but the challenge it faces is to stay coherent. This cache is sometimes referred to as *server acceleration* or *reverse proxy cache*.[29]

The Oracle Web Cache

Oracle[30] has employed the third architecture, depicted in Figure 4(c) augmented with middle-tier data caches at the Web servers or application serves. The front-end cache is called *Web Cache*, whereas the middle-tier caches are referred to as *Data Caches*.

The purpose of a *Data Cache* it to avoid burdening the database backend by caching base data. As base data is considered any collection of data that can be expressed using a *SQL SELECT* statement. They can be a table, or any subset of a table or data from more than one table. The *synchronization policy* of these caches establishes how and how often the cached data are refreshed. The synchronization may be either (a) *incremental*, refreshing only that portion of the data that have been modified, or (b) *complete*, by deleting the locally cached data and retrieving them again from the origin database. The first option is favorable when there is a large amount of cached data whereas the second is better when a large percentage of the data changes or when the bathed updates are loaded into the origin database. The scheduling of synchronization can be done either automatically at specified time intervals or manually with the aid of the *Cache Manager*.

Figure 4: A typical architecture of a Web-powered database in a Web server farm



Oracle9*i*AS *Web Cache* provides server acceleration and server load balancing at the same time. It front-ends a collection of Web and application servers (see Figure 4(c)). It lightens the load of busy Web servers by storing frequently accessed pages in memory, eliminating the need to repeatedly process requests for those pages on middle tier servers. It can cache both static and dynamic content. It can cache full and partial-pages as well as personalized pages (pages containing *cookies* and personalized content e.g., personal greetings).

Oracle9*i*AS *Web Cache* provides both strong and weak consistency through a combination of invalidation messages and expiration. This way, it is able to support applications that can tolerate non-recent data (e.g., weather forecasts) and others, accepting only fresh data (e.g., stock prices).

Data invalidation can be performed in two ways, by the use of an *expiration policy* for the cached objects or by sending an *XML/HTTP invalidation message* to the *Web Cache* host machine.

Expiration Policies. An expiration policy can be set in one of the following three ways:

1. Expire <time> after entering the cache.
2. Expire <time> after object creation (This option relies on the *Last-Modified* header generated by the origin Web server).
3. Expires as per HTTP *Expires* response header.

Expirations are primary used when content changes can be accurately predicted.

***XML*/*HTML* invalidation messages.** When content changes are less predictable and more frequent, then a mechanism based on messages is necessary for maintaining cache coherency. Oracle9*i*AS *Web Cache*'s invalidation messages are *HTTP POST* requests carrying an *XML* payload. The *XML*-formatted part of the POST request informs the cache about which URL's to mark as stale. An invalidation message can be sent in one of the following ways:

1. *Manually*, using the *Telnet* protocol to connect to the *Web Cache*'s host machine or the *Web Cache Manager*.
2. *Automatically*, using database triggers, scripts or applications
   a. Triggers. A trigger stored in a database can include *SQL*, *PL/SQL* or *JAVA* statements to execute as a unit.
   b. Scripts. Since many Web sites use scripts (e.g., *PERL* scripts) to insert new data into the database and the file system, *Web Cache* provides the opportunity for modifying the scripts so as to send an invalidation message to the cache.
   c. Applications. Invalidation messages can be generated by a Web site's underlying application logic or from the content management application used to design Web pages. Oracle9*i*AS *Web Cache* ships with *C* and *JAVA* code that enables developers to embed invalidation mechanisms directly into their applications.

Invalidated objects in the *Web Cache* can be either garbage-collected and thus never be served stale from the *Web Cache* or can be refreshed by sending a request to the origin application Web server. This second option depends on the origin application Web server's capacity.

Prefetching in Commercial Products

Currently, no commercial Web (proxy) server implements prefetching. This is due to the complexity of prefetching in synchronizing the server and the cache and the lack of support from *HTTP 1.1*. Although commercial products do not have prefetching capabilities, several research efforts have resulted in augmenting commercial products with prefetching. To mention the most important of them, the *WebCompanion* (Klemm, 1999) is a client-side prefetching agent implemented as a proxy on the client's browser and is based on prefetching the embedded links of a page. *P-Jigsaw* (Bin & Bressan, 2001) is an extension to Jigsaw Web server, implementing prefetching in the Web server based on a simplified form of association rule discovery (Bin, Bressan, Ooi & Tan, 2001). (Cohen, Krishnamurthy & Rexford, 1998) implements prefetching between Web servers and proxy caches by constructing *volumes* of related resources through the *Dependency Graph*.

