Active Query Caching for Database Web Servers

Qiong Luo Jeffrey F. Naughton Rajasekar Krishnamurthy
Pei Cao Yunrui Li

Computer Sciences Department
University of Wisconsin-Madison
Madison, WI 53706

{qiongluo,naughton,sekar,cao,yunrui}@cs.wisc.edu

ABSTRACT

A substantial portion of web traffic consists of queries to database web servers. Unfortunately, a common technique to improve web scalability, proxy caching, is ineffective for database web servers because existing web proxy servers cannot cache queries. To address this problem, we modify a recently proposed enhanced proxy server, called an active proxy, to enable *Active Query Caching*. Our approach works by having the server send the proxy a *query applet*, which can process simple queries at the proxy. This enables the proxy server to share the database server workload as well as to reduce the network traffic. We show both opportunities and limitations of this approach through a performance study.

Keywords

Active query caching, proxy caching, database web servers.

1. INTRODUCTION

Many web sites are constructed using back-end database systems and provide form-based interface for users to submit queries. We call this kind of system a *database web server*. With the rapid growth of user accesses to the Web, database web servers encounter very heavy workloads and produce a growing percentage of network traffic.

Web caching proxies are today's main solution to improve web performance, share server workload, and reduce wide area network traffic. However, queries and responses of database web servers are un-cacheable by existing web proxies, which cache only static files. This motivates us to investigate the problem of how to answer queries efficiently at a web proxy.

In this paper, we propose a new collaboration scheme between an *active proxy* (an experimental enhanced web proxy server [4]) and a database web server. In our approach, the web server passes simple query processing ability to the proxy when needed, through a "query applet". The proxy can then not only answer queries that are an exact match to cached queries, but also queries whose results are contained in the cached results of more general queries. This increases the cache hit ratio of the proxy, and further decreases the number of trips to the database web server. In turn, this reduces network traffic as well as the load on the server, which allows the system to scale with the addition of multiple proxies.

With the increase of dynamic content on the web, researchers have started studying the general problem of caching dynamic content from various aspects. Challenger et al. [5] have focused on how to efficiently identify and update obsolete pages in the web server cache. Florescu et al. [11] have proposed a customizable cache system at data-intensive web sites. Our approach complements these server-side techniques because it addresses the problem in the context of web proxies and aims at sharing database web server workload and reducing network traffic.

Caching dynamic content at proxies has also been studied in [14] by Smith et al. Their approach allows web content providers to specify result equivalence in generated documents so that the proxy can utilize the equivalence to return a cached result to a new request. However, they do not consider database query containment or evaluate subsumed query at the proxy.

Unlike other proxy caching schemes, which only cache non-executable web objects, the Active Cache scheme [4] allows servers to provide a piece of Java code (called a *cache applet*) associated with a document. An active proxy (a proxy that supports the Active Cache scheme) can cache these cache applets from web servers, along with associated documents. For efficiency or security concerns, the active proxy has the freedom of not invoking a cache applet but directly forwarding a request to the server. However, if the proxy regards a request as a hit in its cache, it will invoke the corresponding cache applet to do some processing rather than just sending back the cached document to the user. Compared with the declarative nature and limited scope of [14], the Active Cache scheme provides a simple and flexible interface to web servers at the price of a possible overhead associated with the mobile code.

Cache applets get their name because of their similarity to Java applets, which are lightweight, originate from a server, and can communicate with the server. Through the cache applets, web servers gain more control over how proxies should behave upon the user requests addressed to the servers. For example, servers can ask the proxy to send them log information about user accesses, which are considered precious to servers. Also servers can execute other important policies such as consistency maintenance through cache applets while the proxy keeps its hands off. The other benefit to servers, in which we are more interested, is that servers can migrate some tasks to active proxies when needed and these tasks may even involve generating dynamic content at proxies.

Our query applet is an extension to the generic cache applet. A straightforward function of the query applet would be to cache the query results at the proxy and return the results to users when they ask queries that are identical to a cached query. We call this *passive query caching*. To further reduce the workload on the server, we have added two more functions to the query applet – query containment checking and simple selection query evaluation. Having these two functions, the query applet can perform active query caching, where the proxy not only answers queries that are identical to a cached query, but also answers queries that are more restrictive than a cached query.

Finally, our active query caching can be viewed as a special case of answering queries using views ([12]) if we consider cached queries as materialized views. However, these views come and go dynamically because of the nature of caching. Moreover, as a first step of query caching at web proxies, we only consider answering a query using one view instead of multiple views and thus reduce the problem to simple query containment checking.

