Using Semantic Information to Improve Transparent Query Caching for Dynamic Content Web Sites

Gokul Soundararajan and Cristiana Amza
Department of Electrical and Computer Engineering
University of Toronto
{gokul, amza}@eecg.utoronto.ca

Abstract

In this paper, we study the use of semantic information to improve performance of transparent query caching for dynamic content web sites. We observe that in dynamic content web applications, the most recently inserted items are also the ones that register the highest activity. For example, the newest books in a bookstore are also the ones more frequently browsed and bought. Hence, assuming repeatable queries, a particular read-only query response is likely to incrementally change as new rows are added to the query's tables. We avoid the cached query response invalidations that would otherwise occur due to the addition of new items by keeping the newly inserted rows in small temporary tables. This allows us to reuse cached responses for partial coverage of query results. A query result is then obtained from merging an existing cached response with one or more lightweight residual query results that involve the temporary tables. In addition, we enhance our cache with other partial coverage techniques based on per-query semantic information such as sub-range queries for all queries that match a specific template.

We implement semantic query caching on top of an existing template-based cache with column-based invalidations. Our evaluation is based on a dynamic content site using the Apache web server with Tomcat Java servlets and the MySQL relational database. We use the industry-standard TPC-W e-commerce benchmark as our benchmark application. We conclude that augmenting transparent query caching with the ability to retrieve partial results from the cache improves performance substantially in terms of latency and to a lesser extent in terms of hit-rate and throughput.

1 Introduction

In this paper, we study transparent query caching techniques providing both performance enhancements and strict consistency guarantees to the user. Most previous work in the area of dynamic content caching has concentrated on specialized caching solutions [15, 23, 10, 5, 8]. These existing solutions require some type of programmer or site administrator intervention to explicitly specify the cacheable fragments of web pages and their lifetime. Other query caching solutions assume relaxed consistency semantics [2]. The use of these previous caching solutions implies either additional design and programming effort to achieve the right consistency semantics for each application, or force the user to handle inconsistent results. At the other end of the spectrum are basic transparent caching techniques which provide strong consistency, but use coarse-grained table-based or column-based invalidations [17, 9].

In this paper, we design and implement optimizations based on semantic information for a fully transparent dynamic content cache suitable for any web application, including applications with strong transactional requirements such as e-commerce workloads. Transparency requires that cached entries be invalidated automatically as a result of writes. Specifically, a cached query response needs to be invalidated when an underlying data item in the database changes. Our focus is on reducing the impact of automatic invalidations by either making them unnecessary or reducing the penalty of a cache miss through partial coverage of query results.

Our first transparent cache optimization is based on addressing the high rate of invalidations caused by the addition of new items to a database table in a cache with coarse grain (table or column-based) automatic invalidations. Semantic information here means that the optimization is driven by knowledge of common application patterns such as the high frequency of browse-type access to newly inserted items (e.g., browsing the *new* books).

We maximize the probability that a read-only query can be largely satisfied from the cache, by keeping track of newly inserted items in separate small temporary tables. A query result is then obtained from merging an existing cached response with one or more lightweight residual query results that may need to be computed on the temporary tables. By keeping the temporary tables small, the overall perceived response time for a query is low.

A second dimension to our semantic cache scheme is using per-query information to determine full and partial coverage for query results. Full or partial coverage occurs when a query response is fully or partially contained within one or more cached responses. Semantic information in this case is using per-query knowledge to check for containment of the current query response within an already cached response. For this purpose we parse each query to infer query result ranges and possible coverage or partial coverage for all queries that match a specific template.

The combination of partitioning the new and old items through the use of temporary tables driven by application patterns and using per-query semantic information work in synergy. In particular, partitioning increases opportunities for partial coverage for each individual query as explained above. On the down side, our semantic cache optimizations come at the expense of additional processing for containment checking, filtering responses for useless rows in cases of partial coverage and even reimplementing a limited fraction of the database functionality (such as re-ordering, or re-counting rows during merging results for our partitioning scheme).

We implement semantic query caching on top of an existing template-based cache with column-based invalidations as available with the open-source C-JDBC database cluster middleware [9]. The C-JDBC query cache accesses the database through a JDBC driver, and does not rely on any special database features (such as availability of triggers or any other database functionality).

