The VLDB Journal manuscript No.
(will be inserted by the editor)

An Architecture to Support Scalable Online Personalization on the Web*

Anindya Datta[1], Kaushik Dutta[1], Debra VanderMeer[1], Krithi Ramamritham[2], Shamkant B. Navathe[3]

[1] Georgia Institute of Technology and Chutney Technologies, Atlanta, GA
[2] University of Massachusetts-Amherst and IIT-Bombay
[3] Georgia Institute of Technology, Atlanta, GA

Received: date / Revised version: date

Abstract Online personalization is of great interest to e-companies. Virtually all personalization technologies are based on the idea of storing as much historical customer session data as possible, and then querying the data store as customers navigate through a web site. The holy grail of on-line personalization is an environment where fine-grained, detailed historical session data can be queried based on current online navigation patterns for use in formulating real-time responses. Unfortunately, as more consumers become e-shoppers, the user load and the amount of historical data continue to increase, causing scalability-related problems for almost all current personalization technologies. This paper chronicles the development of a real-time interaction management engine through the integration of historical data and on-line visitation patterns of e-commerce site visitors.

This paper describes the scientific underpinnings of the system, as well as the architecture and a performance evaluation. The experimental evaluation shows that our caching and storage techniques deliver performance that is orders of magnitude better than those derived from off-the-shelf database components.

1 Introduction

Given the current hyper-competitive nature of the e-commerce marketplace, coupled with razor-thin margins, online personalization is of great interest to e-companies. The attractiveness of online personalization technologies derives from the claim by consumer behaviorists that a personalized experience leads to increased *buy probabilities* on the part of e-shoppers [21]. The basic idea behind virtually all personalization systems specifically, and web-based customer interaction schemes in general, is simple: accumulate vast amounts of historical data (consisting of navigational, transactional, and third party data), transform this data into *static profiles* (often through the use of data mining techniques) and then query these profiles in the context of "current" visitation patterns to provide a personalized experience. For example, suppose that analysis of an online bookseller's historical data were to suggest a strong link between interests in Legal Thrillers and Astrophysics. Based on this knowledge, the bookseller might recommend a recent bestseller from the Legal Thriller category to all shoppers browsing the Astropyhsics books. Most current personalization software, such as NetPerceptions [18] or LikeMinds [4], offer the above-mentioned functionality, popularly known as *recommendations*. It so turns out that the technology underlying virtually all on-line recommendation engines is a clustering algorithm called *collaborative filtering* [23]. While static profiling has indeed provided benefits, there is significant room for improvement. Consider, for instance, a situation where an e-shopper purchases a gift for a friend, and finds that the recommendations he receives from the site continually suggest books that represent his friend's tastes.

The goal of the work presented in this paper is the amelioration of the above-mentioned limitation of static profiles by introducing the notion of *Dynamic Profiles*. These incorporate, in addition to whatever information may be statically available, the changing interests of the user during a particular session. For example, consider an e-shopper who purchases Microbiology textbooks for his daughter from an online seller, and later returns to the site looking for books for himself. After a few clicks the system should recognize that the user is not interested in Microbiology books in his current visit, suppress its knowledge of his past behavior and adjust its

Send offprint requests to: Corresponding author:
Debra VanderMeer, e-mail: `deb@cc.gatech.edu`

* An early version of this work appears in *Proceedings of the* 2^{nd} *ACM Conference on Electronic Commerce* (EC'00), October 17-20, 2000, Minneapolis, Minnesota.

responses to be in tune with his more recent behavior. This is possible only if the system can recognize changing behavior patterns.

The problem in supporting and tracking dynamic behavior, of course, is one of scale – it is extremely difficult to track tens of thousands of e-shoppers in real time, and even more difficult to access a database online to provide real-time responses. The problem is only exacerbated by ever-growing visitor loads and attempts to store increasingly fine-grained historical information. As more and more customers visit e-commerce sites, and as e-companies gather increasingly larger amounts of information from each customer visit, the size of the data warehouses storing all this information continues to grow as well. This results in huge data stores, and increasingly higher response times for queries over these data sources. This is one of the reasons why virtually all of the personalization and customer relationship management technologies on the market rely on one of two basic techniques: (a) delayed responses in the form of emails to customers, and (b) static profiling techniques that fit customers into one of a small number of predefined static profiles (usually based on statically pre-declared information, e.g., login or zip code) and provide canned online responses.

We have been interested in dynamic profiling and personalization and have developed technology that will allow highly scalable, online personalization based on the real-time integration of a vast amount of extremely fine-grained historical data and on-line visitation patterns of e-commerce site visitors. In this paper, we introduce the *eGlue server*, which we have been building as a manifestation of this technology. The eGlue Server brings together a number of technologies that impact storage and retrieval from data warehouses, as well as caching and prefetching.

Essentially, our system (1) tracks a large number of users (potentially tens of thousands) in real time as they navigate through a site, (2) performs retrievals from a large data warehouse in real time, and (3) delivers an appropriate user response, in real time, based on the system's knowledge of the user's current behavior and the information retrieved from the data warehouse.

To achieve our goals, we store two basic types of data: (1) navigational (i.e., where users go in the site) and (2) transactional (e.g., what customers purchase). This session knowledge is encoded simply as a set of rules of the type generated by standard data mining techniques. In particular, the system stores two types of rules: (1) *Action Rules*, which *anticipate* the user's next click on the site, allowing the site to *prefetch* the necessary content before the user actually clicks; and (2) *Market Basket Rules*, which make *recommendations* as to information that is likely to interest the user. The personalization server uses this type of rule to generate *hints*, which can be used either directly by page generation code running on an application server, or in concert with standard profiling techniques (as additional available information) for use in selecting the content of the web page generated for a user.

Our scheme allows users to remain *anonymous*, i.e., user actions are associated with a session_id to track behavior across sessions, but *individual users are not recognized across sessions*, i.e., if a user u_i visits the site twice, his behavior in his second visit is not explicitly correlated to his behavior in his first visit.

This paper makes three key contributions. First, we present the design, implementation, and performance testing of a component-based system to address the problem of real-time user interaction. Performance tests comparing our implementation show that our system give an order of magnitude performance gain over an implementation using off-the-shelf database back end. Second, we present the design of an efficient caching mechanism for user profiles. Third, we develop a web-site and user-navigation model specifically designed for e-commerce sites.

The remainder of the paper is organized as follows. We discuss related work in Section 2. In Section 3, we describe the underlying model of e-commerce interaction used in this paper. The architecture of the eGlue server is presented in Section 4, while Section 5 describes the technology underlying the Profiler. In Section 6, we present the results of our most recent performance tests. Section 7 concludes the paper.

2 Related Work

Within the academic literature, web personalization is emerging as a major field of research. We review some of the important literature in this field here, and mention a few commerical products. At present, there are two main approaches to personalization: collaborative filtering and data mining for user behavior patterns.

Collaborative filtering [23] is a means of personalization in which users are assigned to groups based on similarities in rankings of content of web pages, i.e., users with similar interests will rank the same pages in the same way. [6] presents an analysis of predictive algorithms for collaborative filtering. [8] describes a non-invasive method of collecting user interest information for collaborative filtering.

Much work has been published in the area of user navigation pattern discovery based on web logs, e.g., [7, 25], is quite complementary to our work. In [22], the authors take the novel approach of placing the user profiler on the client side using remote Java applets. In [10], a detailed description of data preparation methods for mining web browsing patterns is presented. [26] proposes a web mining usage tool. Much of the work in web navigation and usage mining is based on fundamental work in data mining for association rules [2] and sequential patterns [3,27]. This type of work is quite complementary to our work; in fact, we base our navigation pattern discovery methods on some of the ideas found in these papers.

Another important area of work related to this paper is efficiently delivering web content, i.e., caching and prefetching. Caching, discussed in [31], refers to the practice of saving content in memory in the hope that another user will request the same content in near future, while prefetching ([17,11]) involves guessing at which content will be of interest to the user, and loading it into memory. Both methods involve attempting to reduce latency by predicting user access patterns. A general discussion of methods for predicting user access patterns can be found in [9]. [24] discusses caching and buffering methods for database systems.

Most of the work in this area concerns caching web content, i.e., the actual files that are sent to the user. The caching described in this paper is different. Rather than caching actual content, we store predictive information, e.g., where the user is likely to navigate in his next click.

