

Application-level Caching for Web-based Database Access

Yanqing Lu and Rajshekhar Sunderraman¹

Department of Computer Science

Georgia State University

Atlanta, GA 30303

Email: raj@cs.gsu.edu

Abstract

Many web applications, especially those that need to access databases, are time-consuming. When a large number of clients make queries of such applications at the same time, the application will appear to be very slow. This paper presents an application-level caching mechanism that speeds up the process. The center of this mechanism is a cache manager and a cache agent. The application uses the cache agent to make caching requests to the cache manager. The paper describes the design and implementation of the cache mechanism, as well as a sample web application that demos this mechanism.

¹ Contact Author

Application–level Caching for Web–based Database

Access

1. Introduction

Many web applications need to access databases. Complicated queries may be time–consuming. When a large amount of web clients ask for the same database–access application at the same time, the database will be very busy and the application will appear to be very slow. These kinds of situation are not uncommon. For example, in the late March this year, the stock market crashed. Many stock brokerage websites suddenly became very slow due to the fact that too many people wanted to access the website at the same time. A similar situation takes place each year during the holiday season before Christmas day, when millions of people want to buy gifts on the Internet. It was reported last year that some gift–selling websites were so busy that many people had to spend a whole day to buy a gift, or even worse, some were blocked out completely.

When analyzing these situations, we find that before people make a transaction or buy a gift, they usually do a large number of searches on the database. For example, they want to know the history of stocks they are interested, or information about the gift they want to buy. These are typically complicated database queries whose execution adds to the response time. Many times, the databases are stable and do not change frequently. If we can cache popular query results, the servers will not need to query the database again

and again for the same information. It will be a great relief for the website servers and thus for those who browse them.

Caching mechanism has existed in computer arena for a long time. Certain network servers cache static information, but they do not apply to dynamic data (e.g., proxy server). Various application server programs running in the server machines usually cache programs. This may increase the efficiency, but they do not cache results. The result of the same program may vary since the parameters passed to the program may be different each time the browser invokes the program. Web browsers also have cache facilities, but they cache only common web pages. They will do nothing with forms, which invoke programs in the server machine.

The question to ask is: Could we take advantage of the stability of certain query results and offer some kinds of high-level caching mechanism to reuse the query results? The answer is "Yes". Some researchers have focused on caching programming objects [1]. In this paper, we present a query-result based alternative, **application-level caching mechanism**, for Web-based database access that can be a simpler solution for the situations described above.

At the center of this mechanism is a program called **Cache Manager** running in the background. The application modules that actually do the queries communicate with the **cache manager** to accomplish the caching mechanism. The application modules store query results into cache files. The **cache manager** records and manages these files. These files will be reused when the same query results are inquired. Thus the same database query is executed only once instead of several times when web clients ask for the same result over and over again. This paper will discuss the system design,

implementation of this mechanism, as well as a sample application that is used to demo the caching mechanism.

2. System Design

2.1 System Architecture

The overall system architecture is shown in Figure 1. The cache system consists of a **cache manager**, a **cache repository**, and a **cache agent**. The **cache manager** is a stand-alone process. It communicates with application modules that serve client requests, records and manages cache file names, and returns correct instructions on cache file inquiries. The **cache repository** is a designated directory that holds cache files, which are generated by application modules through the **cache agent**. The unit of cache is a cache file. The content of the cache file is defined by the application. Typically it contains the result of one request from the client. The **cache agent** is a framework interface used by application modules to communicate with the **cache manager**. It is also responsible for writing the query results into and retrieving cached results from the **cache repository**. The system is implemented in Java.

