An Architecture for Scaling Database-backed Web Applications

Sara Sprenkle
sprenkle@cs.duke.edu

May 2, 2002

Abstract

The bottleneck of Web content delivery is dynamic content, which requires data processing. We propose using a database cache to offload the demand on a database-driven Web service. This paper presents the design of a database cache and preliminary results on the performance of such a system.

1 Introduction

After static document caching improved the performance of the Internet, the new performance bottleneck is dynamic content, which does not receive the benefits of traditional content caching. Considering the growth of popularity of Web-based applications and that dynamic content generation requires orders of magnitude more resources than serving static pages, dynamic content will become a greater bottleneck for Internet resources and performance. Dynamic Web services respond to user requests on-demand by executing programs that return documents created from data stored on the server and elsewhere, the state of the server, and other information resources. The generated documents cannot be cached because they were created in response to a specific request; furthermore, the same request may return a different response because of changes to the data used to create the response, changes in the server's state, etc.

We propose a new caching infrastructure to handle dynamic content: a Web application proxy. At a Web application proxy, instead of caching responses, cache the data used to generate dynamic content and then execute the applications on the cached data.

We believe that caching the data used to generate dynamic content is a promising approach to reducing the client-perceived latency of dynamic content. The problem with response caches is that they do not handle dynamic content; a proxy stores the response to a specific request but does not know how the response was generated. A response cache is unaware of the underlying data that was used to generate the document and therefore does not know when a cached response is stale because of changes to the data.

By caching the data, we push the data closer to the users, decreasing the latency of responses, network utilization, and Web application server load. Furthermore, the cache is an alternate, if stale, data source when the primary database is unreachable from a client because of system or network failure.

For this project, we focus on content generated from applications executing over a database, which covers a large subset of dynamic content. Using a database cache introduces some interesting problems, such as how to load the cache, how to implement cache replacement, and how to maintain cache consistency. We will address these problems in Section 3. In Section 4, we present our Web application prototype for an e-commerce application, specified by TPC-W, and outline our design options and decisions. In Section 5, we describe our experimental set up for comparing our design decisions. The results are presented and discussed in Section 6.

2 Related Work

The problem of caching dynamic content has received a lot of attention in recent years, although

the proposed solutions have been unsatisfactory.

Semantic caching, proposed in [6], is one promising approach to data caching. In semantic caching, the contents of the cache are described by a constraint formula. Cache misses can then be resolved by requesting only the non-resident data, i.e., fetch the data that answers the query minus the data resident in the cache. Using remainder queries reduces communication and bandwidth requirements and thus response latency. While semantic caching has benefits over tuple or page caching, it has several limitations. Semantic caching requires disjoint constraint formulas that are confined to only one relation to simplify the replacement policies. This requirement limits the power and usefulness of semantic caching. In this proposal, we present some generalized semantic caching ideas to ease its deployment in a dynamic content caching infrastructure.

Cao's Active Caches [3, 9] is a Java-based approach to caching dynamic content. The Active Cache implementation utilizes Java's security features to perform computation on cached documents. A cache applet is attached to a document so that proxies, without server interaction, can perform the necessary processing on a cached document to make it current.

IBM's trigger monitor [4, 5] pre-generates the dynamic responses to user requests. The responses' data dependencies on underlying data are used to determine when a response should be regenerated, i.e., when data changes occur, the responses that depend on the changed data are regenerated. Pre-generating pages greatly decreases the workload on the server. Since disk space is now inexpensive, pre-generating large numbers of pages is an option. However, if data changes at a much higher rate than pages containing the data are requested or the data affects many pages that probably will not be requested, pre-generating the pages wastes resources.

3 Approach

We propose introducing a Web application proxy into the Internet infrastructure, as shown in Figure 1. The tradeoffs of using the Web application proxy is similar to the tradeoffs in using other caches—a cache hit will result in reduced response time while a cache miss will increase the response time as compared to the primary server handling all requests. A cache will also increase the complexity of the service while improving its scalability.