On the commercial side, several companies offer products in the Internet Personalization space. The two functions supported are *recommendations*, (i.e., static, predetermined responses to actions) based on collaborative filtering techniques [23] and *reporting* (i.e., analysis of a web site's traffic). Recommendation products include Net Perceptions Recommendation Engine [18], Engage Technologies [30], and LikeMinds [4]. Personify [20] and Accrue [1] offer analysis and reporting based on log data. Broadvision [29], another player in the web-personalization space, uses rules based solely on past behavior to match web content to users' interests. Manna's FrontMind [16] provides recommendations based on mining a customer's past purchases, and placing them in the context of the user's present location on the site.

While the general philosophy of our work is similar to that of the above personalization products in that we use a database and cache to store and retrieve our information, we depart from current personalization efforts in two key ways. (1) We do not rely on static profiles; rather, we generate *dynamic profiles* based on the information available at the time a user clicks. Here, a dynamic profile is not a dynamic version of a standard static profile; rather it is a *dynamic lookahead profile*, which allows us to anticipate which action a user is likely to take in his next click, based on his current behavior. (2) We merge historical information with knowledge of a user's current behavior in order to generate these dynamic decisions, thus enabling more personalized responses, as opposed to delivering canned responses to pre-defined actions.

3 Site Interaction Model and Preliminaries

In order to understand the technology being described in the paper, it is first necessary to understand the underlying data and operations model. Accordingly, in this section, we provide an overview of a model of an e-commerce site as well as a model of interaction of users with such a site. A more formal treatment of this model can be found in [14].

We believe that this model is significantly different from, and more comprehensive than, existing e-commerce site models available in the literature. In particular, this model supports, simultaneously, the notion of a *product catalog* as well as the notion of *user navigation* over the site (i.e., the catalog). While we found several models supporting either of the two, we could not find a single model to address both. A general model for a web site can be found in [15]. However, it does not include the notion of a product catalog, or user tracking. In [22], a simple model of user navigation is presented; however, it assumes a static web site (i.e., a site in which all pages are predefined).

Our model, in contrast, supports the notion of a product catalog, user navigation over this catalog and dynamic content delivery. Here, a script runs for each user request, generating a page on the fly for that user. The use of page generation scripts allows for enormous flexibility in the content eventually presented to the user. In fact, each page generated could be entirely unique, and composed of content from various nodes. In this scenario, it is easy to see how the mapping of pages to nodes, familiar in the context of sites composed of static pages, is no longer valid. Thus, we assume no direct mapping between pages and nodes.

Before embarking on our model discussion, we provide a reference to the notation used throughout this section, as well as the remainder of the paper. Table 1 contains this information. We begin by describing the notion of a *product catalog*, which forms the essence of any e-commerce site (or, for that matter, virtually any web site). We then describe how users navigate through

Symbol	Description
N_i	Navigation click on a link for i
B_i	Buy item i
D	Depart from the web site
CSL	Clickstream length
RA	Rule Antecedent
RC	Rule Consequent
WES	Web/E-commerce application server
eGS	eGlue Server
HR	Hint Request
PG	Page Generator
CM	Clickstream Monitor
CC	Clickstream Cache
CSL	Clickstream Length
PM	Profile Manager
PC	Profile Cache
FP	FastPump Profile Warehouse
QH	Query Handler
CMG	Cache Manager
PQ	Profile Query
CCL	Cache Cleaner
CL	Cacheline
CQ	Cache Queue
LFD	Lowest Following Distance
LD	Lookahead Distance
MLD	Maximum Lookahead Distance
ART	Average Response Time
CUP	Concurrent User Process

Table 1 Table of Notation

the site. Finally, we develop the idea of *dynamic profiles*, which model user interactions with the site.

3.1 Catalogs at an E-Commerce Site

A product catalog is a labelled, directed, acyclic graph in which leaf nodes represent product instances (SKUs in retail lingo) and internal nodes represent various hierarchical groupings of products. Figure 1 shows an example of a hierarchical product catalog for a fictional online bookseller, Papyrus.com (the root node of the product catalog). The site sells several categories (i.e., internal nodes in the hierarchy) of books, e.g., Fiction and History. These categories are further divided into subcategories, e.g., Historical Fiction. Leaf nodes in the tree are products, (namely, books), e.g., M_i (a specific Mystery book), HF_j (a Historical novel), and B_k (a Biography).

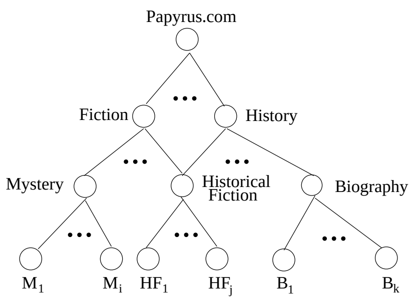

Fig. 1 Product Catalog Example

Definition 1 Node. *A node is a generalized, abstract structure which serves as the foundation for the product catalog model. A node $\mathcal{N}$ is defined as a 5-tuple $\langle L, C, P, D, A\rangle$ of descriptive information where:*

- **L** *is unique label for the node, e.g., History.*
- **C** *is a set of child labels, e.g., the set of child labels for the Mystery node is $\{M_1...M_i\}$ (i.e., books in the mystery category).*
- **P** *is a set of parent labels, e.g., the set of parent labels for the Historical Fiction node is {Fiction, History}.*
- **D** *is a descriptor for the node, containing pointers to objects needed to represent the node in HTML, e.g., links, images and text that describe a product (a leaf node in the product catalog) or a class of products (an internal node in the product catalog). These objects are* **elements**, *i.e., atomic static items that can be displayed on a web page. There are three basic types of elements (based on [5]):*
  - *An* **information element** *is a text, audio, or video object. These objects can be drawn via queries from a variety of data sources, e.g., an XML or database source.*
  - *A* **link element** *is a hypertext link. These links have two parts: (A) a text description of the link destination, and (B) a reference to one of the following:*
    1. *another node in the product catalog (identified by its label),*
    2. *an informational page (i.e., online help, company information, customer service information, etc.), or*
    3. *an external (i.e., outside the e-commerce site) URL.*
  - *A* **form element** *is a portion of a form for user entry. Form elements are collected into* forms, *which are used to facilitate user information entry (e.g., credit card information, customer service requests, etc.).*

  For example, the descriptor for node M_i, a specific Mystery book, might contain a pointer to a text element containing the book's title and a pointer to an image element containing a picture of the book jacket, as well as links to the node's parent (i.e., the **Mystery** *node), and links to other nodes representing mysteries by the same author.*
- *A is a set of permissible* actions *on the node. $A \subseteq \mathcal{A}$, where $\mathcal{A}$ is entire space of possible actions on a product catalog. We define six possible actions on a catalog:*
  1. *A* **navigation** *action is simply a click on a navigational link. For example, if user U is at the home page of an online bookseller (say for example, Papyrus.com in Figure 1), and wishes to see more information about Fiction, he would undertake this navigation action by clicking on the Fiction link.*
  2. *A* **buy** *action is a click indicating the user's intention to buy an item. On an e-commerce site, this occurs when a user chooses to place an item in his shopping cart. Clearly, this action is available only from nodes which offer items for purchase. Further, an item is only available for purchase from the node that represents it, e.g., the action of purchasing Mystery M_i is only available from node M_i.*
  3. *An* **un-buy** *action occurs when a user removes an item from his shopping cart, in effect undoing a previous* buy *action. This action is available at any point after the user has selected an item for purchase.*
  4. *A* **check-out** *action occurs when a user completes the purchase of the items in his shopping cart, thereby actually creating a purchase transaction in the system. This action is available at any point at which the user has at least one item in his shopping cart.*
  5. *A* **form-submit** *action sends user-entered information to the web server. This type of action can be* optional *(e.g., a user can choose to respond*

to a survey), or mandatory *(e.g., the user must enter credit card information and a shipping address in order to complete a* **check-out** *action, or enter a logon and password to enter a protected site).*

6. *A* **departure** *action occurs when a user departs from the site. Note that an e-commerce site's web server cannot explicitly detect a user's departure, since the HTTP request goes to another site's web server. Typically, a user is considered to have departed a web site after some threshold amount of time has elapsed since his most recent click.*

Note that the catalog model presented above ascribes several properties to an e-commerce site hosting the catalog. Below we list two important properties.

1. Through the notion of *elements*, the model supports the notion of *dynamic web pages*, i.e., pages that are not defined statically using HTML, but rather created on demand by putting together a number of elements on the fly. As the newer web servers and browsers come equipped with dynamic content generation technologies such as ASP, JSP, JavaBeans, etc., it is imperative that any meaningful model of an e-commerce site be able to support dynamic content generation.