EMERGING AND FUTURE TRENDS

Content Delivery Networks

The Internet "miracle" is based on a growing set of standardized, interconnected networks and a standard for information publishing and viewing, the Web and browsers. We all know that the Internet, and consequenlty the Web, faces performance problems. Performance problems arise in any of the following three general areas:

- *Web server processing delays.* Servers can't keep up with peak loads presented, unless the site is built with overcapacity.
- *Internet delays.* Beyond USA, network capacity diminishes rapidly. Moreover, the data exchange points between the various networks that constitute the Internet (peering points) become overloaded and lose packets thus requiring the packets to be resent.
- *"Last-mile" delays* (delays between the subscriber and the Internet e.g., due to a slow dial-up modem connection).

Increasing the number of Web (and application) servers (Web server farms) does provide a solution for the first problem but can do nothing for the other two. Caching at various points in the network can improve performance substantially, but these caches usually suffer from low hit rates (Kroeger et al., 1997). The idea in alleviating these problems is to make content delivery from origin servers more "distributed", moving some of their content to the "edge" of the Internet. Based upon this idea the *Content Delivery Networks (CDN)* (Akamai, InfoLibria, etc.) emerged recently. *CDN* are designed to take advantage of the geographic locations of end users. Rather than serving content from the origin Web site, the *content distribution model* makes copies of "key" content on multiple content delivery servers sites distributed through the Internet, close to the users requesting that content. "Key" content may represent popular or bandwidth-demanding objects (graphics, streaming media), usually static content.

A *Content Delivery Network*, like Akamai's *FreeFlow*, runs on thousands of servers distributed across the Internet at Network Provider operations centers, universities, corporate campuses and other locations with a large number of Web visitors. A *CDN* contracts with content providers (e.g., Yahoo!) to deliver their content. The content publisher can control what pieces of content to "outsource" to the *CDN* provider, by replacing existing *HREF* tags in the content's owner's *HTML* with tags that point to the *CDN* provider's domain, which has already obtained a copy of the content to be delivered. Thus, a client request first goes to the origin Web server, which will return an *HTML* page with references for graphics and other objects to the content delivery network. Then, the client will request the "outsourced" content from the *CDN* provider. Figure 5 depicts how the *content distribution model* operates.

Content distribution services address efficiently the aforementioned performance problems.

- With the "hottest" content "outsourced", the load on the origin server is reduced.

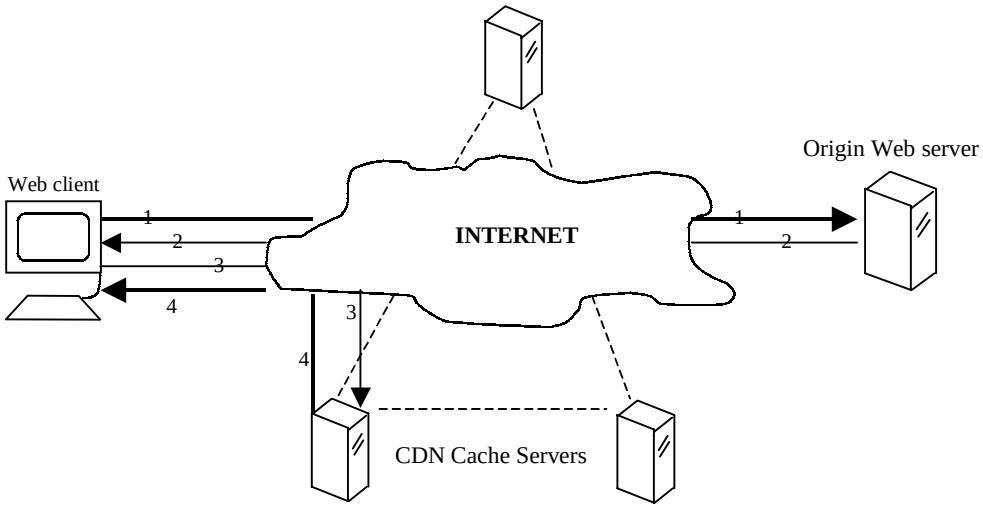
- The connection from a local content delivery server is shorter than between the origin Web server and the user, thus reducing latency.
- Since the *CDN* servers are shared by many users this service greatly increases the hit ratio.

