Form-Based Proxy Caching for Database-Backed Web Sites

Qiong Luo Jeffrey F. Naughton

University of Wisconsin-Madison
Computer Sciences Department
1210 West Dayton Street
Madison, WI 53715, USA
{qiongluo, naughton}@cs.wisc.edu

Abstract

We explore a new proxy-caching framework that exploits the query semantics of HTML forms. We identify a common class of form-based queries, and study two representative caching schemes for them within this framework: (i) traditional passive query caching, and (ii) active query caching, in which the proxy cache can service a request by evaluating a query over the contents of the cache. Results from our experimental implementation show that our form-based proxy is a general and flexible approach that efficiently enables active caching schemes for database-backed web sites. Furthermore, handling query containment at the proxy yields significant performance advantages over passive query caching, but extending the power of the active cache to do full semantic caching appears to be less generally effective.

1 Introduction

Many web sites managing significant amounts of data use a database system for storage. When users access such a web site, clicking on a URL in the HTML page they are viewing causes an application at the web site to generate database queries. After the DBMS executes these queries, the application at the web site takes the result of the queries, embeds it in an HTML page, and returns the page to the user. Figure 1 illustrates such a configuration. Under heavy loads, the database system can become the bottleneck in this process. Our goal in this paper is to explore proxy-caching techniques to alleviate this bottleneck.

Throughout the Internet, proxy caches are used to improve performance and share server workload. There are two kinds of deployment for these proxies. One is a traditional deployment, in which the proxies serve the content from the Internet to a group of users. In this case, the web sites being proxied may not even know of the existence of the proxies. An example is a campus proxy for speeding up the Internet access of local users. The other is ***reverse proxy caching***, in which the proxies serve a specified set of servers to general Internet users. In this case the web sites and the proxies can collaborate. For example, web sites often set up their own reverse proxies or contract with the Content Delivery Network services to use theirs.

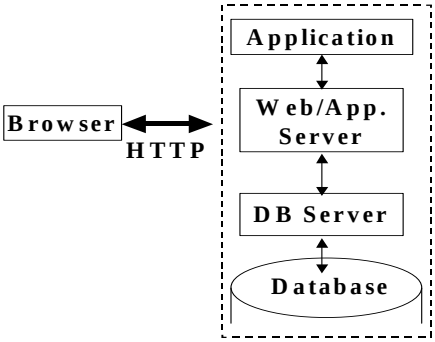

Figure 1: DB-Backed Web Site

In either deployment scheme, the function of these proxies is simple – if a proxy has seen a URL before, and has cached the page corresponding to that URL, it can return the cached page without accessing the web site that is the "home" for that page. When extending a proxy cache to handle access through a form-based interface, one needs to consider the relationship between the user, the form on the HTML page, and the queries that are generated at the database system at the web site.

**Proceedings of the 27th VLDB Conference,
Roma, Italy, 2001**

If clicking on a given URL always generates the same database query (that is, the request generated by clicking on the URL embeds no information from the user), then the proxy can work as if the URL referred to a static page stored at the web server. We call this scheme ***passive caching***, because it caches a page and returns it on a hit without any extra processing on the page. Unfortunately, in general things are not this simple, because instead of clicking on a URL, users are filling in forms. The user input from these forms is incorporated in the queries that eventually get executed by the database system. A common example of this might be in a book selling web site, where a user keyword search on the book title might generate a SQL query containing a "LIKE" predicate with the keywords provided by the user.

One can still use passive caching in such a scenario – the proxy cache associates cached pages with (URL, user input) rather than just with the URL. However, this means that the proxy cache will only be able to service a request if it has cached a previous request for the same form with the exact same user input. Our goal is to see if we can do better than that – we want to extend the proxy cache so that it can not only service requests that exactly match previous requests, but it can also service requests that can be answered by processing results of previous requests. We term this kind of caching ***active caching***, because the proxy is actively functioning in a limited query processing role.

Caching in this context poses a number of challenges not found in other database caching applications; many of these challenges arise because there is a high degree of independence between the database system and the proxy cache. In our work, we characterize what can be done in terms of how closely the web site is willing to collaborate with the proxy cache. For example, we show that if the web site will give no information at all, only passive caching is possible. If the web site is willing to expose the text of the queries its applications generate from the forms, then containment-based active caching is possible. Finally, if the web site provides a facility whereby the proxy can submit modified queries to the server, the proxy can do semantic active caching, exploiting query overlap as well as containment.

Also, if an active proxy scheme is to be widely useful, it must not require custom modifications to existing proxy servers, nor can it require programming effort on behalf of the individual web sites that are being served by the proxy. In our implementation, the caching module is a Java servlet for the unmodified Apache Tomcat servlet engine [2], and there is no programming required of the represented web sites.

In addition to defining and implementing this framework, we have performed experiments with our implementation using the TPC-W benchmark and modifications of that benchmark. These experiments show that query containment active caching generally provides a substantial improvement over purely passive caching; however, extending this to full semantic caching was only effective in specially crafted workloads. Finally, we validated these synthetic workload results with an experiment in which we proxied a real online bookseller using real-world user traces.