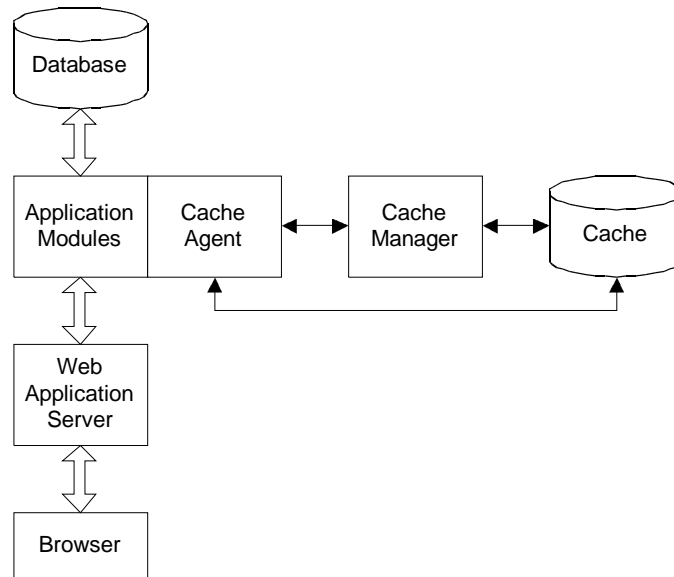


Figure 1 System Architecture

The interaction of the different components of the system is explained next. When a web client requests a page, it will invoke one of the application modules. The application module talks to the **cache manager** through the **cache agent**. The module first asks the **cache manager** for the page. The **cache manager** searches the cache records to check whether the page is in cache. If it is, the **cache manager** will answer yes and the module will output the cached page directly. Otherwise, the module will go on to query the database. When the module finishes querying the database, it will ask the **cache manager** again to check if there is someone writing the same answer to cache. If someone is already writing the same cache content, the module will only output the answer directly. Otherwise, it will output the answer to web browser as well as write the result data to cache. It will also inform the **cache manager** when it finishes writing the cache, and the **cache manager** will put the page into the cache records and it will be available for future use. The procedure is shown in Figures 2 and 3.

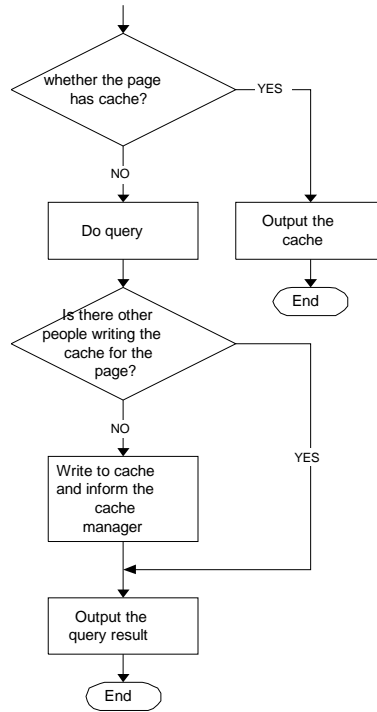


Figure 2: Flowchart for application module

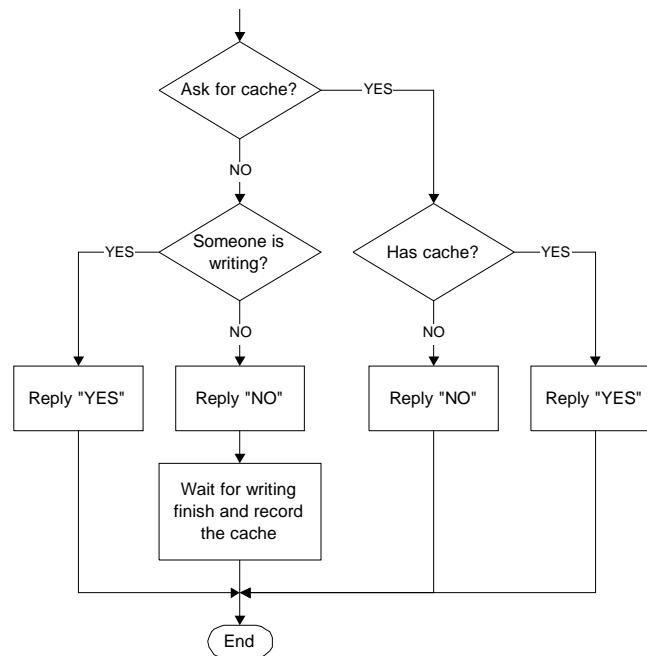


Figure 3: Flowchart for communication part of cache manager

2.2 Cache Manager Framework

The **cache manager** is the main component of the caching mechanism. It has two primary tasks:

(1) Manage the cache files.