Though *CDN* servers operate like a conventional cache, they differ in that they get *only* requests for those objects that they are contracted to serve. Thus, their "hit ratio" is 100%. *CDN* do not replace normal caches since they work in an "orthogonal" axis. *CDN* optimize content access for specific paying content publishers, while an Internet cache optimizes content access for a community of subscribers.

CDN's have been proven successful in delivering static content (e.g., streaming media), but services in the Web move quickly from *read-only* to *transactional*. Millions of users might visit a site per day and perform an analogous number of transactions e.g., in an on-line auctions site. Thus, scalability issues must be effectively addressed in these sites. The primary question is whether *CDN* can be employed in delivering dynamic content.

Currently, several companies (Akamai, Zembu) proposed architectures for delivering dynamic content from the "edge" of the Internet. The basic idea is to separate the *Content Generation Tier* from the *Content Assembly and Delivery Tier*. The former is typically centrally maintained in an enterprise data center and its primary function is application coordination and communication to generate the information that is to be published. It typically consists of application servers, policy servers, data and transaction servers and storage management. The latter, residing in the "edge" of the Internet, consists of servers ("edge servers") that perform content caching and assembly of pages. These two tiers communicate through a simple *Integration Tier*, consisting of a few Web servers serving as the *HTTP* communication gateways. Although this distributed infrastructure is commercialized by some companies (e.g., *Akamai's EdgeSuite*) and employed by several content

Figure 5: The content distribution model



providers (CNN, McAffee), work is still needed in the field of cache management in order to address the issues of scalability.

In such a large scale distribution, where "edge servers" are geographically distributed over a wide area network (Internet), the issue of cache invalidation and update scheduling is very crucial, in order to guarantee strong cache consistency. Recent work on update scheduling focuses only on centrally managed Web-powered databases (Labrinidis & Roussopoulos, 2001) and does not address many important issues like update deadlines, staleness tolerance. Moreover, cache management in the "edge servers" raises some issues for the cache replacement, as well. Traditional policies like *LRU*, are not be adequate for these caches, and novel ones are required (e.g., *Least Likely to be Used (LLU)* (Datta et al., 2001)) that take into account the derivation dependencies between cached objects.

Processing Power to the Clients

Apart from employing the sophisticated solution of *CDN*s in order to relieve the origin data servers from heavy load, simpler solutions could help towards this goal, as well. Such solutions could exploit current proxy caches, which should only become "smarter".

Semantic caching mechanisms (Dar, Franklin, Jonsson, Srivastava & Tan, 1996) could provide significant performance benefits provided that they become more sophisticated than the current proposals (Chidlovskii & Borghoff, 2000), maybe through the cooperation with the origin data server, as proposed in (Luo & Naughton, 2001).

Another alternative would be to migrate not only data closer to the clients but some data processing capabilities, as well. So having the server to provide along with the data, code portions that implement part of its processing logic could improve performance and reduce latency significantly. This is because cached data can be processed by their associated code in order to answer queries different from those that resulted in caching the specific data. Pioneering work based on this idea was implemented for proxy caches in (Luo, Naughton, Krishnamurthy, Cao & Li, 2000).

CONCLUSION

In this chapter, we have examined the issues of cache replacement and consistency as well as that of prefetching for Web caches. We were particularly concerned for caches that store data originating from *Web-powered databases*. We demonstrated that caching is essential for improving the performance of such a multi-tier system.

We presented the most important policies proposed so far for the issue of cache replacement. Through this presentation became clear that any successful replacement algorithm should take into account the object's size and cost in such a manner that it presents no complexity in making replacement decisions. We discussed how cache consistency can be maintained and concluded that, since Web applications

require both strong and weak consistency, a combination of invalidation and expiration policies is the best solution.

We also described a common context so as to treat existing predictive prefetching algorithms as *Markov predictors*. Through this description, the superiority of WM_o become clear, since it was specifically designed to address the requirements of the Web environment, namely, higher order dependencies and non-consecutiveness between correlated references.