Our evaluation uses a dynamic content site with the Apache web server [1], Tomcat Java servlets as the application server and the MySQL [17] relational database. We use the industry-standard TPC-W [22] e-commerce benchmark's browsing and shopping workload mixes, as our benchmark application. We compare our semantic cache and the original C-JDBC basic cache version. We conclude that semantic caching improves performance significantly in terms of response time, especially for workloads with a higher fraction of writes (e.g., the TPC-W shopping mix). For both of the TPC-W browsing and shopping mixes, column-based invalidation enhanced with partial coverage detection is more effective than column-based invalidation alone. Hence, for both workloads, the benefits of higher hit ratios outweigh the costs of additional processing in our semantic cache. The remainder of this paper is structured as follows. Section 2 provides the necessary background on transparent caching with coarse-grained automatic invalidations. Section 3 introduces our semantic cache design. Section 4 describes our benchmark and experimental platform. We experimentally investigate how our semantic cache improves performance in Section 5. Section 6 discusses related work. Section 7 concludes the paper and presents avenues for future work.

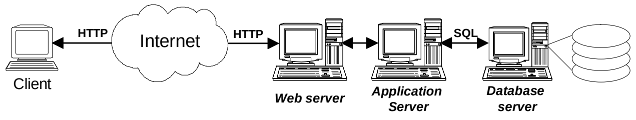

Figure 1. Common architecture for dynamic content sites

2 Background

Dynamic content sites commonly use a three-tier architecture, consisting of a front-end web server, an application server implementing the business logic of the site, and a back-end database (see Figure 1). The (dynamic) content of the site is stored in the database. A number of application programs provide access to that content. A client request for dynamic content causes the web server to invoke a method in the application server. The application server executes the application program, issues SQL queries, one at a time, to the database and formats the results as an HTML page. The web server then returns this page in an HTTP response to the client.

To speed up the delivery of dynamic web pages, database *query caching* has been proposed [4, 11, 14, 19, 24]. With query caching, the results of recent queries are cached locally and are reused on later queries. By caching, both the latency of retrieving the results and the load on the database back-end is reduced.

Our cache optimizations are implemented on top of an open-source caching middleware called C-JDBC which in turn accesses the database through a JDBC driver. JDBC (Java Database Connectivity) allows two methods of declaring queries: (1) using templates where queries are predeclared as prepared-statements and parameters are set through function calls and (2) non-templated versions where the query is a string concatenation of query body and arguments. The C-JDBC cache assumes the template version is in use for queries which suits the parameterized design of web pages well. Henceforth, we assume that query templates are accessible to the cache.

The cache functions as a transparent proxy between the application logic and the database. To the application logic, the cache appears as the database and to the database, the cache appears as the application server. The cache takes as its input the database queries generated by the application logic. On a read query, the cache checks whether the results of the query reside in the cache, and, if so, returns them immediately to the application. Otherwise, the cache forwards the query to the database. The database returns the results of the query to the cache, where they are inserted and forwarded to the application. On an update, insert or delete query, the cache performs the necessary invalidations and forwards the write query to the database. The cache may be located on the same machine as the front-end, on a separate machine, or on the same machine as the database.

The C-JDBC cache supports two transparent invalidation schemes (table and column based). In these schemes, for each cached query response, the query's dependencies are recorded in terms of database tables, or in terms of database columns. In the case of table-based (column-based) invalidation, each table (column) object contains references to the cache entries that are dependent on this table (column). When an update, insert or delete query is received, the cache invalidates all cache entries dependent on either the affected tables or the affected columns and forwards the query to the database. To keep the size of the cache manageable, the C-JDBC cache implements an LRU replacement strategy.

3 Semantic Cache Design

In this section we describe our techniques for using query results from the cache as partial results towards computing a query response. Section 3.1 presents an overview of our partitioning scheme for transforming misses caused by insert queries into partial hits. Section 3.2 presents our method for obtaining partial answers from the cache using both our optimization for inserts and more generally per-query semantic information.

