A Personal Assistant for Web Database Caching

Beat Signer, Antonia Erni, and Moira C. Norrie
{signer, erni, norrie}@inf.ethz.ch

Institute for Information Systems
ETH Zurich, CH-8092 Zurich, Switzerland

Abstract. To improve the performance of web database access for regular users, we have developed a client caching agent, referred to as a personal assistant. In addition to caching strategies based on data characteristics and user specification, the personal assistant dynamically prefetches information based on previously monitored user access patterns. It is part of an overall multi-layered caching scheme where cache coherency is ensured through cooperation with a server-side database caching agent. The personal assistant has been implemented in Java and integrated into the web architecture for the OMS Pro database management system.

1 Introduction

Significant gains in performance for web access to databases can be achieved with a shift away from server-side access architectures towards web-based client-server database architectures. By moving components of the database system to the web client, user access times can be improved by reducing both server load and communication costs. Communication costs can be decreased through a combination of using socket-based communication instead of the slower Hypertext Transfer Protocol (HTTP) and eliminating some of the communication through client-side caching mechanisms and processing. In addition, further reductions in user response times can be achieved by predictive prefetching of data into an active cache based on observed user access patterns.

We have implemented such a web-based architecture for the object-oriented database management system OMS Pro [NW99,KNW98] in which client-side DBMS components include a generic object browser, schema editor, presentation editor and also caching agents [ENK98,EN98]. The client-side caching framework may optionally contain a *personal assistant* which provides registered users with an adaptive, persistent cache on their local machine. The personal assistant monitors a user's access behaviour in order to dynamically prefetch data in accordance with predicted access patterns. Cache coherency is ensured in cooperation with the server-side database agent where communication is established through the use of persistent socket connections between the client-side agents and the database agent. Changes to the database can therefore be propagated to all client-side agents ensuring that only current information is stored in the client agent's caches. The resulting multi-level caching scheme, together with its

prefetching strategies, has enabled us to significantly improve performance while keeping the caching overhead small.

The personal assistant's cache provides benefits not only in terms of performance, but also in terms of mobile computing in that it allows objects of interest to be explicitly cached and then accessed without an active connection to the server side. We note however that this is not the main purpose of the cache and full, disconnected operation of OMS Pro databases with bi-directional, synchronised updates is supported through a general database connectivity mechanism, OMS Connect [NPW98].

Our architecture is specifically designed to cater for the requirements of web databases and the access patterns of their users in contrast to other proposals for advanced web caching schemes [Wes95,CDF+98,CZB98,BBM+97]. General web caching strategies typically do not cache dynamically generated documents, nor do they deal with the case of client-side applications retrieving query results from a database on the server side. In contrast, we recognise the browsing-style access patterns of many web-based information systems where users navigate from single entry points and frequently issue the same or similar requests to a database as is reflected by the heavy use of the "back"-button in web browsers.

Clearly, building client-server database systems on the web is similar in concept to any form of client-server database architecture. However, the fact that we access the database via a web browser with the client components implemented as Java applets presents some special technical challenges including problems of security restrictions, the general dynamic and temporary nature of the client-server relationship, interaction with browsers and non-uniformity across different browsers.

In Sect. 2, we present our general web-based client-server architecture in terms of the various caches and the agents managing them. Section 3 goes on to detail the functionality of the personal assistant and the active cache it maintains. The personal assistant's user interface is outlined in Sect. 4 and some implementation details are given in Sect. 5. Performance issues and preliminary measurement results are presented in Sect. 6, while concluding remarks are given in Sect. 7.

2 Web Client-Server Database Architecture

Currently, most architectures for web access to databases are server-based in that the whole database functionality resides on the server side [ACPT99]. To avoid the processing overheads associated with CGI architectures and provide support for interactive user sessions, Java servlets provide a general means of improving the efficiency and ease of development of server-based application systems [HCF98]. In the case of database systems, several general and DBMS-specific web servers tailored for database system access have been developed. However, with all of these solutions, processing tends to remain on the server side and communication is via the HTTP protocol with results presented as Hypertext Markup Language (HTML) documents.

While Java applets are being used increasingly to move processing to web clients, it tends to be the case in web information systems that there is a separation of application and database logic and all components of the DBMS remain on the server side. For example, an application applet may use a JDBC database server to retrieve information from a database [Ree97]. There are however disadvantages of this approach resulting from the fact that the applet and the database are only loosely-coupled, meaning that the applet cannot exploit the full functionality of the database system and has to do all the work of interpreting and assembling query results and then generating their presentation.