§ Record which files are in cache

§ Record which files are being written to cache

§ Limit the number of files to a reasonable amount (implement a cache replacement algorithm)

(2) Exchange cache information with application modules.

The **cache manager** does not actually read or write cache files. It is the application modules that are responsible for calling functions of the **cache agent** to generate the cache files. How does the **cache manager** record and manage cache files? It manages file names. The main advantage of this strategy is that the application module does not need to pass the whole query result to the **cache manager**, which would be a tedious job. Thus, the **cache manager** does not depend on any particular application module, and can manage all kinds of cache files, no matter which file format the file is in. Since the **cache manager** cares only about file names, each name should be unique. This is easy to achieve if we use the criteria of queries or parameters of forms to construct the file name.

The **cache manager** also records the latest time of a cache file when it is inquired. The time value is used to delete out-of-date cache files. Although this functionality is a future work, the time value is still recorded for future use.

When the **cache manager** is shut down, the cache information is stored in a persistent storage – a file in the **cache repository** directory. When the **cache manager** is

brought up, it will first retrieve the information into memory. The **cache manager** will manage the cached files through the records in the memory, and refresh the records to the persistent storage periodically.

2.3 Framework Interface – The Cache Agent

The **cache agent** is a framework interface used by application modules to communicate with the **cache manager**. It is also responsible for writing the query results into and retrieving cached results from the **cache repository**.

In order to use the **cache manager**, the application modules need to talk to the **cache manager** correctly. Letting the applications do the connection and handle the protocol is too much burden for the application developers. Besides, the implementation of the protocol is unlikely to change from application to application. So we propose a wrapper class `CacheAgent` to implement the application side of the protocol. `CacheAgent` has the following three static methods, `checkCache()`, `readCache()`, and `checkWriting()`:

Method	Description
<code>checkCache()</code>	Responsible for asking the cache manager if a page is in the cache.
<code>readCache()</code>	Reads the desired cached content.
<code>checkWriting())</code>	Writes the page into cache if no one is already writing that page. Otherwise it will do nothing.

Table 1: Cache Agent Functions

All the application module needs to do is to pass the file name and other necessary parameters to these functions and use their return values, as shown in Figure 4. The module first uses `checkCache()` to check if the page has been cached. If yes, it uses

`readCache()` to read the cache and output. Otherwise, it will go on to do the query on the database. After the query, it will ask `checkWriting()` to see if there is someone writing the same page to cache. If there is no one, `checkWriting()` will output the page to cache and inform the **cache manager**. Then the module outputs the page to the client. These three methods are the interface of the **cache manager** framework. Any application module that want to use **cache manager** can reuse them.

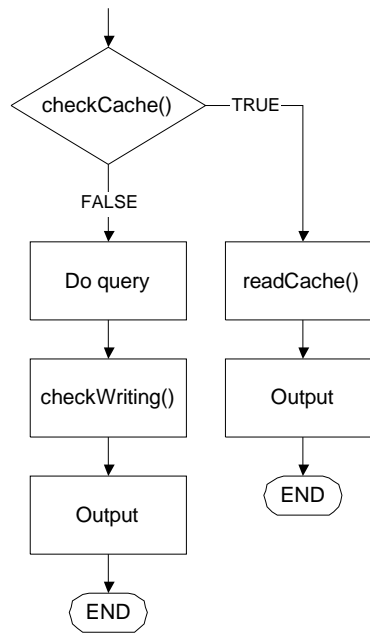


Figure 4: Flowchart for application modules using cache agent

3. Implementation

3.1 *Client–Server Architecture and Sockets*

3.1.1 Structure Design

The cache mechanism is implemented using a client–server architecture. The **cache manager** acts as a server, and the application modules are the clients. They talk to each other through TCP sockets. The client–side socket communication procedure has been wrapped into the **cache agent**.