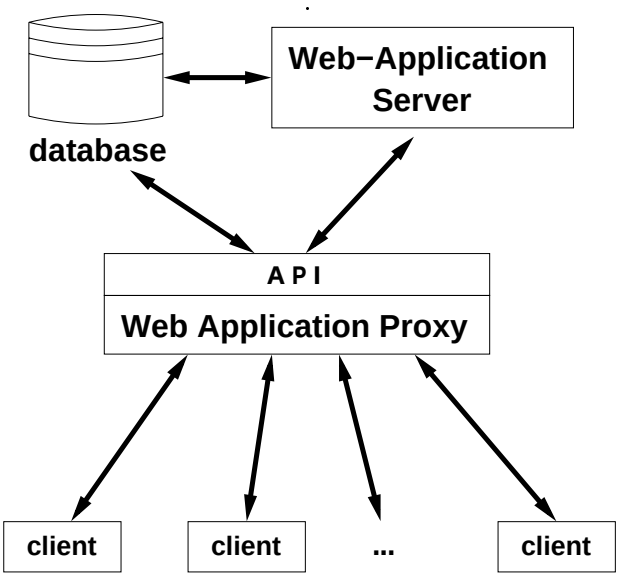

Figure 1: Internet architecture featuring a database-backed Web-application proxy. Clients make dynamic content requests, which are intercepted at a Web-application proxy. The proxy attempts to handle all requests. Requests that the proxy cannot handle are serviced by the primary Web-application server.

The Web application proxy offloads some of the workload of the primary Web server. As requests come into the proxy, the proxy attempts to dynamically generate the content based on the data stored in its local cache. The requests that it cannot answer are passed along to the primary Web application server. The response—in the form of a database result set, not the processed content—to the request can be cached at the proxy for use in future requests. We choose to fault in complete tuples—rather than processed results—based on the assumption that requests for a subset of a tuple's attributes will be followed by requests for other attributes, as was argued in [7].

The cache is a smaller replica of the original database that stores the same relations with the full

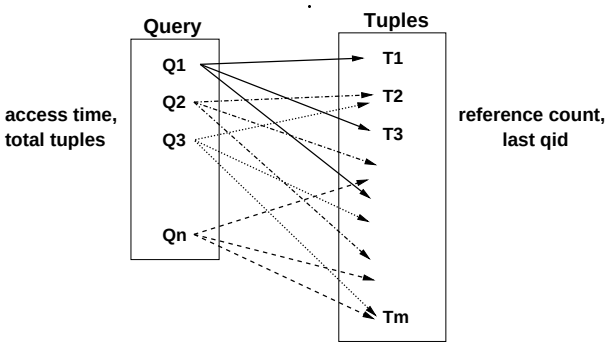

Figure 2: The cache maintains information about tuples' query membership. We augment these tables with information for implementing cache replacement policies.

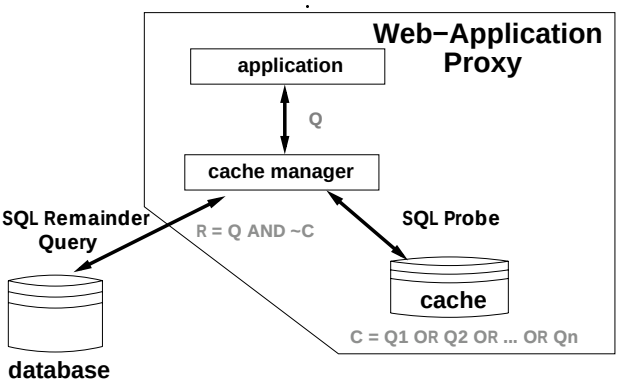

Figure 3: Design of a database-backed Web-application proxy. An application makes a query on the database cache through the cache manager. The cache manager handles the request by answering the query if possible and sending a remainder query to the primary database to fetch all non-resident tuples that satisfy the query. The contents of the cache are described by the union of a set of queries, C.

schema. A proxy backed by a database cache introduces challenges for cache management, which we will discuss in the next section.