We are instead interested in making a DBMS web-capable in the sense that components of the DBMS can execute on the client side. Thus, it is not the case of client applications connecting via the web to server database systems, but rather adapting the DBMS to a dynamic client-server architecture in which the clients run through a web browser. The client-server architecture is dynamically generated when downloading the client as an applet over the Internet.

A major advantage of this approach is that database-specific caching strategies can be implemented on the client machine. Other advantages come in terms of the ease of providing clients with extended database functionality, eliminating unnecessary mappings through intermediate standard interfaces and generally providing the look and feel of a DBMS through the web for both developers and end-users. To this end, we have developed web components for OMS Pro.

In addition to client-side components for browsing and caching, we also have schema and object presentation editors which execute on the client side and enable databases to be designed and prototyped over the web. The general browser and developer components are heavily used in teaching since they enable students to browse, query, update and create databases via the web. Furthermore, they are valuable in providing on-line demonstrations of our OMS Pro system. The caching components with their own presentation layer are part of the Internet OMS framework which can be used by any web application system based on OMS Pro.

In Fig. 1, we show the web architecture of the resulting Internet OMS system [EN98], with respect to the general querying and browsing interface and cache management.

OMS Pro has an integrated server and we can therefore assume that the database on the server side is ready to accept requests. We show that, although the database is on the server side, it may be located on a machine other than that of the web server.

Initial access to the system is through a normal web browser using the HTTP protocol as indicated by step 1. An applet is returned and a front-end agent started on the client side. This front-end agent is responsible for communication with the server side, presentation of objects and the maintenance of a short-term session cache. Thus, when query requests are issued, the front-end agent checks whether they were previously requested in which case the results are already stored in the local session cache. Therefore, if users browse a database, going "back" to previous objects, no communication with the server is required.

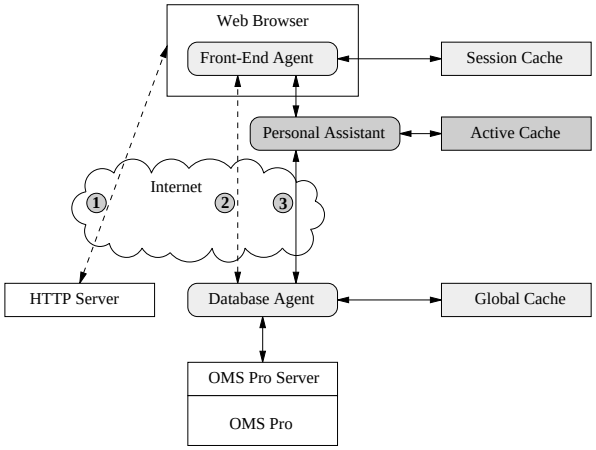

Fig. 1. Architecture overview of Internet OMS

A query result contains the required database data with additional metadata information such that the front-end agent can generate the appropriate view of the object and present it to the user.

The front-end agent sends requests to a database agent on the server side as shown in step 2. The database agent manages a global cache and notifies client-side caching agents of database updates that invalidate data objects in their caches. This is possible since we use two-way communication over persistent connections instead of HTTP. The global cache is beneficial across the user community in caching query results already requested by other users or by the same user in previous sessions. This proves to be extremely advantageous since web information systems usually have only a few entry points and the queries corresponding to points at the beginning of navigation paths tend to be repeated frequently and are relatively stable in terms of their results.

To further improve the performance for regular users, it is possible to locally install an adaptive caching agent, referred to as a *personal assistant*, which manages a client-side persistent cache. The personal assistant registers with the database agent on the server side and its cache is maintained in cooperation with the database agent. It manages a short-term session cache and long-term personal and prefetching caches. Separate caches are provided for image files. The personal assistant maintains a user access profile which is used to dynamically prefetch data during user sessions based on access predictions. In addition, users can explicitly specify data of interest and the personal assistant ensures that this data is always stored in the local cache and is current. The caches managed by the personal assistant are persistent in that they are written to disk every time the personal assistant is shut down (for security aspects see Sect. 5). In the case that a personal assistant is running on a client machine, the database agent informs the front-end agent and all communication is redirected through the personal assistant as indicated by step 3 in Fig. 1.