The rest of the paper is organized as follows: Section 2 presents the system overview, Section 3 describes active query caching in more detail, Section 4 shows performance results, and finally discussion and conclusions are given in Section 5.

2. SYSTEM OVERVIEW

We have developed a prototype system of a database web server and an active proxy with active query caching capability. The goal is to answer as many queries as possible at the proxy while keeping the overhead of the query applet low. The system architecture along with the handling process is shown in Figure 1. The shaded parts represent the components implemented by us.

In this system, we used a modified version of the active proxy [4], which was originally developed on the CERN httpd code base [17]. The modifications included allowing CGI requests with query strings in GET or POST methods to be cached, and loosening certain security inspections and resource limits on cache applets. We also used a CERN httpd as the web server. The database server was the IBM DB2 Universal Database Server V5.2 with a JDBC driver.

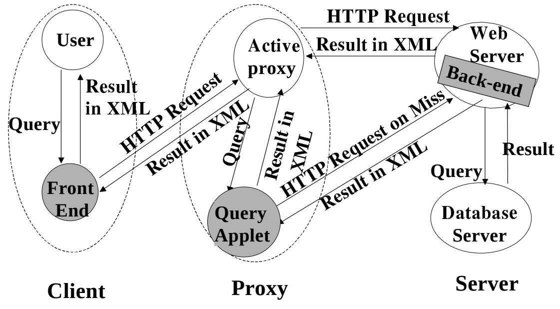

Figure 1. System architecture

As illustrated in Figure 1, the three components we have implemented are the query front-end, query applet, and query back-end. They reside on the client side, the proxy (after the proxy gets the applet from the server), and the web server correspondingly.

When a user submits a query to the query front-end, the front-end program will convert the user query into an HTTP request and send it to the proxy. The proxy then examines the URL to see if it is a cache hit at the proxy. If it is a cache hit and the server URL has a corresponding query applet in the proxy, the proxy will invoke the query applet. Otherwise, the proxy will forward the request to the web server.

On the web server, the query back-end program transforms an HTTP request into a SQL query and sends it through JDBC to the back-end database. It then retrieves result tuples from the database server, wraps them into an XML file with an XML representation of relational data proposed by Bos [2], and sends the XML file to the proxy. If the server decides to send a query applet to the proxy, the query back-end program will send a query applet header along with the query result.

If a query applet header is sent to the proxy along with the document, the proxy will obtain the applet from the server and associate it with the server form URL. The next time the proxy receives a request to the server form URL with a query, it will invoke the corresponding query applet.

The query applet maintains a mapping between the cached queries and their corresponding results. When the query applet is invoked upon a request, it extracts the query from the request parameters and examines its own cache. If a new query is the same as a cached query, the cached result will be returned; if the new query is more restrictive than a cached query, it is then evaluated on the result of the cached query, and new query results are generated and sent back to the user. Otherwise, the query applet forwards the request to the web server and caches the query and result from the server before passing the result to the client.

Note that in practice one HTTP request may be transformed into several SQL queries or involve more complex operations at the server side. However, the proxy does not need to know about it because all it sees is single table views expressed by forms. Also in our implementation we only dealt with XML files not HTML files. This scenario is possible in automatic business data exchange applications. If HTML pages are needed, servers can modify the query applet code to generate the HTML.

In our implementation each query applet corresponds to a form URL at a web server and so it answers all the queries submitted to that form. When multiple query applets are present at the proxy, each of them manages its own query cache.

3. ACTIVE QUERY CACHING

3.1 Query Caching Scheme

We chose caching at the query level rather than at the table level or semantic region level for a number of reasons. The most prominent reason is its low overhead, which is crucial to the proxy. The queries that database web servers allow users to submit are usually form-based. At the proxy, these form-based queries are treated as selection queries with simple predicates over a single table view whose columns are the form attributes. This simplifies query containment checking and query evaluation at the proxy because the actual queries on the back-end database schema at the server may be much more complex. For instance, queries submitted through a search form of an online bookstore may be viewed as selections with some title, author, and price range predicates on a virtual table “books”.

Moreover, the query level granularity fits well in the Web context. Firstly, each query corresponds to an individual user request so that later refinement queries from a user can be answered easily based on earlier queries. Secondly, if there are some hot queries during a period of time, many queries can be answered from the results of these hot queries. Finally, caching individual queries is convenient for possibly maintaining user specific information and generating tailored results for individual users.