2 Form-Based Proxy Caching Framework

The goal of this framework is to efficiently facilitate active caching mechanisms for database-backed web sites in a general way. Despite the large volume of user queries that these web sites must handle, those queries are not arbitrary SQL; instead, they are usually submitted through simple HTML forms. Our key observation is the following: form-based queries enable a useful variety of active caching schemes that would be impractical for arbitrary SQL queries. Inspired by this observation, we built a proxy-caching framework based on ***query templates***, which are parameterized query definitions that are instantiated with the parameter values in user requests at run time.

2.1 Forms and Query Templates

Figure 2: Example HTML Form

We start with a running example in Figure 2. The HTML form shown is a simplified search request page for an on-line bookstore as given in the TPC-W benchmark [20]. When a user types "Java Programming" in the text box and clicks the "Submit" button, an HTTP request containing the user input is sent to the server side. No matter what application program implementation the server side uses, be it a CGI script, a Java servlet, an Active Server Page, the HTTP request will result in a SQL query for the backend DBMS to execute. A corresponding SQL query from the example form is given in Figure 3. Notice that when the user input changes, only the string in the LIKE predicate changes in the SQL query. The form in Figure 2 can be abstracted into a query template as shown in Figure 4.

```
SELECT TOP 50 i_title, i_id, a_lname, a_fname
FROM     item, author
WHERE  a_id = i_a_id AND
           i_title LIKE '%Java Programming%'
ORDER BY i_title
```

Figure 3: Example Form-based Query in SQL

```
SELECT TOP 50 i_title, i_id, a_lname, a_fname
FROM     item, author
WHERE  a_id = i_a_id AND
           i_title LIKE '%$search_string%'
ORDER BY i_title
```

Figure 4: Example Query Template in SQL

We emphasize that these templates and queries are not executed at the proxy; rather, the proxy uses them for analysis purposes, so that it can exploit the semantics of the query for more sophisticated caching schemes than exact-match passive caching.

2.2 Implementation

We implemented a Java servlet on top of the Apache Tomcat servlet engine [2]; together they serve as a caching proxy. The cache servlet runs in the same process space as Tomcat, and a pool of multiple threads in the servlet engine handles simultaneous requests. We chose the Tomcat servlet engine for ease of development, portability, and performance, but the same approach can be applied to the Apache web server, the Squid proxy, or other enterprise application servers.

Our proxy cache stores the results of queries and uses them to answer subsequent queries. One question that must be addressed is how these query results should be represented in the cache. Because XML is the emerging data transfer format on the Web, we chose to cache query results in XML format. This frees us from data representation issues and allows us to cache for web sites without any format translation as long as they provide their query results in XML. However, this is not a requirement for our approach; any storage scheme at the proxy cache will work as long as one provides translators from the form result format into this cache format, and then again from the cache format to the browser format.

Each query template is a text file containing a parameterized query such as the one in Figure 4. In addition, associated with a query template, there is a query template information file in XML, which specifies the correspondence between the form parameters and the query template parameters.

Figure 5 shows the query template information file for the form in Figure 2. The information file specifies that the query template is for the form queries sent to the URI "/tpcwSearchRequest.xsql" and the parameter "search_type" in the requests should have the value "i_title". In addition, it specifies that the parameter "search_string" in the HTTP requests from the form corresponds to the parameter "$search_string" in the query template. By specifying query templates and their associated mapping information with forms in this declarative manner, we separate the proxy caching functionality from the implementation and data representation issues of the web sites.

```
<queryTemplateInfo>
        <URI>/tpcwSearchRequest.xsql</>
        <paramPair>
                <paramName>search_type</>
                <paramValue>i_title</>
        </paramPair>
        <paramNameMapping>
                <requestParam>search_string</>
                <queryParam>$search_string</>
        </paramNameMapping>
</queryTemplateInfo>
```

Figure 5: Example Query Template Info File

For passive query caching, the proxy just needs to map an incoming HTTP request to a file name according to the parameter descriptions in the query template information, and check if this file is cached on disk. If the file is not cached, the proxy forwards the request to the server, caches the result by that file name when the result comes back from the server, and returns the result to the user. Otherwise, the proxy reads the cached file and returns the content to the user. For active caching, the proxy goes through a similar process of checking the cache using query templates, although the proxy processing and server interaction is more complex. We discuss form-based active caching in detail in Section 4.

2.3 Deployment Issues

The only difference between deploying a regular proxy and deploying our form-based proxy is that for active caching our proxy needs to know the query semantics of the forms. This is necessary because the application at the web server can perform arbitrary computations based upon the user input. Thus, to enable active caching, we require that the web site provide the text of the SQL query corresponding to each form. It can do so through the use of query templates.

Query templates are provided in the configuration step. In the configuration step of a regular proxy, the proxy administrator specifies which URLs the proxy should cache by adding them into the configuration file. When configuring our form-based proxy, the administrator specifies which forms that the proxy should cache by adding the query template files and associated

information files to the appropriate directories at the proxy.

Consistency is always an issue in caching. We regard consistency as an interesting area for future work that is largely orthogonal to this paper. The web currently works surprisingly well with a very relaxed attitude toward consistency. It is possible that many applications will be well served by simply providing a facility for the web site to invalidate data and/or templates stored at a proxy.