3.1 Partitioning Scheme for Alleviating Insert Induced Invalidations

We keep newly inserted rows in separate tables called *temp tables*, one temp table per regular database table. The value of each field in the temp table rows (including key values) is the same as if the rows were present in the main table. Upon receiving a select query, the cache splits it into two queries: the original query and one or more residual queries. The original query is the unmodified query working on the regular table. A residual query is the same query on the corresponding temp_table. These separate results are obtained and cached as usual. The final query result is obtained by merging these query results. This optimization potentially benefits query results that would otherwise be invalidated by inserts into the accessed tables. In the example shown in Figure 3, a `SELECT` which fetches a large number of rows (A) would be invalidated by a subsequent `INSERT` query. By placing newly inserted rows in a temporary table, we avoid invalidating the result of the first `SELECT` and we can compute the result for a subsequent matching `SELECT` quickly by merging the cached response (A) with a small residual response (B) computed from the corresponding temporary table (Figure 4).

Although the residual queries on the temporary tables are lightweight, we choose to cache their results as well. Hence, these residual results could either be returned from the cache if valid or otherwise would need to be recomputed at the database similarly to the treatment of any other query.

Upon an `INSERT` query, the cache redirects the query to insert it in the temporary table. For `UPDATE` and `DELETE` queries the cache sends them to both the original and the temporary tables. In the case of queries containing joins of two or more tables, the query is split into the corresponding sub-queries necessary to compute the join according to the formula in Figure 2 (shown for 2 tables). Checks for attributes with null values and additional filtering of the result set are inserted to handle the case of outer joins.

$$\begin{aligned} A \bowtie B &= (A \cup tA) \bowtie (B \cup tB) \\ &= (A \bowtie B) \cup (A \bowtie tB) \\ &\quad \cup (tA \bowtie B) \cup (tA \bowtie tB) \end{aligned}$$

Figure 2. Join Formula for Partitioned Tables

Partitioning a select query could lead to $O(2^n)$ partitioned queries, where n is the number of tables referenced in the query. However, in practice, only certain tables are partitioned at any given time (usually 1 or 2). Their number depends on how many of a query's tables registered inserts in the recent past. Furthermore, the joins that involve the temporary tables are fast since these tables contain a few rows (a maximum of 100 in our scheme). When an insert table exceeds the threshold size, we remove and reintegrate all entries from the insert table into the corresponding regular table. Since the number of sub-queries is kept low and the temp tables are small, computing the residual queries in the case of joins is relatively fast compared to recomputing the original result.

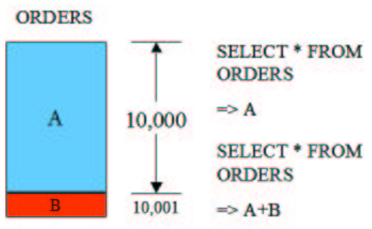

Figure 3. Incremental change for a query response (A) upon inserting a new row (B)

3.2 Detecting Coverage for Query Results

The cache can detect both *Full Coverage* and *Partial Coverage* of a query response. In *Full Coverage* (see Figure 5), the query response of the current query is fully contained within a cached response. In this case, (at least) one

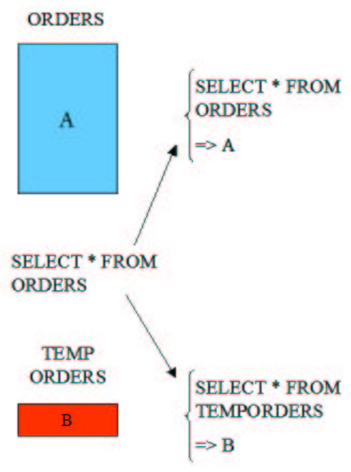

Figure 4. Using a separate temporary table for new inserts allows reuse of previous query response (A) for partial coverage

of the cached query responses is a superset of the current query response. In *Partial Coverage*, a query result is obtained from merging an existing cached response with one or more residual query results that may need to be computed at the database.

