ELSEVIER

Data & Knowledge Engineering 39 (2001) 165–189

DATA & KNOWLEDGE ENGINEERING

www.elsevier.com/locate/datak

Web log data warehousing and mining for intelligent web caching

F. Bonchi [a,b], F. Giannotti [a], C. Gozzi [a], G. Manco [a,b], M. Nanni [b], D. Pedreschi [b], C. Renso [a], S. Ruggieri [b,*]

[a] *CNUCE-CNR, Pisa Research Area, Via Alfieri 1, 56127 Ghezzano (PI), Italy*

[b] *Dipartimento di Informatica, Università di Pisa, Corso Italia 40, 56125 Pisa, Italy*

Received 19 June 2001; received in revised form 24 July 2001; accepted 24 July 2001

Abstract

We introduce intelligent web caching algorithms that employ predictive models of web requests; the general idea is to extend the least recently used (LRU) policy of web and proxy servers by making it sensitive to web access models extracted from web log data using data mining techniques. Two approaches have been studied in particular, frequent patterns and decision trees. The experimental results of the new algorithms show substantial improvement over existing LRU-based caching techniques, in terms of *hit rate*. We designed and developed a prototypical system, which supports data warehousing of web log data, extraction of data mining models and simulation of the web caching algorithms. © 2001 Elsevier Science B.V. All rights reserved.

Keywords: Web caching; Log data warehousing; Data mining; Frequent patterns; Association rules; Decision trees

1. Introduction

If data mining is aimed at discovering regularities and patterns hidden in data, the emerging area of web mining is aimed at discovering regularities and patterns in the structure and content of web resources, as well as in the way web resources are accessed and used [11,19,25–27,32,41,42]. In this paper we describe one particular data/web mining application based on data warehouse technology: the development of an intelligent web caching architecture, capable of adapting its behavior on the basis of the access patterns of the clients/users. Such usage patterns, or models, are extracted from the historical access data recorded in log files, by means of data mining techniques.

More precisely, the idea is to extend the *least recently used* (LRU) cache replacement policy adopted by web and proxy servers by making it sensitive to web access models, extracted from web log data. To this end, we introduce several ways to construct intelligent web caching

[*] Corresponding author. Tel.: +39-050-2212782; fax: +39-050-2212726.
E-mail address: ruggieri@di.unipi.it (S. Ruggieri).

algorithms that employ predictive models of web requests. The goal of these algorithms is to maximize the so-called *hit rate*, namely the percentage of requested web entities that are retrieved directly in cache, without requesting them back to the origin server.

The general idea is motivated by the following observation. The LRU policy – drop from cache the least recently used entities – is based on the assumption that the requests that occurred in the recent past will likely occur in the near future too. This assumption is often true in practice, and this explains why LRU is effective: in particular, when requests are characterized by temporal locality (like the case of web requests [7,9,12,24,30]). Now, the more information we can extract from the history of recent requests, the more informed the cache replacement strategies that can be devised. This is a clear indication to mine the web log data for access models which may be employed for the above purpose.

When compared with the many alternatives and variations to LRU caching presented in the literature, our approach has a unique feature: its adaptiveness to changes in the usage patterns, which are rather natural in the web. This is due to the fact that the proposed caching strategies are parametric w.r.t. the data mining models, which can be recomputed periodically in order to keep track of the recent past.

We adopt two data mining techniques, which yield two classes of web caching algorithms: *frequent patterns* and *decision trees*. In the first case, we extract from past web logs patterns of the form $A \rightarrow B$, where A and B are web entities: one such pattern means that when A is requested, then B is also likely to be requested within the same user session. A pattern $A \rightarrow B$ may be used as follows within the cache replacement algorithm on new requests: if A is requested, and therefore assigned a high priority due to the LRU principle, then B should also be treated analogously.

In the second case, we develop on the basis of the web logs a decision tree, i.e., a model capable of predicting, given a request of a web object A, the time of the next future request of A given other properties of A itself. Again, the prediction is based on the historical data contained in the web logs.

A prerequisite of either technique is the development of a data warehouse of log data, as raw log files are a thoroughly unsatisfactory starting point for data mining. We therefore developed a fully automated acquisition process which migrates log data into a carefully designed data mart, oriented to the analysis required for intelligent web caching. In the data mart, the web log information is consolidated, cleaned, selected and prepared for the data mining analysis; for instance, new derived attributes are introduced, and missing values for attributes are approximated.

As a final step in the knowledge discovery process, we designed a reference web caching model as a means to evaluate the models extracted by data mining. The architecture, which emulates a cache, and is parametric to the replacement strategy, supports the evaluation and comparison of the various replacement policies, according to two selected metrics: the hit rate and the weighted hit rate, which also takes into account the size of the requested entities.

The overall process, from log data acquisition to model extraction up to evaluation by the web cache architecture, is formalized and implemented within a database management system, Microsoft's SQL Server 2000 [34]. The prototype system was used for a first round of extensive experiments over large log files from two web servers. A thorough presentation of the experimental results is outside the scope of this paper; moreover, further benchmarking is needed and is currently pursued. However, the performance figures obtained so far indicate that the developed methods, compared with LRU, substantially increase in the hit rate, and outperform on this metric many traditional caching strategies. This indication, together with the adaptiveness of data-mining-based caching, definitely motivates further study on this subject.

2. Towards intelligent web caching

2.1. Web caching

Web caching inherits several techniques and issues from caching in processor memory and file systems [10]. However, the peculiarities of the web give rise to a number of novel issues which call for adequate solutions.

Caches can be deployed in different ways, which can be classified depending on where the cache is located within the network. The spectrum of possibilities ranges from caches close to the client (browser caching, proxy server caching) to caches close to the origin server (web server caching). Proxy server caching provides an interface between many clients and many servers. The effectiveness of proxy caching relies on common accesses of many users to the same set of objects. Even closer to the client we find the browser caches, which perform caching on a per-user basis using the local file system. However, since they are not shared, they only help single-user activity. Web server caching provides an interface between a single web server and all of its users. It reduces the number of requests the server must handle, and then helps load balancing, scalability and availability.

In all cases, it is recognized that deploying caching in the world wide web can improve the net traffic in several ways [1,3,6,7,14,21,23,24,33,36,39,46,47], including the reduction of bandwidth consumption, network latency and server load. Web caching, however, poses some issues which risk reduction in its applicability and effectiveness. We mention the issues of cache consistency [13] (cached copies may become stale), dynamic objects [15] (e.g., web resources personalized on-the-fly for each client), security and legal issues.

In order to evaluate the quality of a web caching system, several measures can be applied, depending on which resource we are focusing on – usually the bottleneck of the system under consideration. The most commonly used criteria are basically the following three:

- *Hit rate*, i.e., the ratio of requests fulfilled by the cache, and then not handled by the origin servers. It is a straight measure of the saved workload on the CPU of the web server, and so it becomes particularly important when the web server computational resources are the main issue. Moreover, it gives a rough estimation of both the saved network traffic and the reduction of latency perceived by the clients.
- *Weighted hit rate*, i.e., the ratio of bytes served to the client by the cache. It measures how much network traffic is saved to/from the web server, and so it is particularly relevant whenever the network bandwidth between the web server and the cache is an issue.
- *Latency*, i.e., the time that an end-user waits for retrieving a resource. While usually a short latency is simply desirable, for strongly time-related resources (such as multimedia data) that can become a strict requirement.

In this work we choose to evaluate our novel caching strategies by simulating them over a set of collected logs from web servers. We abstract from the architectural details needed to estimate the latency measure, so restricting consideration of hit rate and weighted hit rate.

Caching algorithms are characterized by their replacement strategy, which mainly consists of ordering objects in cache according to some parameters (the arrival order, the request frequency, the object size, and compositions of them): objects are evicted according to such an order. Among the best-known replacement strategies [1–3], we mention **FIFO** (order by arrival time), **SIZE** (order inversely to object size), **LFU** (order inversely to number of requests), **LRU** (order by last request time), **SLRU** (order inversely to $\Delta T \cdot \mathit{Size}$, ΔT being the number of requests since the last request to the object), **LRU-Min** (when the cache has to make room for an object of size S, first it tries to do it by evicting objects of size S or greater in LRU order; if that fails, it tries with objects of size $S/2$ or more, then $S/4$ or more and so on).

In later sections of this paper, another strategy is referred to, which we call **ORCL** (ORaCLe strategy): it is off-line (i.e., uses information not available at the time of the requests) and extends the SLRU strategy. At each time, the weight of an entity is $\Delta' T \cdot \mathit{Size}$, where $\Delta' T$ is the number of requests that *will be* received until the next access to the object. Entities with higher values of $\Delta' T \cdot \mathit{Size}$ are removed first. Indeed, we observe that in the SLRU schema ΔT is a simple on-line estimation of $\Delta' T$. Of course, the ORCL strategy can be simulated only on historical data, since we need to know future requests in order to compute $\Delta' T$.

Many other algorithms can be found in literature, very similar to or extending the schema of the above strategies [2,3,14,20,21,23,28,29,33,36,39,46]. Also, we refer to [5,8] for the state of the art in theoretical complexity analysis of on-line and off-line caching strategies. However, we notice that all such approaches and refinements are of a *static* nature, since they follow some fixed criteria usually originated from an a priori analysis of the general behavior of requests traffic. They yield then *data-independent* optimizations, which do not take into account the *structural* characteristics of the *actual* requests flow. Moving from one web context to another with different characteristics (e.g., from web to proxy caching), or staying within a context having a time-varying behavior (see [9]), the performances of such static strategies may significantly change. In contrast, as explained in later sections, the approaches proposed in this paper are of an *adaptive* nature, since they make use of a *mining model* which can be modified from time to time according to the recent request history of the cache.

2.2. A reference model for intelligent caching

This section introduces a general model for caching strategies where data mining algorithms are integrated with replacement policies. Before introducing the model, we give a brief presentation of the caching specification of the HyperText Transfer Protocol (HTTP) [44].