In contrast, table level caching does not seem to apply naturally for proxy caching. It requires the proxy to get the base data, store all of it, translate simple form queries into complex queries on base data, and evaluate them at the proxy. This is undesirable in terms of resource consumption, efficiency, and proxy autonomy. To take advantage of the dynamic nature of caching, we choose query level caching, which seems more feasible and efficient than table level caching at this point.

Semantic region caching ([7],[8]) has a finer granularity than query level caching and has the nice feature of non-redundancy. However, this advantage does not come for free. The expense of checking overlap among regions, coalescing regions, splitting queries among regions, and merging regions into the final query result is a lot more expensive than simple query containment checking and selection query evaluation. The small size of web query results causes region fragmentation and constantly triggers coalesce. Finally, it is complex to determine how "current" a coalesced region is in cache replacement.

3.2 Query Containment Checking

Query containment testing for general conjunctive queries is NP-complete [6]. However, there are polynomial time algorithms for special cases [15]. For simple selection queries, which are a very special case, we identify a sufficient condition to recognize subsumed queries efficiently. The worst-case time complexity of our query containment recognition algorithm is polynomial in terms of the number of simple predicates in the Conjunctive Normal Form (CNF) query condition. The following illustrates our checking criteria.

Query1	Query2
SELECT List1	SELECT List2
From Table1	From Table2
WHERE WhereCondition1	WHERE WhereCondition2

Given the above two queries, Query1 and Query2, whose where-conditions have been transformed into the CNF, we recognize that Query1 is subsumed by Query2 (we call Query2 a *super-query* of Query1) if all of the following conditions are satisfied:

- Table1 and Table2 are the same table (view).
- Fields in List1 are a subset of the fields in List2.
- WhereCondition1 is more restrictive than WhereCondition2.
- If WhereCondition1 and WhereCondition2 are not equivalent, all fields that appear in WhereCondition1 also appear in List2.

In general the last condition is not a necessary condition. We specify it because eventually we need to evaluate the subsumed query on the query result of the super-query. Thus, we must guarantee that the result of the super-query contains all fields that are evaluated in the where-condition of the subsumed query. So we use the current sufficient condition for simplicity and efficiency.

At this point, our query containment checking reduces to the problem of recognizing if one CNF condition is more restrictive than another CNF condition. The following two propositions further reduce the problem to testing if a simple predicate is more restrictive than the other simple predicate.

Proposition 1

Given $WhereCondition1 = P_1 \text{ AND } P_2 \text{ AND } \ldots P_m,$

$WhereCondition2 = Q_1 \text{ AND } Q_2 \text{ AND } \ldots Q_n,$

WhereCondition1 is more restrictive than WhereCondition2 if

$\forall i, 1 \le i \le n,\ \exists k, 1 \le k \le m,\ P_k$ ***is more restrictive than*** Q_i

Proposition 2

Given $P_k = R_1 \text{ OR } R_2 \text{ OR } \ldots R_x,$

$Q_i = S_1 \text{ OR } S_2 \text{ OR } \ldots S_y,$

P_k ***is more restrictive than*** Q_i ***if***

$\forall v, 1 \le v \le x,\ \exists u, 1 \le u \le y,\ R_v$ ***is more restrictive than*** S_u

Finally, given two simple predicates F1 op1 c1, F2 op2 c2, it is straightforward to test whether the former is more restrictive than the latter. Intuitively, F1 and F2 should be the same field, and the relationship among the two operators op1, op2, and the two constants c1, c2, should make the first predicate more restrictive than the second one. For example, “price <= 10” is more restrictive than “price < 20”.

3.3 Query Cache Management

Since our cached query definitions use the CNF format, we transform user queries into the CNF and store the AND-OR tree format at the proxy. The query cache consists of these query trees and their corresponding query results. A mapping table (called *query directory*) is used to record the correspondence between queries and their results. Note that query definitions and their actual results are stored separately because query containment checking can be done by only comparing query trees and do not need the actual query results.

There is a choice whether we should cache the query result of a subsumed query. One argument for caching it is that we may answer new queries faster on it because its result size is smaller relative to its super-query's. The problem is the large redundancy between this query and its already cached super-query. Since web queries tend to return a small size of records per request, we chose not to cache any subsumed queries of a cached query. As a result, cache hit ratio is improved because of less data redundancy in the cache.