The reason to choose this architecture is to centralize the **cache manager** and make it stand–alone, thus the **cache manager** does not need to rely on any particular application module. The application modules can run, no matter whether the **cache manager** is running or not. Similarly, the **cache manager** can run no matter whether there is any application module needing it. Furthermore, the **cache manager** can even be shared by multiple instances of the applications running on several servers, and TCP sockets conveniently support such a configuration. Only one of the servers needs to run the **cache manager**; the rest that want to use cache service can connect to the same **cache manager** through TCP.

Besides, TCP sockets are natively supported by Java, and works satisfactorily on all popular platforms like Windows and Unix, etc. The choice of using client–server architecture and sockets increases the program’s portability.

3.1.2 Cache Records

Two linked lists are maintained in **the cache manager**. One is the cache list, which records the information for each of the cache files. The other is the writing list, which records the cache files that are being written. The writing list is used to prevent two threads from writing the same file at the same time. The reason to have two linked lists is due to the fact that the files that are being written cannot be used as cache yet.

3.1.3 Protocol Sequence

The protocol sequence of the socket communication in the program is as follows:

1. Check to see if a file is cached – `CacheAgent.checkCache()`

	Application module (client side)	Cache manager (server side)
1	Send command string "cache" to inform the cache manager that it wants to check the cache records.	Receive the command string to decide whether the module wants to check the cache or write the cache. The server knows it should check the cache from the "cache" command.
2	Send the file name string to see if there is a matching record.	Receive the file name string.
3	Receive the answer.	Reply "YES" for cache hit or "NO" for cache miss.

Table 2: Protocol Sequence of Check Cache Module

2. Check to see if there may be any conflicts writing the cache – `CacheAgent.checkWriting()`

	Application module (client side)	Cache manager (server side)
1	Send command string "writing" to inform the cache manager that it wants to check to see if the cache of the current query result is already being written.	Receive the string and find out that the client wants to check the writing list .
2	Send the file name string to check to see if somebody else is writing the same cache.	Receive the file name string.

3	Receive the answer.	Send string "YES" or "NO" to inform the module. If it answers "NO", it also records the file name into the writing list .
---	---------------------	--

If the client receives "NO" from the **cache manager**, it writes the cache, then continues with the cache writing protocol sequence:

4	Send command string "FINISH" to inform the cache manager that writing to cache is finished.	Receive string "FINISH" and transfer the record from the writing list to the cache list .
---	--	---

Table 3: Protocol Sequence of Check Writing Module

3.2 Threads in the Cache Manager

There are three kinds of threads in **cache manager**. One is a **housekeeping thread** that manages the cache files; the others are **service threads** that handle requests from application modules. Of course, there is also a main thread. It creates the **housekeeping thread**, and listens to the server port and spawns the **service threads** when a request arrives.

The **housekeeping thread** runs in the background. During its execution, it periodically checks the number of cache files existing in the system to limit the amount to a reasonable number, and refreshes cache records to the file `CacheRecord.txt` periodically. Essentially it implements a cache-replacement algorithm, which will be discussed in next section. The flowchart of this thread is shown in Figure 5.

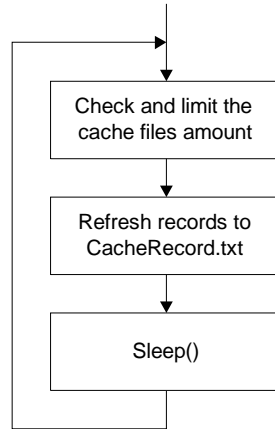


Figure 5: Flowchart of the housekeeping thread in the cache manager

3.3 Cache-replacement Algorithm: Least Recently Used (LRU) Algorithm

The **cache manager** needs to limit the number of cache files to a reasonable number. If there are too many cache files, it needs to delete some of them. It is reasonable that the page that has not been used for the longest period of time should be deleted. This approach is the Least Recently Used (LRU) algorithm [2].