3.1 Cache Management

We propose to use semantic caching-like techniques to manage the cache. The constraint formulas to describe the cache contents are queries; the union of all the queries describe the cache. (We assume that the queries are conjunctive queries.) In contrast to semantic caching, we do not require disjoint constraint formulas; however, we believe that our system is an approximation of a more general semantic cache. The cache maintains tables containing the query membership of tuples and total the number of tuples that are contained in each query, as shown in Figure 2; the cache may augment the tables with information for implementing cache replacement policies. The cache description table does not include simple search-by-key queries on a single relation to reduce space overhead and complexity. Since the cache can quickly search by key, storing the queries and its results is not space efficient.

The description of the cache—the union of all the queries—could become quite complex. The ideal policy for reducing the number of queries to describe the cache is to ignore queries whose tuple membership overlaps existing queries, i.e., the tuples it describes are not unique to that query. We believe such a policy is difficult to implement and propose a simpler policy: do not store information about a query whose tuple membership is smaller than some threshold t of the cache. Choosing an appropriate threshold for keeping a query string is an important subproblem to solve to optimize cache performance.

Our approximated semantic cache model requires reconsidering how to handle requests, how to implement cache replacement, and how to manage cache consistency. We will now address each of these issues.

3.1.1 Answering queries

Our cache can answer queries in a similar manner to the semantic cache. When a request comes in, the cache will attempt to answer the query from its cache. If the cache cannot answer the query, it sends a remainder query to the primary database to fetch the required tuples. For a query Q and cache description query C, the cache sends a remainder query, $R = Q \wedge \neg C$. The cache stores the tuples in its cache and updates the cache description table, if the number of tuples is above a given threshold.

The proposed design is shown in Figure 3.

There are several possibilities for optimizing this approach. The cache can execute the two queries—the probe query and the remainder query—in parallel; the cache can then calculate the set union of the two queries to process the final response.

One potential problem is that the cache description can become too complicated for the primary database to process efficiently. Instead of sending the full cache description, the cache could instead send a subset of the queries—preferably the most filtering queries. Choosing the query subset is another interesting subproblem to be investigated.

We believe that a simple pulling or faulting policy is sufficient for loading the cache; however, our cache model does not limit the cache loading policy. In the future, we may also want to consider push policies similar to [4] or prefetching policies. The pushed/prefetched tuples can be named by a query and added to the cache description table.

3.1.2 Cache Replacement

Since our model does not restrict the cache description to disjoint constraint formulas, cache replacement is not as straightforward as semantic caching. We can use our cache description table to approximate the semantic cache replacement algorithm. The cache description table maintains which tuples are contained by each query. When the cache chooses which tuples to evict, it evicts based on queries. The cache description table is updated to reflect the query eviction. A tuple is evicted when it is not contained in any query. The query eviction process continues until there is enough space in the cache.

We have not addressed how we choose a query to evict. Should we keep track of how frequently or how recently a query was requested? Should we evict queries that overlap with the most other queries? With the fewest other queries? Several replacement policies were discussed in [8], and one of these may also be appropriate for our application.

3.1.3 Cache Consistency

Semantic caching does not address cache consistency. We maintain the cache based on tuples so that we can incrementally update the cache. An update to a tuple may change the queries that contain the tuple. Tuple updates may result from remainder requests to the server or pushes from the server during its idle time. Updates to the commit should be transactional such that, for every updated tuple, we can update the cache description table.

In this stage of design, we are focusing on cache management and will consider the complications caused by updates at a future date.

4 Implementation

We implement our Web-application proxy prototype for a representative e-commerce application—an on-line bookstore based on the TPC-W benchmark [1]. The cache-describing structure and remainder query techniques we describe are not specific to TPC-W, but we will use TPC-W for examples.