The HTTP is a request/response protocol. A client sends a request to a server (or to a proxy) specifying a request *method* for accessing a resource, a resource identifier (called a URL), and, possibly, a message containing request modifiers and client information. The HTTP methods include "GET", i.e., client requests for resources, "POST", i.e., requests to process information, "PUT", i.e., requests to store resources, and "HEAD", i.e., client requests for meta-information about resources. The server responds to requests with a return code, and an *entity* containing an entity-header and an entity-body content. The HTTP/1.1 protocol considers an "expiration" time provided by the origin server or estimated by a cache consistency algorithm, which is used to evaluate consistency of entities in cache. Till the expiration time, the entity is considered *fresh*, i.e., an exact copy of the entity stored at the origin server. After the expiration time has passed, the entity in cache becomes *stale*. In order for the cache to use a stale entity as a response to a client's request, it must *validate* it. Validation may be performed by means of a conditional GET message to the origin server (or to an intermediate cache), including a *validator*, such as the `Last-Modified` time of the resource or a timestamp, that allows the origin server to check whether the entity has changed.

Traditional caching strategies can be modeled by considering entities in cache as belonging to a priority queue. In this way, the cache replacement strategy coincides with the priority queue weight assignment policy. While for traditional caching strategies the weight assignment policy is fixed once and for all, we support strategies that are "intelligent", in the sense that they adapt to the flow of requests, possibly affecting the weight assignment policy. We call a policy that exploits knowledge extracted from past requests a `DataMiningWeightModel`. Of course, when the weight assignment policy is a fixed policy, then it boils down to traditional caching strategies. The generic model for intelligent caching strategies is reported in Fig. 1.

First of all (line `1`), an initial data mining model `DMM` is built on past data.

For a requested entity `t`, if the cache contains the entity and it is fresh (line `5`), then the entity is returned to the client. Concerning the performance measures (lines `6`–`7`), this case is considered as a *hit*. Also, the bytes sent back to the client are counted for the *weighted hit rate* measure. Note that the size of the entity is not necessarily the same as the number of bytes sent back to the client. They differ, for instance, when the client closes the connection before receiving the whole entity. The weight of the entity in cache, and possibly the weights of other elements in cache, is updated (line `8`) according to the weight assignment policy expressed by `DMM`.

On the contrary, if the entity is not in cache or it is stale, (line `9`), we have a *miss*. The entity is deleted from the cache (line `10`) and a fresh version is retrieved and pushed into the cache (lines

```
        PriorityQueue Cache;
        DataMiningWeightModel DMM;
        CacheEntry t, t_fresh;
        long hits = 0;

1.      DMM.build();
2.      loop forever {
3.          do {
4.                  get_request(t);
5.                  if (Cache.contains_freshcopy(t)) {
6.                      hits += 1 ;
7.                      whits += t.bytes_sent_to_client ;
8.                      Cache.update(t, DMM);
9.                  } else {
10.                     Cache.delete(t);
11.                     Retrieve_freshcopy(t, t_fresh);
12.                     Cache.push(t_fresh, DMM);
13.                     while (Cache.size > Cache.maxsize)
14.                         Cache.pop();
15.                 }
16.         } while (condition);
17.         DMM.updatemodel();
18.     }
```

Fig. 1. Intelligent caching reference model.

`11–12`). The `push` method consists of assigning a weight to the entity and, possibly, updating the weights of other elements in cache. If the inclusion makes the cache exceed the maximum size, then entities are popped out from the cache according to their weights (lines `13–14`). Such an approach is known as running the replacement policy *on demand*, i.e., each time cache size overflows.

Finally, the data mining model `DMM` may be periodically updated when some *condition* becomes false (line `16`), e.g., at fixed time intervals or when the cache performance decreases. We model such an update by the method `updatemodel` (line `17`). Notice that the update of the model is decoupled from the on-line caching of entities, and it can be performed in parallel, e.g., by some external "agent".

Two further details of real-world caching were abstracted away in the general model. First, the requested entity may not be cacheable [44] (e.g., responses to "POST" and "PUT" requests, entities with a `no-cache` directive). We abstract from the cacheability issue, by assuming that all requests refer to cacheable documents. With this assumption, our performance measures, *hit rate* and *weighted hit rate*, refer to the percentage of *cacheable* entities (resp., bytes) requested by clients and that are found in cache. We will not assume that log files include information about cacheability of entities, i.e., we assume that transactions in our data mart refer to cacheable documents. Second, we abstract from cache consistency by assuming that each transaction in the data mart includes the size of the entity at the time of the request. This means that an entity in cache is considered to be *fresh* if its size coincides with the size of the entity at the time of the request. Using the HTTP/1.1 terminology, we assume the entity size as a validator (see Section 3.3).

3. A data mart of web log data

We have developed a data mart for web logs specifically to support intelligent caching strategies. The data mart is populated starting from a web log data warehouse (such as those described in [18,31,35]) or, more simply, from raw web/proxy server log files that we assume containing some very basic fields. The data mart population consists of a number of preprocessing and coding steps that perform data selection, cleaning and transformation. The data mart has been implemented as a relational database, using Microsoft SQL Server 2000 [34]. Interestingly, also the processes of populating the data mart are formalized and automated within the same framework.

Log files produced by web/proxy servers are text files with a row for each HTTP transaction. We assume that an example row follows the *common log file format* [45], which includes the following information:

```
... 213.213.31.41 [15/Apr/2000:04:00:04 +0200] "GET http://www.un-
ipi.it/images/h/h_home.gif HTTP/1.1" 200 1267 ...
```

Most of the fields accomplish the HTTP standards [43,44]. In the example above, `213.213.31.41` is the IP address of the client (the computer originating the request), `15/Apr/2000:04:00:04` is the date/time of the transaction, `GET` is the *method* of the transaction, `http://www.unipi.it/images/h/h_home.gif` is the URL of the resource requested by

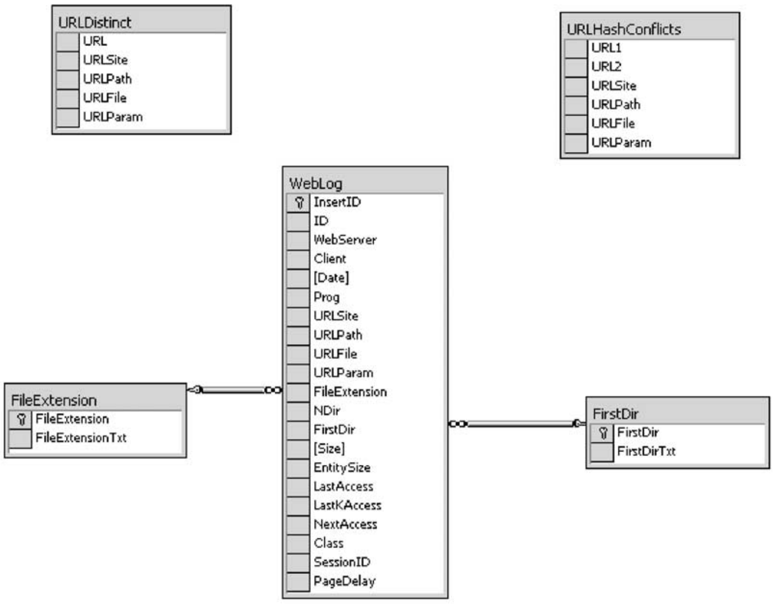


Fig. 2. Data mart tables and relations.

the client, `HTTP/1.1` is the HTTP protocol used by client, `200` is the HTTP return code (`200` means OK), and `1267` is the size in bytes of the response sent to the client.

It is worth noting that we are assuming a set of basic fields, which is normally available from web/proxy servers. Additional information may be present, either in transaction logs or on-line, such as HTTP headers describing transmission parameters or content properties. At one extreme, the whole resource is available on-line, so that (web mining) techniques may be adopted to provide deep information on the structure of a resource (e.g., incoming and outcoming hyperlinks, degree of relevance or authority) or its content (such as a classification into interest categories – e.g., news, science, sport, etc.). While we think that it is worth investigating such a situation, in this work we restrict evaluation of our approach under the assumption that such deep information is not available or it is computationally expensive to obtain or to use.

3.1. A data mart for mining log data

The data mart is implemented as a relational database, whose schema is shown in Fig. 2. The main table of the data mart database is `WebLog`. Each row of this table stores a transaction from a web/proxy server. Let us describe some of its fields.

`InsertID` is a unique identifier of the transaction as it is inserted in the database.

`ID` is a unique identifier of the transaction that respects the temporal date/time order of transactions.

`WebServer` identifies the real [1] server from which the transaction comes.

[1] For load balancing reasons, web/proxy servers are often implemented as a network of computers (called *real servers*) and a switch that routes client's requests.

`Prog` is a unique identifier of the order of the transaction among all transactions at the same time. This information is needed to maintain the order of transactions in the relational database.

`URLSite`, `URLPath`, `URLFile`, `URLParam` are hash codes that identify the requested URL, where the components of host name, path, file name and parameters are separated. The need for hash coding will be discussed later on.

`Size` is the number of bytes sent to the client.

`EntitySize` is the size of the requested entity at the time of the request. It is not a basic field in our raw file logs. It is computed as will be described in Section 3.3.

`FileExtension` is a numeric code of the file extension in the requested URL.

`NDir` is the depth in the server file system of the requested resource. For instance, the `NDir` value of `/images/h/h_home.gif` is 2.

`FirstDir` is a numeric code of the top directory in the path name of the requested resource. For instance, the `FirstDir` value of `/images/h/h_home.gif` is (the code of) `images`.

The remaining fields are particularly suited for data mining analysis and will be described in depth later. In addition, the database contains three tables that are useful to *decode* the hash values into text URLs, and to decode the numeric values of `FileExtension` and `FirstDir`. Since hash functions may assign the same code to two or more URLs (which means that those URLs will be considered as the same in all queries to the data mart and in all phases of the data mining analysis), the distinct pairs of conflicting URLs are stored in a table, named `URLHash-Conflicts`, in order to evaluate the need for more powerful hash codings. It is worth noting, however, that once the hash function has been demonstrated sufficiently powerful, the decoding tables and the `URLHashConflicts` table are unnecessary.

3.2. Data mart population

The processes of data preparation and data mart population have been designed using SQL Server 2000 Data Transformation Services (DTS), a tool that allows the specification of import/export/transformations processes of data through text files, databases or applications. The tasks are defined in a few DTS packages. One is `MineFast010_Part1`, shown in Fig. 3. The central task of the package is a Perl script that performs preprocessing of raw file logs. According to the assumptions of Section 2.2, we select transactions such that: the request method is "GET", i.e., client requests for resources; and the HTTP return status is "200", which means *OK*; and the requested URL does not contain parameters, i.e., dynamically generated entities are not considered. [2] On the contrary, HTTP transactions with methods "POST" and "PUT" are not considered since they are not cacheable [44]. Also we omit considering "HEAD" transactions and "GET" transactions involving partially successful responses, redirections, or client/server errors.