Finally, recent research in the web caching community has focused on adding application logic to the proxy from remote sites while the proxy is running. For example, the Active Cache Protocol [4] allows small software modules to be shipped from the web servers to the proxy on demand, specifying application-specific caching policies, while the Dynamic Content Cache Protocol [19] supports application-specific headers specifying caching policies. Our caching modules could also be shipped on-demand if the Active Cache Protocol were supported, while the application-specific query template information for our framework could also be easily shipped from web sites if either of the protocols were supported. In this way proxies could dynamically implement our active caching schemes "on the fly" without manual intervention.

3 The Class of Queries Handled by the Cache

3.1 Queries in the Web Site Application

While our framework can be applied to forms containing arbitrary database queries, the efficiency of caching techniques is related to the characteristics of the queries. As a first step in applying this framework, we concentrate on a simple but common class of form-based web queries, which we call ***top-n conjunctive keyword queries (TCKQ)***. The class of web queries can be expressed in an SQL-like syntax (Figure 6).

The characteristics of the form-based queries include:

```
SELECT TOP n selection_list
FROM   target_relations
WHERE  search_predicate(search_field,
       $search_string) AND other_predicates
ORDER BY orderby_fields
```

Figure 6: Class of Form-Based Queries

- Select-project-join (SPJ)
- A ***parameterized*** search predicate
- An order by clause
- A top-n operation
- The search-by and order-by fields appear in the selection list.

As simple as it looks, this class of queries represents a large number of forms on the web, including those used in on-line catalog search forms and on-line bibliography search forms.

Although keywords can be connected using "OR" and "NOT", users on the web seldom use them. We examined a 1-million entry Excite Search Engine log and found only 361 entries used "NOT" and 519 entries used "OR". A report [18] on a 1-billion entry AltaVista search engine log also showed that 80% of the queries did not have any operators (+, -, AND, OR, NOT, and NEAR). Thus, we focus on conjunctive keyword predicates.

3.2 Queries Executed in the Cache

While our proxy handles query templates that look like the one in Figure 6 this does not mean our proxy executes joins. Rather, we treat all queries from a given form as simple top-n selection queries on a single table view with a keyword predicate. This is because under each query template, the only difference among the queries is the search strings in the search predicate. This is one strength of our approach – we cache tuples that may have been generated by complex processing at the server, and avoid that complex processing in the proxy.

In the remainder of this section we discuss ***queries from the same form***. Whenever appropriate, we omit the n value of the top-n clause, the fields in the selection clause, the target relations in the from-clause, the search field in the search predicate, the other predicates in the where-clause, and the order-by fields. We use terminology from relational databases as well as from XML interchangeably. For example, fields correspond to elements, and tuples correspond to sets of elements. Because of order-by and top-n operations, we need to include list semantics as well as set semantics. These definitions and facts are not new; we repeat them here to make this paper self-contained.

3.2.1 Definitions

A *list* is an ordered set. A list L1 is a *sub-list* of another list L2 if and only if the elements in L1 all appear in L2, and in the same order ignoring absent elements. L2 is then a *super-list* of L1. We also define a list intersection, union, and equivalence to be a set intersection, union, and equivalence with order correspondingly. We use the symbols $\subseteq$, $\not\subset$, $=$, $\cap$, $\cup$, to denote operators between sets as well as between lists.

We extend the standard definitions of ***query containment and equivalence*** to lists. A query Q_1 is contained in another query Q_2, denoted $Q1 \subseteq Q2$, if and only if for any database D, the result of the former, $Q_1(D)$, is always a subset (or sub-list, if order is required) of the latter, $Q_2(D)$. Q1 and Q2 are equivalent if and only if $Q1 \subseteq Q2$ and $Q2 \subseteq Q1$. Two queries Q1 and Q2 are ***disjoint*** if and only if for any databases D, $Q1(D) \cap Q2(D) = \emptyset$. Q1 and Q2 ***overlap*** if and only if $Q1 \not\subset Q2$, $Q2 \not\subset Q1$, and Q1 and Q2 are not disjoint.

Next, we explore conjunctive keyword queries.

Definition 1. [Conjunctive keyword predicate] An n-ary conjunctive keyword predicate is of the form contains(e,{k1,k2,…,kn}), where e is a field name, and {k1,k2,…,kn} is a set of distinct words. The predicate contains(e,{k1,k2,…,kn}) is true if and only if all of the keywords k1,k2,..,kn (not necessarily in that order) appear in the field e. ■

In relational databases a conjunctive keyword predicate can be simulated using the string "LIKE" predicates. Also, our keyword predicate corresponds to a Boolean query in Information Retrieval with e being the top-level document.