Prefetching of information on the web is not a new idea. Several approaches have already been proposed, such as proxy servers prefetching and storing popular pages based on the observation of client access patterns, thereby allowing sharing of retrieved results among clients [CY97]. A proxy-side pre-push technique by [JC98] relies on proxies sending (pushing) predicted documents from the proxies to the browsers. The term *speculative service* used by [Bes95] defines prefetching in terms of server-side statistics and actions, as opposed to clients controlling what and when documents are prefetched. Our approach goes more in the direction of cooperative caching, having client and server agents cooperate with each other to prefetch the correct data. A similar prefetching approach for general web caching is described in [SW97]. These approaches all concentrate on prefetching HTML documents from proxies or web servers to web browsers and neither proxies nor browsers cache dynamically generated pages. Since we are interested in prefetching dynamically generated query results, we provide cooperating agents which prefetch such results, not by using proxy or browser caches, but rather by providing our own caching hierarchy as shown in the dynamic client-server architecture in Fig. 1.

The partitioning of the local active cache and combination of various caching and prefetching mechanisms has enabled us to significantly improve performance while keeping the caching overhead in terms of space, processing and communication costs to an acceptable level. In the following sections, we describe the operation of these various cooperative agents and their caches in detail.

3 Personal Assistant and its Active Cache

There are three different "subcaches" building up the active cache managed by the personal assistant: The *session cache*, the *personal cache* and a *prefetching cache* (see Fig. 2). These three caches are disjoint, i.e. the same information will be cached in at most one of the caches. They differ from the front-end agent's session cache in that their content is made persistent and is available in future sessions.

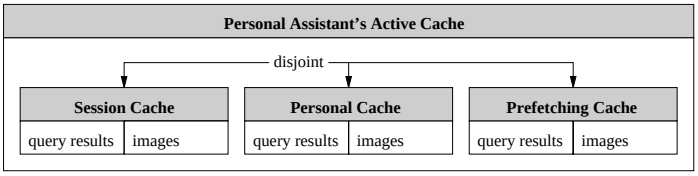

Fig. 2. Personal assistant's caching framework

In recognition that multimedia files, especially images, form a vital part of most web-based information systems, we provide special support for the caching of images. Each of the three subcaches is divided into a part for caching query results and a part for caching images. We have chosen this approach because the image sizes are large relative to the query results and often the same image

is required by different queries. Due to this separation, an image will always be cached at most once, even if it is needed by more than one query. The caching space saved by this strategy, i.e. by eliminating image redundancy, is used to cache additional query results. Every time a query result is removed from a cache, we check if an associated image is still required by any other queries before removing it from the cache. This was done by introducing a reference counter indicating the number of queries using the image and implementing a *garbage collector*.

Note that the personal assistant will not cache any referenced HTML documents or multimedia files other than images, e.g. sound and movies.

3.1 Session Cache

The session cache, as its name suggests, acts as a short-term cache and will only cache query results recently used. Therefore the caching strategy used by the session cache is a simple least recently used (LRU) strategy (for further information about caching strategies see [Tan92,SG94,Fra96]). By using an LRU replacement strategy, the session cache will profit from the locality of a single session.

3.2 Personal Cache

A user may explicitly specify data of interest by defining query results to be permanently cached. These query results are then managed by the personal cache. To ensure that the data in the personal cache is always up to date, every time a new entry is added to the cache, the personal assistant registers it with the database agent on the server side. Whenever data changes in the database, the database agent checks whether registered queries are affected. If so, the database agent sends a cache invalidation message to the corresponding personal assistants, forcing them to update their caches to maintain cache consistency. Removal of a query result from the personal cache will cause the query to be unregistered with the database agent. Further, every time a personal assistant is started, the entire contents of its persistent caches are registered with the database agent.

The personal cache does not have a caching strategy, i.e. if there is no more place in the cache, the cache manager will not select a victim and replace it by the new entry to be added. If a user tries to add a new query result to the personal cache and there is no more space, the caching agent informs him that he either has to manually remove some entries from the cache or to increase the personal cache size at runtime. This "caching strategy" makes sense because the user wants to be sure that all the query results added to the personal cache will be present in the future and not be deleted by the cache manager without his knowledge.

Of the three subcaches, the personal cache has the highest priority since the user explicitly defined the query results to be permanently cached. Therefore, if an entry is added to the personal cache and is already part of either the

session or prefetching cache, it will be removed from these caches to avoid cache redundancy. Further, a query result will not be added to the session or the prefetching cache if it is already present in the personal cache.