For the selected transaction, URLs are first *normalized*, then split into fields of information (site, path, file, parameters), which are then coded into integers using a hash function. According to the standards [43,44], the syntax of an http URL is:

```
'http://' host.domain [':'port] [ abs_path ['?'query]]
```

[2] Note, however, that there may be entities generated dynamically by resources whose URLs contains no parameters, e.g., by considering cookies.

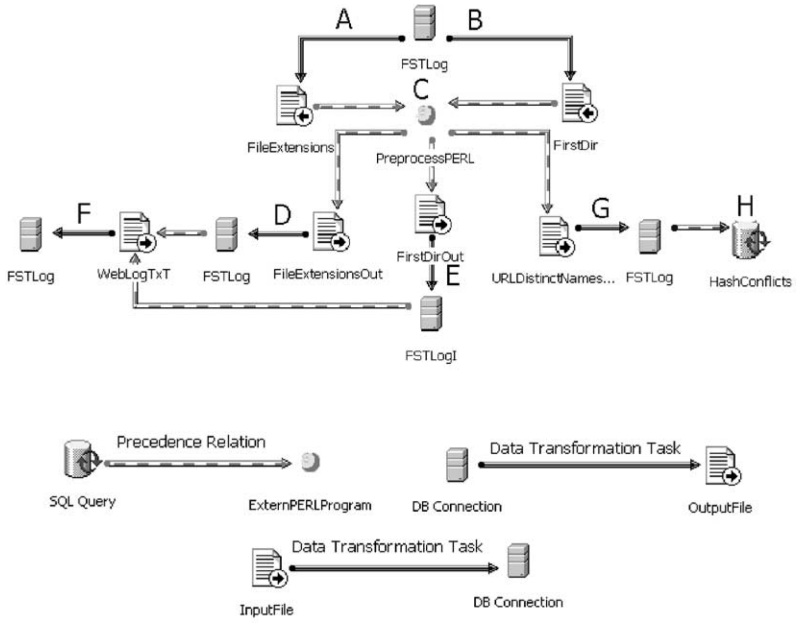


Fig. 3. The `MineFast010_Part1_Load` package. (Below: Legend of symbols in SQL Server DTS packages.)

where

- `host.domain[:port]` is the name of the server site;
- `abs_path` is the absolute path of the requested resource in the server file system. We further consider `abs_path` of the form `path ['/' filename.fileextension]`;
- `query` is a collection of parameters, to be passed as an input to a resource that is actually an executable program, e.g., a CGI script.

For instance, in the example URL:

```
http://www.UNIpi.IT/kdd/public/intro.html?State=base
```

we distinguish the site server (`www.UNIpi.IT`), the path relative to the file system of the server (`kdd/public`), the requested file name and extension (`intro` and `html`), and some parameters (`State=base`). During preprocessing of log files, a set of normalizations (using some heuristics) is performed on URLs. These are necessary in order not to distinguish two syntactically distinct URLs that actually refer to the same resource.

Finally, the need for URL coding traces back to problems and inefficiency of databases management systems and data mining algorithms to work with strings. Among the advantages of using a numeric coding, we mention optimization of disk space, efficient comparison and sorting, and privacy concerns. As an additional requirement, since the coding mechanism may be employed on-line for implementing intelligent caching strategies, it should be both efficient and allow for coding a URL independently of the coding of other URLs. For the reasons above, each of the fields among site, path, file name and parameters of a normalized URL is coded using a hash function into 32 bit integers. The hash function is obtained as a bit-to-bit *xor* of the 128 bit code returned by the MD5 hashing algorithm [38].

3.3. *Approximating* `EntitySize`

In Section 2.2, we assumed the availability of the size of the requested entity at the time of the request. However, such a field is not basic in our log data. We construct the field `EntitySize` by the following heuristics. For every transaction in the data mart, we assign to `EntitySize` the maximum value of `Size` in the previous 100 requests to the same URL of the transaction. If, at the end, a request records a zero `EntitySize`, we discard it from the data mart. Assume that an entity size never changes or increases only. If clients always receive the whole entity, the heuristics correctly assigns the `EntitySize` field. If some client closes the connection before receiving the whole entity, then we assign the correct entity size if in at least one out of 100 previous requests some client receives the whole entity. Assume now that an entity size decreases at some time. Then our heuristics assigns an incorrect entity size from that request up to the next 99 requests. Some form of approximation is unavoidable for any on-line cache consistency algorithm that must estimate the expiration time of entities.

We are now in the position to assign a unique identifier `ID` to transactions according to their date/time, the order of registration in the log file (stored in the `Prog` field) and the real web server identifier. Note that, with such an order, we assume that, at a fixed date/time, the order of arrival of requests to real web servers is interleaved.

3.4. *Additional data mart fields*

In addition to the fields described so far, the `WebLog` table includes further information particularly suited for the data mining algorithms that will be described later on. In particular, the fields `LastAccess` and `NextAccess` count how long in the past and in the future the same URL is requested. The count is absolute, i.e., it considers the number of requests. Note that `LastAccess` is set to a large constant ∞ if there is no previous request in the data mart. Analogously, `NextAccess` is set to ∞ in case there is no further access in the data mart. The field `LastKAccess` counts the number of accesses to the same URL in the last K requests, where K is a parameter definable by the data mart user. Since K is fixed for all transactions, `LastKAccess` actually denotes frequency of a requested URL in the previous K requests.

`Class` is a field freely definable by the data mart user, e.g., using SQL `UPDATE` queries. In particular, it will be used in one of the data mining algorithms to store a classification of transactions.

`PageDelay` is the distance in milliseconds of a request from the previous request of the same user. `PageDelay` is set to a sufficiently large constant ∞ in case of the very first request of an user. Finally, `SessionID` is a unique integer identifier of the user session the transaction belongs to. The concept of user session deserves a detailed discussion.

From a semantical point of view, a user session could be defined as the set of URLs accessed by a user for a particular purpose. Since it would be impossible to distinguish the "purpose" of accessing an object without information on the content of the object, one usually restricts consideration of some heuristics [16–18].

Often, a fixed *time window* is used to break a user clickstream into user sessions. Another heuristics is the *maximal forward reference*, where the session is identified as the maximum number of links followed by the user before a backward reference is made. Here, we adopt the *reference length* approach, which is based on the assumption that the amount of time a user spends examining an object is related to the interest of the user in the object contents. On this basis, a model for user sessions could be built by distinguishing the *navigational* objects (i.e., containing only links interesting to the user) from the *content* objects (i.e., containing the information the user was looking for). The distinction between navigational and content accesses is related to the distance (in time) between a request and the next one. If between two accesses `A` and `B` there is a time delay greater than a given threshold (to be estimated), then `A` can be considered as a *content* URL; otherwise, it is a *navigational* URL. On this basis, a user session is a sequence of navigational URLs followed by one content URL. Starting from `PageDelay`, we assign a `SessionID` to each URL request by grouping requests whose distance is below a given threshold.

Note that we identify users with IP addresses. This assumption, however, is, again, an heuristics. It fails in case IP corresponds to proxies, or dynamic IPs, or computers shared among many users. A correct identification of users is only possible in the presence of some form of authentication (e.g., login or cookies).

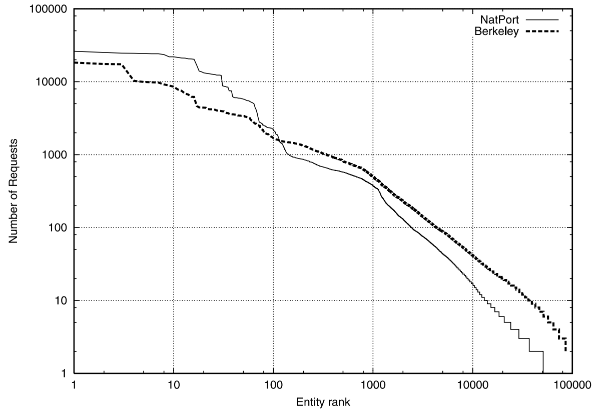
3.5. Characterization of workloads used in experiments

There is an extensive literature on understanding the statistical characteristics of workloads of web/proxy servers [7,9,12,24,30] and their impact on caching policies [7,28,29]. In this section, we introduce two workloads that will be used in experimentation and show that their statistical distributions reflect the typical characteristics reported in the literature, with particular reference to the work of Arlitt and Williamson [7].

We have populated the data mart with data from two web servers:

- `NatPort`: four days' log data from a leading Italian portal web server, for a total of 2.27 million requests and for a total of 29.7 Gbytes transmitted. There are 81,079 IPs (no proxy) and 85,888 distinct entities (i.e., normalized URLs), for a total web server size of 3.8 Gbytes.
- `Berkeley`: the July 2000 log data from the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley [22], for a total of 3.1 million requests and for a total of 70.2 Gbytes transmitted. There are 153,812 IPs (including proxies) and 125,412 distinct entities, for a total web server size of 12.6 Gbytes.

In both workloads, the MD5 hash coding algorithm gave rise to zero conflicts. Among the *invariants* observed by Arlitt and Williamson [7], we can confirm the following: the predominance of requests for `html` and multimedia documents (97% for `NatPort` and 96% for `Berkeley`); the small median transfer size (13.7 Kbytes for `NatPort` and 23.7 Kbytes for `Berkeley`); the small fraction of distinct resources (i.e., normalized URLs) requested over the total number of requests (3.8% for `NatPort` and 4% for `Berkeley`); the significant percentage of resources requested only once (41% for `NatPort` and 21% for `Berkeley`). In addition, most of the cited papers observe a *concentration of references*, i.e., with most of the requests referring to a small fraction of URLs. This is reflected in the distribution shown in Fig. 4 (left) of references over entity rank, i.e., from the most-requested entity to the less-requested ones. Those distributions follow a Zipf law: $f \sim r^{-\alpha}$, where f is the number of requests to an entity, r is the entity rank, and α is a parameter ($\alpha = 1.35$ for `NatPort` and $\alpha = 1.14$ for `Berkeley`).