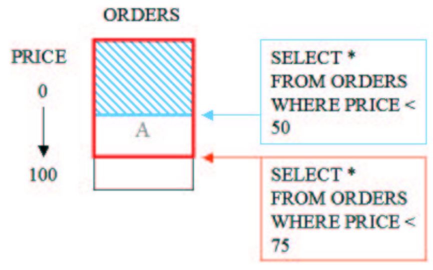
ORDERS
PRICE
0
100
A
SELECT * FROM ORDERS WHERE PRICE < 50
SELECT * FROM ORDERS WHERE PRICE < 75

Figure 5. Coverage of Query Results: The response for the "PRICE < 50" query is fully contained within a previous cached entry

Full coverage is classified as a cache hit, while partial coverage is currently classified as a cache miss. Checks for either full or partial coverage are done for all types of queries (including sub-queries of a partitioned query). The cache first checks whether there is an exact match with a previously cached query. If not, the algorithm proceeds to check if any previously cached query can provide a full answer (full coverage). If full coverage is detected, the cache filters out any rows that are not necessary to satisfy the current query from the cached entry and returns the resulting query response.

Otherwise, the cache algorithm proceeds to check if any previously cached query can provide a partial answer. If such a query exists, the cache sends a remainder query to the database. When the database returns the result for the remainder query, the cache merges the result with the partial result obtained from the cache. In the following section we provide more detail on how we determine whether a cached result satisfies the incoming query (i.e., how we implement full coverage and partial containment checks), how to generate a remainder query in the case of partial coverage and the merging algorithm.

3.2.1 Coverage Checking

We follow the general coverage testing described by Larson et al. [13] which defines that a query Q_1 *covers* Q_2 if the following three conditions hold:

1. *Attribute Coverage:* This states that the columns in the `SELECT` clause of Q_2 should be a subset of Q_1.

2. *Tuple Coverage:* This means that the tuples addressed by Q_2 should be a subset of the tuples addresses by Q_1. In other words, for the `WHERE` predicates, P_1 of Q_1 and P_2 of Q_2, $\forall(ttupleofQ_2)P_2 \rightarrow P_1$.

3. *Selectability:* This requires that the query must be entirely evaluated using the cache entries. That is, Q_2 should not refer to any columns in other `SQL` constructs (such as `ORDER BY`, and `WHERE`) that is referred by Q_1.

We currently consider only conjunctive `WHERE` clauses for containment (coverage) checking. Furthermore, we can detect coverage only conservatively for queries matching a given query template (e.g., differing only in the inequality expressions appearing in the `WHERE` clause).

In more detail, the containment checking algorithm takes as input the incoming query Q and a list of cached queries C matching Q's template. Each query contains a list of predicates P contained in the `WHERE` clause. For each cached query, the algorithm checks whether all the equality constraints are satisfied. Any cached query not satisfying an equality constraint is removed from further consideration for this match. Then, for each inequality predicate, the algorithm selects the cached query that *covers* the most tuples. In case of ties, the first query on the list is returned.

3.2.2 Generation of the Remainder Query

To complete the result obtained from the cache, a remainder query obtains the missing tuples from the database. Formally, the remainder query is described as $R = Q - C$ where Q is the incoming query and C is the cached partial result. The remainder query is generated in a straightforward way. We consider the inequality predicates, one at a time. For each predicate, if the cached query does not cover the incoming query, the predicate is rewritten to fetch the tuples not referred by the cached query.

For example, consider the following queries conforming to template T below, where the new query Q is presented to the cache, while the results of C1, C2 and C3 are already cached (only the `WHERE` clause is shown for these queries):

```
T: (SUBJECT = ?) AND (PRICE < ?)
Q: (SUBJECT = 'KIDS') AND (PRICE < 100)
C1: (SUBJECT = 'KIDS') AND (PRICE < 50)
C2: (SUBJECT = 'KIDS') AND (PRICE < 75)
C3: (SUBJECT = 'ARTS') AND (PRICE < 100)
```

The containment checking algorithm first checks whether the equality constraints are satisfied. After this stage, `C3` can be removed from future consideration. Then, we proceed to check if there are any overlapping regions between Q and any of the cached queries. In this example, it is easy to see that both `C1` and `C2` partially cover `Q`, but `C2` is a better candidate since it contains a larger percentage of the final result. Therefore, `C2` is selected as the best result from the cache. `Q` is rewritten to

```
Q2: (SUBJECT = 'KIDS') AND
        (PRICE >= 75 AND PRICE < 100)
```

The rewritten query is sent to the database and its results are merged with the cached results to form the final answer that is returned to the client.