There are three cache replacement schemes available in our implementation: LFU (Least Frequently Used), LRU (Least Recently Used), and benefit-based. The first two are straightforward. The third one is a combination of the other two in that it uses reference frequency and recency as parameters of the benefit. We define the benefit of a cached query as a weighted sum of the reference frequency and the recency. The heuristic behind the benefit metric is intuitive. If a query was used as a super-query for many new queries, it is likely that it will serve later queries also. This is a reflection of spatial locality – that query covers a hot region of data. If a query was used as a super-query recently, we believe that it will probably be used as a super-query for subsequent queries soon if users are doing query refinement. This can be thought as temporal locality.

4. EXPERIMENTS

4.1 On Excite Query Trace

Many web caching studies have used real traces ([3], [9], [16]) or synthetic web workloads ([1]). However, these real traces or generated workloads usually do not include CGI script requests or queries. What we really needed was a trace that recorded user queries to a specific database web server. Fortunately we obtained a real query trace of around 900K queries[1] over one day from a popular search engine, Excite [10].

Search engines have their special features that may differ from other database web servers. The main differences include: their search forms conceptually have only one column (keywords), their query results are URLs, and these results are sent page by page upon user requests. Despite these differences, we feel that it is useful to investigate the effect of active query caching on search engine queries, because these queries represent web user query patterns to a popular class of web information sources.

A recent study [13] by Markatos has shown that 20-30% of the 900K queries in the Excite query trace can be answered from cache if the query results are cached. This caching is equivalent to what we called passive query caching. We set out our experiments to examine how much more opportunity exists for active query caching.

We transformed the search engine trace into a SQL query stream on two columns – keywords and page number and ran it through our query caching module. All started from a cold query cache. The cache replacement policy was LRU.

We compared hit ratios of active query caching and passive query caching at various cache sizes. The legend "20KQ passive" in the following figure means passive query caching using a cache of 20K queries. Other legends have likewise meanings. If we use the assumption of 4KB query result page in [13], the query cache sizes of 20KQ, 50KQ, and 100KQ can be roughly translated into 80MB, 200MB, and 400MB correspondingly. Cache sizes used in [13] vary between 500MB to 2GB.

From Figure 2 we can clearly see that there is much more opportunity for active query caching than passive query caching. The whole trace of 900K queries has 29% non-unique queries, which is the upper limit of the cache hit ratio of passive query caching with a sufficiently large cache. In contrast, we can achieve 45%-65% hit ratios in active query caching with a small to medium size cache. Notice that active query caching with a query cache size of 20K queries outperforms passive query caching with a cache size of 100K queries by a factor of three.

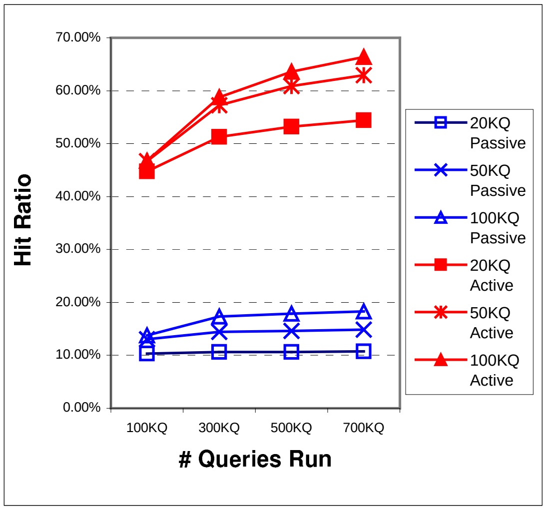

Figure 2. Query cache hit ratios on Excite query trace

4.2 On Synthetic Data and Queries

No matter how high the cache hit ratios are, they are just part of the story. To identify the performance implications of our prototype system, we generated synthetic relational data and query streams for our experiments and measured actual query response times.

In the following experiments we ran the CERN httpd server and the active proxy on two Sun Ultra10 300Mhz machines with the SunOS 5.6 Operating System. The DB2 server was running on a Pentium Pro 200MHz PC with Windows NT Operating System. All machines are in our department local area network and query caches start cold.

---

[1] When we say a query, we mean users' request of a specific page of the query results of a specific keywords sequence.