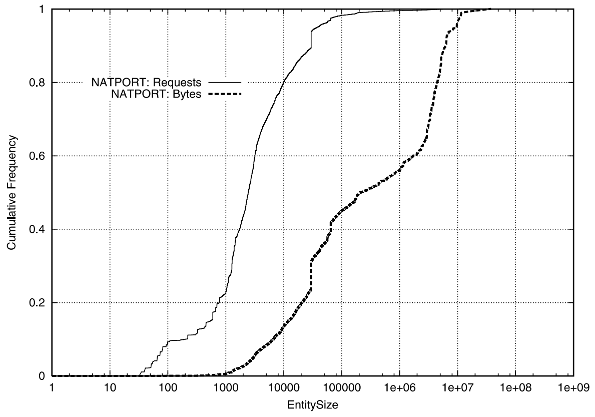
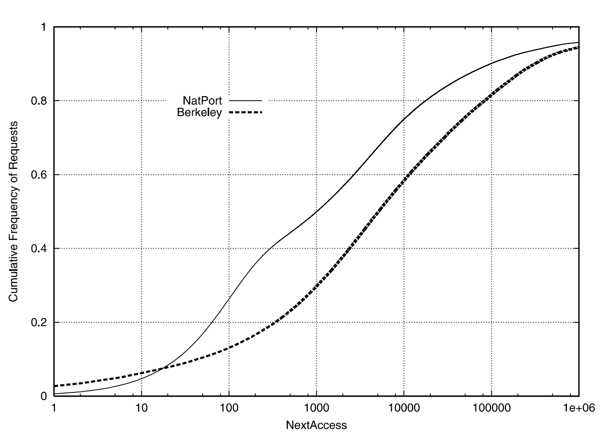


Fig. 4. Distribution of requests over entity rank (left). Distributions of requests and bytes transmitted over entity size (right).

Moreover, Arlitt and Williamson [7] observe that the cumulative frequency of requests over `EntitySize` is *heavy-tailed.* A similar observation holds for the cumulative frequency of bytes transmitted, yet the distribution seems to be less heavy-tailed. Such characteristics are present in our workloads, as shown in Fig. 4 (right) for the `NatPort` workload. Entities with size between 1000 and 100,000 bytes account for almost all requests, but only for less than 45% of bytes transmitted.

Also, *locality of references* has been observed, i.e., with many requests for a resource within a small amount of time. This is confirmed in our workloads. Fig. 5 shows the distribution of requests over `NextAccess`. In the case of `NatPort`, more than 75% of the time the next request for a URL occurs within 10,000 requests.

Finally, let us consider user sessions. We have already mentioned that the presence of proxies could confuse distinct user sessions into a single session. The `NatPort` data, however, show no IP with a predominant number of requests. On the contrary, for the `Berkeley` workload the top



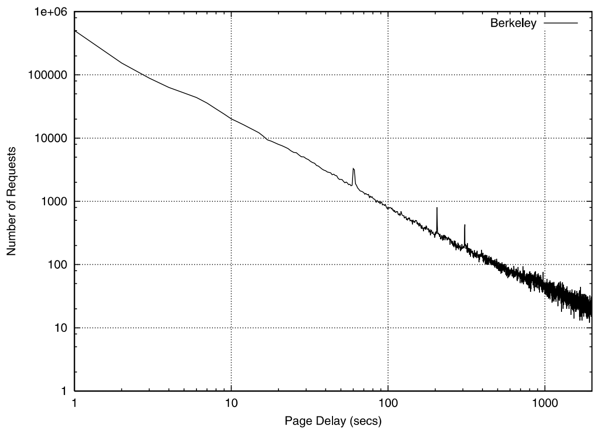


Fig. 5. Distribution of requests over `NextAccess` and over `PageDelay`.

three IPs account for 15% of the total number of requests. Fig. 5 relates Berkeley distribution of requests to PageDelay (in seconds), i.e., the time distance between a request and the previous one by the same client. The plot fits well the distribution $f \sim p^{-\alpha}$, where f is the number of requests to an entity, p is the PageDelay value (in seconds) and α is a parameter ($\alpha = 1.25$ for NatPort and $\alpha = 1.56$ for Berkeley). This confirms our expectation that most user accesses are navigational.

4. Deploying the reference model with data mart data

We present in this section two instantiations of the general intelligent caching model of Fig. 1, the first one based on frequent patterns, and the second one on decision trees. For each of them, a brief introduction to the general mining task is given, followed by the description of its application to the caching strategy and a summary of the results of experimental simulations.

For both the approaches, the results of simulations are obtained by building and then simulating an *extended LRU* over the two workloads introduced in Section 3.5. Each workload is split into two datasets of almost the same size: the *training set* is used for building a data mining model, while the *validation set* is used for validating the extracted model. [3] We simulate the extended LRU strategy over the validation set for cache sizes from 0.2 to 409.6 Mbytes (corresponding to 0.005–3.2% of Berkeley server size, and to 0.0015–10.5% of NatPort server size), with exponential growth. Fig. 6 shows the hit and weighed hit rates of traditional caching strategies on the validation set of Berkeley. Notice that all strategies tend to a limit hit/weighted hit rate as cache size grows. Such limits are the maximum hit rate and weighted hit rate achievable with a sufficiently large cache. For instance, for the Berkeley workload the limit hit rate is 93.2% and the limit weighted hit rate is 88.2%.

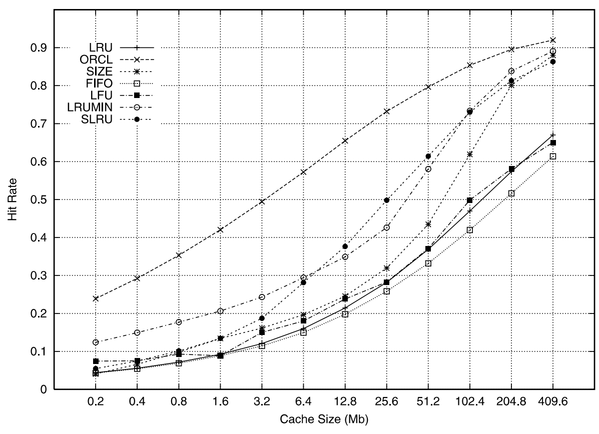
4.1. Frequent patterns discovery

In this section, we introduce the notion of frequent patterns discovery, as defined in [4,26], and discuss their use in intelligent web caching. We mainly consider two forms of frequent patterns:

- *association rules*, where (general) correlations among items in user transactions are discovered (in our context, a user transaction is a user session), and
- *sequential patterns*, where temporal correlations among items in user transactions are discovered.

Consider a user transaction as a set of items. Given a database of user transactions, an *association rule* is a rule of the form $A \Rightarrow B[S, C]$, where A and B are sets of items; S is the *support* of the rule, defined as the percentage of user transactions containing all items in $A \cup B$; C is the *confidence* of the rule, defined as the ratio of S to the percentage of transactions containing all items in A. In probabilistic terms, S approximates the probability that all items in the rule appear together in a transaction, while C approximates the (conditional) probability that items in B appear in a transaction given that items in A appear. Since the number of association rules which can be built over a set of transactions grows exponentially with the number of possible items, usually only rules with support and confidence greater than some given thresholds (respectively called *mini-*

[3] Note that we are actually experimenting one **do** ... **while** cycle of the model in Fig. 1.



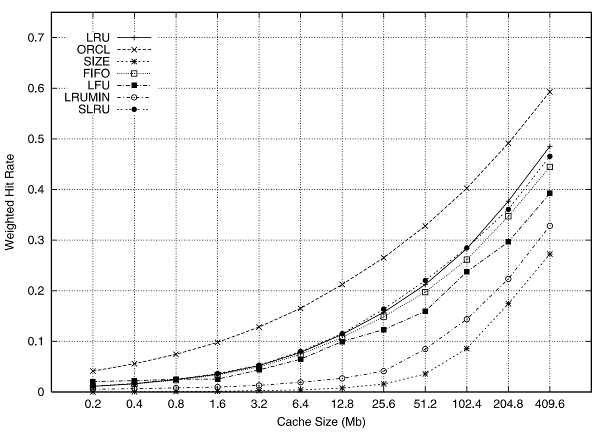


Fig. 6. Berkeley: hit and weighted hit rates w.r.t. cache size.

mum support and *minimum confidence*) are considered. In our application domain, items correspond to web resources, while user transactions correspond to user sessions. Thereby, a rule such as $\texttt{res}_1 \Rightarrow \texttt{res}_2$ informally means that if $\texttt{res}_1$ appears in a user session, $\texttt{res}_2$ is expected to appear in the same session, though possibly in the reverse order and/or not necessarily consecutively.

Association rules do not take into account the time component. For this reason, we will also use an adaptation of sequential patterns discovery to our context. Consider a user session as a time-ordered sequence of requested resources. Given a database of user sessions, a sequential pattern is an expression of the form $res_1 \lhd res_2 \lhd \cdots \lhd res_n[S]$, where res_i ($i \in \{1, \ldots, n\}$) is a resource; S is the support of the sequential pattern, defined as the percentage of user sessions containing the subsequence $res_1, res_2, \ldots, res_n$.[4] As for association rules, we are interested in *frequent sequential patterns*, i.e., we only consider patterns with a support greater than a given minimum threshold.

In this paper, we experiment association rules and sequential patterns of restricted form, mainly due to the concerns of designing a cache strategy whose implementation is efficient on-line. On the one hand, we only consider sequential patterns of length 2, i.e., of the form $res_1 \lhd res_2$. Also, we only consider one-to-one association rules, i.e., of the form $res_1 \Rightarrow res_2$.

4.1.1. Frequent-pattern-based intelligent caching

The LRU strategy essentially consists of assigning to the requested entities a time-increasing priority value (e.g., the value of a counter which is incremented at each request). In our approach we keep the LRU criteria for assigning priorities, while the overall strategy is *extended* by modifying the priorities of the entities already in cache as a *reaction* to the new incoming requests.