2. A more subtle, but important property of the above-described model is that it confines a user's possible movements within a site across nodes of the site's catalog (or course, it also models the fact that a user can depart). In other words, the model mandates that at any given time during his visit, a user is on a given node in the catalog, and any action on his part will result in one of the following three results: (a) he will remain on the same node (in case of, say, a transaction action), (b) he will move to a different node (in the case of a navigation action), or (c) he will depart the site. Note, however, that the model does not imply that only a single node can be displayed on a page delivered to a user. Rather, the **descriptor** component of a node, say X, may contain references to other nodes, say Y and J. Thus when the user navigates to X, the contents of Y and J (or some specified subset of the contents) will be displayed, along with the contents of X.

The above discussion naturally leads us to describing a model of user navigation through a site.

3.2 User Navigation through an E-Commerce Site

As discussed already, we can think of a user navigating through an e-commerce site as a user navigating over the product catalog, since the web pages are simply presentations of different views over the product catalog. Thus, a user can be said to be *located* at some node in the product catalog throughout his visit to the site. In particular, after the user's i^{th} click, he is said to be located at the node pointed to by the link chosen in his i^{th} click. Thus, if the user is located at the Fiction node before his i^{th} click, and chooses the link to Historical Fiction as his i^{th} click, then after the i^{th} click he is at the Historical Fiction node.

Based on this, we model user interaction with the site as a sequence of actions (of the types described in 3.1) called a *clickstream*. For example, a clickstream modeling a user navigating from the root of the Papyrus.com product catalog (shown in Figure 1) to book HF_3, purchasing HF_3, and then departing from the site might consist of the following sequence: $\langle N_{Papyrus.com}$, $N_{Fiction}$, $N_{HistoricalFiction}$, N_{HF_3}, B_{HF_3}, $D\rangle$, where N_i denotes navigation to node i, B_j denotes buying item j, and D denotes departure from the web site.

The reader will remember that a user can buy item i (i.e., place item i in his shopping cart) only if he is located at node i, i.e., buying item i is a permissible action only at node i (see Section 3.1 for a discussion of actions). Since a user can be located at one node at most, our model would seem to disallow buying at least one of the products displayed on the page. To handle this, when a user chooses to buy a product and is not currently located at the appropriate node, we insert a click in his clickstream leading him to the appropriate node, thus indicating his interest in the item. For example, consider the situation in which a user is located at leaf (i.e., product) node M_a, and the web page he is viewing reveals items M_a and M_b. If the user were to decide to buy M_b, then his clickstream would be $\langle N_{M_a}, N_{M_b}, B_{M_b}\rangle$.

Next we move on to describing the key notion of this work, i.e., the notion of a *dynamic user profile*.

3.3 Dynamic User Profiles

A key focus of this paper is generating *dynamic lookahead profiles*, rather than static profiles, as noted in Section 2. Here, we describe dynamic profiles in detail.

Essential to providing online personalization is an ability to anticipate what a user is likely to do, across several dimensions, e.g., which pages a user is likely to access, which product categories he is likely to navigate, and when he is about to leave. For the purposes of this paper, we consider three major options open to a user at any given web page in an e-commerce site : (1) the user can buy a product (assuming one is offered on the page), (2) the user can depart from the site, and (3) the user can choose a link to another page in the e-commerce site.[1] If we can predict which of these three actions a user is likely to take, an array of personalization options become available. For example, if we know which link a user is likely to choose next, then we can preload the page content from disk, providing a fast response time

[1] Clearly, other actions are available to the user at the site; including these actions in the context of dynamic profiles is a trivial extension.

to the user. A dynamic profile of a user is simply a collection of information that provides a prediction of what the user's next action is likely to be. Note that, for the remainder of this paper, we will refer to a dynamic profile simply as a profile.

Owing to recent concerns over online privacy [19] as well as a preference for keeping our strategies as general as possible, we consider *anonymous users* in this paper. In other words, users are identified only by a *session id* while visiting the site, and no user-specific information is maintained between user visits. Clearly, since we collect only traces of users' sessions through the site, we do not cluster users in the traditional sense. Rather, users who behave similarly can be treated similarly, resulting in a much more fine-grained interaction between a user and the site than is possible with technologies based on static profiling.

For the purposes of this paper, the reader can assume a profile to consist simply of a set of rules corresponding to the observed as well as anticipated interaction patterns of a user with an e-commerce site catalog. To explain this in greater detail, we now describe the two classes of rules we use in the eGlue Server, *action rules* and *market basket rules.*

3.3.1 Action Rules Action rules predict what action a user will perform, based upon his current visitation clickstream, i.e., a sequence of actions. A clickstream representing a user's entire visit at a site is referred to as a *session*. We track users as they visit the site and log the session information. Note that session tracking is a well understood technology [7,25,22]. These session logs are then mined to extract the *action rules*, which are of the form $Action_1, Action_2, ...Action_{CSL} \rightarrow Action_R$; *confidence* $= C$ *and support* $= S$, where $Action_i$ is a navigation click, a purchase click, or a (virtual) departure click.

We generate these rules using standard sequential pattern mining tools (e.g., with IBM's Intelligent Miner). The result of the mining process is a set of sequential patterns of the type shown above. The antecedents of the rules are of a (configurable) maximum length *CSL*, and correspond to certain minimum confidence, and minimum support thresholds. The rules are stored in a rule warehouse (referred to as FastPump, and described in Section 5).

Referring back to our product catalog in Figure 1, an example of such a rule, where *CSL*= 3, might be: $N_{HistoricalFiction}, N_{HF_6}, B_{HF_6} \rightarrow Depart$, *confidence* $= x$, and assuming that a configurable minimum support threshold is met. Here, our user is navigating from "Historical Fiction" to the book HF_6, and purchasing HF_6. This rule predicts that he will depart from the site with probability x (where the rule's $confidence$ serves as the probability that the rule will fire, assuming that the minimum support threshold is met). The eGlue server uses rules of this type to generate hints about a user's next action, thus enabling a wide range of customization or prefetching possibilities.

3.3.2 Market Basket Association Rules Market basket association rules provide predictive information about items that tend to be purchased in groups. Such a rule, e.g., $A, B, C \rightarrow D$ with confidence x, might show us the following: if a user's shopping basket contains the books A, B, and C, then he is likely to also purchase the book D with $confidence = x$ and meeting a configurable minimum support threshold. The eGlue server uses such rules to support recommendations and to determine the appropriate content for a page. While recommendation is an important function in the eGlue Server, it is also well-understood and supported in most commercial personalization products. As such, it is not a major focus of this paper and we refrain from further discussions of recommendations and market basket rules.

Having described the rules used in profiles, we now define the precise structure of a dynamic profile. We note that the profile contents in the eGlue Server product are significantly more complex than the profile described below. Because the exact structure of the eGlue profile is proprietary, the reader should consider the profile discussion below, and throughout the remainder of the paper, as illustrating the notion of a profile, rather than describing the actual structure of a profile.

Definition 2 *A* **Profile** *is a 2-tuple* $\langle RA, RC\rangle$ *of information where:*

1. *RA is a rule antecedent, i.e., a user clickstream of length* CSL.
2. *RC is a set* $\{c_1, c_2, ..., c_n\}$ *of rule consequents* c_i*, where each* c_i *is itself a 3-tuple* $\langle A, L, p\rangle$ *where:*
   (a) *A is an action, as described in Section 3.1.*
   (b) *L is a node label, as described in Section 3.1.*
   (c) *p is the conditional probability of the consequent, given the antecedent.*

Having described the structure of a profile, we now provide an example to make the above discussion more concrete. Consider the situation where our user Ulysses has arrived at the Papyrus.com home page, and has chosen the links for "Fiction" and "Historical Fiction", in that order. Based on this clickstream, i.e, $< N_{Papyrus.com}, N_{Fiction}, N_{HistoricalFiction} >$, the eGlue Server might find the following rules in the rule warehouse whose antecedent corresponds to the current visitation clickstream:

1. $N_{Papyrus.com}, N_{Fiction}, N_{HistoricalFiction} \rightarrow N_{HF_3}$ with *probability* $= 40$
2. $N_{Papyrus.com}, N_{Fiction}, N_{HistoricalFiction} \rightarrow N_{HF_4}$ with *probability* $= 20$

These rules suggest that Ulysses will navigate to the node representing HF_3 with probability 40%, or to the

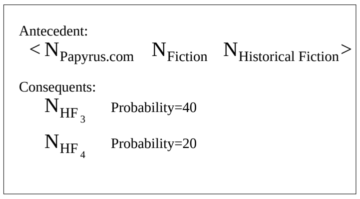

Fig. 2 Dynamic Profile Example

node representing HF_4 with probability 20%. The contents of Ulysses' dynamic profile at this point are shown in Figure 2. The profile contains the rule antecedent, i.e., Ulysses' current clickstream, as well as the two matching rule consequents.