Definition 2. [**SORT**] A sort operation is of the form $SORT_o(T)$, where o is a list of fields, and T is a set of tuples whose fields are a superset of the fields in o. The operation returns a list of all tuples from T ordered by o. For simplicity, we will use SORT(T) when appropriate. ■

Definition 3. [**Top-n**] A top-n operation is of the form $TOP_n(L)$, where n is a natural number, and L is a list of tuples. The operation returns a list of the first min(n, cardinality(L)) tuples from L. For simplicity, we will use TOP(L) when appropriate. ■

Definition 4. [**CKQ**] A conjunctive keyword query (CKQ) is of the form $Q_e(\{k1,k2,\ldots,kn\})$ where Q_e is a query with a keyword predicate contains(e,{k1,k2,…,kn}). The query returns a set of tuples. For simplicity, we will use Q({k1,k2,…,kn}) when appropriate. ■

Definition 5. [**OCKQ**] An Order-by conjunctive keyword query (OCKQ), denoted OQ({k1,k2,…,kn}), is defined as SORT(Q{k1,k2,…,kn}) where Q is a CKQ. The query returns a list of tuples. ■

Definition 6. [**TCKQ**] A top-n conjunctive keyword query (TCKQ), denoted TQ({k1,k2,…,kn}), is defined as TOP(OQ{k1,k2,…,kn}). The query returns a list of tuples. ■

3.2.2 Properties of Queries

From definitions in Section 3.2.1, we have the following simple but useful facts and properties about the queries that we are caching.

Fact 1. $Q(\{k1,k2,\ldots,kn\} \cup \{j1,j2,\ldots,jm\}) = \sigma_{contains(e,\ \{k1,k2,\ldots,kn\})}(Q(\{j1,j2,\ldots,jm\}))$ ■

Fact 2. $Q(\{k1,k2,\ldots,kn\} \cup \{j1,j2,\ldots,jm\}) = Q(\{k1,k2,\ldots,kn\}) \cap Q(\{j1,j2,\ldots,jm\})$ ■

These two facts tell us how to answer more restrictive conjunctive keyword queries from less restrictive CKQs, by selection or intersection. Similar facts hold for OCKQs except the set semantics is replaced by the list semantics. However, these facts do not hold for TCKQs.

Fact 3. $TQ(\{k1,k2,\ldots,kn\} \cup \{j1,j2,\ldots,jm\}) \supseteq \sigma_{contains(e,\ \{k1,k2,\ldots,kn\})}(TQ(\{j1,j2,\ldots,jm\}))$ ■

Fact 4. $TQ(\{k1,k2,\ldots,kn\} \cup \{j1,j2,\ldots,jm\}) \supseteq TQ(\{k1,k2,\ldots,kn\}) \cap TQ(\{j1,j2,\ldots,jm\})$ ■

Next we show that CKQ and OCKQ have similar properties on containment and equivalence, but TCKQ do not.

Proposition 1. A CKQ Q1 = Q({k1,k2,…,kn}) is contained in a CKQ Q2 = Q({j1,j2,…,jm}) if and only if {k1,k2,..,kn} is a superset of {j1,j2,..,jm}. This also holds for OCKQ. ■

Proposition 2. A TCKQ TQ1 = TQ({k1,k2,…,kn}) is contained in a TCKQ TQ2 = TQ({j1,j2,…,jm}) implies {k1,k2,..,kn} is a superset of {j1,j2,..,jm}, but not vice versa. ■

For TCKQs the following stronger proposition holds.

Proposition 3. A TCKQ TQ1 = TQ({k1,k2,…,kn}) is contained in a TCKQ TQ2 = TQ({j1,j2,…,jm}) if and only if {k1,k2,..,kn} = {j1,j2,..,jm}. ■

For query equivalence, similar results hold for the family of conjunctive keyword queries.

Proposition 4. A CKQ Q1 = Q({k1,k2,…,kn}) is equivalent to a CKQ Q2 = Q({j1,j2,…,jm}) if and only if {k1,k2,..,kn} = {j1,j2,..,jm}. The same holds for OCKQs and TCKQs. ■

The following result says that if a CKQ is contained in a union of CKQs, it is contained in at least one of the CKQs in the union. Similar results hold for OCKQs.

Proposition 5. A CKQ Q1 = Q({k1,k2,…,kn}) is contained in a union of other CKQ's Q2∪Q3∪… ∪Qx, if and only if for some Qy, 2<=y<=x, Q1 is contained in Qy. ■

Finally, two CKQs are never disjoint because we can always find a database in which there is an answer to satisfy both of them:

Proposition 6. For any two CKQs Q1 = Q({k1,k2,…,kn}), Q2 = Q({j1,j2,…,jm}), Q1 and Q2 are not disjoint. The same holds for OCKQs and TCKQs. ■

4 Form-Based Active Caching

4.1 Design Decisions

In this paper we consider active proxy caching in which the cache can execute top-n conjunctive keyword queries. Certainly other classes of queries are possible (range queries are one obvious alternative), but top-n conjunctive keyword queries are a useful class and general enough to illustrate the strengths and limitations of our approach.

From the properties we studied in the previous section, we know that limiting the result size with top-n implies that one query contains another only when the two are

equivalent (Proposition 3), which prohibits anything other than passive query caching. Therefore, we cache only order-by conjunctive queries at the proxy. A cache of order-by conjunctive keyword queries is immediately useful if the form being cached issues such queries; it is also useful if the web site being proxied provides facilities by which the proxy can "strip off" top-N operators. In the latter case we cache order-by conjunctive queries without a top-N, applying the top-N predicate at the proxy before returning results to the user.

Given a cache of the union of results from order-by conjunctive keyword queries, when a new query comes in, there are three possibilities: the result of the new query could be contained in the cache, it could intersect with the cache, or it could be disjoint from the cache.