Using doubly linked list is particularly appropriate for software implementations of LRU algorithm and this is the data structure we choose to maintain the cache. When a cache file is newly generated, its cache record is added to the head of the linked list. When a cache file is requested, its record is removed from the linked list, the time field of the record is updated to the current time, and then the record is re-inserted to the head of the linked list. This way, the head of the linked list is always the most recently used file and the tail is the least recently used file. When the **cache manager** needs to delete files, it always deletes from the tail. In a production environment, it may be helpful to add a tree to speed up the filename searching if the number of cache files is huge.

4 Sample Application

A sample application module is implemented to demo the cache mechanism. This application module is a servlet application related to stock market data. It selects stocks based on the growth rate and the volatility.

The data is stored in an Oracle database. In the database, there are 27 stocks. Each stock has 3 years of price data from 10/01/1997 to 9/30/2000. They are simulated data, generated by programs, which represent the close price of stock each day.

The user selects the stock criteria from a form in an HTML page. The page is shown in Figure 6. The form sends the parameters to a servlet program called "CacheAnalysis" and invokes the program. For each stock and for the period the user selects, the servlet does the following calculation. It examines the growth rate using the first day and the last day data. It calculates the exponential rate using the first day and the last day data, and uses it to calculate the ideal price of each day. The distance between the ideal price and the actual price over the ideal price is the volatility of that day. The overall volatility is the average over the whole period. The stocks that meet the criteria will be put into the result set. Since the search is an intensive database search and involves intensive calculation, the module needs a lot of time to get the result if there is no cache mechanism provided.