3.3 Prefetching Cache

In addition to the short-time session cache and the user defined personal cache, the prefetching cache will analyse user behaviour and try to predict the information the user will need in the near future. It dynamically adapts to a user's preferences without any interaction.

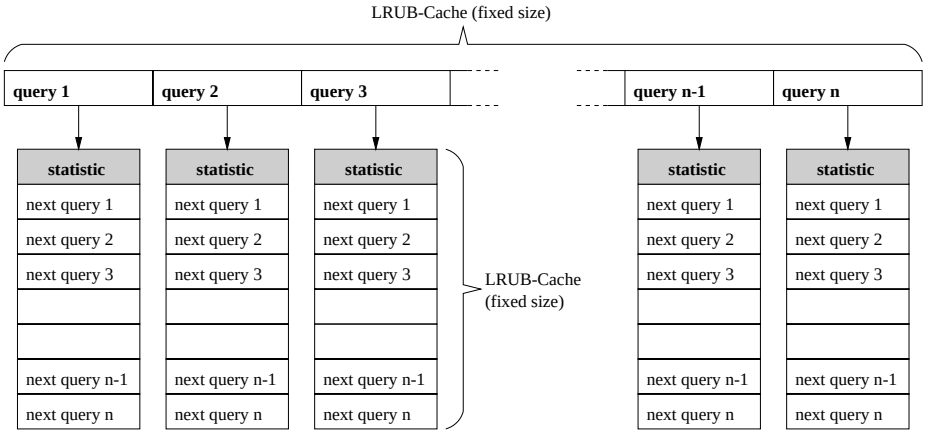

Fig. 3. Structure of the prefetching statistic

For each query requested by the user (*query 1* to *query n* in Fig. 3), the prefetching cache maintains a statistic of the succeeding queries and the number of times they were requested immediately after the current query (*next query 1* to *next query n* in Fig. 3). The statistic is in turn represented by a cache of limited size. So for each query, only a limited amount of possible succeeding queries – the queries which will most likely follow the current query – are considered in the statistic. The simplest replacement strategy for these statistic caches would be an LRU strategy. The problem is that these caches should act as long-term caches, i.e. the statistic will grow over a long period. Therefore we also have to consider the number of accesses for each cache entry when selecting a victim for removal. As a result, the LRU strategy is extended by the number of times a cache entry was requested resulting in an LRUB caching strategy where the number of cache hits acts as a *bonus* for entries which where often used in the past but have not been accessed recently. Considering these facts, we use the following weighting formula for cache entry i

$$weight_i = \frac{\alpha + (1-\alpha)\, H_i}{T_i} \qquad 0 \leq \alpha \leq 1 \tag{1}$$

where H_i is the number of hits for entry i and T_i is the time since the last access of entry i. Each time an element has to be added to a full cache, the cache entry with the smallest weight will be removed from the cache. The influence of earlier cache hits can be adjusted by changing the value of α in (1). The smaller the value of α, the greater the weighting given to the number of cache hits (by setting $\alpha = 1$ we have a simple LRU cache).

The queries with their statistic are themselves maintained in a cache of limited size using the same LRUB caching strategy. Therefore, only for the queries most likely to be requested in the future will a statistic of the succeeding queries be present in the prefetching statistic. Each time a user requests a new query, the personal assistant checks whether there exists a statistic for it. If a corresponding statistic is present, the queries with the highest weights (the queries which will most likely succeed the current query) are selected. The corresponding query results will then be prefetched into the prefetching cache. The maximal number of query results to be prefetched is defined by the prefetching width. By increasing the prefetching width, the cache hit rate can be slightly improved but, as a negative effect, the database agent will have to handle an additional amount of queries.

The prefetching cache will profit from the fact that, in web interfaces to databases, users often browse in a similar way, i.e. as users navigate through the database, there exist "chains of queries" often requested in the same order. While a user analyses the result of one query, the personal assistant prefetches the predicted next query results. Over slower network connections, this active prefetching leads to better usage of the available bandwidth and may employ otherwise paid for, but idle, modem connection time.

The great advantage of the proposed dynamic prefetching mechanism over a "static" strategy of globally caching the most frequently used query results is that the prefetching cache, like a sliding window, always focuses on the current query. As a result, it only contains the data most likely required in the near future and therefore needs much less memory than a global strategy trying to achieve the same hit rate.