Finally, we presented the ideas behind the emerging trend of *Content Distribution Networks*, that attempt to make the Web a really distributed database by moving data closer to their consumers. We highlighted the challenges related to caching behind their architectural design. Moreover, we pointed out some areas where future work should concentrate. We believe that future work should concentrate on two targets. Firstly, to move data closer to the clients, as *Content Distribution Networks* do currently and secondly to move some "application logic" closer to the clients, in order to improve the scalability.

ENDNOTES

1 The number of appearances of an item in a stream is called the *popularity profile* of the item.

2 This path will be called *data flow path* in the sequel.

3 The reader can refer to (Oracle, 2001) for more information on this topic.

4 Longest Forward Distance.

5 Cost Inverse Forward Distance.

6 Presented in (Abrams, Standridge, Abdulla, Fox & Williams, 1996).

7 Not referenced in the future.

8 Policies like *LRU*, *LFU*, *SIZE*, can be regarded as *function-based* policies as well, where the utility function is the inverse of recency, frequency and inverse of size, respectively. They can be regarded as *key-based* policies as well having only a primary key.

9 These two assumptions determine the *Independent Reference Model*.

10 The number of such groups is usually small.

11 Details on how the value of the constants c, α_1, α_2 is computed can be found in (Rizzo & Vicisano, 2000).

12 Available in the response message from the server.

13 Here, by "client" we mean any location in the *data flow path* (see Section "Introduction") that caches data originating from a "server" location.

14 In the Web there is usually a single "writer" (the server) and multiple "readers" (the clients).

15 More info on the techniques for identificating clients can be found in (Kristol, 2001) and (Oracle, 2001).

16 In case the original server is unreachable due to failure or network partition, then the cache can inform the client that the data are potentially stale and the client can take an "application-depended" decision.

17 Provided that it has recorded it on the secondary storage.

18 Leases fall into *server-driven* protocols since it is the server that grants the leases.

19 Recorded in Web server log files.

20 By the term subsequence, we mean that the elements of *S* need not be necessarily consecutive in *Tr.*

21 Or by the same application or the same process.

22 We assume standard knowledge of the problem of mining association rules and of the *Apriori* algorithm (Agrawal & Srikant, 1994).

23 The itemsets whose support exceeds a user-defined support threshold.

24 We use the term "itemset" for the *WMo* as well, although we should use the term "itemsequence".

25 Oracle is a registered trademark of Oracle Corporation.

26 Objects are maintained in sorted order of their last access in a list (*list-based LRU*) and not in a heap (*heap-based LRU*).

27 Instead of using *Round Robin DNS.*

28 Note that for Web-powered databases, replicating only the Web server is not enough for scaling up the whole system. The application server must be replicated as well.

29 A proxy server caches content from an infinite number of sources, whereas a server accelerator caches content for one or a handful of origin servers.

30 All information in Subsection "The Oracle Web Cache" is based on (Oracle, 2001). Consequently, newer releases of the product may turn this material obsolete.

REFERENCES

Abrams, M., Standridge, C., Abdulla, G., Fox, E. and Williams, S. (1996). Removal policies in network caches for World Wide Web documents. *Proceedings of the ACM Conference on Applications, Technologies, Architectures and Protocols for Computer Communications (ACM SIGCOMM'96)*, 293-305.

Aggrawal, C., Wolf, J. and Yu, P. (1999). Caching on the World Wide Web. *IEEE Transactions on Knowledge and Data Engineering (IEEE TKDE), 11*(1), 94-107.

Agrawal, R. and Srikant, R. (1994). Fast algorithms for mining association rules in large databases. *Proceedings of the 20th International Conference on Very Large Databases (VLDB'94)*, 487-499.

Almeida, V., Bestavros, A., Crovella, M. and de Oliveira, A. (1996). Characterizing reference locality in the WWW. *Proceedings of the 4th IEEE Conference on Parallel and Distributed Information Systems (IEEE PDIS'96)*, 92-103.

Apache. (2001). Apache 1.2.6 HTTP server documentation. Retrieved August 30, 2001, from http://www.apache.org.

Atzeni, P., Mecca, G. and Merialdo, P. (1998). Design and maintenance of data-intensive Web sites. *Proceedings of the 6th International Conference on*

Extending Database Technology, (EDBT'98), Lecture Notes in Computer Science, 1377, 436-450.