First, we measured the response times of a query stream when the workload of the database server was varied. We used a stream of 100 queries to measure the response time when the database server was idle, when it had 6 other clients, and when it had 12 other clients. The measurements were made with R20, R40 and R60 query streams (RX reads that X% queries are subsumed queries). In this experiment, a cache size of 10 queries was sufficient to achieve the performance gain, since we generated subsumed queries immediately after their super- queries. In practice the cache size should be sufficiently large to ensure that the super-queries are in the cache when their subsumed queries arrive.

The response time variation in Figure 3 shows the impact of subsumed query distribution on response times with active query caching. Unlike the case without caching, the query response times with active query caching decrease when the percentage of subsumed queries increases. For the R40 and R60 query streams, the response time for the proxy with the cache is better than the case without the cache, which means these hit ratios offset the query applet overhead at the proxy. Although for the R20 query stream the response time with caching is slightly more than the response time without a cache, the proxy with caching can still share 20% of the workload with the server.

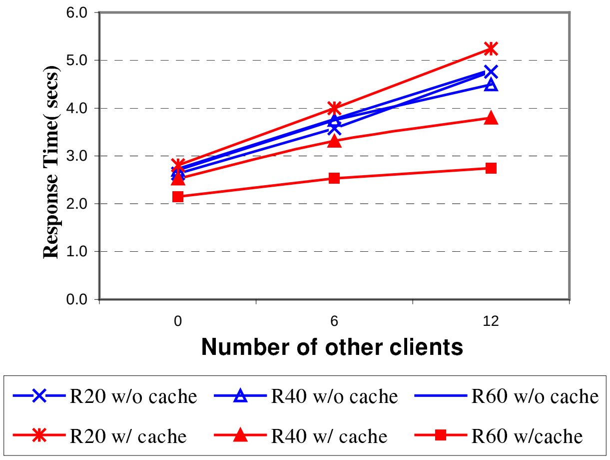

Figure 3. Response time as server workload varies

We then measured the breakdown of the time spent by a query at the various stages in a query applet. We considered the three cases – the new query could be identical to a query in the cache, be a subsumed query to a cached query, or need to be evaluated at the server. "Load + Save" refers to the time that the query applet takes to load the query directory from disk when it is invoked by the proxy plus the time taken for saving the query directory to disk before it finishes. "Check + Evaluate" includes the time that the query applet spends checking the query cache to see if the new query is subsumed by any cached query, and the time that the query applet spends evaluating the query from the cache. Finally "Fetch from server" is the time spent sending the query to the server and waiting for the result back, if the query cannot be answered from the cache. The results are shown in the following figure.

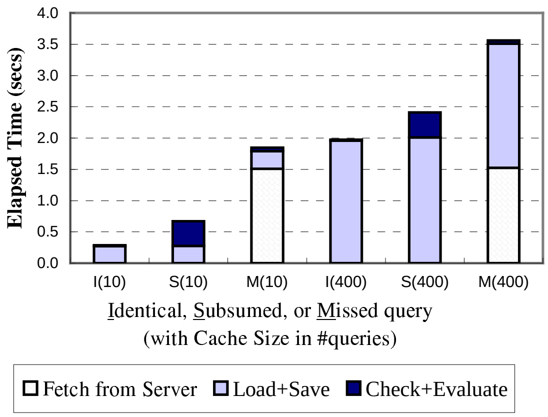

Figure 4. Breakdown of the time spent in the query applet

The breakdown of the time spent at the various stages in the query applet shows that even in an intra-departmental network, the time taken to contact the server and get the result back is a major portion of the total time. From other experiments (not shown here) we observed that roughly 40% of the "Fetch from Server" time was spent at the database server and the remaining 60% was spent on the network. The time taken to evaluate the subsumed query from the result of the cached query is considerable. This time was also seen to be proportional to the size of the result file of the cached query as the file I/O of reading this file and writing the result back was seen to dominate this time. Within "Check + Evaluate" the portion of the time taken to check and find a super-query is quite small. Finally, we see that the time taken to load and save the cache directory is considerable. This time increases almost linearly with increase in cache size, and becomes comparable to the time taken to contact the server when the cache size is 400 queries. This cost can be avoided if the query directory could be kept in memory in the active proxy. However, this is not feasible in the current implementation of the active proxy.

5. DISCUSSION AND CONCLUSIONS

In this paper, we have studied active query caching for database web servers. We have shown the opportunities that active query caching brings through a trace-driven simulation on real query traces and a prototype implementation. We have also identified the performance bottlenecks in the current implementation of the active proxy framework.