3.2.3 Efficient Merging

Since results have to be merged for every partitioned query, special care needs to be taken for queries that contain special clauses such as `ORDER BY`, `COUNT`, `MAX` and other SQL functions. For sorting necessary in `ORDER BY` clauses, we use an algorithm that merges the results in $O(ndc)$ time where d is the number of partitioned tables in the query and c is the number of `ORDER BY` columns. Our algorithm is based on merge-sort [7] and uses the fact that the query result sets to be merged are pre-sorted by the database. The algorithm then generates the sorted final result by interweaving the different streams.

4 Experimental Evaluation

4.1 Hardware and Software Environment

We use the same hardware for all machines running the client emulator, the web servers, the cache, and the database. Each machine has dual AMD Athlon 2400 MP processors (running at 2 GHz), 512MB SDRAM, and a 120 GB disk drive. All machines are connected through a switched 100 Mbps Ethernet LAN.

All machines run RedHat Linux 9. We use Apache Tomcat 4.1 as our Web/Application server. We use MySQL v4.0 as our database server.

4.2 TPC-W Benchmark

We use an industry-standard e-commerce benchmark, TPC-W, from the Transaction Processing Council [22].

Several interactions are used to simulate the activity of a retail store. We implemented the 14 different interactions specified in the TPC-W benchmark specification. Of the 14 scripts, 6 are read-only, while 8 cause the database to be updated. The read-only interactions include access to the home page, listing of new products and best sellers, requests for product detail, and two interactions involving searches. Read-write interactions include user registration, updates to the shopping cart and two interactions involving purchases. With one exception, all interactions query the database server.

TPC-W simulates three different interaction mixes by varying the ratio of read-only to read-write scripts **browsing**, **shopping**, and **ordering**.

The complexity of the interactions varies widely, with interactions taking between 20 ms and 700 ms on an unloaded database in our experimental environment, and read-only interactions up to 30 times more heavyweight. than read-write interactions. The most complex read-only interactions are BestSellers, NewProducts and Search by Subject.

We use the standard TPC-W database containing 100,000 items and 2.8 million customers which gives a database size of about 5 GB.

4.3 Measurement Methodology

In the experiments, we consider a configuration where the client emulator, Apache and Tomcat are co-located on one machine. The query cache is running on a second machine and the MySQL database on a third machine.

The same methodology is used for measuring the performance of both cache versions (the C-JDBC cache with and without the semantic optimizations). In each experiment the first half of the run is used to warm-up the cache and is excluded from the measurements. Each experiment also starts with an identical database. Differences between repeated runs of the same experiment were minimal. To select the load for the experiments, we are driving the server without the cache with increasing the number of clients, until performance peaks. Then, we use the same number of clients to drive the server with the cache enabled, for both the basic C-JDBC cache and the semantically enhanced cache.

We measured the performance in terms of throughput and response time of our cache versus the C-JDBC cache for two TPC-W mixes: browsing and shopping. The browsing mix contains 95% read-only scripts, and the shopping mix 80%.

5 Results

In this section we study the impact of our semantic caching techniques on the performance of a query cache with column-based automatic invalidations.

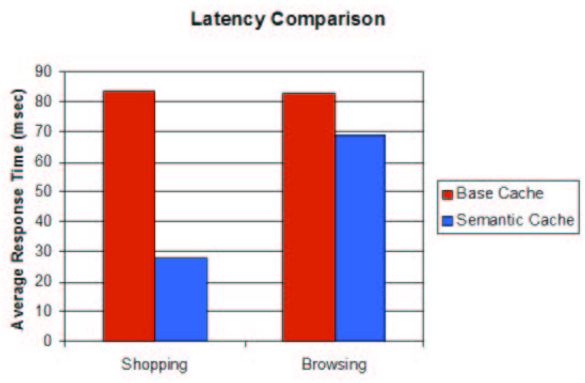

Figure 6. Latency Comparison

Figure 6 shows a latency comparison for the C-JDBC original cache and the semantic cache for the TPC-W browsing and shopping mixes, respectively. The latency represents the average response time for a page as perceived by the client. The results indicate that semantic caching lowers the latency significantly, by a factor of 2.9 for the shopping mix and by a factor of 1.2 for the browsing mix. The shopping mix contains a higher fraction of writes compared to the browsing mix (20% vs. 5%), hence a higher number of invalidations in C-JDBC. The latency gains are mostly due to faster computation of query responses from partial and residual results compared to re-executing the original query.