The reader is reminded that a given user's profile is recomputed, in real-time, after each action undertaken by the user in order to provide up-to-date predictive information about what the user is likely to do next *as he is clicking through the site.* This recomputation is clearly extremely difficult – the system must not only track every user click (for a potentially large number of users), it must also return a profile for each user click *in real time.* This is possible due to a combination of efficient, tailored database technology as well as an efficient profile caching scheme. These will be described in the next section.

4 Architecture

In this section, we present an architecture of the eGlue Server, a component-based system which works with readily available web server and e-commerce application server systems. Figure 3 shows a graphical depiction of an "end-to-end" e-commerce system architecture, including the eGlue Server component. We note here that the "real" eGlue server software system consists of a number of real-time personalization applications (e.g., targeted advertisement delivery) not shown in this diagram, to restrict the expositional complexity of the paper. What we depict as the eGlue server is, in reality, the eGlue server engine whose purpose is to create dynamic profiles that feed the applications not shown. The important thing to note is that the power and scalability feature of the system derives from this engine. We also remind the reader at this point that a table of notation, which will be very helpful in this rather acronym-laden section, can be found in Table 1.

Perhaps the best way to describe the overall workings of the system is to provide an example. Consider a user U, who clicks in an eGlue enabled e-commerce site. This causes an HTTP request to be sent to the *Web/e-commerce Server* (WES). (Note that, although the e-commerce and web servers are actually separate components, we describe them as a single component here for convenience). When the WES receives an HTTP request from U, it forwards U's click information to the *eGlue Server* (eGS) in the form of a *Hint Request* (HR) containing U's most recent click. Upon receiving this information, the eGS performs two tasks. First, *the eGS updates U's clickstream.* Second, *The eGS generates a hint.* A *hint* is simply a set of *action-probability pairs*, which represent actions U is likely to take, along with the corresponding probability that U will choose the action, given his current clickstream. These are drawn from the action rules described in Section 3.3, where a hint consists of a set of rule consequents matching a given antecedent (i.e., clickstream).

When the WES receives a *hint* from the eGS, it uses the hint to generate a customized web page for U. Precisely how the WES uses the hint to generate a personalized web page for a user is dependent on the needs of the web site, and is outside the scope of this paper. Typically, the WES runs a set of *scripts* that describe how a web page should be generated in different situations. In an eGlue-enabled system, these scripts include business logic to handle eGlue hints.

To make the use of these hints more concrete, we provide an example. Consider a situation where our user U has navigated through a series of product catalog nodes, but has not yet purchased anything. Let us further assume that, on U's i^{th} click, the eGS returns a hint to the WES, suggesting that U is highly likely to depart in his next click, given his current clickstream. At this point, the business logic in the WES scripts might send U a web page that includes a special offer, perhaps free shipping or a percentage discount on his order, to entice him to stay at the site and continue shopping.

In addition to the above interaction, each user click causes a client-side *Clickstream Monitor*[2] to send user session (i.e., clicks composed of a session_id and node_id) information to the eGlue Server, where it is stored in the Session Log for later use in updating the ruleset. (Note that this interaction is entirely separate from the user's interaction with the WES). The CM module is essentially a client-side Java applet that sends clicks (including client-side cache hits) to a server, as described in, say [22]. In this way, the eGS obtains a complete account of a user's actions at the site, including those involving cache hits external server (e.g., client-side cache hits). Since the information stored in the Session Log is used for refreshing the rulebase (an offline operation), there is no requirement for this interaction to occur in real time.

Having described the interaction between the main components (i.e., the WES and the eGS) of the system, we now provide a detailed description of these components. For the sake of brevity, we provide only a general description of the WES components, since the workings

[2] Clearly, a server-side session-logging option is possible, and much easier to implement than the client-side method. The intent of the discussion here is not to preclude server-side session-logging, but rather to point out the inaccuracies a server-side logging method could produce.

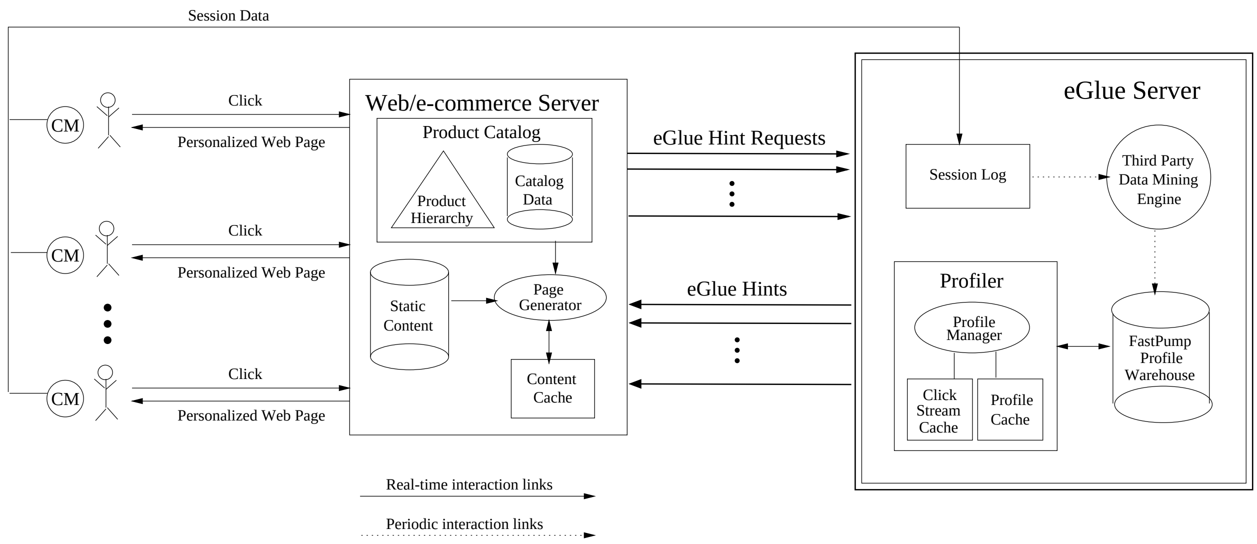

Fig. 3 Real-Time Personalization System Architecture

of these components are well-known. We then move on to a detailed description of the eGlue Server.

4.1 Web/E-commerce Server

The eGlue server works in conjunction with available web servers (e.g., Apache, Netscape, etc.) and e-commerce application servers (e.g., IBM's WebSphere or Pervasive's Tango) to provide the functionalities we have described. We now give a brief description of the various modules in the WES here, noting that a detailed discussion of these products is outside the scope of this paper.

The eGS provides hints to the *Page Generator* (PG), which then generates a personalized page for each user. The PG then synthesizes web pages by integrating content from various sources and formatting the content as necessary (e.g., into HTML pages). Page content is drawn from various sources, including the *Product Catalog* (which stores product information in hierarchical format as well as detailed information about each product), a store of *static content* (i.e., content not associated with a particular product catalog node, e.g., a corporate logo), and a *server cache*. Communication between the eGS and WES follows a client-server paradigm, i.e., the WES requests hints from the eGS, and the eGS provides hints in response.

4.2 The eGlue Server Components

Here, we provide an overview of the interaction of the components of the eGS, which is the primary focus of this paper. We describe the underlying technical details of these components later, in Section 5.

The *Profiler*, technical details of which will be described in Section 5, consists of three components: (1) a *Clickstream Cache* (CC), which stores current clickstream information (of maximum length *CSL*) for each user in the system, (2) a *Profile Cache* (PC), which stores recently-used profiles, and (3) a *Profile Manager* (PM), which generates hints for the WES.

To illustrate the mechanics of the hint-generation process, we return to our user U. Consider a situation where the WES has submitted a *hint request* to the eGS for U's i^{th} click. The PM first checks the CC to find U's previous clickstream (if any), and updates that to include U's latest reported action.

After determining U's current clickstream, the PM checks the PC for a profile matching U's clickstream (i.e., a profile whose RA matches the observed clickstream). If such a profile is found, the PM generates a hint from it, and sends the hint to the WES. If, at this point, another user, say U', were to follow on the same path as U, the reader can easily see that the needed profile would be in the PC (assuming U' follows U closely enough that the profile has not been replaced by a newer profile).