Belady, L. A. (1966). A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal, 5*(2), 78-101.

Berners-Lee, T., Caililiau, R., Luotonen, A., Nielsen, H. F. and Secret, A. (1994). The World-Wide Web. *Communications of the ACM (CACM), 37*(8), 76-82.

Bin, D. L. and Bressan, S. (2001). P-Jigsaw: Extending Jigsaw with rules assisted cache management. *Proceedings of 10th World Wide Web Conference (WWW10)*, 178-187.

Bin, D. L., Bressan, S., Ooi, B. C. and Tan, K.L. (2000). Rule-assisted prefetching in Web-server caching. *Proceedings of ACM International Conference on Information and Knowledge Management (ACM CIKM'00)*, 504-511.

Breslau, L., Cao, P., Fan, L., Phillips, G. and Shenker, S. (1999). Web caching and Zipf-like distributions: Evidence and implications. *Proceedings of the IEEE Conference on Computer Communications (IEEE INFOCOMM'99)*, 126-134.

Cao, P., Felten, E. W., Karlin, A. R. and Li, K. (1996). Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Transactions On Computer Systems (ACM TOCS), 14*(4), 311-343.

Cao, P. and Irani, S. (1997). Cost-aware WWW proxy caching algorithms. *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS'97)*, 193-206.

Cao, P. and Liu, C. (1998). Maintaining strong cache consistency in the World Wide Web. *IEEE Transactions on Computers (IEEE TOC), 47*(4), 445-457.

Cate, V. (1992). Alex–A global file system. *Proceedings of the USENIX File System Workshop*, 1-12.

Challenger, J., Iyengar, A. and Dantzig, P. (1999). A scalable system for consistently caching dynamic Web data. *Proceedings of the IEEE International Conference on Computer Communications (IEEE INFOCOM'99)*.

Chen, M. S., Park, J. S. and Yu, P. S. (1998). Efficient data mining for path traversal patterns. *IEEE Transactions on Knowledge and Data Engineering (IEEE TKDE), 10*(2), 209-221.

Chidlovskii, B. and Borghoff, U. (1999). Semantic caching of Web queries. *The VLDB Journal, 9*(1), 2-17.

Coffman, E. and Denning, P. (1973). *Operating Systems Theory*. Englewood Cliffs, NJ: Prentice-Hall.

Cohen, E., Krishnamurthy, B. anbd Rexford, J. (1998). Improving end-to-end performance of the Web using server volumes and proxy filters. *Proceedings of ACM International Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (ACM SIGCOMM'98)*, 241-253.

Cohen, E., Krishnamurthy, B. and Rexford, J. (1999). Efficient algorithms for predicting requests to Web servers. *Proceedings of the IEEE International Conference on Computer Communications (IEEE INFOCOM'99)*, 284-293.

Curewitz, K., Krishnan, P. and Vitter, J.S. (1993). Practical prefetching via data compression. *Proceedings of the ACM International Conference on Management of Data (ACM SIGMOD'93)*, 257-266.

Dar, S., Franklin, M., Jonsson, B., Srivastava, D. and Tan, M. (1996). Semantic data caching and replacement. *Proceedings of 22nd International Conference on Very Large Data Bases (VLDB'96)*, 330-341.

Datta, A., Dutta, K., Thomas, H., VanderMeer, D., Ramamritham, K. and Fishman, D. (2001). A comparative study of alternative middle tier caching solutions to support dynamic Web content acceleration. *Proceedings of 27th International Conference on Very Large Data Bases (VLDB'01)*, 667-670.

Denning, P. and Schwartz, S. (1972). Properties of the Working-Set model. *Communications of the ACM (CACM), 15*(3), 191-198.

Deshpande, M. and Karypis, G. (2001). Selective Markov models for predicting Web-page accesses. *Proceedings of the 1st SIAM Conference on Data Mining (SDM'01)*.

Dilley, J. and Arlitt, M. (1999). Improving proxy cache performnace: Analysis of three replacement policies. *IEEE Internet Computing, 3*(6), 44-55.

Duvvuri, V., & Shenoy, P. and Tewari, R. (2000). Adaptive Leases: A strong consistency mechanism for the World Wide Web. *Proceedings of the 19th IEEE Conference on Computer Communications (IEEE INFOCOM'00)*, 834-843.