	Shopping Mix	Browsing Mix
Column	23%	37%
Semantic	27%	44%

Table 1. Hit Rates

As seen in Table 1, semantic caching also improves the hit-rate of the cache due to full coverage detection. For the browsing mix, the hit-rate is improved by 8% and for the shopping mix by 4%. We currently classify partial hits as misses.

On the other hand, the most latency improvement comes from partial coverage cases rather than additional hits due to full coverage. Partial hits, especially for queries that involve multiple table joins (e.g., the *BestSeller* query that retrieves the books that were ordered the most in recent purchases) make a difference in terms of latency improvements for the shopping mix. The performance improvements brought about by semantic caching translate in throughput increases as well, however to a lesser extent than for latency, with throughput increases only up to 10% for both mixes.

6 Related Work

6.1 Overview of dynamic data caching

Dynamic Web data can be cached at different stages in its production: the final HTML page (e.g., [3, 12], intermediate HTML or XML fragments (e.g., [8]), database queries (e.g., [16]), or database tables (e.g., [15, 18]). Combination of various caches are also possible (e.g., [5, 23]). Intuitively, caching at the database stage typically offers higher hit ratios, while caching at the HTML or XML stage offers greater benefits in the case of a hit. There is no conclusive evidence at this point that caching at any single stage dominates the others. For instance, Labrinidis and Roussoulos use a synthetic workload and conclude that HTML page caching is superior [12], but Yagoub et al. use TPC-D and conclude that database query caching is more effective [23]. It appears that the different caches are complimentary [20, 23]. This paper is concerned with database query caching. Our methods can be extended to record dependencies between HTML pages or fragments and database data items, and we intend to investigate this in further work.

6.2 Non-transparent approaches

Luo et al. [15] require the database designer to specify which tables are cached. Updates to the cache are performed once a minute. Oracle 9i also provides table-level caching in the middle-tier and invalidation based on time and events (database triggers), but no generalizable solution for generating invalidations [18]. Yagoub et al. [23] describe a declarative system for specifying a web site that allows a designer control over HTML, XML and query caches, including what to insert or to remove from the cache and how to invalidate or update items in the cache. Challenger et al. propose a cache API to control the contents of the API [10, 5]. Datta et al. propose annotating the application logic to inform the cache which HTML fragments are cacheable, similar to the WebLogic cache [8]. In contrast, our approach is transparent, can be applied without additional effort to an existing web site design, and automatically maintains consistency at all times. Nonetheless, we have been able to demonstrate substantial performance benefits.

6.3 Semantic Caching

The concept of semantic caching has been examined in the context of database design [13, 21]. This approach has been explored for LDAP (Lightweight Directory Access Protocol) with existing studies studies [6] focusing on how to reuse results from existing LDAP queries to answer future queries. More recently, Amiri et al. [2] has proposed using semantic information to generate results based on cached query results for dynamic content queries sharing the same query template. Their cache shares similarities with our per-query semantic information optimizations, but it lacks the ability to generate partial results. Furthermore, their intended deployment is in caches with loose consistency at the client edge of the network, while our focus is on providing strong consistency for a central server cache.

7 Conclusions

In this paper, we presented a method of using semantic information to retrieve partial results for queries from the cache. This method has several new features. First, the tables are partitioned to reduce misses due to `INSERT` queries. Second, information from previous queries is used to generate partial results. Finally, the results are efficiently merged for high performance. We have demonstrated that semantic caching improves the performance of query caching for dynamic content web servers. For the TPC-W benchmark, we have shown a factor of 2.9 latency improvement in the shopping mix and a factor of 1.2 latency improvement in the browsing mix.