If a matching profile is not found in the PC, the PM requests the information from the *FastPump Profile Warehouse* (FP) (described below). After FP returns the profile information, the PM generates a hint, and sends it to the WES. This profile will also now reside in the PC until a decision is made to discard it. Note that the PM maintains a profile in cache *even if the FP returns no consequents for a given antecedent*. This allows the system to differentiate between the case of a null consequent and the case of a cache miss.

Currently, a strict mechanism is used to match clickstreams and profiles, i.e., the clickstream sequence must exactly match the antecedent in a profile. Future plans include the development of more "relaxed" matching mechanisms.

Within the context of the eGlue server, FP serves as a *profile warehouse*, storing historical and navigational data in the form of rules, and retrieving this data in response to queries.

The rules stored in FP are generated by an *off-the-shelf data mining engine* which takes a *Session Log*, i.e., clickstream and transaction information generated by various clients clicking in the web site, as input, and generates a set of rules as output. Click and transaction data is added to the session log in real time (as noted by the solid lines in Figure 3), while the mining of the session log and update of the Profile Warehouse takes place offline (shown by the dotted lines in Figure 3).

Rules in the FP are periodically updated, via an offline process, to incorporate recent behavior. Note that generating and maintaining rules is an offline process and is an important feature of the eGS. However, this feature is not native to the eGS itself, but rather is handled by a best-of-breed data mining engine, which is incorporated as part of the overall solution.

Having provided an overview of the components of our system, we move on to a discussion of the technical details of our system.

5 eGlue Server Technical Details

In this section, we describe the underlying technical details of the eGlue Server. As already mentioned, the eGS contains two main components: (1) the FastPump Profile Warehouse, and (2) the Profiler. FastPump is an efficient storage and retrieval engine based a set of new indexing and join strategies. Since a description of the FastPump profile warehouse is available in the published literature [13], we omit further discussion of FastPump in order to save space. Rather, we highlight the details of the Profiler component in this section.

5.1 Profiler Overview

Figure 4 provides a graphical overview of the technology that forms the basis of the Profiler module of the eGlue Server. We describe the interaction between the different components of the Profiler. Further technical details are presented later in this section.

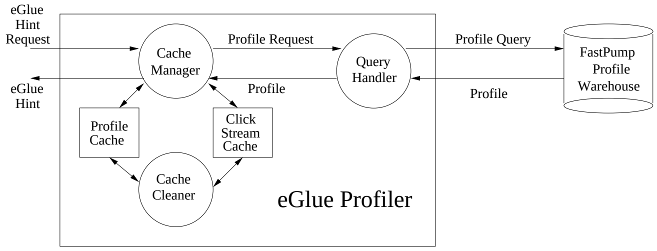

Fig. 4 eGlue Server Profiler Module

The Profile Manager shown in Figure 3 in Section 4 consists of three modules, the *Cache Manager*, the *Query Handler* and the *Cache Cleaner*. We describe each of these modules in turn.

The *Cache Manager*(CMG) monitors incoming hint requests, generates hints from profiles, and maintains the *Profile Cache* (PC) and and *Clickstream Cache* (CC) structures. The details of the cache structures are described later, in Section 5.2.

For *cache misses*, i.e., situations where a needed profile is not found in cache, the CMG generates a *Profile Query* (PQ) to retrieve the needed profile from FastPump. When the PQ has been processed, the CMG receives a profile in response. This profile is then used to generate a hint and update the PC, as described in Section 4.

The *Query Handler* (QH) regulates access to the Profile Warehouse. The QH accepts PQs, formulates the appropriate FP queries, and submits the queries to the FP. The QH formats each query result as a profile, and forwards it to the CMG.

The *Cache Cleaner* (CCL) removes outdated information from the cache. The idea behind the cache cleaner is similar to the clock hand idea in classical operating systems [28].

5.2 Profile Cache

We begin our discussion of the Profile Cache with a note on the basic structure of the cache. The cache consists of a (configurable) number of variable-sized elements, called *Cachelines* (CL). Each CL can store at most one instance of a profile. The CMG accesses a CL through a hash index on the *RA* (i.e., antecedent) portion of each CL's profile. CLs have three possible states:

In the **empty** state, a CL contains no profile. All CLs are *empty* at system startup. A CL returns to this state when is has been selected for replacement, before a new profile has been stored in it.

In the **requested** state, the CL is awaiting a response to a PQ. Here, the *RA* portion of the profile has been instantiated, but the *RC* (i.e., consequent) portion has not. A CL moves from the *empty* state to the *requested* state when the CMG receives a profile request, but does not find the needed information in the PC, i.e., when a query must be submitted to the QH. The CL remains in the *requested* state until the QH receives the needed response from the FP, and instantiates the *RC* portion of the profile.

In the **full** state, the CL contains both the *RA* and *RC* portions of the profile. A CL enters the *full* state when the CMG receives a profile from the QH (in response to a query), and remains in this state until it is selected for replacement.

Having considered the general structure of the cache, we move on to a discussion of the algorithm used to control the contents of the cache. This algorithm serves a dual purpose; it not only acts as a cache replacement mechanism, it also *prefetches* profiles from the Profile Warehouse, based on users' current profiles and likely navigation choices. Since the algorithm is not simply a replacement mechanism, less complex replacement algorithms, such as LRU, will not serve the purpose. We first describe the cache replacement functionality of our cache, then move on to discuss the profile prefetching mechanism.

5.2.1 Cache Replacement Policy As noted above, the CMG stores recent profiles in the PC. The utility of caching such information is clear. Consider, for example, the case of two users, U and U', who are both interested in the same item, book X. In this scenario, U enters the site at the home page, and navigates through the product hierarchy to the page offering book X. U' enters the site just after U arrives, and navigates the same click sequence. Clearly, since the profiles used to generate hints for U are likely to be located in the cache at this time, these profiles can be used for U' as well, saving the expense of querying the warehouse. This is the intuitive basis of the cache replacement policy described below.

In order to ensure that useful profiles are retained in the cache, we would like to be able to detect situations in which one user is following behind another, and utilize this knowledge in our cache replacement policy. For this purpose, we introduce the idea of *following distance* between two CLs.

Definition 3 *Consider two CLs C_i and C_j, containing profiles P_i and P_j, respectively. Assume that these CLs and profiles exist in the context of an e-commerce site with a catalog T. The* **following distance** *from C_i to C_j, denoted by $F(C_i \to C_j)$, is the minimum number of actions required by a user whose clickstream sequence currently matches P_i to match P_j, in the context of T.*

For example, consider a situation where user U has traversed the click sequence $\langle N_a, N_b, N_c, N_d \rangle$. U's traversal of this sequence has caused profiles P_i and P_j to be loaded into the profile cache (PC), where $P_i.RA = \langle N_a, N_b, N_c \rangle$ and $P_j.RA = \langle N_b, N_c, N_d \rangle$ (assuming a clickstream length of 3). Assume that P_i and P_j reside in CLs C_i and C_j, respectively. Clearly, $F(C_i \to C_j) = 1$, because a user with profile P_i, need only click on link c to be ascribed to profile P_j.

Extending our notion to include the possibility that multiple users might be traversing the same click sequence at the same time, we note that we are only interested in the *closest* user to a particular CL in a particular click sequence. Thus, we are interested in the *lowest following distance* (LFD) of a CL.

Definition 4 *Consider a set of CLs $\{C_1, C_2, \ldots, C_n\}$. The lowest following distance of a CL, say C_i, is denoted by* LFD*(C_i) and is given by:* LFD(C_i) $= min[F(C_1 \to C_i), F(C_2 \to C_i), \ldots, F(C_n \to C_i)]$

We compute LFD online for a CL by *broadcasting* a user's position to each CL that is within a configurable *maximum lookahead distance* (MLD) ahead of a user's current CL. Specifically, with each user click, the Profiler updates the distance for each CL within MLD ahead of the current CL (based on permissible actions on each node), where MLD is a configurable parameter. For a CL with an undefined LFD, i.e., where no user will need it within MLD clicks, we set the CL's LFD value to MLD.