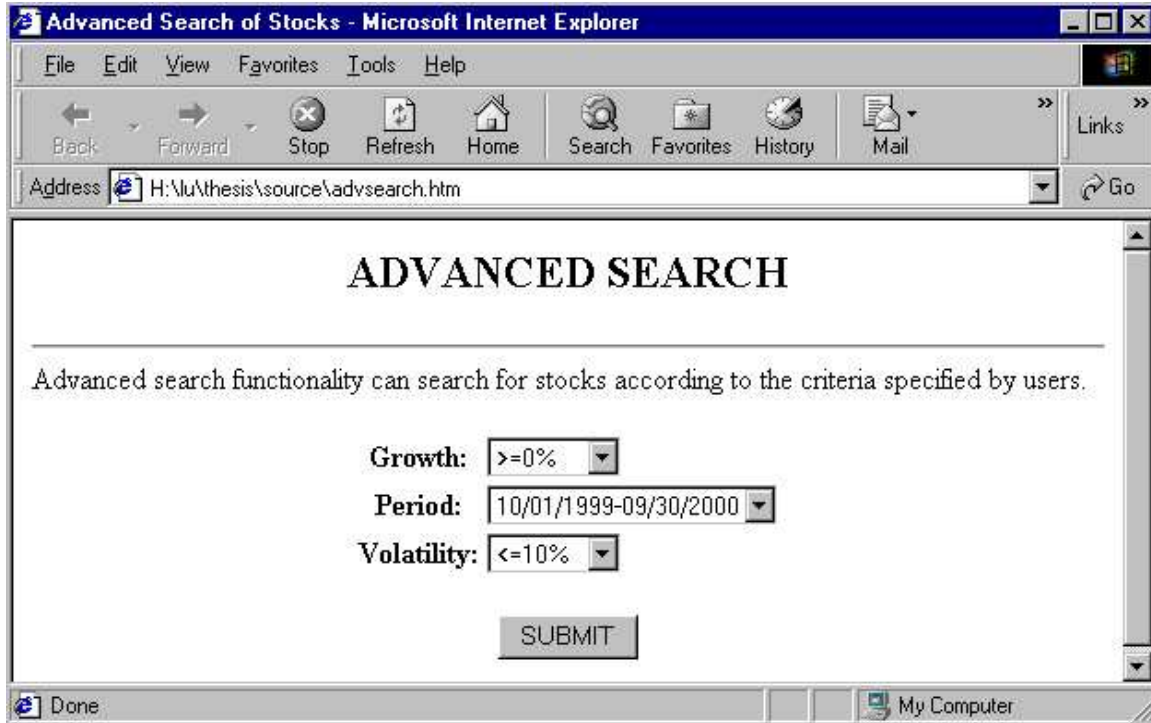


Figure 6: Advanced search page

The application program first constructs the cache file name, and passes it to the `checkCache()` method of the cache agent to check whether it is in cache. If it is in cache, the program calls the `readCache()` method to read the cache content and output it. Otherwise, the program performs the query, and passes the file name to the `checkWriting()` method to check if there is someone writing the same page. Then the program outputs the query result and terminates.

4.3 XML and XSL in the Sample Application

Each query result is stored in an XML format and is cached in an XML file. The XML files store only result data and do not indicate the visual appearance of how a browser displays them. The latter part is handled by the servlets using XSL transformation. The advantage of this strategy is that the result of one set of query

parameters can be generated regardless the display format, and its visual appearance can be easily changed using XSL according to viewers' need. When the visual appearance has to be changed, the cache doesn't need to be flushed. The data storage of the cache is also more compact, since the display information is omitted.

The result of the query in this sample is a set of stocks that qualify the query parameters. Their XML elements are some statistics about the stocks.

4.4 File Name Construction

In order to manage the cache files correctly and conveniently, each XML file is named using the criteria of the query. For example, this servlet application selects stocks based on the growth rate and the volatility, so the growth rate, the query period, and the volatility construct the XML file name. For example,

growth_10_period_1_30-SEP-2000_volatility_20.xml

is a file name which corresponds to the query that asks for those stocks that has a growth $\geq 10\%$, price history of 1 year period ending in 9/30/2000, and volatility $\leq 20\%$. The same query will ask the **cache manager** for the same XML file name.

5 Performance Statistics

In order to calculate the performance statistics of the **cache manager**, we let the sample servlet application write a log file. The application writes the machine time when it is invoked and the time when it returns. The difference between these two time values is the total time it needs to return the corresponding web page to the browser. It also writes this time value to the log file.

Table 4 shows the statistics collected on a Unix platform. The environment information is as follows: Sun Ultra 30 Workstation with 756 MB RAM running Solaris 7, Oracle 8i database server and Oracle Application Server.

Years	1		2		3	
Use Cache	NO	YES	NO	YES	NO	YES
Time Spent to Output a page (ms)	4300	19	5939	17	8075	47
	3432	24	5676	20	8301	17
	3477	26	5819	20	7993	19
	3461	27	5950	24	7944	19
	3678	31	5849	23	8220	20
	3447	27	5771	24	8002	20
	3649	28	5894	26	8045	21
	3383	25	5895	27	8067	21
	3630	26	5875	23	8268	21
	3392	25	5789	23	8316	58
	3455	16	5674	14	7903	17
Average	3573	24	5830	21	8103	25

Table 4: Statistics collected on Unix platform

From the table we can see that using cache to output a page is much faster than not using cache. Using the average 3-year data column values in Table 5 (3-year data: 8103, use cache: 25), we can calculate the effective access time for each hit ratio case by weighing each case by its probability.