Fan, L., Cao, P., Lin, W. and Jacobson, Q. (1999). Web prefetching between low-bandwidth clients and proxies: Potential and performance. *Proceedings of ACM International Conference on Measurement and Modeling of Computer Systems (ACM SIGMETRICS'99)*, 178-187.

Franklin, M., Carey, M. and Livny, M. (1997). Transactional client-server cache consistency: Alternatives and performance. *ACM Transactions On Database Systems (ACM TODS), 22*(3), 315-363.

Garey, M. and Johnson, D. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness.* New York: W.H. Freeman and Company.

Gray, C. and Cheriton, D. (1989). Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. *Proceedings of the 12th ACM Symposium on Operating Systems Principles (ACM SOSP'89)*, 202-210.

Greenspun, P. (1999). *Philip and Alex's Guide to Web Publishing*. New York: Morgan Kaufmann.

Griffioen, J. and Appleton, R. (1994). Reducing file system latency using a predictive approach. *Proceedings of the Summer USENIX Conference*, 197-207.

Hosseini-Khayat, S. (1997). *Investigation of generalized caching*. PhD Thesis, Washington University, Saint Louis, Missouri.

Hosseini-Khayat, S. (2000). On optimal replacement of nonuniform cache objects. *IEEE Transactions on Computers (IEEE TOC), 49*(8), 769-778.

Howard, J., Kazar, M., Menees, S., Nichols, D., Satyanarayanan, M., Sidebotham, R. and West, M. (1988). Scale and performance in a distributed file system. *ACM Transactions On Computer Systems (ACM TOCS), 6*(1), 51-81.

Iyengar, A. and Challenger, J. (1998). Data update propagation: A method for determining how changes to underlying data affect cached objects on the Web. *Technical Report*, IBM Research Division, RC 21093(94368).

Jiang, Z. and Kleinrock, L. (1998). An adaptive network prefetch scheme. *IEEE Journal on Selected Areas in Communications (IEEE JSAC), 16*(3), 358-368.

Jigsaw. (2001). Jigsaw 2.0 HTTP server documentation. Retrieved August 30, 2001 from http://www.w3c.org/Jigsaw.

Jin, S. and Bestavros, A. (2000). Sources and characteristics of Web temporal locality. *Proceedings of the IEEE/ACM Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'00)*.

Jin, S. and Bestavros, A. (2001). GreedyDual* Web caching algorithm: Exploiting the two sources of temporal locality in Web request streams. *Computer Communications, 24*(2), 174-183.

Klemm, R. (1999). WebCompanion: A friendly client-side Web prefetching agent. *IEEE Transactions on Knowledge and Data Engineering (IEEE TKDE), 11*(4), 577-594.

Korth, H., Silberschatz, A. and Sudarshan, S. (1998). *Database System Concepts*. New York: McGraw-Hill.

Kristol, D. (2001). HTTP cookies: Standards, privacy and politics. *ACM Transactions on Internet Technology (ACM TOIT), 1*(2), 151-198.

Kroeger, T., Long, D. E. and Mogul, J. (1997). Exploring the bounds of Web latency reduction from caching and prefetching. *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS'97)*, 13-22.

Labrinidis, A. and Roussopoulos, N. (2000). WebView materialization. *Proceedings of the ACM International Conference on Management of Data (ACM SIGMOD'00)*, 504-511.

Labrinidis, A. and Roussopoulos, N. (2001). Update propagation strategies for improving the quality of data on the Web. *Proceedings of 27th International Conference on Very Large Data Bases (VLDB'01)*, 391-400.

Luo, Q. and Naughton, J. (2001). Form-based proxy caching for database-backed Web sites. *Proceedings of 27th International Conference on Very Large Data Bases (VLDB'01)*, 191-200.

Luo, Q., Naughton, J., Krishnamurthy, R., Cao, P. and Li, Y. (2000). Active query caching for database Web servers. *The World Wide Web and Databases, Lecture Notes in Computer Science*, 1997, Springer-Verlag, 92-104.

Luotonen, A. and Altis, A. (1994). World Wide Web proxies. *Computer Networks and ISDN Systems, 27*(2), 147-154.

Malaika, S. (1998). Resistance is futile: The Web will assimilate your database. *IEEE Data Engineering Bulletin, 21*(2), 4-13.