4 Agent User Interfaces

As can be seen from the description of the active cache in the previous section and the general client-server architecture in section 2, the overall system comprises many levels of caches and associated agents. It is therefore essential that tools be provided to facilitate system installation, administration and tuning. For each agent – database, front-end and personal assistant – a graphical user interface is provided that enables both the agent and associated cache parameters to be set and the operation of both the agent and the cache to be monitored. An example of such an interface is given in Fig. 4 which shows the graphical user interface of the personal assistant.

The parameters option enables the agent's associated properties, such as server host name and port numbers for communication with other agents, to be

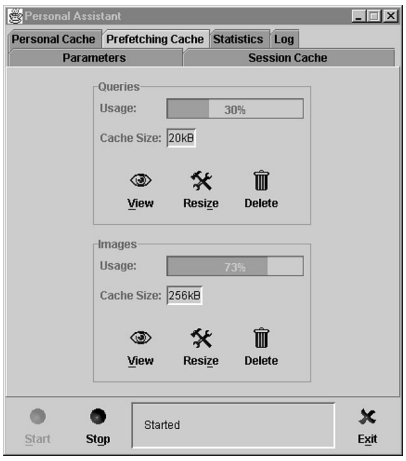

Fig. 4. Personal assistant's prefetching cache

specified. It also provides methods to start and stop the agent. The log option enables a log of agent activities to be inspected. Further it shows additional information about processed queries, clients connecting to the agent and other network specific details. All information presented in the log view is stored in a log file and therefore also can be inspected by means of a standard text editor after the agent has been terminated. The statistics option displays information such as the number of queries processed, their execution time (with or without images), the average execution time and the percentage cache hit rate.

Options are also available for each of the caches managed by the agent – in the case of the personal assistant these are the personal, prefetching and session caches. The graphical user interface shows information about the usage and size of both query and image caches. As shown in Fig. 4, it is possible to view, resize or delete the contents of the different caches.

In Fig. 5, a view of the query part of the prefetching cache is presented. As presented, the cache consists of a number of query results with the associated number of hits shown on the right hand side.

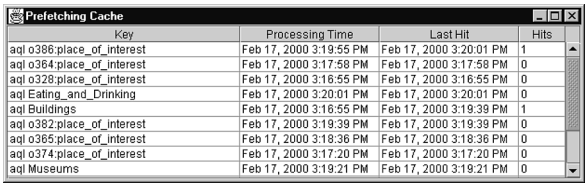

Fig. 5. Prefetching query cache content

The personal cache is the only cache the content of which can be directly manipulated by the user. To make it easy for a user to specify query results to be permanently cached, the general object browser of Internet OMS was

extended by the facility to add a query result to the personal cache by simple menu selection as shown in Fig. 6. To remove a query result from the personal cache, the user may either browse the currently cached query result and select the corresponding menu entry or directly view the personal cache and delete the query entry through the personal assistant's graphical user interface.

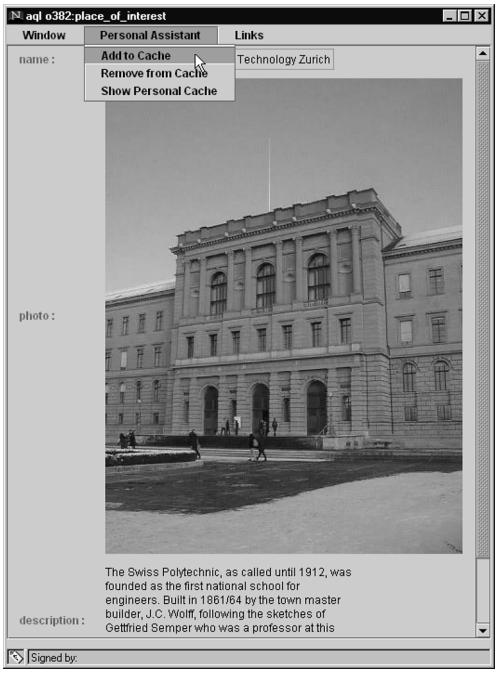

Fig. 6. Personal cache menu

When resizing the personal cache, there is a different behaviour than with the other caches. If the user tries to resize either the personal query or image cache to a size smaller than the total size of all entries currently in the cache, he will be informed that the new size is too small. Query results would have to be manually removed from the cache until the space used is less than the desired new cache size.