By Proposition 5, if an OCKQ is contained in a union of OCKQs, it is contained in at least one of them. Thus we do not need to consider combinations of cached queries, but only need to consider 1-1 relationships between the new query and the individual cached queries. Moreover, we can determine query containment for OCKQs by examining the keywords in the queries (Proposition 1). So for containment, we only need to compare the keywords in the new query and in the cached queries without examining the contents of the cache.

The situation changes for query overlap. If a new query is not contained in a cached query, by Proposition 6, it could overlap with any previously cached query; furthermore, we cannot tell if the query indeed overlaps with previously cached queries without going through the contents of the cache. If upon examining the contents of the cache we find that the query does overlap, we issue a query to the web server for the form to get the answers "missing" from the cache. Using the terminology from semantic caching [7], the query evaluated over the cache is the ***probing query***, whereas the difference query sent to the DBMS is the ***remainder query***. In our context, the remainder query is easy to specify.

Consider a new query *Q*, with keywords *k1*, *k2*, …, *km*. Furthermore, let Q*1(c1)*, …, *Qn(cn)* be the queries that currently appear in the cache, where *ci* is the conjunct of keywords that appear in query *Qi*. Then the remainder query *QR* is just *QR(k1,…, km, not c1, not c2, … not cn)*. We refer to the *not ci* as ***remainder predicates***.

Clearly, with a large cache *QR* will be enormous, and would cause severe problems if sent to the DBMS at the web site. Thus we need to pick out a few remainder predicates that can reduce the remainder result size effectively. Choosing a minimum number of remainder predicates from the cached queries to cover all the cached tuples is a computationally hard problem (it can be shown NP-complete by reduction from the vertex cover problem). Instead, we used simple heuristics to try to pick a fixed number of predicates that cover a large portion of the cache.

Finally, another decision is whether redundancy in overlapping query results is allowed in the cache. We chose to eliminate duplicates when merging results of queries into the cache. As we will see in the experiments, this choice causes some computational overhead but avoids filling the cache with duplicates.

4.2 Implementation

If a new query presented to the cache is contained in a previous query, we simply execute the conjunctive keyword query over the contents of the cache. If the query is not contained in a previous query, then things are more complex. Here the probing step is a selection query with the current search predicate on the cached query results. If we are using full semantic caching, we need to send a remainder predicate to the server. When the web server responds with the result of the remainder query, our cache merges this result with the result of the probe query, and sends the combined result on to the user. Furthermore, our cache merges the result of the remainder query in with the existing cache contents, and adds the original query to the list of cached queries.

An important special case occurs if we decide to handle only containment relationships and to ignore query overlap. In this case, we never send a remainder query; rather, we always pass on the original query to the web server, and merge the result of that query in with the current cache contents. This case is important because it does not require any special collaboration between the proxy cache and the web server (since no "new" queries need to be sent to the web server, it only sees requests that it would see in the absence of our proxy cache.)

When there is a top-N operator in the class of cached queries, we once again require closer collaboration with the web server, because we handle such queries by "stripping off" the top-N operator before sending the queries on to the web server. To support this class of query we also have a top-N operator in the cache, so that the proxy can apply it to the full result before it is passed to the user.

As we see from Figure 7, each cache consists of a row of cached queries from the same query template, a set of cached result tuples, and a lexicon of the words in the search field in the cached result tuples. The queries that exactly match or are contained in a previously cached query are not added to the cache, in order to keep the number of cached queries small. The cached tuples are the union of all the result tuples from previously cached queries. We used LRU for cache replacement.

The list of cached queries is used to answer a new query if it is an exact match to a cached query, or is contained in a cached query. Both exact match and contained queries can be answered completely at the proxy. If a new query is neither an exact match nor a contained query, the cached tuples are examined through the lexicon indexes to pick out satisfying tuples (those in the overlap between the query and the cache) for the new query.

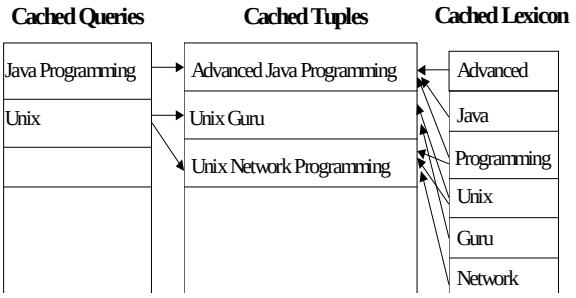

Figure 7: Example Cache Organization

5 Experiments

In this section, we first exercise the proxy-caching framework using the TPC-W book title search query traces. We then use modified workloads to investigate properties of active caching not revealed by the simple TPC-W traces.

5.1 Experimental Setup

There are four computers involved in the experiments. The four machines all have a Pentium III 800Mhz CPU and 256MB memory. The machine for the database server has 20GB disk space, while the other three machines each have a 9GB disk. All the machines are on a 100Mbit/second Ethernet.

All four machines use the RedHat Linux 6.2 operating system. The RBE program (Remote Browser Emulator) and proxy servlet are homegrown. The servlet engine is the Apache Tomcat Servlet Engine version 3.1, which supports the Java Servlet API v2.2. The database server in our experiments is Oracle 8.1.6 Enterprise Edition with the InterMedia Text 8.1.6 index server. We use Oracle XSQL servlet version 1.0.1.0 at the server side to process form-based queries and generate query results in XML. Table 1 summarizes the configuration.