1. Not using cache or hit ratio 0 %: effective access time = 8103 millisecond

2. hit ratio 20 %:

$$\text{effective access time} = 0.80 \times 8103 + 0.20 \times 25 = 6487.4 \text{ millisecond}$$

$$\text{speed-up percentage} = (8103 - 6487.4) / 8103 \times 100 \% = 19.9 \%$$

3. hit ratio 40 %:

$$\text{effective access time} = 0.60 \times 8103 + 0.40 \times 25 = 4871.8 \text{ millisecond}$$

$$\text{speed-up percentage} = (8103 - 4871.8) / 8103 \times 100 \% = 39.9 \%$$

4. hit ratio 80 %:

$$\text{effective access time} = 0.20 \times 8103 + 0.80 \times 25 = 1640.6 \text{ millisecond}$$

$$\text{speed-up percentage} = (8103 - 1640.6) / 8103 \times 100 \% = 79.8 \%$$

Of course, the higher the hit ratio, the better the performance. The hit ratio is related to the maximum cache-file number of the **cache manager**.

6 Discussion and Limitations

During the development of the system, we also performed intensive testing of the programs. The following situations are beyond the control of our system:

1. The first time the application module executes, the time it needs may appear longer than that during later executions. The cause of this is that the Java virtual machine needs to load necessary libraries, the database server also needs to load a lot of data, and thus the extra delay.
2. If we keep running the sample servlet application for a long time, say, about a hundred queries, there may be a few times that when there is actually a cache hit, the result to be displayed in the browser seems to need nearly the same time as in the case of a cache miss. We checked the log file of the execution and found that when this situation happens, the time the servlet module needs is the same as that of other cache hit cases. However, when we look at the system task manager, it shows that Internet Explore occupies over 95% of CPU time. Obviously, the IE is busy doing something at that time! That is beyond the control of the server program.

The main limitation of our approach is that not all applications can use such a caching mechanism. The query results of some applications contain fast changing data. The **cache manager** introduced in this paper is inappropriate for these applications.

7 Future Extensions

There can be several improvements and future extensions to the **cache manager**.

1. The cache record of each cache file has a time field to record the latest time the cache file is used. A new functionality can be added to the **cache manager** to delete out-of-date cache files using this time field. (For example, in some applications, a cache file that has not been used for a certain days can be considered out-of-date.)
2. The maximum number of files in the **cache manager controller** may be inconsistent with the actual max file number in the **cache manager** when the controller restarts, because it does not read from the **cache manager** at that time. It would be better that when the **cache manager controller** started, it exchanged this information with the **cache manager**, so the information could be consistent.
3. The **cache manager** can be hooked up with a trigger mechanism. Each database-access application is associated with some tables in the database. When the content of one or more of the tables changes, the cache files associated with them can be considered out-of-date, and should be deleted. So if we can record which tables relate to each cache file, then when the database changes, it can trigger the **cache manager** to delete out-of-date cache files. Furthermore, The cache record can contain information that helps to indicate what kind of table

modification should trigger the deletion, thus improving the deletion accuracy. Obviously that will be application dependent. Using such trigger mechanism can guarantee that the cache files are up-to-date.

8 Bibliography

1. Laura M. Haas, Donald Kossmann and Ioana Ursu, *Loading a Cache With Query Results*, Proceedings of the 25th VLDB Conference, Edinburgh, Scotland, 1999.
2. Abraham Silberschatz and Peter Baer Galvin, *Operating System Concepts*, Fifth Edition, Addison–Wesley, 1998.
3. Elliotte Rusty Harold, *XML Bible*, IDG Books Worldwide, Inc., 1999.
4. Elizabeth Castro, *HTML for the World Wide Web*, Peachpit Press, 1998.
5. Rajshekhar Sunderraman, *Oracle8 Programming: A Primer*, Addison–Wesley, 1999.
6. Cay S. Horstmann and Gary Cornell, *Core Java*, Sun Microsystems Press, 1999.
7. Daniela Florescu, Alon Levy and Dan Suciu, *Optimization of Run–time Management of Data Intensive Web Sites*, Proceedings of the 25th VLDB Conference, Edinburgh, Scotland, 1999.