First of all, a set of sequential patterns is extracted from the training set, composed of past log data. Consider now new requests. If resource A is requested and $A \lhd B$ is in the set of previously

[4] Formally, a sequence $B_1, B_2, \ldots, B_n$ contains the subsequence $A_1, A_2, \ldots, A_m$ if there exists $B_{k_1}, B_{k_2}, \ldots, B_{k_m}$ such that: (1) for $i \in [1, m-1]$, $k_i < k_{i+1}$; (2) for $i \in [1, m]$, $A_i = B_{k_i}$.

extracted patterns, then we *predict* that the resource B will be requested soon. This suggests that if B is already in cache, its eviction should be delayed: we accomplish that by assigning to B the priority it would have if it was requested immediately after A.

The above strategy is characterized by two (related) parameters: the minimum support (specifying the *local* efficacy of a sequential pattern) and the number of patterns (specifying the *global* efficacy of the approach). A large number of patterns with a high support provides a substantial modification of the LRU strategy: the *impact* of each pattern is quantified by its support, while the number of patterns provides a quantification of the number of resources that are influenced.

A problem that may raise is the management of a large number of patterns. In our implementation, we use a hash table to guarantee efficient access to the patterns to examine. Each bucket of the hash table represents a URL, and points to the list of all patterns $A \triangleleft B$ having the URL A as antecedent. Note that the number of patterns $A \triangleleft B$ may be large as well: this may lead to a large number of weight re-assignments to resources B in cache. In order to bound those re-assignments, we adopt a support-based pruning of the extracted patterns. An interesting comparison can be done with an approach using association rules instead of sequential patterns. We now extract association rules from the training set, and modify the priority of a resource B in cache when A is requested and the rule $A \Rightarrow B$ is in the set of extracted rules. The rule $A \Rightarrow B$ subsumes (modulo the confidence threshold) the sequential pattern $A \triangleleft B$. Hence, by comparing the two strategies, we evaluate whether temporal correlation improves over pure correlation.

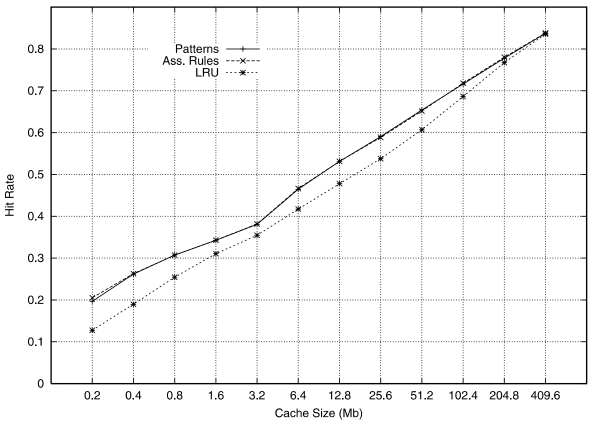
4.1.2. Results of experimentation

The `NatPort` workload is split into a two days' training set and a two days' validation set: the first two days of log data have been used for extracting sequential patterns over user sessions, with minimum support threshold ranging from 2.5% to 0.153125%, and sessions with maximum `PageDelay` equal to 240 s. The resulting extended LRU strategy has been simulated over the remaining two days of log data.

Fig. 7 shows some of the results obtained by simulating an LRU cache, a *pattern-based extended LRU* and an *association-based extended LRU* . The left-hand side of Fig. 7 shows hit rates. The three curves have a similar shape, and, as expected, converge as the cache size grows. The graph reveals a significant hit improvement of our two novel approaches over the standard LRU for all cache sizes, ranging from an absolute gain [5] of +8.28% (with a cache of size 25.6 Mbytes) to +0.55% (for the largest cache). The right-hand side of Fig. 7 shows an analogous plot for the weighted hit rate. Here the improvement is smaller (from an absolute +3.01% to +0.2%) due to the fact that the pattern- and association-based approaches try to maximize only hits, not taking into account the size of the cached entities.

The hit rates of the pattern-based and the association-based extensions are almost identical, and the weighted hit rates are very similar. This means that for the `NatPort` workload, the temporal aspect does not play a relevant role in the extracted patterns. The `Berkeley` workload shows a different behavior. It has been split into a 15 days' training set and a 16 days' validation set. Since the network traffic per day is much smaller than in the previous workload (less than a

[5] Here and in the rest of the paper, *absolute gain* and *absolute improvement* stand for the difference between two hits rates.



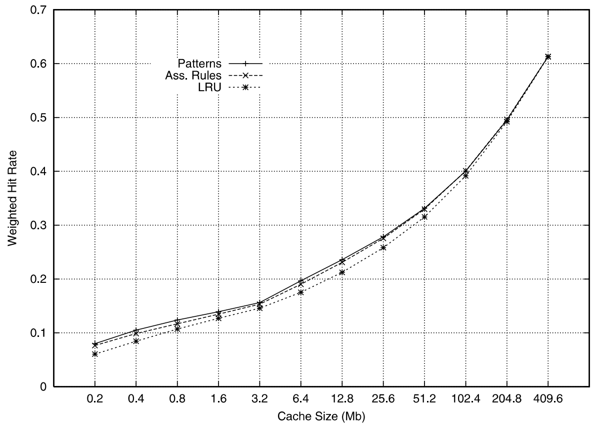
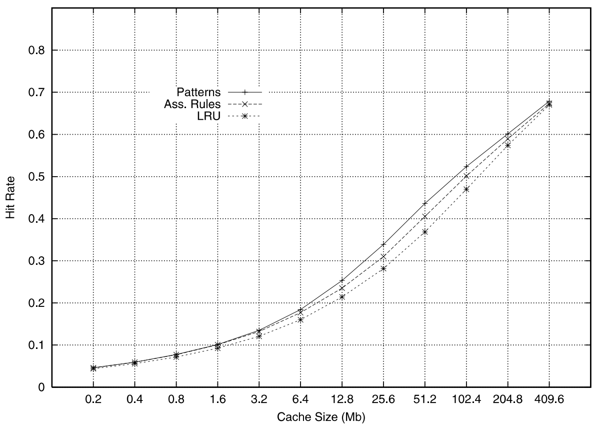


Fig. 7. NatPort: hit rates and weighted hit rates w.r.t. cache size.

third) and it is caused by a larger number of clients, we need to choose a larger time window for reconstructing user sessions of significant length. Different thresholds for PageDelay have been tried, and the value of 8 minutes has been chosen, which yields reasonable sessions.

Another factor that may influence the results for the Berkeley workload is the large percentage of requests originated from a few proxies. A potentially large set of real clients may be hidden behind a proxy, so several sequences of requests coming from different but not distinguishable clients are mixed. Indeed, sessions longer than 1000 requests cover more than 20% of the requests. As a consequence, many misleading patterns can be extracted, thus compromising performances. In order to avoid such a drawback, we imposed a maximum session length threshold: sessions longer than a given user-defined threshold (set to 100 requests in our experiments) are broken into shorter ones. We used lower support thresholds than in the NatPort workload in order to extract the same number of rules from the training set. Fig. 8 shows the results of simulations. We obtain a maximum absolute gain of +6.74% for hit rate, and of +2.83%



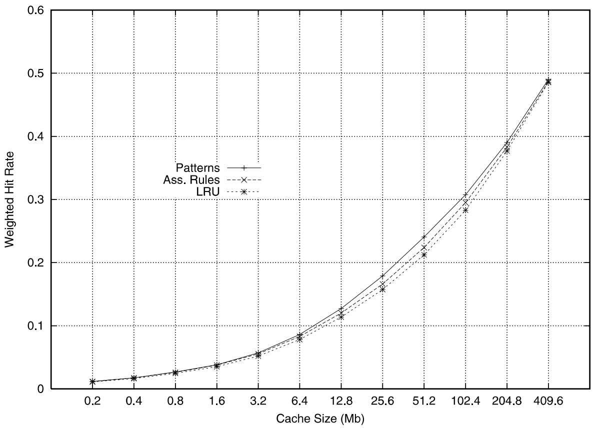


Fig. 8. Berkeley: hit rates and weighted hit rates w.r.t. cache size.

for weighted hit rate. Here the pattern-based strategy yields a hit rate significantly higher than the association-based one, particularly for caches of medium size.

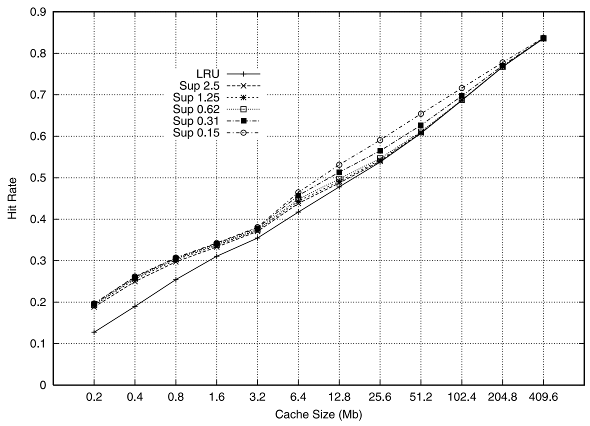
Differently from the `NatPort` workload, the enhanced strategies perform poorly on the `Berkeley` workload on small-sized caches. Here is our intuition on such a behavior. When a resource A is requested and the pattern $A \lhd B$ is considered, if B is not in cache then no action is taken. In small-sized caches, the probability of having B already in cache is low, especially for workloads with many distinct resources, as in the case of `Berkeley`. Therefore, the intelligent strategies rarely re-assign weights. As cache size grows, the probability that B is in cache increases, and then the intelligent strategies re-assign weights more often.

Finally, it is worth investigating the global efficacy of the approach. Fig. 9 show how performances scale with respect to patterns minimum support. In both plots of Fig. 9, supports are tuned in order to obtain a sufficiently large number of patterns (at least 36,000). Lower support values yield better performances. However, when the minimum support is extremely low, the strategies lead to the worst performances.

4.2. Decision trees

In this section, we recall the notion of predictive classification models based on decision trees, and show how to use them to enhance the LRU strategy.

A *classifier* is a model that describes a discrete attribute, called the *class*, of an entity in terms of other attributes of the entity, called the *observed attributes*. In supervised learning, a classifier is constructed from a set of entities (the *training set*) whose class values are known, and can be used to predict the unknown class values of entities in another set. Among the most popular classification models, we use decision trees for our analysis, and, in particular, the well-known C4.5 [37,40] algorithm. A *decision tree* is a tree data structure consisting of *decision nodes* and *leaves*. A decision node specifies a *test* over one of the observed attributes. For each possible outcome of the test, a child node is present. A path from the root to a leaf of the decision tree can be followed based on the observed attribute values of an entity. The class specified at the leaf is the class *predicted* by the decision tree.



