Form-Based Proxy Caching for Database-Backed Web Sites: Keywords and Functions

QIONG LUO[1] JEFFREY F. NAUGHTON[2] WENWEI XUE[1]

[1] *Department of Computer Science, Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong, China.*

[2] *Computer Sciences Department, University of Wisconsin-Madison, Madison, WI 53706, USA.*

Contact email: luo@cs.ust.hk
Phone: (852) 2358-6995
Fax: (852) 2358-1477

Abstract

Web caching proxy servers are essential for improving web performance and scalability, and recent research has focused on making proxy caching work for database-backed web sites. In this paper, we explore a new proxy caching framework that exploits the query semantics of HTML forms. We identify two common classes of form-based queries from real-world database-backed web sites, namely, ***keyword-based queries*** and ***function-embedded queries***. Using typical examples of these queries, we study two representative caching schemes within our 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.

Keywords: web proxy caching, database-backed web sites.

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.

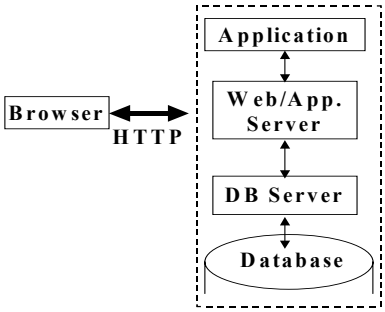

Figure 1: A DB-backed web site

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.

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.

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 refers 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 an 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 full 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 [4], and there is no programming required of the represented web sites.

We identify two representative classes of queries from real-world web sites to test our proxy caching schemes. One is *keyword-based queries*, which contain keyword search predicates, and the other is *function-embedded queries*, which embed calls to user-defined functions. Although simple, these two classes of queries are common in practice and pose challenges for active caching in a web proxy. Among the most prominent challenges are how to extract the query semantics of these two classes of queries and how to perform efficient active caching for these queries in an environment with limited collaboration from web sites. Moreover, function-embedded queries are harder to handle than keyword-based queries, due to the black-box nature of user-defined functions. Note that the keyword-based queries we handle are for the backend SQL databases of the web sites and are not web search queries over documents. Also, as a first step of studying active caching for function-embedded queries, we focus on the user-defined functions with spatial selection semantics.

In addition to defining and implementing this framework, we have performed experiments with our implementation using the TPC-W benchmark [42] and modifications of that benchmark. Moreover, we validated these synthetic workload results with experiments in which a real online bookseller and the SkyServer [38] web sites were proxied and real-world user traces were sent through our proxy to the original sites. These experiments show that query containment active caching generally provides a substantial improvement over purely passive caching; however, extending to full semantic caching was only effective in specially crafted workloads.

The remainder of the paper is organized as follows. In Section 2, we present our form-based proxy-caching framework. In Section 3, we define the two classes of queries handled in our framework. In Section 4, we describe our form-based active caching schemes for the two classes of queries. In Section 5, we present our experimental results. We discuss related work in Section 6 and draw conclusions in Section 7.

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, these 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. For function-embedded queries, we further define *function templates* to describe the query semantics of user-defined functions.

2.1 Forms and Query Templates

We start with a running example of keyword-based queries. The HTML form shown in Figure 2 is a simplified search request page for an online bookstore as given in the TPC-W benchmark [42]. 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 an SQL query for the backend DBMS to execute. A corresponding SQL query from the example form is given in Figure 3.

tpcwSearchForm.html

Figure 2: The title-search HTML form

```
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: An SQL query from the title-search form

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 keyword-based 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 '%$search_string%'
ORDER BY i_title
```

Figure 4: The title-search query template

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 queries for more sophisticated caching schemes than exact-match passive caching.

2.2 Forms and Function Templates

Having seen an example HTML form and query template for keyword-based queries, we continue with function-embedded queries.

Traditionally, user-defined functions are employed in database applications that handle complex data, such as Geographical Information Systems (GIS) and Computer Aided Design (CAD) systems. With the proliferating use of the Web, certain types of database-backed web sites also heavily utilize user-defined functions in answering user queries. These function-embedded queries appear in applications for functionality extension, performance improvement, and user convenience. Take the nearest-neighbor query as an example. First, application-specific distance functions used in the query are likely to fall out of the range of DBMS built-in functions. In such a case, users must code their own functions and register the functions with the DBMS. Second, users can write their distance functions in the high-level

programming language of their choice and employ optimization techniques based on their domain knowledge. As a result, the compiled and registered user-defined functions can run efficiently in queries. Last, other users can call the registered functions without knowing the implementation details of the functions, which is especially convenient for non-expert users.