Note that, in the case of the prefetching cache, deletion of either the prefetching query or image cache causes only the cache entries to be deleted and not the statistics on which the actual prefetching is based.

5 Implementation

All web client components of Internet OMS and also the server-side database agent are implemented in Java. The front-end agent is implemented as a Java

applet accessed via a web browser. Optionally a personal assistant may be downloaded and installed as a local Java application.

By default, Java applets have several security restrictions including the fact that they can only open network connections to the host machine from which they were downloaded. In the case of our architecture, this presents a problem as it is necessary for the front-end agent applet to be able to communicate with a locally installed personal assistant.

Fortunately, recent browsers have provisions to give trusted applets the ability to work outside the usual restrictions. For this power to be granted to an applet, additional code has to be added to the applet requesting permission to do any "dangerous" actions and it has to be digitally signed with an unforgeable digital ID (private key). Each time a user runs a signed applet requesting some form of access to the local system, the browser checks its granted privileges for the corresponding signer. If the user already granted the requested additional power to the signer during an earlier session, the browser allows the access without interrupting the user's activities; otherwise the browser will show a Java security dialog box asking if the user will trust the developer the applet was signed by.

Every time a personal assistant is started, it automatically sends its IP-address and listening port number to the database agent. When a new front-end agent is executed on that client machine, it first connects to the database agent, sending it the IP-address of the user's machine and asking for the IP-address and port to which it should connect. The database agent will check if there exists a registered personal assistant with the same IP-address as the address received from the front-end agent. If a personal assistant is registered on the client, the database agent will send the IP-address and port of the personal assistant to the front-end agent. Otherwise the database agent will send its own IP-address and port so that the front-end agent connects directly to the database agent. For a user starting the front-end applet, the process will be the same whether he has installed a personal assistant or not.

6 Performance

To prove the efficiency of our agent architecture, we have built performance predictive models for the caching hierarchies of our system as described in [MA98]. We provide a formal way of dealing with performance problems of our architecture based on the theory of queuing networks. Performance models have been developed for the case where no caches are used at all, where only a global cache on the server side is used and also for the system taking advantage of the personal assistant's and front-end agent's caches [Ern00]. In this paper, we present first measurement results showing the advantages of using our personal assistant to reduce user response times over the case of accessing the database directly.

For the measurements, we queried the *City Database* which was developed to teach object-oriented database concepts in various lecture courses using the

OMS Pro system. The database contains information about cities and their places of interest such as hotels, museums, sightseeing tours etc.

The public web-interface for the City Database allowed us to collect query traces from a large amount of users. Since we are interested in testing performance improvements while using a personal assistant on the client side, we were in need of user traces of single persons to simulate their browsing behaviour. To build a realistic test environment for our simulations, we stressed the server and the network with a moderate workload consisting of our own collected traces.

We tested the system for a number of individual user traces. It is obvious that the personal assistant will achieve the best results when the browsing style of a specific user has been studied over a long period. We refer to this as the *best case*. If the personal assistant instead does not know anything about the browsing behaviour of a user, it is likely that the predicted queries will not be the next ones requested by the user and therefore the wrong data is prefetched into the personal assistant's cache. This is the *worst case* that can occur. The querying behaviour of every user lies somewhere between these two boundaries.

How can these two cases be analysed? For the best case, we provide the personal assistant with history data of user behaviour studied in previous sessions. In addition, we run the simulation feeding the system with one query after the other as provided by the query trace reproducing exactly the browsing behaviour of an individual user. The worst case can be simulated by providing no history data to the personal assistant and selecting queries from the query trace file at random. By randomly selecting the queries, the prefetching of the personal assistant is of little or no benefit. The two scenarios were simulated for several users without and with a personal assistant installed. The average response times are shown in Fig. 7.