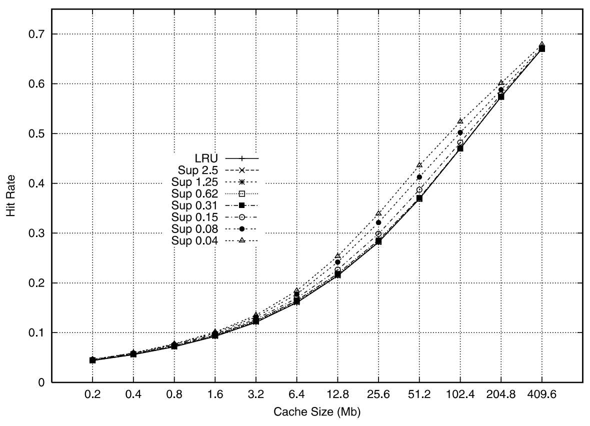


Fig. 9. `NatPort` (left) and `Berkeley` (right): hit rate w.r.t. minimum support.

What is in our context a useful notion of class for intelligent caching? We observe that the field `NextAccess` (the distance from the next access to a URL) described in Section 3.4 can be used to implement the off-line ORCL caching strategy. We recall that ORCL assigns the weight to an entity as the product of `EntitySize` and the distance from the *next* request to the entity. However, `NextAccess` can only be calculated for historical data, as it is obviously not available on-line. The central idea of this section is to train a classifier, in the form of a decision tree, that is able to *predict* the value of `NextAccess`, which is therefore chosen as the class variable. Since `NextAccess` is a continuous attribute, we use a discretization of its values into a few classes.

4.2.1. Classification-based intelligent caching

In order to use decision trees as data mining models instantiating the general reference model, we are required to make choices on the following parameters: the discretization method to be applied on `NextAccess`, the set of observed attributes to be used in constructing decision trees, and the way a decision tree is integrated in the caching strategy weight assignment policy.

In this paper, we mainly present one selected strategy, called S2. The S2 strategy uses a general rule for defining classes and weights, in order to scale up over cache size. The intuition is that, if we intend to approximate the ORCL strategy, we must train a decision tree and use a weight assignment formula that is tightly related to the behavior of ORCL on the training set for a given cache size. First of all, let us fix some notation:

- **ORCLCache**(s) is a cache of size s that uses the ORCL strategy and is run on requests from the training set,
- **ORCLAvgDSize**(s) is the average entity size in ORCLCache(s),
- **ORCLTertile**(t, s), where $t \in \{1, 2, 3\}$, is the number n such that $t \cdot 33.33\%$ of the time ORCLCache(s) contains no more than n entities, and
- **ORCLMax**(s) is the maximum number of entities in ORCLCache(s), or, in other words, ORCLTertile($3, s$).

Note that these values are computed on the training set of a workload, where the field `Next-Access` is available; hence the ORCL strategy can be simulated.

Discretization of `NextAccess`. This choice means both deciding the number of classes and how to discretize values for `NextAccess` into intervals. We experimented several possibilities, with the result that a small number of classes turns out to be preferable. The S2 strategy uses a discretization into four classes, defined as follows:

- **Class 0**, defined as `NextAccess` $\in [1, \mathrm{ORCLTertile}(1, s)[$,
- **Class 1**, defined as `NextAccess` $\in [\mathrm{ORCLTertile}(1, s), \mathrm{ORCLTertile}(2, s)[$,
- **Class 2**, defined as `NextAccess` $\in [\mathrm{ORCLTertile}(2, s), \mathrm{ORCLMax}(s)]$,
- **Class 3**, defined as `NextAccess` $\in]\mathrm{ORCLMax}(s), \infty]$.

The intuition behind this choice is that, if we intend to approximate the ORCL strategy, we should be able to predict how likely ORCL is to successfully cache a resource that will appear again after `NextAccess` requests. This probability depends on the number of entities the ORCL cache may hold for a cache size of s. So, if `NextAccess` is in the interval $[1, \mathrm{ORCLTertile}(1, s)[$, then 66.66% of the time the ORCL cache may hold (in the training set) at least `NextAccess` entities. Assume now a request with `NextAccess` in the interval above. In the extreme case that all intermediate requests refer to distinct resources, this means that there are from 66.66% to 100% possibilities the entity may remain in the ORCL cache until its next reference. The possibilities reduce to 33.33–66.66% if `NextAccess` is in the interval [ORCLTertile$(1,s)$, ORCLTertile$(2,s)$[; and they reduce to 0–33.33% if `NextAccess` is in the interval [ORCLTertile$(2,s)$, ORCLMax(s)]. Summarizing, a simulation of the ORCL strategy should assign higher priority to requests with lower class values.

Weight assignment. Following the intuition of approximating the ORCL strategy, we adopt a weight assignment function that *corrects* the LRU weight by adding a displacement that is related to the priority of the request in the ORCL strategy. The LRU weight function on the ith request can be defined as $\mathscr{W}_{\mathrm{LRU}}(E_i) = i$, i.e., as assigning weight i to the ith request for an entity E_i. Here, a low weight value means low priority. The ORCL priority of an entity E_i at each time is inversely proportional to $\Delta' T$, i.e., the number of requests that will be received until the next request of the entity – in our context, `NextAccess` – and it is inversely proportional to the size, $E_i.size$, of the entity – in our context, `EntitySize`. Therefore, we would like to define a weight function for S2 of the form: $\mathscr{W}_{\mathrm{S2}}(E_i) = i + \alpha(\Delta' T) \cdot \beta(E_i.size)$, for some inversely proportional functions $\alpha()$ and $\beta()$.

Concerning $\alpha(\Delta' T)$, we do not know $\Delta' T$ on-line, alias `NextAccess`. We can approximate it by means of the class $c \in [0,3]$ predicted for E_i by some decision tree. We approximate $\alpha(\Delta' T)$ as $\alpha(c)$, where: $\alpha(3) = \mathrm{ORCLMax}(s)$, $\alpha(2) = 2 \cdot \alpha(3)$, $\alpha(1) = 2 \cdot \alpha(2)$, $\alpha(0) = 2 \cdot \alpha(1)$. Intuitively, we assign $\alpha(3) = \mathrm{ORCLMax}(s)$ since if the predicted class is correct then $\Delta' T$ is at least ORCLMax(s) – this is immediate by the discretization adopted for S2. Moreover, since $\alpha()$ must be inversely proportional, we assign $\alpha(c+1) = 2 \cdot \alpha(c)$ for $c \in [0,2]$ – i.e., giving to class $c+1$ twice the value we assign to class c. The formula $\alpha(c+1) = 2 \cdot \alpha(c)$ is related to the fact that requests in class $c+1$ have higher probability of being successfully cached by ORCL than requests in class c.

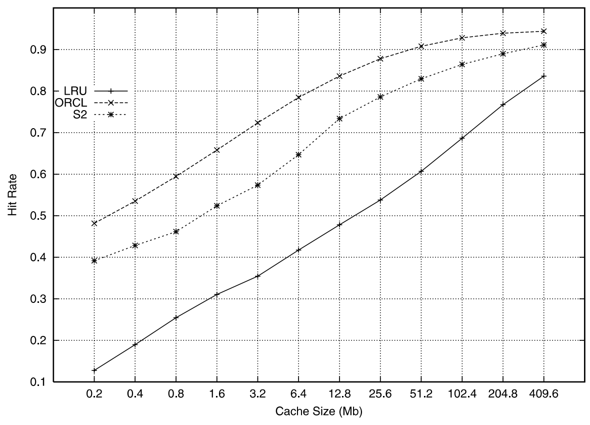
Concerning $\beta(E_i.size)$, we note that while $E_i.size$ is expressed in bytes in the ORCL strategy, the LRU assignment function *measures* "number of requests". Therefore, instead of $E_i.size$, we will use $E_i.size/ORCLAvgDSize(s)$, namely the number of requests that ORCLCache(s) would satisfy, in average, if there were $E_i.size$ free bytes. Since $\beta()$ must be inversely proportional, we choose experimentally $\beta(x) = 1/x$. Summarizing, we define $\beta(E_i.size) = ORCLAvgDSize(s)/E_i.size$.

Finally, it is worth observing that the weight assigned to an entity by $\mathscr{W}_{\mathrm{S2}}()$ does not change until the next reference. This makes the implementation of S2 efficient, and departs from ORCL, where the weight changes at any request.

Observed attributes. We must specify the set of observed attributes the tree is built on among the ones available at the time of URL request. Attributes such as `Client`, `Date`, `URLPath`, `URLFile`, `URLParam`, `RefSite`, `RefPath`, and `RefFile` are too fine-grained to be useful. On the contrary, `NDir`, `FirstDir`, `Hour`, `FileExtension`, `EntitySize`, `LastAccess`, and `LastKAccess` are suitable for selection. Strategy S2 uses the basic fields `EntitySize`, `NDir`, and the hour part of `Date` as continuous attributes, and the basic fields `FileExtension` and `FirstDir` as discrete ones.