Computer	RBE	Proxy	Server	Database
Software	RBE	Tomcat + servlet	Tomcat + XSQL	Oracle8i

Table 1: Software Deployment in the Experiments

5.2 On TPC-W Query Traces

To measure the effects of proxy caching on response times, we set up the TPC-W databases [20] at three scales: 10K, 100K, and 1M (in terms of the cardinality of the *item* table) in Oracle. The cardinality of the *author* table is ¼ of that of the *item* table. The ASCII data files of the two tables are of a total size of about 5MB, 50MB, and 500MB respectively. We used the default buffer pool size of 16MB in Oracle. We used the TPC-W search-by-title workload (form in Figure 2 and queries as in Figure 4).

The *i_title* field of the *item* table was generated using the TPC-W WGEN utility. In this dataset each title gets one "signature word", and each signature word is inserted into an average of five titles. The search string in a TPC-W query is a signature word. This causes each query to return an average of five books, and two queries in the trace are either identical (if they have the same search string) or have disjoint results (otherwise). This is the worst case for active caching because there is no query containment or overlap.

We ran a ten thousand query trace to the three scales of the TPC-W databases. This query trace contains two thousand distinct queries, and the caches reach a hit ratio of 80%. At the end of the experiment, both caches contained nearly 10K items. No cache replacement was triggered.

We compare timings in four cases: RBE directly to the server (Direct), RBE through the proxy without any cache (NC), RBE through the proxy with a passive query cache (PQ), and RBE through the proxy with an active query cache sending no remainder predicates (AQ0). The response times were measured in the RBE. Because the timings in the non-cache proxy case were almost identical to those of a miss in the PQ setting, we only show the three cases in Table 2.

Database scale		10K	100K	1M
Direct	Overall	74	384	4144
PQ	On hit	11	11	12
	On miss	110	442	4215
	Overall	31	98	853
AQ0	On hit	11	13	12
	On miss	262	539	4499
	Overall	61	118	905

Table 2: TPC-W Average Response Times (in ms)

From Table 2, we see that the database web server processing time dominates (comparing PQ cache misses with the direct-to-server case) and this gets worse when the scale of the database increases. Passive query caching achieves an overall average response time ¼ of that of the direct-to-server case. On a miss, passive query caching adds less than 70 milliseconds of overhead when compared to the direct-to-server case. The active cache adds another 100-280 milliseconds overhead on miss because of its more sophisticated query cache management. As the scale of the database increases, this overhead is dominated by the server time.

5.3 Adding Overlap in Queries

Since the TPC-W query trace generates queries with only disjoint small results, we generated another set of traces, which we term *NounPhrase* traces, from the TPC-W

vocabulary. NounPhrase traces explore how well the active cache performs when a new query is contained in a cached query or intersects with some data in the cache.

Trace	Noun100	Noun80	Noun60	Noun40
1-noun	20%	20%	20%	20%
2-noun	20%	20%	20%	20%
3-noun	20%	20%	20%	0
4-noun	20%	20%	0	0
5-noun	20%	0	0	0
Dummy	0	20%	40%	60%

Table 3 : Composition of NounPhrase Traces

The four NounPhrase traces we experimented with were Noun40, Noun60, Noun80, and Noun100. Each trace contains two thousand queries; which can be queries with one noun, two nouns, ... five nouns, or a dummy word as the search string (their percentages in the traces are shown in Table 3). Each noun was chosen independently from one another with a Zipfian distribution from the 100 most popular nouns in the TPC-W vocabulary. The dummy words in each trace were distinct and returned no answers. The different percentages of noun queries in the traces were designed to yield similar exact match ratios but different containment ratios across the traces. As a result, the exact match ratios of the four traces were all around 20%, and the ratios of contained queries were 12%, 33%, 52%, and 71%.

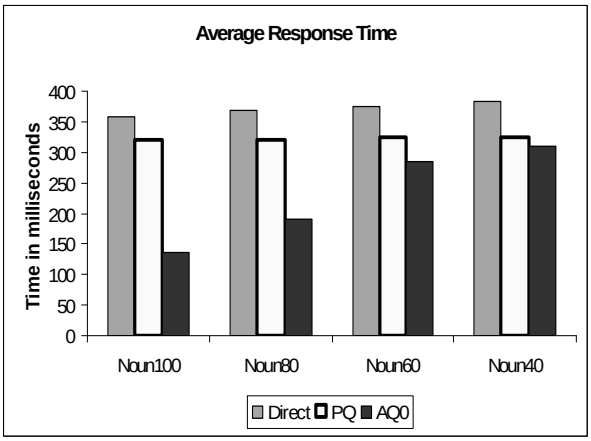

Figure 8: NounPhrase Average. Response Times

Figure 8 shows the average response times of the four NounPhrase traces on the 100K-scale TPC-W database running directly to the server (Direct), through a passive query caching proxy (PQ), or through an active caching proxy with no remainder predicates (AQ0). Recall that this (AQ0) is the case that does not require close collaboration between the web server and the proxy cache. We see that: (1) When the number of noun queries on the fixed vocabulary increases, the ratio of exact matches does not change much and passive query caching has a limited performance. (2) When the number of noun queries on the fixed vocabulary increases, the ratio of contained queries increases and benefits active caching to a larger extent.