References

[1] The Apache Software Foundation. http://www.apache.org/.

[2] K. Amiri, S. Park, R. Tewari, and S. Padmanabhan. Scalable template-based query containment checking in web semantic caches. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, Bangalore, India, 2003.

[3] K. S. Candan, W.-S. Li, Q. Luo, W.-P. Hsiung, and D. Agrawal. Enabling dynamic content caching for database-driven web sites. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, May 2001.

[4] J. Challenger, P. Dantzig, and A. Iyengar. A scalable system for consistently caching dynamic web data. In *Proceedings of the 18th Annual Joint Conference of the IEEE Computer and Communications Societies*, New York, New York, 1999.

[5] J. Challenger, A. Iyengar, and P. Dantzig. A scalable system for consistently caching dynamic web data. In *Proceedings of IEEE INFOCOM'99*, pages 294–303, Mar. 1999.

[6] S. Cluet, O. Kapitskaia, and D. Srivastava. Using LDAP directory caches. pages 273–284, 1999.

[7] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. Introduction to algorithms (second edition). McGraw-Hill and MIT Press.

[8] A. Datta, K. Dutta, H. M. Thomas, D. E. VanderMeer, K. Ramamritham, and D. Fishman. A Comparative Study of Alternative Middle Tier Caching Solutions to Support Dynamic Web Content Acceleration. In *Proceedings of the 27th International Conference on Very Large Databases*, pages 667–670, Sept. 2001.

[9] E. Cecchet et al. ObjectWeb Open Source MiddleWare: Clustered JDBC, 2003.

[10] A. Iyengar and J. Challenger. Improving web server performance by caching dynamic data. Dec. 1997.

[11] A. Labrinidis and N. Roussopoulos. WebView materialization. pages 367–378, 2000.

[12] A. Labrinidis and N. Roussopoulos. WebView Materialization. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 367–378, May 2000.

[13] P.-Å. Larson and H. Z. Yang. Computing queries from derived relations. In A. Pirotte and Y. Vassiliou, editors, *VLDB'85, Proceedings of 11th International Conference on Very Large Data Bases, August 21-23, 1985, Stockholm, Sweden*, pages 259–269. Morgan Kaufmann, 1985.

[14] Q. Luo, S. Krishnamurthy, C. Mohan, H. Pirahesh, H. Woo, B. G. Lindsay, and J. F. Naughton. Middle-tier database caching for e-business. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 600–611. ACM Press, 2002.

[15] Q. Luo, S. Krishnamurty, C. Mohan, H. Pirahesh, H. Woo, B. Lindsay, and J. Naughton. Middle-tier database caching for e-business. In *Proceedings of the 2002 ACM International Conference on Management of Data*, pages 600–611, June 2002.

[16] Q. Luo and J. F. Naughton. Form-based proxy caching for database-backed web sites. In *Proceedings of the 27th International Conference on Very Large Databases*, pages 667–670, Sept. 2001.

[17] MySQL. http://www.mysql.com.

[18] Oracle. Oracle9*i* Application Server Web Caching, Oct. 2000.

[19] K. Rajamany. *Multi-tier caching of dynamic content for database-driven web sites*. PhD thesis, Rice University, Houston, Texas, 2000.

[20] K. Rajamany. *Multi-tier caching of dynamic content for database-driven web sites*. PhD thesis, Rice University, Aug. 2000.

[21] D. J. Rosenkrantz and H. B. H. III. Processing conjunctive predicates and queries. In *Sixth International Conference on Very Large Data Bases, October 1-3, 1980, Montreal, Quebec, Canada, Proceedings*, pages 64–72. IEEE Computer Society, 1980.

[22] Transaction Processing Council. http://www.tpc.org/.

[23] K. Yagoub, D. Florescu, V. Issarny, and P. Valduriez. Caching strategies for data-intensive web sites. In *Proceedings of the 26th International Conference on Very Large Databases*, pages 188–199, Sept. 2000.

[24] K. Yagoub, D. Florescu, V. Issarny, and P. Valduriez. Caching strategies for data-intensive web sites. In *Proceedings of the 26th International Conference on Very Large Databases*, 2000.