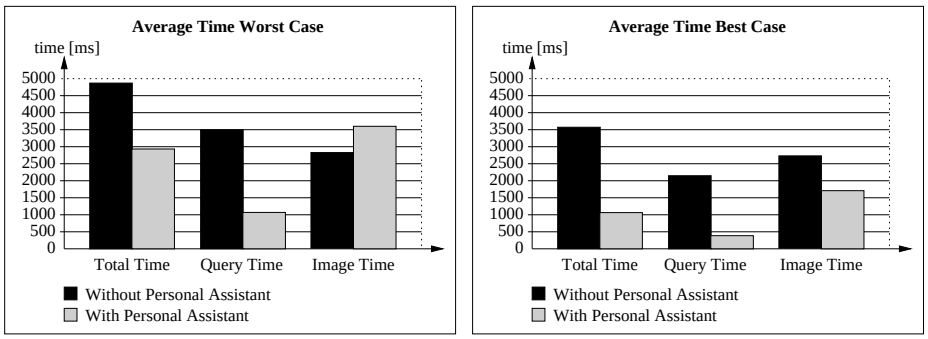

Fig. 7. Average response times

The left bar chart shows the results for the worst case where no history data is delivered to the personal assistant and the queries are randomly selected from the query trace. The *total time* represents the average time to query the database, receiving not only the data but also all images a query result contains. The *query time* is the average time required to retrieve data only (without any

images). Finally, the *image time* represents the mean time necessary to fetch images. As shown, the total response time is reduced by the personal assistant even in the worst case scenario where absolutely no information about the user's browsing behaviour is available. The reduced response times result from the fact that the front-end agent's session cache size is increased by the overall cache size of the personal assistant.

The right part of Fig. 7 shows the results for the best case where the personal assistant's prefetching cache reduces user response times significantly. These results were achieved with a relatively small caching overhead in terms of space. For the database agent, a global cache of 20 kBytes is used while no image cache is necessary, since all images are already stored on the web server. The personal assistant has an image cache of 512 kBytes and a data cache of 40 kBytes counting the personal, prefetching and session cache together. In addition, we also have the session cache of the front-end agent, storing 20 kBytes of data and 512 kBytes of images. The query caches can be kept so small since the results contain data as well as metadata only in textual form. For images, only the references are listed within the query results, whereas the related pictures are stored in the image cache. The number of query results to be prefetched (prefetching width) is set to three, i.e. for each query at most three queries will be executed and the results prefetched. Of course, using larger caches would provide additional advantages in performance since more objects and pictures could be stored, but already with relatively small caches significant performance gains were achieved.

An interesting phenomenon occurs with the average image response times in the worst case scenario (left bar chart of Fig. 7). A more detailed version of the image response times with four different user sessions is shown in the left part of Fig. 8.

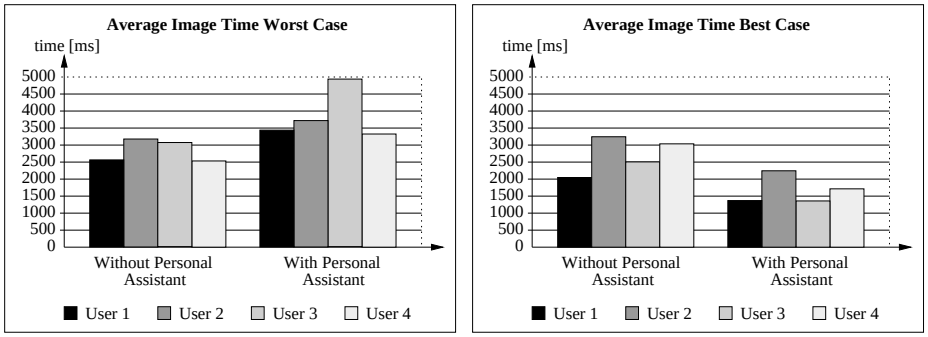

Fig. 8. Image response times

For all four users more time is required to download the images with a personal assistant installed. If the front-end agent requests an image and the image is not in the personal assistant's cache, it first has to be downloaded into the personal assistant's image cache and then forwarded to the front-end agent. This

overhead becomes significant if the personal assistant's image hit rate is low (the average cache hit rates are shown in Table 1). Otherwise when correct data is prefetched as shown in the right part of Fig. 8, time for accessing images is drastically reduced.

Table 1. Average cache hit rates

	Worst Case		**Best Case**	
	queries	images	queries	images
Front-End Agent	13%	4%	33%	29%
Personal Assistant	29%	28%	75%	68%

The results shown by these preliminary experiments are promising. Additional future simulations are planned for heavy workloads and larger data and query sets. For the basis of these experiments, we first want to record typical querying patterns for other operational web-based application systems based on the OMS Pro database.

7 Concluding Remarks

In this paper, we presented our web-based agent architecture for accessing databases over the Internet. Cooperating agents on the client and server sides are responsible for managing local and global caches and ensuring cache coherency. In addition, a personal assistant can be installed on the client side to further improve user response times. The personal assistant monitors user behaviour and assists the user by prefetching the most frequently requested data into a local prefetching cache. Users may further explicitly declare data of interest to ensure that this data is permanently stored in the personal assistant's local cache.