4.1 Prototype for TPC-W

A freely-available, Java-based TPC-W release [2] was developed at the University of Wisconsin. The release includes Java data structures and servlets and testing software. We augmented the TPC-W release to use a database cache, providing us with a framework to investigate various cache loading and management techniques.

We built upon this framework to develop the Web application proxy. The primary component we added is the data cache, the interface between the application and the backend databases, which implements the techniques we described in Section 3. The data cache, written in Java, manages its database contents with a semantic cache manager.

The cache is responsible for separating single-relation, search-by-key queries from other queries. The cache can answer those queries without accessing the cache description table and fault in the tuple if it is not resident.

4.2 Semantic Cache Manager

The design for the semantic cache manager was described in Section 3.1. The cache manager is driven by a Java class that synchronizes access to the database cache description tables and provides an interface to the data cache. The cache manager uses three MySQL tables–a query table, a tuple table, and a query id lookup table. As described before, the query and tuple tables maintain query membership information. The query id lookup table maps SQL query strings to IDs. The query table is augmented with information about the last access to a query and the number of tuples covered by a query. The tuple table also keeps a total reference count and a pointer to the last query that contained the tuple.

When a query is added to the cache, the query and its associated tuples are inserted into the respective cache description tables. Since full, non-aggregate tuples are faulted into the cache, query projections, as well as group and sorting information, are ignored because the cache can answer any subsequent query on the same relation with the same condition, regardless of what columns are projected out or how the data is aggregated or sorted. For example, the SQL query

```
SELECT title, cost FROM item WHERE id <
50 ORDER BY cost DESC
```

is stored in the cache description tables as

```
SELECT * FROM item WHERE id < 50 .
```

4.2.1 Cache Replacement Policies

The information in the cache description tables is sufficient to implement four cache replacement policies:

- Random: choose a random query to evict
- Least Recently Used: choose the least-recently accessed query to evict
- Smallest Query First: choose the query containing the fewest tuples to evict
- Most unique tuples: choose the query with the most unique (fewest overlapping) tuples to evict. The intuition for this policy is that tuples accessed by many different queries will be accessed more frequently than tuples accessed by fewer queries.

The best replacement policy may depend on the application.

4.2.2 Remainder Query

The data cache accesses the cache description tables when it needs to create a remainder query to request query results from the primary database. In Section 3, we explained the ideas behind using the remainder query. In practice, several issues make generating a remainder query challenging. The goal is for remainder queries to describe the cache completely to reduce the communication and processing overhead of faulting tuples while not burdening the primary server with excessively complex queries.

Since we allow queries over one or many relations, the conditions we store in the cache description tables may reference one or many relations. Parsing join conditions to produce local selection conditions is not trivial. Another problem is caused by not keeping track of all queries that created the cache: the description of the cache is incomplete, missing a small subset of the tuples.

We consider two approaches to generating the remainder query. In general, the remainder query for a query with join condition j over relations A and B, $A \bowtie_j B$, and with cache contents A_C and B_C, should look like

$$(A \bowtie_j B) - (A_C \bowtie_j B_C)$$

In the first approach, the cache is described by queries whose relations are the same as the relations in the original query. In other words, for the query with join condition j over relations A and B, $A \bowtie_j B$, the cache contents are described by

$$C = A \bowtie_{j_1} B \cup A \bowtie_{j_2} B \cup \cdots \cup A \bowtie_{j_n} B,$$

where $j_1, j_2, ...j_n$ are join conditions over A and B, and the remainder query is

$$R = (A \bowtie_j B) - (A \bowtie_{j_1} B \cup A \bowtie_{j_2} B \cup \cdots \cup A \bowtie_{j_n} B)$$

For example, a remainder query for a query over the items and authors of mystery books may look like

```
R = SELECT * FROM item, author
   WHERE ((item.a_id = author.id) AND
(item.subject = 'MYSTERY'))
   EXCEPT (SELECT * FROM item, author
   WHERE ((item.a_id = author.id) AND
(author.l_name = 'KING')))
```

This approach reduces the number of queries used to describe the cache contents by ignoring queries that do not access all of the relations in the original query. However, this approach does not filter as many tuples as an approach that filters on each relation locally.