4.2.2. Results of experimentation

Figs. 10 and 11 report the performances of strategies S2, LRU and ORCL in terms of hit rate and weighted hit rate for the validation sets of the two workloads. Although we consider splits into sets of approximately equal size, we mention that different splits (e.g., validation set twice or half of the training) have yielded similar results. The hit rate figure is very similar for both workloads, and exhibits an impressive improvement of S2 w.r.t. LRU: consistently around 25% absolute improvement. The simulation of ORCL achieved by S2 is rather impressive: the gain between LRU



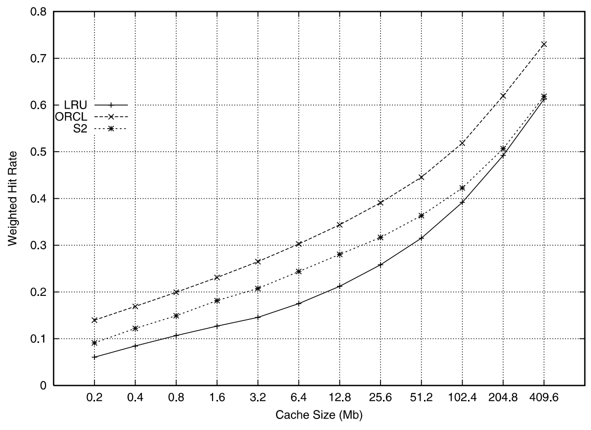
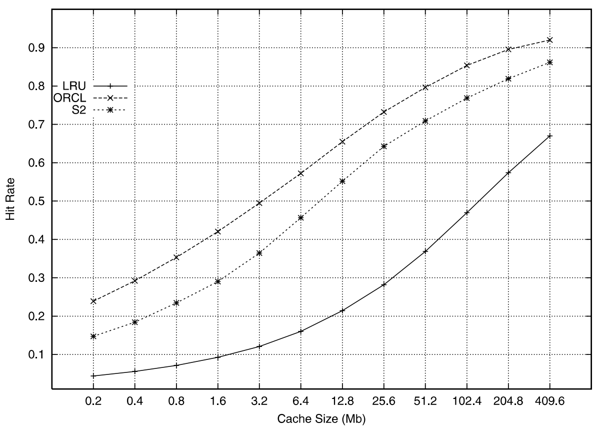


Fig. 10. `NatPort`: hit and weighted hit rates w.r.t. cache size.



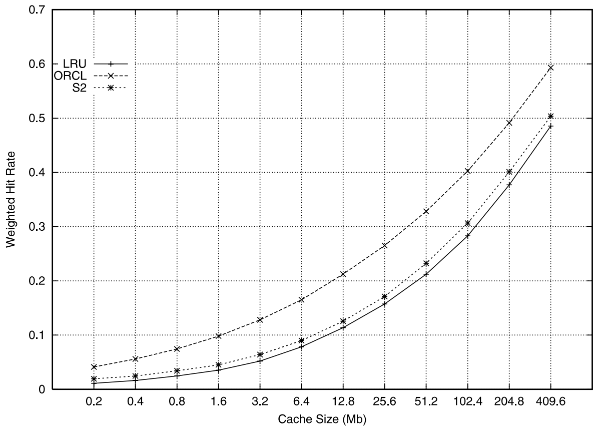
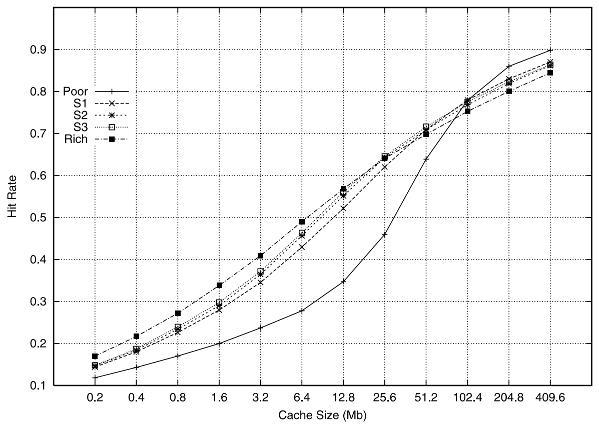


Fig. 11. `Berkeley`: hit and weighted hit rates w.r.t. cache size.

and S2 is consistently 50–75% of the gain between LRU and the off-line, theoretical strategy ORCL. Note that the achieved hit rate outperforms also the other traditional strategies reported in Fig. 6. The weighted hit rate performances are instead different in the two workloads. Although both performances improve on all traditional strategies (see, e.g., Fig. 6), there is a consistent gain w.r.t. `NatPort` for small-size caches, and a modest gain w.r.t. `Berkeley`. We attribute this behavior to the lower prediction accuracy of the decision trees for the `Berkeley` workload (from 70% to 80% of correctly classified requests on the validation set) compared to the `NatPort` workload (from 84% to 87% of correctly classified requests). It is then natural to ask ourselves what the limits are of the approach of strategy S2. We can partly answer this question by investigating the relevance of the decision tree under the assumption that all other parameters are fixed – i.e., discretization and weight formula $\mathscr{W}_{S2}()$. Let us consider the following variants of strategy S2:

- `Poor`: this is as strategy S2 except that the decision tree always assigns class 0 – i.e., it does not perform any distinction between entities.
- `S1`: this is as strategy S2 except that the decision tree is built on the only observed attribute `EntitySize`.



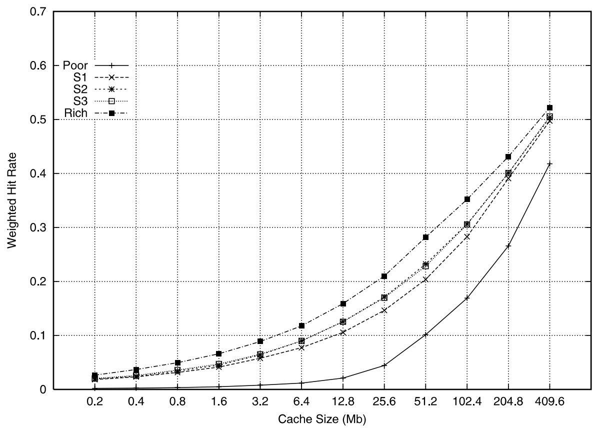


Fig. 12. `Berkeley`: hit and weighted hit rates w.r.t. cache size.

- `S3`: this is as strategy S2 except that the decision tree is built on the observed attributes used in S2 with, in addition, `LastKaccess` – the number of requests for a same URL in the previous K requests. We have experimented several values for K with similar experimental results. Here, we consider $K = 10 \cdot ORCLMax(s)$ for a cache size of s.
- `Rich`: this is as strategy S2 except that the decision tree always assigns the *correct* class an entity belongs to.

Intuitively, the strategies above show an increasing *predictive* accuracy of the classifier adopted. However, this does necessarily means that the strategies exhibit an increasing performance in terms of hit rate and weighted hit rate – due to the role of the other parameters in the definition of a strategy. Fig. 12 shows the performances of the strategies above on the `Berkeley` workload. Concerning the hit rate, we note that higher predictive accuracy means higher performance for cache sizes lower than 102.4 Mbytes. Also, it is worth noting that improving accuracy of the decision tree (or any other classification model!) used by S2 does not result into better performance. This implies that we must act on the other parameters of the strategy in order to improve hit rates. Consider now the weighted hit rate. For all cache sizes, higher predictive accuracy means higher performance. However, strategies S2 and S3 substantially coincide: this means that the additional information (i.e., the field `LastKAccess`, namely, frequency of requests) available in S3 does not contribute to improve the overall performance.

As a final comment on efficiency, on a 650 MHz Pentium III PC with 10,000 rpm hard disk, for the `Berkeley` workload and a cache size of 409.6 Mbytes, the overall S2 strategy took less than one hour. The decision tree produced can be stored in 1.5 Mbytes of memory and its depth is 20. On average, our prototype searched the decision tree 137,000 times per second.

5. Conclusions

We have presented two approaches to enhance LRU-based web server caching with data mining models (frequent patterns and decision trees) built on historical data. Also, the design of a suitable data mart has been presented, together with the main problems that such a design must solve. The performance figures of the developed methods, compared with LRU from one side and the theoretical off-line strategy ORCL from the other side, indicate substantial increase in the hit rate. Also, the decision-tree strategy S2 outperforms many traditional fixed strategies (SIZE, FIFO, LFU, LRU-Min, SLRU) both on hit rate and on weighted hit rate. Although further extensive benchmarking is required (tailoring to proxy server workloads, combination of the two approaches into one, extension of other traditional caching strategies, extension to dynamic resources), there is a strong indication that data-mining-based caching systematically enhances hit and weighted hit rates. All in all, the intelligence of our web caching strategy lies in the capability of tuning the cache replacement algorithm according to the recent history of requests extracted from the web log data: any fixed strategy, albeit extremely efficient in certain situations, is bound to get into difficulties as the pattern of web usage changes.

Acknowledgements

Research partially supported by FST – Fabbrica Servizi Telematici, Cagliari, Italy, under grant no. FR-22-2030 (*Web mining project MineFaST*). We also acknowledge support from Microsoft Research, Cambridge, UK, under grant no. 2000-23. We are grateful to the team at FST Research: R. Fenu, M. Magini, O. Murru, L. Petrella and L. Sannais, for giving us the opportunity to conduct this research, and for many useful discussions.

References

[1] M. Abrams, C.R. Standridge, G. Abdulla, S. Williams, E.A. Fox, Caching proxies: limitations and potentials, in: Proceedings of the 4th International World-Wide Web Conference, 1995, pp. 119–133.

[2] C.C. Aggarwal, J.L. Wolf, P.S. Yu, Caching policies for web objects, Technical Report RC20619, IBM Research Division, New York, 1997.

[3] C.C. Aggarwal, P.S. Yu, On disk caching of web objects in proxy servers, in: F. Golshani, K. Makki (Eds.), Proceedings of CIKM-97, ACM Press, New York, 1997, pp. 238–245.

[4] R. Agrawal, R. Srikant, Fast algorithms for mining association rules in large databases, in: Proceedings of International Conference on Very Large Databases (VLDB), 1994, pp. 487–499.

[5] S. Albers, S. Leonardi, On-line algorithms, ACM Computing Surveys 31 (1999).

[6] M.F. Arlitt, R. Friedrich, T. Jin, Performance evaluation of Web proxy cache replacement policies, in: Proceedings of the International Conference on Computer Performance Evaluation: Modelling Techniques and Tools (TOOLS), Lecture Notes in Computer Science, vol. 1469, 1999, pp. 193–206.

[7] M.F. Arlitt, C.L. Williamson, Internet Web servers: workload characterization and performance implications, IEEE/ACM Transactions on Networking 5 (1997) 631–645.

[8] A. Bar-Noy, R. Bar-Yehuda, A. Freund, J. Naor, B. Schieber, A unified approach to approximating resource allocation and scheduling, in: Proceedings of the 32nd Annual ACM Symposium on Theory of Computing, 2000, pp. 735–744.

[9] P. Barford, A. Bestavros, A. Bradley, M.E. Crovella, Changes in Web client access patterns: characteristics and caching implications, World Wide Web 2 (1999) 15–28.

[10] L.A. Belady, A study of replacement algorithms for virtual storage computers, IBM Systems Journal 5 (1996) 78–101.

[11] B. Berendt, M. Spiliopolou, Analysis of navigation behaviour in web sites integrating multiple information systems, VLDB Journal 9 (2000) 56–75.