Broadcasting LFDs occurs via a set of CL pointers associated with each node. Specifically, CL C_i contains a set of pointers to other CLs, corresponding to the links available from the last node in C_i's antecedent. For example, consider the situation where C_i holds profile P_i, with $P_i.RA = \langle N_a, N_b, N_c \rangle$ and CL C_j holds profile P_j with $P_j.RA = \langle N_b, N_c, N_d \rangle$. Let us further assume that a user can traverse the click sequence $\langle N_a, N_b, N_c, N_d \rangle$. In this situation, there exists a pointer from C_i to C_j, denoting that C_j is *reachable* from C_i, shown graphically in Figure 5.

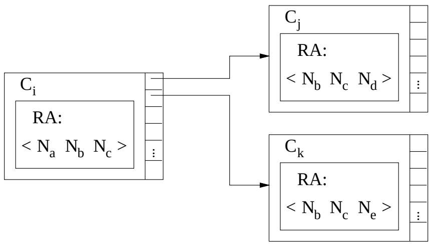

Fig. 5 CL Pointer Example

The CMG updates these CL pointers for each newly instantiated CL, i.e., when a CL enters the *full* state. For example, consider the situation where C_j holds profile P_k with $P_k.RA = \langle N_b, N_c, N_e \rangle$, and where a profile P_i, with $P_i.RA = \langle N_a, N_b, N_c \rangle$ is about to be added to the cache into CL C_i. We assume, for the purposes of this example, that a user can traverse the click sequence $\langle N_a, N_b, N_c, N_e \rangle$. In this situation, the CMG sets a pointer in CL i pointing to CL k, as shown in Figure 5.

Having laid the groundwork for our cache replacement policy, we move on to discuss this policy in detail. We note first that the most useful CLs in the cache are those with low LFDs, since these are the items likely to be needed soonest. Thus, we would like our cache replacement policy to take following distance into account in selecting CLs for replacement. However, in addition to LFD, we would also like to take time into account. In particular, we would like to give CLs with low LFDs priority over CLs with higher LFDs, and for CLs of the same LFD, we would like to give preference to CLs that have been used most recently over "older" CLs, i.e., those that have not been used as recently.

We implement this cache replacement policy using a series of *priority queues*, shown in Figure 6 as *Cache*

Queues (CQ), numbered 1 to MLD. Each queue represents a specific LFD, e.g., CQ_1 stores CLs with a LFD value of 1, CQ_2 stores CLs with a LFD of 2, and so on. Each queue is ordered by time, with the oldest CL, i.e., the least recently used CL, at the front of the queue.

We maintain these cache queues in the following manner. As user navigates on a click sequence, the LFD values for CLs on that sequence will decrease. When a CL's LFD values changes, it is moved to the end of the appropriate cache queue for its new LFD value. When a CL is referenced (i.e., a *cache hit* occurs), the CL is moved to the end of the queue, thus maintaining the queue's time ordering.

We replace CLs first based on LFD. More specifically, we first choose CLs for replacement from the MLD queue. If there are no CLs in the MLD queue, we replace CLs from the (MLD - 1) queue, and so on. Within each queue, we give priority to CLs in most-recently-used order, choosing the least recently used CL for replacement first. Since each queue is ordered based on time, the CL at the front of the queue is always the least recently used, and is the first chosen for replacement.

We note here that this prioritization scheme leaves open the possibility that CLs may sit unused for a long time in a cache queue. This occurs when a CL's distance is updated because it is on a possible future path for a user. If the user chooses an alternate path, the above prioritization scheme has no means of resetting the CL's distance value to the default value. We take care of this by noting the most recent access time of a CL with a timestamp. The Cache Cleaner module periodically checks all CLs in the cache, and remove CLs with timestamps more than T_{max} time in the past, where T_{max} is a configurable threshold value.

Having considered the cache replacement policy, we move on to discuss how requests for profile warehouse data are handled.

5.2.2 Serving Profile Warehouse Requests The Profiler supports two types of Profile Requests: (1) requests for *cache misses*, i.e., requests made when a needed profile is not found in the cache, and (2) *anticipatory requests*, i.e., requests for profiles that users are likely to need. These profile requests are simply CLs in the *requested* state, queued for processing by the FP.

We would like to prioritize handling of the profile requests in much the same manner that we prioritize for our cache replacement policy. In particular, we would like to process requests for cache misses before processing anticipatory requests. In addition, we would like to give priority to requests for profiles that are likely to be needed in a user's next click over those likely to be needed after 2 clicks, and so on.

Like CLs containing fully instantiated profiles, request CLs are assigned LFD values. Here, the LFD value refers to how soon the profile is needed. Thus, a request generated by a cache miss has an LFD value of 0, indicating that it is needed immediately, while an anticipatory request for a profile that a user can reach in 1 click has an LFD value of 1, and so on.

We prioritize request CLs in *Data Request Queues*, shown graphically in Figure 6. As each request CL is generated, it is placed at the end of the data request queue matching its LFD value. The QH module services requests in increasing priority order; within each queue, requests are serviced in first-in-first-out order.

We note here that this queuing mechanism may generate requests for information that is never needed, e.g., if a user chooses a different path from the path the Profiler predicted. Here, a cleaning mechanism similar to the Cache Cleaner (described above) is used to remove unneeded requests.

Having described our Profile Cache structure, we move on to the Clickstream Cache.

5.3 Clickstream Cache

As noted above, the Clickstream Cache stores user clickstreams for users active on the site. A CCL cache element has a very simple structure – it need only store a userID and the most recent *CSL* user clicks. Since CCL cache elements are simple data structures, and are not reusable for multiple simultaneous users, we can use an off-the-shelf technology, such as an object database, as our caching mechanism. As this caching structure is not the focus of our work, we do not elaborate further on it. Rather, having described our caching mechanisms, we move on to our performance evaluation.

6 Performance Evaluation

In this section, we discuss our most recent performance results. We describe our implementation, then our experimental methodology and the metric we used to measure performance. Finally, we discuss our experimental results.

6.1 Implementation Description

The implementation of eGlue server used for these experiments consists primarily of two modules, the *FastPump Warehouse* and the *Profiler*. Both the FastPump the Profiler are written in C/C++ using ObjectSpace STL and Communication Toolkits, and compiled with Visual C++ V6.0. FastPump and the Profiler run on separate PentiumIII (450MHz) single-processor NT Server V4.0 (SP5) machines, each with 18GB disk and 256MB RAM. Communication between modules is implemented with sockets over a local Ethernet network (with significant other traffic).

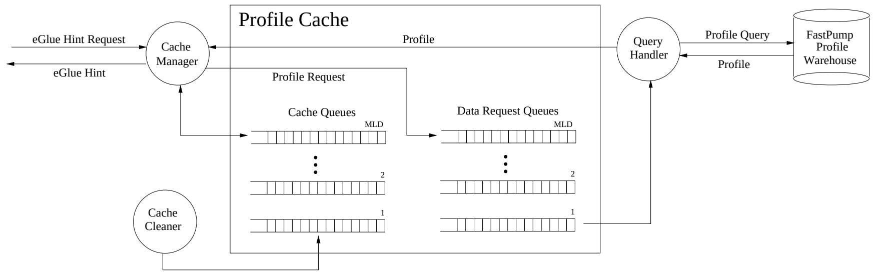

Fig. 6 eGlue Server Profiler Cache Detail

6.2 Experimental Methodology

We simulate load on the system by generating user requests, which represent clicks on an e-commerce site. Specifically, each user is modeled as a UP models a specific user's interaction with the site. These UPs are generated on an NT workstation (SP5) using a user simulator developed in JDK 1.2, and submitted as requests over a local Ethernet network. Each CUP submits requests as follows: As soon as the CUP enters the system, it submits a request. As soon as it receives a response, it immediately submits a new request, i.e., we do not include "reading time" in our experiments in order to simulate a higher user load. The maximum number of permissible concurrent UPs is denoted as *CUPLevel*. Clearly, *CUPLevel* denotes load on the system.

CUPs arrive as a Poisson process with interarrival times averaging *ArrivalRate*, and depart after making *NumClicks* requests to the system, where*NumClicks* is distributed normally with mean *MeanClicks* and standard deviation *StdClicks*. CUPs navigate through the product catalog. Each CUP is presented with a set of links from which he can choose his next click. The link choice is based on a Zipfian distribution[12] drawn from the action probabilities in the rulebase.

Simulated data is used for the product catalog and rulebase. Each product catalog consists of a number of items, *NumItems*, which appear as leaf nodes in the product catalog hierarchy. For each product catalog size, CUP traversals were simulated to generate a ruleset. Each ruleset consists of *RuleBaseSize* rules. The rules in the rulebase and the Zipfian distributions used for CUP navigation during the experiments are both based on the same underlying navigational dataset.