Mattson, R., Gecsei, J., Slutz, D. and Traiger, I. (1970). Evaluation techniques for storage hierarchies. *IBM Systems Journal, 9*(2), 78-117.

Nanopoulos, A., Katsaros, D. and Manolopoulos, Y. (2001). Effective prediction of Web-user accesses: A data mining approach. *Proceedings of the International Workshop on "Mining Log Data Across All Customer TouchPoints" (WEBKDD'01)*.

Nanopoulos, A., Katsaros, D. and Manolopoulos, Y. (2002). A data mining algorithm for generalized Web prefetching, *IEEE Transactions on Knowledge and Data Engineering (IEEE TKDE)*, to appear, 2002.

Nelson, M., Welch, B. and Ousterhout, J. (1988). Caching in the Sprite network file system. *ACM Transactions On Computer Systems (ACM TOCS), 6*(1), 134-154.

Niclausse, N., Liu, Z. and Nain, P. (1998). A new efficient caching policy for the World Wide Web. *Proceedings of the Workshop on Internet Server Performance (WISP)*.

O'Neil, E., O'Neil, P. and Weikum, G. (1993). The LRU-K page replacement algorithm for database disk buffering. *Proceedings of the ACM International Conference on Management of Data (ACM SIGMOD'93)*, 297-306.

Oracle. (2001). Oracle9*i*AS Web Cache (*White paper*), June.

Padmanabhan, P. and Mogul, J. (1996). Using predictive prefetching to improve World Wide Web latency. *ACM SIGCOMM Computer Communication Review, 26*(3).

Palpanas, T. and Mendelzon, A. (1999). Web prefetching using partial match prediction. *Proceedings of the 4th Web Caching Workshop (WCW'99)*.

Patterson, H.R., Gibson, G.A., Ginting, E., Stodolsky, D. and Zelenka, J. (1995). Informed prefetching and caching. *Proceedings of the ACM Symposium on Operating System Principles (ACM SOSP'95)*, 79-95.

Rizzo, L. and Vicisano, L. (2000). Replacement policies for a proxy cache. *ACM/IEEE Transactions on Networking (ACM/IEEE TON), 8*(2), 158-170.

Robinson, J. and Devarakonda, M. (1990). Data cache management using frequency-based replacement. *Proceedings of the ACM Conference on Measurement and Modeling of Computer Systems (ACM SIGMETRICS'90)*, 134-142.

Rodriguez, P., Spanner, C. and Biersack, E.W. (2001). Analysis of Web caching architectures: Hierarchical and distributed caching. *ACM/IEEE Transactions on Networking (ACM/IEEE TON), 9*(4), 404-418.

Shim, J., Scheuermann, P. and Vingralek, R. (1999). Proxy cache algorithms: Design, implementation and performance. *IEEE Transactions on Knowledge and Data Engineering (IEEE TKDE), 11*(4), 549-562.

Squid. (2001). Squid 2.4 Stable 1 HTTP server documentation, Retrieved August 30, 2001 from http://squid.nlanr.net.

Tanenbaum, A. (1992). *Modern Operating Systems*. Englewood Cliffs, NJ: Prentice-Hall.

Wang, J. (1999). A survey of Web caching schemes for the Internet. *ACM SIGCOMM Computer Communication Review, 29*(5).

Wooster, R. and Abrams, M. (1997). Proxy caching that estimates page load delays. *Computer Networks, 29*(8-13), 977-986.

Yagoub, K., Florescu, D., Issarny, V. and Valduriez, P. (2000). Caching strategies for data-intensive Web sites. *Proceedings of the 26th International Conference on Very Large Databases (VLDB'00)*, 188-199.

Yin, J., Alvisi, L., Dahlin, M. and Iyengar, A. (2001). Engineering server-driven consistency for large scale dynamic Web services. *Proceedings of the 10th World Wide Web Conference (WWW10)*, 45-57.

Yin, J., Alvisi, L., Dahlin, M. and Lin, C. (1999). Volume Leases for consistency in large-scale systems. *IEEE Transactions on Knowledge and Data Engineering (IEEE TKDE), 11*(4), 563-576.

Young, N. E. (1994). The k-server dual and loose competitiveness for paging. *Algorithmica, 11*(6), 525-541.

Section IV

Heterogeneous and Distributed Systems