[12] L. Breslau, P. Cao, L. Fan, G. Phillips, S. Shenker, Web caching and Zipf-like distributions: evidence and implications, in: Proceedings of the INFOCOM'99 Conference, 1999, pp. 163–174.

[13] P. Cao, C. Liu, Maintaining strong cache consistency in the world wide web, IEEE Transactions on Computers 47 (1998) 445–457.

[14] P. Cao, S. Irani, Greedydual-size: a cost-aware www proxy caching algorithm, in: Proceedings of the USENIX Symposium on Internet Technologies and Systems, 1997, pp. 193–206.

[15] P. Cao, J. Zhang, K. Beach, Active cache: caching dynamic contents on the web, in: Proceedings of IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98), 1998, pp. 373–388.
[16] M.S. Chen, J.S. Park, P.S. Yu, Data mining for traversal patterns in a web environment, in: Proceedings of the International Conference on Distributed Computing Systems, 1996, pp. 385–392.
[17] R. Cooley, B. Mobasher, J. Srivastava, Grouping web page references into transactions for mining world wide web browsing patterns, in: Proceedings of the IEEE Knowledge and Data Engineering Exchange Workshop (KDEX-97), 1997.
[18] R. Cooley, B. Mobasher, J. Srivastava, Data preparation for mining world wide web browsing patterns, Knowledge and Information Systems 1 (1999) 5–32.
[19] R. Cooley, P.-N. Tan, J. Srivastava, Discovery of interesting usage patterns from web data, in: Proceedings of the Web Usage Analysis and User Profiling Workshop, Lecture Notes in Computer Science, vol. 1836, 2000, pp. 163–182.
[20] B.D. Davison, Web caching and content delivery resources, http://www.web-caching.com.
[21] B.D. Davison, A survey of proxy cache evaluation techniques, in: Proceedings of the International Web Caching Workshop (WCW'99), 1999, pp. 67–77.
[22] Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, Berkeley CS HTTP logs, http://www.cs.berkeley.edu/logs/http.
[23] J. Dilley, M.F. Arlitt, Improving proxy cache performance: analysis of three replacement policies, IEEE Internet Computing 3 (1999) 44–50.
[24] B.M. Duska, D. Marwood, M.J. Feeley, The measured access characteristics of world-wide-web client proxy caches, in: Proceedings of the USENIX Symposium on Internet Technologies and Systems, 1997.
[25] O. Etzioni, The world-wide web: quagmire or gold mine?, Communications of the ACM 39 (1996) 65–68.
[26] J. Han, M. Kamber, Data Mining: Concepts and Techniques, Morgan Kaufmann, San Mateo, CA, 2000.
[27] S. Bradley, S. Sarawagi, U.M. Fayyad (Eds.), Internet Data Mining, ACM SIGKDD Explorations 2 (2000).
[28] S. Jin, A. Bestavros, GreedyDual* Web caching algorithms: exploiting the two sources of temporal locality in Web request streams, in: Proceedings of the International Web Caching and Content Delivery Workshop, 2000.
[29] S. Jin, A. Bestavros, Popularity-aware greedydual-size algorithms for web access, in: Proceedings of the International Conference on Distributed Computing Systems (ICDCS), 2000.
[30] S. Jin, A. Bestavros, Temporal locality in web request streams: sources, characteristics, and caching implications (poster), in: Proceedings of the SIGMETRICS 2000 Conference, 2000, pp. 110–111.
[31] K.P. Joshi, A. Joshi, Y. Yesha, R. Krishnapuram, Warehousing and mining web logs, in: Proceedings of ACM CIKM Workshop on Web Information and Data Management (WIDM'99), 1999, pp. 63–68.
[32] R. Kosala, H. Blockeel, Web mining research: a survey, ACM SIGKDD Explorations 2 (2000) 1–15.
[33] E.P. Markatos, Main memory caching of web documents, Computer Networks and ISDN Systems 28 (1996) 893–906.
[34] MicroSoft Corporation, MicroSoft SQL Server 2000, http://www.microsoft.com/sql.
[35] J. Han, O.R. Zaiane, M. Xin, Discovering web access patterns and trends by applying OLAP and data mining technology on web logs, in: Proceedings of the Advances in Digital Libraries Conference (ADL'98), 1998, pp. 19–29.
[36] J.E. Pitkow, M.M. Recker, A simple yet robust caching algorithm based on dynamic access patterns, in: Proceedings of the 3rd International WWW Conference, 1994.
[37] J.R. Quinlan, C4.5: Programs for Machine Learning, Morgan Kaufmann, San Mateo, CA, 1993.
[38] R. Rivest, The MD5 Message-Digest algorithm, Technical Report RFC 1321, The Internet Engineering Task Force, 1992, http://www.ietf.org/rfc/rfc1321.txt.
[39] L. Rizzo, L. Vicisano, Replacement policies for a proxy cache, IEEE/ACM Transactions on Networking 8 (2000) 158–170.
[40] S. Ruggieri, Efficient C4.5, IEEE Transactions on Knowledge and Data Engineering (to appear). http://www.computer.org/tkde/.
[41] M. Spiliopoulou, L.C. Faulstich, WUM: a web utilization miner, in: Workshop on the Web and Data Bases (WebDB98), Lecture Notes in Computer Science, vol. 1590, 1998, pp. 109–115.
[42] J. Srivastava, R. Cooley, M. Deshpande, P. Tan, Web usage mining: discovery and applications of web usage patterns from web data, ACM SIGKDD Explorations 1 (2000) 12–23.
[43] L. Masinter, T. Berners-Lee, R. Fielding, Uniform resource identifiers (URI): generic syntax, Technical Report RFC 2396, The Internet Engineering Task Force, 1998, http://www.ietf.org/rfc/rfc2396.txt.
[44] L. Masinter, T. Berners-Lee, R. Fielding, Hypertext transfer protocol – HTTP/1.1, Technical Report RFC 2616, The Internet Society, 1999, http://www.w3.org/Protocols.
[45] W3C httpd, Logging control in w3c httpd, http://www.w3.org/Daemon/User/Config/Logging.html.
[46] S. Williams, M. Abrams, C.R. Standridge, G. Abdulla, E.A. Fox, Removal policies in network caches for world-wide web documents, in: Proceedings of ACM SIGCOMM, 1997, pp. 293–305, revised August 1997, http://vtopus.cs.vt.edu/nrg/williams.html.
[47] J. Yang, W. Wang, R.R. Muntz, Collaborative web caching based on proxy affinities, in: Proceedings of the SIGMETRICS 2000 Conference, 2000, pp. 78–89.

Francesco Bonchi holds a Laurea degree in Computer Science (University of Pisa, 1998). Since 1999 he has been a Ph.D. student in Computer Science at the University of Pisa. He has been a visiting fellow at the Kanwal Rekhi School of Information Technology, Indian Institute of Technology, Bombay (2000). His current research interests are data mining query optimization, meta learning and web mining.

Fosca Giannotti was born in 1958 in Italy. She graduated in Computer Science in 1982 at the University of Pisa. From 1982 to 1985 she was a research assistant, Dipartimento di Informatica, Università di Pisa. From 1985 to 1989 she was a senior researcher at R&D Labs of Sipe Optimization and Systems and Management, Pisa. In 1989–1990 she was a visiting researcher of MCC, Austin, TX, USA, involved in the LDL (Logic Database Language) project. She is currently a senior researcher at CNUCE, Institute of CNR (Italian National Research Council) in Pisa. Her current research interests include knowledge discovery and data mining, spatio-temporal reasoning, and database programming languages design, implementation, and formal semantics, especially in the field of logic database languages.

Cristian Gozzi was born in 1974 in Lucca, Italy. He graduated in Computer Science (Laurea in Informatica) at the University of Pisa in October 2000. Since November 2000 he is working with the Pisa KDD Laboratory at CNUCE-CNR as a contributor in the MineFaST research project, aimed at analysis of web access data and development of intelligent caching techniques. His current research interests are in the areas of databases, data mining and knowledge discovery.

Giuseppe Manco holds a Laurea degree (University of Pisa, 1994) in Computer Science and a Ph.D. in Computer Science (University of Pisa, 2001). He is currently senior researcher at the Institute for Systems Analysis and Information Technology of the National Research Council of Italy. He has been contract researcher at the CNUCE Institute in Pisa (April 1999–January 2001), and visiting fellow at the CWI Institute in Amsterdam (1998). His current research interests include deductive databases, knowledge discovery in databases and data mining, web databases and semi-structured data management.

Mirco Nanni holds a Laurea degree in Computer Science (University of Pisa, 1997). Since 1998 he has been a Ph.D. student in Computer Science at the University of Pisa. From April to June 1999 he was a visiting fellow in College Park – University of Maryland, working on probabilistic agent programming. His current research interests include deductive databases, data mining and knowledge discovery, distributed interactive systems and agent programming.

Dino Pedreschi was born in 1958 in Italy, and holds a Ph.D. in Computer Science from the University of Pisa, obtained in 1987. He is currently a full professor at the Dipartimento di Informatica of the University of Pisa. He has been a visiting scientist at the University of Texas at Austin (1989–1990), at CWI Amsterdam (1993) and at UCLA (1995). His current research interests are in logic in databases, and particularly in data analysis, deductive databases, the integration of data mining and database querying, spatio-temporal reasoning, and formal methods for deductive computing.

Chiara Renso was born in Italy in 1968 and holds a Masters degree and a Ph.D. in Computer Science (University of Pisa, 1992 and 1997). She is currently a researcher at CNUCE Institute of CNR, Italy. She has been working on extensions of logical languages with modularity, defining the language MedLan as a proposal to perform semantic integration of different data sources. Her current research interests are: extensions to logic programming to perform spatio-temporal and uncertain reasoning on geographical data and use of web mining techniques for intelligent web caching.

Salvatore Ruggieri holds a Laurea and a Ph.D. in Computer Science (University of Pisa, 1994 and 1999). He is currently a researcher at the Dipartimento di Informatica of the University of Pisa. He has been an ERCIM fellow at the Rutherford Appleton Laboratory at Oxford (1995). His current research interests are data analysis in deductive databases, (parallel) tree-induction algorithms for data mining, formal methods in logic programming and intelligent multimedia presentation systems.