The Profile Cache consists of *CacheSize* cachelines (i.e., elements), where *CacheSize* is expressed as *CachePct*, a percentage of the rulebase size. Specifically, *CacheSize*= *CachePct* × *RuleBaseSize*. The Profile Cache is organized on the basis of *LookaheadDistance* Cache and Data Request Queues.

The rules in the FP profile warehouse are stored in proprietary table structures corresponding to the eGlue Server's profile structure. While we cannot divulge the actual schema, we can provide a general overview of the schema, as well as the query structure used to retrieve profile information, in order to make the experimental methodology clearer. The rulebase schema consists of four tables arranged in a star schema, one fact table consisting of eleven integer- and string-valued attributes, and three dimension tables, each consisting of four integer- and string-valued attributes. The specific query used in these experiments is a four-way join across all tables, projecting 8 columns.

Table 2 shows the minimum and maximum values of the various system load and data size parameters used in our experiments.

Parameter	Minimum	Maximum
NumItems	5K	60K
ArrivalRate	2	2
MeanClicks	30	30
StdClicks	5	5
CUPLevel	10	125
CacheSize	2K	36K
RuleBaseSize	175K	600K
LookaheadDistance	1	4

Table 2 Nominal Parameter Values

6.3 Performance Metric

For the purpose of evaluating our system, we use *Average Response Time* (ART), i.e., the average time from a user's click (i.e., request) to eGlue's response. Thus, ART is simply the sum of the response times for all clicks in the system, divided by the total number of clicks made in the system:

$$ART = \frac{\Sigma(t_{response} - t_{click})}{TotalClicks}$$

We chose these timing points, omitting time required for some typical events, e.g., the time required for a web/e-commerce server to build a web page for the user, in order to highlight the performance of the eGlue server, i.e., we wish to examine the performance of the storage, retrieval, and caching mechanisms we have implemented.

All performance graphs exhibit mean values that have relative half-widths about the mean of less than 10% at the 90% confidence level. We only discuss statistically significant differences in the ensuing reporting section. Each data point represents the average of five experimental values.

6.4 Baseline Experiments

In these experiments, we seek to place our work in the context of existing web-based customer-interaction technology. In particular, one question that arises is why we chose to design our own storage and retrieval engine (i.e., FastPump), and not use existing commercial data warehousing software. Virtually all existing personalization technologies use commercial databases for storage and retrieval. Our contention is that this is one of the major factors contributing to the scalability of the software. In order to show this, we compare the performance of the eGlue Server (i.e., FastPump and the Profiler) to the performance of a system combining a widely-used commercial data warehousing product with the Profiler. We chose Oracle 8*i* for this purpose, since it is a common data warehouse choice in existing personalization products.

For these experiments, identical datasets were loaded into both FastPump and Oracle 8*i* (using bitmap indexes for all primary key and foreign key columns in Oracle). Oracle was run on a separate machine under Windows NT Server, connected via SQL-Net over a local Ethernet network. All other modules in the system, including the Profiler, were held constant.

We first consider experiments comparing the performance of FastPump with the Profiler and Oracle with Profiler. Figure 7[A] shows two curves for FastPump with the Profiler (FP-Profiler), i.e., the eGlue Server, and Oracle with the Profiler (Oracle-Profiler), i.e., replacing FastPump with Oracle in the eGlue Server, for *CachePct*, *RuleBaseSize*, and LD remain constant at 5%, 375K, and 3, respectively. The points plotted on the curves show steady state values for ART for each *CUPLevel* value.

We first consider the FP-Profiler curve, which will serve as a baseline for the remainder of the experiments in this paper. We begin with a discussion of the general shape of the curve. The curve is exponential, i.e., as *CUPLevel* increases, the rate of increase of the slope increases. At low loads, i.e., between *CUPLevels* of 10 and 50 the slope of the curve is fairly low. In this range, the system is not overwhelmed by requests. As the load on the system increases, the rate of increase of the slope of the curve also increases. Here, a larger number of users in the system leads to an increase in the portion of the rulebase needed to respond to Hint Requests. This, in turn leads to higher cache miss rates and a larger number of requests in the data request queue. A longer data request queue leads to higher response times for cache misses, which leads to higher overall ART values.

The Oracle-Profiler curve in Figure 7[A] has the same general shape as the FP-Profiler curve, i.e., it is exponential. However, the ART values for Oracle-Profiler are much higher than the ART values for FP-Profiler. In addition, the rate of increase of the slope is much higher for the Oracle-Profiler curve than for the FP-Profiler curve. These differences are all due to Oracle's service rate, which is much slower than FastPump's [13]. Since Oracle requires more time to respond than FastPump, the data request queue grows more quickly, and to a larger size. In addition, items are added to the cache more slowly, leading to an even longer data request queue. This leads to the higher overall ARTs, as well as the faster growth of ARTs as load on the system increases.

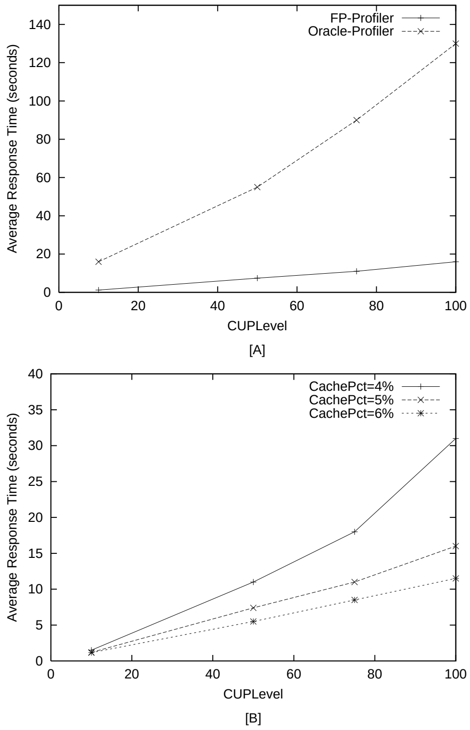

Fig. 7 [A] Effect of the Underlying Database: FastPump vs. Oracle; [B] Effect of Varying *CachePct*

Based on the evidence above, we chose FastPump for this application. For the remainder of these experiments, we report only results for experiments using FastPump with the Profiler.

Having considered the effects of replacing FastPump with off-the-shelf data warehousing technology, we move on to discuss our sensitivity experiments.

6.5 Effect of Varying CachePct

In this experiment, we show the performance (i.e., ART) of the eGlue Server as *CUPLevel*, i.e., the load on the system, increases. Figure 7[B] shows three curves for *CachePcts* (i.e., percentages of the *RuleBaseSize*) of 4%, 5%, and 6%, while LD and *RuleBaseSize* remain constant at 3 and 375K, respectively. We note that the curve for *CachePcts*=5% is exactly the same as the FP-Profiler curve in Figure 7[A].

The curve for *CachePct*=4% has the same basic shape as the curve for *CachePct*=5%; however, the ART values are higher, and the difference in ART values between

CachePct=4% and *CachePct*=5% increases as *CUPLevel* increases. This occurs because the smaller cache size causes increased cache contention for *CachePct*=4%. Here, the cache miss rate for *CachePct*=4% is higher than the rate for *CachePct*=5%. This leads to longer data request queues, and higher overall ARTs.

The curve for *CachePct*=6% has the same general shape as the curve for *CachePct*=5%, but has lower ART values. This is due to decreased cache contention – a larger cache retains more useful profiles, leading to fewer requests for the profile warehouse, and shorter data request queues. This, in turn, leads to lower ARTs.

Based on the results of this experiment, we conclude that larger cache sizes will lead to lower ARTs, while smaller cache sizes will lead to higher ARTs.

6.6 Effect of Varying Lookahead Distance

In this experiment, we show the effect of prefetching profile data. Figure 8[A] shows four curves, each showing the change in ART for different *lookahead distances* (LDs) (see Section 5 for a discussion of LD) as *CUPLevel* increases, while the *CachePct* and the *RuleBaseSize* remain constant at 5% and 375K, respectively.