In the second approach, the cache is described as the union of local selection conditions over each relation.

In other words, for the query with join condition j over relations A and B, $A \bowtie_j B$, the cache contents are described by

$$C = \sigma_{s_1 \cup s_2 \cup \cdots \cup s_n} A \cup \sigma_{t_1 \cup t_2 \cup \cdots \cup t_m} B,$$

where $s_1, s_2, \ldots, s_n$ and $t_1, t_2, \ldots, s_m$ are selection conditions over A and B, respectively, and the remainder query is

$$R = (A \bowtie_j B) - (\sigma_{s_1 \cup s_2 \cup \cdots \cup s_n} A \bowtie_j \sigma_{t_1 \cup t_2 \cup \cdots \cup t_m} B)$$

We implemented and evaluated the first approach because it is the most straightforward to implement in SQL.

5 Evaluation Methodology

We evaluate our Web application proxy using performance metrics such as response latency, bandwidth, and cache hit rate. We have identified several important problems and subproblems for the Web-application proxy. For each approach to these problems, we would like to determine if we made the correct choices.

The cache/database and the Web server (Java's WebServer) are housed on separate Linux machines on the same LAN. Both machines are 1 GHz machines. The database machine has 2.5 GB of memory, and the web server machine has 256 MB of memory.

For our initial experiments, we used a relatively small database, scaled by the test parameters, summarized in Table 1. We tested our cache using the remote emulated browser provided with the Wisconsin TPC-W release. We chose the browsing mix to reduce the number of updates to the database. The browsing mix is dominated by search queries—such as searches by book title, author, or subject, searches for new products, and searches for related books. Operations like administrative updates and best seller searches—which requires accessing the large customer order tables—are directed to the primary database. Each run of the experiment lasted 10 minutes, beginning with a cold cache but a warm primary.

For every test run, the cache parameters were set as shown in Table 2. For a query to be included in the description of the cache, the query must cover at least TUPLE_THRESHOLD percent of the current cache size. As the cache is loaded, queries must describe larger numbers of tuples.

We monitored query and tuple cache hit rates as well as the number of tuples that were faulted into the cache. Cache misses are the tuples that are faulted in from the primary and not already resident on the cache. Cache hits are successful key field and non-key field searches. Before a cache requests the result from the primary database, it computes the result of the query in its cache. The tuples in this result are also considered cache hits. Faulted tuples, the bandwidth metric, include all tuples received by the cache from the primary, regardless of if the tuples were already resident.

For these experiments, the cache only monitors tuples from the "item" and "author" tables. Tuples from other relations may be stored in the cache, and these tuples are included in the tuple cache hit and miss rates, but the queries used to access these tuples fell below the TUPLE_THRESHOLD.

We set the CACHE_LIMIT to less than half of the total number of tuples for items (1000) and authors (250), combined.

Parameter	Comments	Value
NUM_ITEMS	Number of "books" in database	1000
NUM_RBE	Number of load generators	5
MIX	*Browsing* or *Shopping* or *Ordering*	*Browsing*
RAMP_TIME	Startup & shutdown allowance	10 sec
INTERVAL	Measurement interval	600 sec

Table 1: TPC-W parameters used in all experiments.

Parameter	Comments	Value
TUPLE_LIMIT	Max tuples in cache	600
TUPLE_THRESHOLD	Min percentage of cached tuples for query to be in description	2%
EXCEPT_THRESHOLD	Max queries in "EXCEPT" clause of remainder query	5

Table 2: Cache parameters used in all experiments.