Next we examine in detail the time spent by individual queries at the proxy.

We compare four cases at the active cache: an exact match (MATCH), a containment (CONT), an overlap (INTER), and a miss (MISS). For the passive query cache, this is simply MISS or HIT. Because the response time of a query depends on many factors, such as the current contents of the cache, the result size, and the database web server status, we ran the Noun40 trace three times, chose four representative queries in the trace, and show their times averaged from the three runs.

From Table 4 we see that both caches have similar response times on an exact match query (Query 515). A contained query (Query 511) also has similar response time to an exact match (Query 515) in the active cache, which is much better than a miss in the passive query cache. Query 514 is a dummy query returning no answers, and an active cache miss on it is 27% more expensive than a passive cache miss. Query 510 is a 2-noun query returning 50 tuples (top 50), and an active cache intersection is three times slower than a passive cache miss. This is because in the passive query cache case, only the top 50 tuples are obtained from the server, returned to the user, and saved into the cache while in the active cache case the active cache gets 62 result tuples from the cache, gets 510 result tuples (the whole answer set) from the server, merges these two parts of answers to eliminate duplicates, returns the top 50 to the user, and caches the un-cached answers.

Query ID		510	511	514	515
AQ0	Status	INTER	CONT	MISS	MATCH
	Time	2683	18	472	17
PQ	Status	MISS	MISS	MISS	HIT
	Time	664	361	376	18

Table 4: Numbers of Remainder Tuples of Query 510

We conducted further experiments on the Noun40 trace and found that increasing the number of remainder predicates had a very limited effect on reducing the number of remainder tuples (as an example, we show this for Query 510 in Table 5). This was because in the TPC-W database there is very little overlap among titles.

#Remainder predicates	0	10	20	30	40
#Remainder tuples	510	502	491	484	480

Table 5: Response Times of Four Cases (in ms)

5.4 Adding Overlap in Datasets

Because the TPC-W dataset had so little overlap, we generated a dataset with the same TPC-W *item* schema

but used a 10-word vocabulary {*w0,w1,w2,…,w9*} for the title field. This data set was tailor-made to benefit remainder processing.

In this dataset, each title field had three words: the id, *wi*, and *wj*, where 0=<i, j <9. There were 100 distinct combinations of the (*wi,wj)* pairs, but the id field was unique so that each title was unique. We generated 1000 tuples with each combination of (*wi,wj)* appearing in 10 titles and appended these 1000 tuples to the 100K TPC-W database. We then ran then ten queries *w0*, *w1*, …, *w9*, and compared the performance of the 10th query with varying numbers of remainder predicates. Note that here the selection heuristic used for remainder predicates is not important, because in this scenario all remainder predicates are equivalent. Table 6 shows the number of remainder tuples of Query 10 and Figure 9 shows the timing breakdown, averaged over three runs.

#Remainder predicates	0	5	10
#Remainder tuples	190	90	10

Table 6: Numbers of Remainder Tuples of Query 10

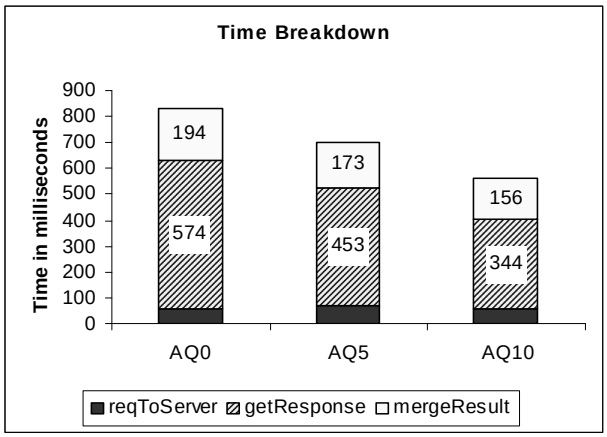

Figure 9: Time Breakdown of Query 10

The legends from left to right in Figure 9 correspond to the portions bottom up in the bars. The time spent on probing the cache and sending the remainder query to the server (reqToServer) was small. The time taken waiting for the server response (getResponse) and merging the probe results and the remainder results (mergeResult) were comparable. Both the server response time and the proxy result-merging time decreased when the number of remainder predicates increased. We also experimented with a dataset one magnitude larger than this one (10,000 special tuples inserted into the 1M TPC-W database) and observed the same pattern.

5.5 Experiments Omitted

Due to space limitations, in this paper we omit experiments investigating the effect of "combinatorial blowup" on caching schemes that do not eliminate duplicates. Similarly, we omit experiments that tested our cache as an accelerator for real-world online book selling web sites. The interested reader can find details in [13].

6 Related Work

To our knowledge, this paper is the first that explores an active proxy-caching framework for database-backed web sites explicitly based on query templates. Caching and materialization for databases on the Web has received a lot of attention recently ([3], [5], [9]). These studies all consider passive caching of the HTML or XML pages generated from DBMS-resident data. In contrast, our main focus is active caching.