Our architecture has been tested by simulating user sessions and comparing the results achieved with and without a personal assistant. First results show that significant gains in performance are achieved through the installation of a personal assistant on the client side. Even in the worst case scenario where the personal assistant cannot take advantage of learnt query access patterns for predictive prefetching of data, faster access to the data was achieved resulting from the increased overall cache size. Further, the multi-level caching scheme, together with its prefetching strategies, has enabled us to achieve significant performance improvements while keeping caching space to an acceptable level.

References

[ACPT99] P. Atzeni, S. Ceri, S. Paraboschi, and R. Torlone. *Database Systems: Concepts, Languages and Architectures.* McGraw-Hill, 1999.

[BBM+97] M. Baentsch, L. Baum, G. Molter, S. Rothkugel, and P. Sturm. Enhancing the web infrastructure — from caching to replication. *IEEE Internet Computing*, 1(2):18–27, March/April 1997.

[Bes95] A. Bestavros. Using speculation to reduce server load and service time on the www. In *Proceedings of the International Conference on Information and Knowledge Management, CIKM'95*, Baltimore, MD, November 1995.

[CDF+98] R. Cáceres, F. Douglis, A. Feldmann, G. Glass, and M. Rabinovich. Web proxy caching: the devil is in the details. In *Proceedings of Workshop on Internet Server Performance (WISP'98)*, Madison, WI, June 1998.

[CY97] K. Chinen and S. Yamaguchi. An interactive prefetching proxy server for improvement of WWW latency. In *Proceedings of the Seventh Annual Conference of the Internet Society (INET'97)*, Kuala Lumpur, June 1997.

[CZB98] P. Cao, J. Zhang, and 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.

[EN98] A. Erni and M. C. Norrie. Approaches to Accessing Databases through Web Browsers. *INFORMATIK, Journal of the Swiss Informaticians Society*, October 1998.

[ENK98] A. Erni, M. C. Norrie, and A. Kobler. Generic Agent Framework for Internet Information Systems. In *Proceedings of IFIP WG 8.1 Conference on Information Systems in the WWW Environment*, Beijing, China, July 1998.

[Ern00] A. Erni. *A Generic Agent Framework for Internet Information Systems.* PhD thesis, ETH Zurich, to be published 2000.

[Fra96] M. J. Franklin. *Client Data Caching: A Foundation for High Performance Object Database Systems.* Kluwer Academic Publishers, 1996.

[HCF98] J. Hunter, W. Crawford, and P. Ferguson. *Java Servlet Programming.* O'Reilly & Associates, 1998.

[JC98] Q. Jacobson and P. Cao. Potential and limits of web prefetching between low-bandwidth clients and proxies. In *Proceedings of the 3rd International WWW Caching Workshop*, Manchester, England, June 1998.

[KNW98] A. Kobler, M. C. Norrie, and A. Würgler. OMS Approach to Database Development through Rapid Prototyping. In *Proceedings of the 8th Workshop on Information Technologies and Systems (WITS'98)*, Helsinki, Finland, December 1998.

[MA98] D. A. Menascé and V. A. F. Almeida. *Capacity Planing for Web Performance: Metrics, Models and Methods.* Prentice Hall, 1998.

[NPW98] M. C. Norrie, A. Palinginis, and A. Würgler. OMS Connect: Supporting Multidatabase and Mobile Working through Database Connectivity. In *Proceedings of Conference on Cooperative Information Systems*, New York, USA, 1998.

[NW99] M. C. Norrie and A. Würgler. OMS Pro: Introductory Tutorial. Technical report, Institute for Information Systems, ETH Zurich, CH-8092 Zurich, Switzerland, March 1999.

[Ree97] G. Reese. *Database Programming with JDBC and Java.* O'Reilly & Associates, 1997.

[SG94] A. Silberschatz and P. Galvin. *Operating System Concepts.* Addison-Wesley, 1994.

[SW97] J. Sommers and C. E. Wills. Prefetching on the web using client and server profiles. Technical report, Worcester Polytechnic Institute, June 1997.

[Tan92] A. S. Tanenbaum. *Modern Operating Systems.* Prentice-Hall, 1992.

[Wes95] D. Wessels. Intelligent caching for world-wide web objects. In *Proceedings of INET'95*, Hawaii, 1995.