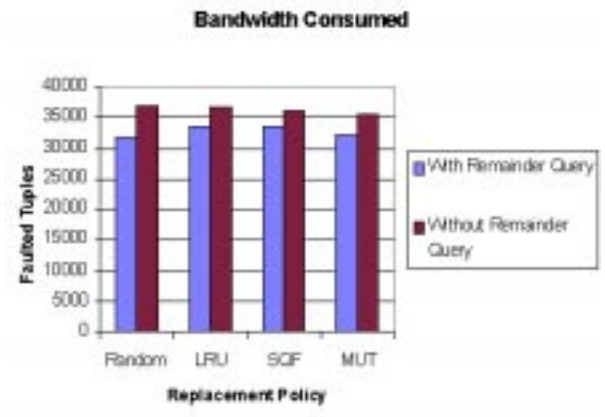

Figure 4: Comparison of remainder query and cache eviction policies in terms of bandwidth consumed, measured in tuples. On average, using remainder queries descreased the number of tuples faulted by 3000 in 10 minutes.

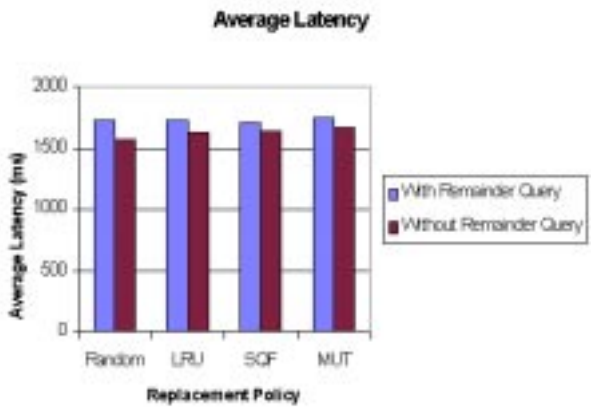

Figure 5: Comparison of remainder query and cache eviction policies in terms of latency, measured in milliseconds. Using remainder queries, latency increased slightly.

6 Results

The averaged results of several experiments are shown in Table 3. The results include the average web interactions per second (WIPS), latency of operations, cache hit ratios, and the bandwidth consumed by faulting.

The results show that using a remainder query significantly reduced the number of tuples faulted into the cache, as illustrated in Figure 4. The experiment ran for 10 minutes; in that time, approximately 3000 (on average 5/sec) fewer tuples were faulted into the database when the remainder query is used.

Latency (Figure 5) and throughput (Figure 6), on the other hand, did not improve with the use of the remainder query. While not intuitive, we attribute this result to several factors. The results of the remainder query may be scattered and more difficult for the database to optimize. Furthermore, the choice we made for generating remainder

Experiment	WIPS	Latency (ms)	Tuple Hit Rate	Query Hit Rate	Faulted Tuples
Random w rem query	3.481	1721	.9809	.0386	31700
Random w/o rem query	3.807	1575	.9828	.0333	36800
LRU w rem query	3.449	1732	.9814	.0350	33500
LRU w/o rem query	3.685	1622	.9827	.0344	36700
SFF w rem query	3.517	1697	.9809	.0302	33400
SFF w/o rem query	3.652	1640	.9823	.0314	36000
MUT w rem query	3.447	1742	.9808	.0366	32100
MUT w/o rem query	3.606	1657	.9824	.0346	35600

Table 3: Experimental Results

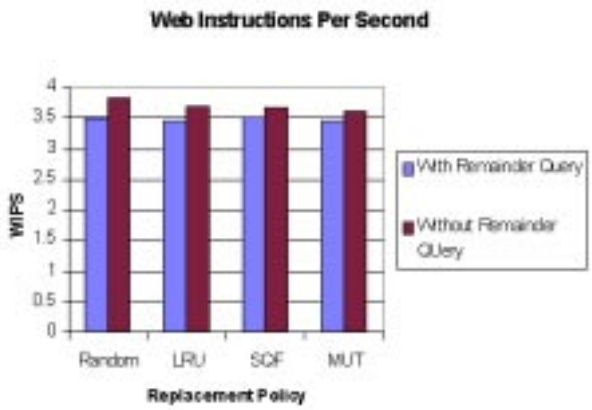

Figure 6: Comparison of remainder query and cache eviction policies in terms throughput, measured as web interactions per second. Using remainder queries, throughput dropped slightly.