Research in web caching that is most closely related to ours includes [4], [14], [15], and [19]. Studies [4], [15], and [19] did not consider database queries. Our previous work [14] focused on how a custom proxy caching protocol could be used to distribute caching code for select-project-join queries to proxies on the fly. However, it did not study the main issues we focus on here, including how forms can be used in the definition and deployment of caching schemes, and how well these schemes perform for keyword-based queries over the web.

There has been a large body of work ([1], [6], [8], [10], [11], [12], [17]) in data caching, query caching, and answering queries using views. Some of them ([6], [8], [10], [12], [17]) dealt with relational queries while others ([1], [6], [11]) focused on caching for heterogeneous sources. Our work builds on semantic caching as presented in [6], and is closely related to [6] and [11]. [11] focused on algorithms for choosing the best matching query in the context of semantic caching for range queries. While [6] studied semantic caching for keyword queries over search engines, we focus on using query templates to enable active caching for database-backed web sites. Also, [6] did not present a performance study.

Finally, there is an increasing commercial interest in caching for database web servers. The Oracle 9i Application Server [16] includes the Oracle Database Cache and the Oracle Web Cache. The Oracle Web Cache does passive caching. The Oracle Database Cache currently caches full tables; caching selected rows and columns, and caching query results may be available in the future release. To be used in a proxy cache scenario, the table level caching approach requires the DBMS data to be replicated to the proxy and an SQL query processor at the cache. This shifts the entire query computation from the DBMS to the proxy. Our approach, on the other hand, caches query results, thereby avoiding re-computation and requiring much simpler computation at the cache. Furthermore, unlike our approach, full table caching cannot take advantage of caching only "hot regions" of the result space. However, also unlike our approach, full table caching with a SQL processor can answer arbitrary

queries on those tables. A detailed comparison of the two approaches is an interesting area for future work.

7 Conclusions and Future Work

We have described a form-based proxy-caching framework for database-backed web servers. We studied two representative caching schemes for web queries using a full system implementation and evaluation. We show that while passive query caching is sufficient for the TPC-W workloads, active caching is more promising for other generated traces and real workloads. More specifically, answering contained queries results in a significant performance gain, but answering cache-intersecting queries is probably not worthwhile for the top-n conjunctive keyword queries. Finally, different caching schemes rely on different degrees of collaboration from servers. Passive query caching does not need query semantics information from the server, handling top-n queries needs some facility for getting the full answers from the server, and full semantic caching needs the server to handle remainder queries.

Acknowledgements

Thanks to our database group for valuable feedback, and Hongfei Guo for the TPC-W data generator. Funding for this work was provided by NSF through CCR-9734437, CDA-9623632 and ITR 0086002, and DARPA through NAVY/SPAWAR Contract No. N66001-99-1-8908.

References

[1] Sibel Adali, K. Selçuk Candan, Yannis Papakonstantinou, and V. S. Subrahmanian. Query Caching and Optimization in Distributed Mediator Systems. *SIGMOD Conference 1996*: 137-148.

[2] The Apache Tomcat Servlet Engine. http://jakarta.apache.org/tomcat/index.html

[3] K. Selçuk Candan, Wen-Syan Li, Qiong Luo, Wang-Pin Hsiung, and Divyakant Agrawal. Enabling Dynamic Content Caching for Database-Driven Web Sites. *SIGMOD Conference 2001.*

[4] Pei Cao, Jin Zhang, and Kevin Beach. Active Cache: Caching Dynamic Contents on the Web. *Middleware '98.*

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

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

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

[8] Arthur M. Keller, Julie Basu. A Predicate-based Caching Scheme for Client-Server Database Architectures. *VLDB Journal 5(1)*: 35-47 (1996).

[9] Alexandros Labrinidis and Nick Roussopoulos. WebView Materialization. *SIGMOD Conference 2000*: 367-378.

[10] Per-Åke Larson and H. Z. Yang. Computing Queries from Derived Relations. *VLDB85*: 259-269.

[11] Dongwon Lee and Wesley W. Chu. Caching via Query Matching for Web Sources. *CIKM99*: 77-85.

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

[13] Qiong Luo and Jeffrey F. Naughton. Form-based Proxy Caching for Database-backed Web Sites (Full version). Available at http://www.cs.wisc.edu/niagara/papers/formProxyFull.pdf.

[14] Qiong Luo, Jeffrey F. Naughton, Rajasekar Krishnamurthy, Pei Cao, and Yunrui Li. Active Query Caching for Database Web Servers. *WebDB 2000*: 29-34.

[15] Evangelos P. Markatos. On Caching Search Engine Query Results. In the *Proceedings of the 5th International Web Caching and Content Delivery Workshop*, May 2000.

[16] Oracle Corporation. Oracle Internet Application Server Documentation Library. http://technet.oracle.com/docs/products/ias/doc_index.htm

[17] Timos K. Sellis. Intelligent caching and indexing techniques for relational database systems. *Information Systems 13(2)*: 175-185 (1988).

[18] Craig Silverstein, Monika Henzinger, Hannes Marais, and Michael Moicz. Analysis of a Very Large AltaVista Query Log. SRC Technical Note 1998-014. Compaq, October 1998.

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

[20] Transaction Processing Performance Council (TPC). TPC Benchmark™ W (Web Commerce) Specification Version 1.1. June 27, 2000.