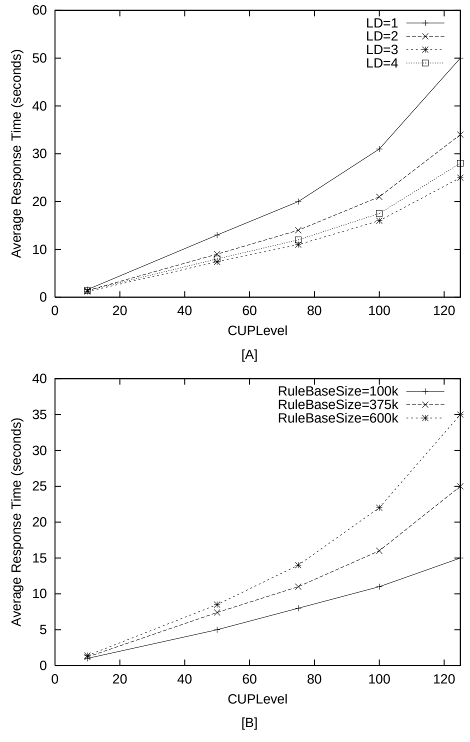

Fig. 8 [A] Effect of Varying *Lookahead Distance*; [B] Effect of Varying *RuleBaseSize*

The curve for LD=3 is exactly the same as the curve for *CachePct*=5% curve in Figure 7[B]. The curve for LD=2 has the same shape as the LD=3 curve, but appears above the curve for LD=3. This occurs because the system *caches ahead* further for LD=3 than for LD=2. Thus, needed items are less likely to be in cache for LD=2 than for LD=3. This leads to higher response times for LD=2 than for LD=3. This line of reasoning extends to the curve for LD=1 (which also has a shape similar to that of the LD=3 curve) and explains why the ART values for LD=1 are higher than for LD=2, i.e., as the lookahead distance decreases, ART increases.

The curve for LD=4 has a shape similar to the LD=3 curve, but has higher ART values than the LD=3 curve. This occurs because the Profiler with LD=4 *looks too far ahead*, i.e., replaces cache items that are highly likely to be reused with items that have been fetched in anticipation of their need.

Based on the results of this experiment, we conclude that there exists an optimal LD. Clearly, choosing an LD that caches too far ahead will have an effect similar to that of not caching far enough ahead – both lead to sub-optimal ARTs.

6.7 Effect of Varying the RuleBaseSize

Figure 8[B] shows curves for the change in ART for *RuleBaseSizes* of 100K, 375K, and 500K rules as *CUPLevel* increases, while the *CachePct*, and LD remain constant at 5% and 3, respectively. We note that the curve for 375K is exactly the same as the *CachePct*=5 curve in Figure 7[B]. The curve for 100K rules is similar to the curve for 375K rules, but appears below the curve for 375K rules, i.e, shows lower overall ART values. Clearly, a smaller rulebase size will provide lower data warehouse response times, which results in lower response times for the eGlue Server.

Similarly, a larger rulebase size of 600K results in higher warehouse response times. This is why the 600K curve has a shape similar to the 375K curve, but shows higher ART values than the 375K curve.

Based on the results of this experiment, we conclude that larger rulebases lead to higher overall ARTs, while smaller rulebases give lower ARTs.

7 Conclusion

In this paper, we described our experience building a scalable, real-time interaction engine for web personalization. We described our model of an e-commerce web site and how users navigate the site. We next discussed the overall architecture of the system and the underlying technical details of the eGlue Profiler. Finally, we evaluated the performance of our system.

Our performance evaluation showed that the eGlue Server (i.e., FastPump and the Profiler) outperforms the combination of Oracle with the profiler. In addition, we found that a larger cache size provides better response times that a smaller cache size, that a smaller rulebase size give better performance that a larger rulebase size, and that there exists an optimal lookahead distance.

Our plans for future work include designing and implementing a distributed e-commerce architecture, expanding the site and interaction model to better capture actions and data on a web site, and developing improved mining techniques to include clustering mechanisms, e.g., multiple levels of granularity, and temporality, the notion of time spent on a page.

References

1. Accrue. Accrue technologies. www.accrue.com, 1999.
2. R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 207–216, May 1993.
3. R. Agrawal and R. Srikant. Mining sequential patterns. In *Proceedings of the Eleventh International Conference on Data Engineering*, pages 3–14, March 1995.
4. Andromedia. Likeminds. www.andromedia.com/products/likeminds, 1999.
5. P. Atzeni, G. Mecca, and P. Merialdo. Semistructured data in the web: Going back and forth. *SIGMOD Record*, 26(4):16–23, 1997.
6. J.S. Breese, D. Heckerman, and C. Kadie. Empirical analysis of predictive algorithms for collaborative filtering. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, pages 43–52, July 1998.
7. A.G. Buchner, M. Baumgarten, S.S. Anand, M.D. Mulvenna, and J.G. Hughes. Navigation pattern discovery from internet data. In *Proceedings of WEBKDD'99: Workshop on Web Usage Analysis and User Profiling*, 1999. http://www.acm.org/sigs/sigkdd/proceedings/ webkdd99/toconline.htm.
8. P.K. Chan. A non-invasive approach to building web user profiles. In *Proceedings of WEBKDD'99: Workshop on Web Usage Analysis and User Profiling*, 1999. http://www.acm.org/sigs/sigkdd/proceedings/ webkdd99/toconline.htm.
9. E. Cohen, B. Krishnamurthy, and J. Murphy. Efficient algorithms for predicting requests to web servers. In *Proceedings of the 18th Conference on Computer Communications*, pages 284–93, March 1999.
10. R. Cooley, B. Mobasher, and J. Srivastava. Data preparation for mining world wide web browsing patterns. *Knowledge and Information Systems*, 1(1):5–32, 1999.
11. C.R.Cunha and C.F.B. Jaccoud. Determining www user's next access and its application to pre-fetching. In *Proceedings of the 2nd IEEE Symposium on Computers and Communications*, pages 6–11, July 1997.
12. C. Cunha, A. Bestavros, and M. Crovella. Characteristics of www client-based traces. Technical Report 1995-010, Boston University, April 1995.
13. A. Datta, H. Thomas, and K. Ramamritham. Curio: A novel solution for efficient storage and indexing in data warehouses. In *Proceedings of 25th International Conference on Very Large Data Bases*, pages 730–733, 1999. NB: Curio is the former name of the FastPump data warehouse.
14. A. Datta, D. VanderMeer, K. Ramamritham, and Sham Navathe. Toward a comprehensive model of the content and structure, and user interaction of a web site. In *Proceeding of the Workshop on Technologies for E-Services (in cooperation with VLDB 2000)*, 2000.
15. D. Florescu, A. Levy, D. Suciu, and K. Yagoub. Optimization of run-time management of data intensive web sites. In *Proceedings of the 25th VLDB Conference*, pages 627–638, September 1999.
16. Inc. Manna. Frontmind. http://www.mannainc.com/.
17. V.N. Padmanabhan and J.C. Mogul. Using predictive caching to improve world wide web latency. *Computer Communication Review*, 26(3):22–36, July 1996.
18. Net Perceptions. Net perceptions recommendation engine. www.netperceptions.com, 1999.
19. K. Perine. Ftc backs its online privacy report. In *The Standard*, May 25, 2000. Available via http://www.thestandard.com/article/ display/0,1151,15439,00.html.
20. Personify. Personify technologies. www.personify.com, 1999.
21. F.F. Reichheld and W.E. Sasser. Zero defections: quality comes to services. *Harvard Business Review*, 68:105–7, September-October 1990.
22. C. Shahabi. Knowledge discovery from users web-page navigation. In *Proceedings of RIDE'97 - Seventh International Workshop on Research Issues in Data Engineering*, 1997.
23. U. Shardanand and P. Maes. Social information filtering: algorithms for automating "word of mouth". In *Conference Proceedings on Human Factors in Computing Systems*, pages 210–217, May 1995.
24. A. Silberschatz, H.F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw Hill, 1998.
25. M. Spiliopoulou, L.S. Faulstich, and K. Winkler. A data miner analyzing the navigaitional behavior of web users. In *International Conference of ACAI'99: Workshop on Machine Learning in User Modelling*, 1999.
26. M. Spilioupoulou, L.C. Faulstich, P. Atzeni, A. Mendelzon, and G. Mecca. Wum: A tool for web utilization analysis. In *International Workshop WebDB'98: World Wide Web and Databases*, pages 184–203, 1998.
27. R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *Advances in Database Technology - EDBT'96, 5th International Conference on Extending Database Technology*, pages 3–17, March 1996.
28. A.S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 1992.
29. Broadvision Technologies. Broadvision. www.broadvision.com, 1999.
30. Engage Technologies. Engage e-commerce product suite. www.engage.com, 1999.
31. J. Wang. A survey of web caching schemes for the internet. *Computer Communication Review*, 29(5):36–46, October 1999.