queries may not be optimal; we should investigate the second approach we outlined in Section 4.2.2. Finally, and probably most importantly, the implementation was sub-optimal. Occassionally, the database server was unable to process a remainder query because of a JDBC exception. The JDBC driver on the server side was unable to handle SQL queries above a certain size. When an exception occured, the cache simply sent the original query, without a remainder query of any size. Exceptions added latency to responses and also increase the bandwidth consumed by faults. We believe that allocating more memory to the server's JDBC driver will improve latency and bandwidth, widening the performance gap of using remainder queries.

None of the evaluated cache replacement policies dominates as the best or worst choice. Both the tuple hit rate and the query hit rate remainded relatively constant among the caching policies. The relative hit rates also show that queries have tuple-locality rather than query-locality. For example, when a query is executed, subsequent queries are likely to access the same tuples as the first query but will not duplicate the first query exactly. The low query hit rate is not detrimental to performance.

7 Conclusions and Future Work

We proposed a new infrastructure to scale database-backed Web applications. We believe that the Web application proxy is a promising architecture for addressing the challenges of scaling dynamic content. We introduced generalized semantic caching techniques to manage the cache, thus allowing easier deployment while receiving the benefits of semantic caching. We implemented a prototype for TPC-W, a representative e-commerce application. Preliminary results show that our generalized semantic caching techniques can significantly reduce bandwidth consumption.

Future work includes improving the prototype implementation. Since the cache can return resident results, the Web application proxy should be able to process partial results, then add in the faulted results to reduce the overall latency of serving requests. The implemented remainder query generation techinques should be augmented with local selection conditions. Adding those selection conditions may significantly reduce the number of

tuples faulted into the cache.

Before the Web application proxy can be deployed in the Internet infrastructure, the cache should handle updates, remaining consistent with the primary database within some staleness threshold.

Acknowledgments

We would like to thank Kevin Walsh for his extensions to the Wisconsin release of TPC-W to correct the code to match the TPC-W specifications and to make the code more general and more usable.

References

[1] *Transaction Processing Performance Council*. http://www.tpc.org/tpcw/.

[2] Harold W. Cain, Ravi Rajwar, Morris Marden, and Mikko H. Lipasti. An architectural evaluation of Java TPC-W. In *The Seventh International Symposium on High-Performance Computer Architecture*, January 2001.

[3] P. Cao, J. Zhang, and K. Beach. Active cache: Caching dynamic contents. In *IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98)*, The Lake District, England, September 1998.

[4] J. Challenger, A. Iyengar, and P. Dantzig. A scalable system for consistently caching dynamic data. In *IEEE INFOCOM '99*, March 1999.

[5] J. Challenger, A. Iyengar, K. Witting, C. Ferstat, and P. Reed. A publishing system for efficiently creating dynamic web content. In *IEEE INFOCOM 2000 Conference*, Tel-Aviv, Israel, March 2000.

[6] Shaul Dar, Michael J. Franklin, Bjorn r Jonsson, Divesh Srivastava, and Michael Tan. Semantic data caching and replacement. In *The VLDB Journal*, pages 330–341, 1996.

[7] Laura M. Haas, Donald Kossmann, and Ioana Ursu. Loading a cache with query results. In *VLDB Conference*, Edinburgh, Scotland, September 1999.

[8] Yannis Kotidis and Nick Roussopoulos. DynaMat: a dynamic view management system for data warehouses. In *SIGMOD*, pages 371–382, 1999.

[9] Qiong Luo, Jeffrey F. Naughton, Rajasekar Krishnamurthy, Pei Cao, and Yunrui Li. Active query caching for database web servers. In *WebDB (Selected Papers)*, pages 92–104, 2000.