The active proxy made it possible for us to study active query caching at proxies for database web servers. Nevertheless, since the active proxy is in its prototype stage and active query caching is a brand-new application of the active proxy, we learned a few lessons from our experience. Due to space constraints, we briefly discuss one major issue – the active proxy does not provide any memory-resident structure for cache applets. This is not a limitation of the Active Cache protocol but is related to the CERN proxy design and implementation. Two factors are involved. One is that the CERN proxy does not have memory-resident cache. The other is that CERN proxy forks one process per request and so the cache applet's memory structure cannot be directly passed back to the proxy. This limitation had a strong negative effect on our implementation's performance. Finally we note that Java in its current stage does have performance complications in spite of its attractive features of portability, security, and ease of implementation.

As the first step of active proxy query caching for database web servers, this prototype is a simple functional system rather than a mature one. There are many ways that our work can be extended. We are investigating ways of sharing memory structure between the proxy and the query applet to address the bottleneck. We plan to utilize indices on the query directory or other techniques to further reduce the time of query containment checking and query evaluation. We are also investigating other query caching schemes and cache replacement policies in this framework.

6. ACKNOWLEDGMENTS

We would like to thank Jin Zhang for his technical support and helpful discussions on the active proxy prototype system, and Evangelos Markatos for his discussion on Excite search engine query trace. We would also like to thank Kevin Beyer, Jim Gast, and Chun Zhang for their comments.

Funding for this work was provided by DARPA through NAVY/SPAWAR contract No. N66001-99-1-8908 and NSF through NSF Award CDA-9623632.

7. REFERENCES

[1] P. Barford and M. E. Crovella. Generating Representative Web Workloads for Network and Server Performance Evaluation. *Proc. Performance '98/ACM SIGMETRICS '98.*

[2] Bert Bos. XML representation of a relational database. *http://www.w3.org/XML/RDB.html*

[3] Ramon Caceres, Fred Douglis, Anja Feldmann, Gideon Glass, and Michael Rabinovich. Web Proxy Caching: The Devil Is in the Details. In *Workshop on Internet Server Performance*, 1998.

[4] Pei Cao, Jin Zhang, and Kevin Beach. Active Cache: Caching Dynamic Contents on the Web. *Proc. IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98).*

[5] Jim Challenger, Arun Iyengar, and Paul Dantzig. A Scalable System for Consistently Caching Dynamic Web Data. *Proc. IEEE INFOCOM 99.*

[6] Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Conference Record of the Ninth Annual ACM Symposium on Theory of Computing*, May 1977, pages 77-90.

[7] Boris Chidlovskii, Claudia Roncancio, and Marie-Luise Schneider. Semantic Cache Mechanism for Heterogeneous Web Querying. Proc. *8th World-Wide Web Conference (WWW8)*, 1999.

[8] Shaul Dar, Michael J. Franklin, Björn Þór Jónsson, Divesh Srivastava, Michael Tan. Semantic Data Caching and Replacement. *VLDB* 1996.

[9] Fred Douglis, Anja Feldmann, Balachander Krishnamurthy, and Jeffrey Mogul. Rate of Change and other Metrics: a Live Study of the World Wide Web. In *Symposium on Internet Technology and Systems.* USENIX Association, December 1997.

[10] Excite Search Engine. *http://www.excite.com/*

[11] Daniela Florescu, Khaled Yagoub, Patrick Valduriez, Valerie Issarny. Caching Strategies for Data-Intensive Web Sites. *INRIA Technical Report*, INRIA, December 1999.

[12] Alon Y. Levy, Alberto O. Mendelzon, Yehoshua Sagiv, Divesh Srivastava. Answering Queries Using Views. *PODS*, 1995 95-104

[13] Evangelos P. Markatos. On Caching Search Engine Results. *Technical Report 241, ICS-FORTH*, January 1999.

[14] Ben Smith, Anurag Acharya, Tao Yang, Huican Zhu. Caching Equivalent and Partial Results for Dynamic Web Content. *Proc. of 1999 USENIX Symposium on Internet Technologies and Systems.*

[15] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems*, Volume II. Computer Science Press 1989, pp. 877 - 907.

[16] Craig E. Wills, and Mikhail Mikhailov. Examining the Cacheability of User-Requested Web Resources. In *Proc. of the 4th International Web Caching Workshop*, 1999.

[17] W3C. http://www.w3.org/Daemon/