Unlike keyword strings, user-defined functions have diversified and complex semantics that may be very different from application to application. Consequently, for function-embedded queries, caching for real-world user traces is more meaningful than synthetic benchmarks. In our work, we use the SkyServer web site [38] as a case study to investigate proxy caching for function-embedded queries.

The SkyServer web site serves terabytes of the public Sloan Digital Sky Survey (SDSS) data for both astronomers and science educators. The web site contains many pre-defined functions, such as *fGetNearbyObjEq(ra, dec, radius)*, *fGetObjFromRect(min_ra, max_ra, min_dec, max_dec)*, and *fPhotoFlags (FlagName)*. These functions are frequently called to serve queries submitted through HTML search forms. In addition, web users can input arbitrary function-embedded queries directly by using the "SQL" section of the "Search" facility provided by the web site. The semantic information of the functions as well as the text of the function-embedded queries is available through various online public documentations of the web site.

In this paper, we focus our study on *table-valued functions*, which return a set of tuples, as opposed to *scalar functions*, which return a scalar value, because the special properties of the former bring additional challenges as well as opportunities for active caching in our form-based proxy. For example, compared with the single value returned by a scalar function, the multiple result tuples of a table-valued function consume more network resource for shipping, but accumulate more data for future query answering.

Figure 5 shows the radial search form of the SkyServer web site (http://skyserver.sdss.org/en/tools/search/radial.asp). This form is one of the search facilities provided by the SkyServer. It searches the celestial sky around a given point and returns objects (tuples) within a given radius. It has a corresponding

function-embedded query template calling a table-valued function named fGetNearbyObjEq(). The query template of the form is shown in Figure 6.

Figure 5: The SkyServer radial search form

```
SELECT   [top n]
         p.objID, p.run, p.rerun, p.camcol, p.field, p.obj, p.type, p.ra, p.dec,
         p.u, p.g, p.r, p.i, p.z, p.Err_u, p.Err_g, p.Err_r, p.Err_i, p.Err_z
FROM     fGetNearbyObjEq($ra, $dec, $radius) n,   PhotoPrimary p
WHERE    n.objID = p.objID [AND other_predicates]
```

Figure 6: The radial search query template

As shown in Figure 6, the table-valued function fGetNearbyObjEq() is called in the FROM clause of the SQL statement. Moreover, the parameters of the function correspond to the user input parameters in the form. The query template is a join between the result table of the function and the PhotoPrimary table. From the documentation of the SkyServer, we know that the join with the PhotoPrimary table is for the purpose of tuple filtering and attribute list expansion. When a user fills out the form with some input values and clicks the "Submit" button on the HTML page, the function parameters are instantiated and the result table of the function can be computed. Then the query can be evaluated as an SQL query on the two tables.

In addition to the join condition, there are other optional predicates (the range predicates on the attributes u, g, r, i, z) in the WHERE clause. We use "other_predicates" to represent them, since they are not our focus in this work and

they do not affect the properties of the function-embedded queries we study. Finally, there is an optional top-n operation on the tuples of projected attributes. In general, these optional components of function-embedded queries are seldom used by web users in real-world scenarios, as we observed in the query traces of various HTML forms extracted from the SkyServer web logs.

If we consider active caching for queries of such a function-embedded query template, we immediately encounter a problem: active caching requires checking the relationship between two queries, but we can not tell the relationship (except an exact match) between two queries from this template even after the function parameters are instantiated. The fundamental reason is that we do not know the processing logic of the function. To solve this problem, we propose to use function templates to capture the semantic information of user-defined functions for the purpose of enabling the relationship checking between two function-embedded queries from the same template. Note that this is not a problem for keyword-based queries, since the processing logic of keywords in database systems, i.e., matching substrings, is well known.

There are at least two problems associated with providing information about functions. The first is the diversity – user-defined functions vary widely from one application to another. How do we design general templates to cover as many cases as possible? The second problem is the complexity – some user-defined functions may have a mix of control statements and SQL statements, and may be implemented in some host languages such as C or Java. For such complex cases, how do we extract useful information about the functions?

Considering these two problems, we propose to define a function template based on its *query semantics*. This has two implications. First, if a function performs updates, we regard it unsuitable for active caching and do not provide a function template for it. Second, different function definitions may have different application semantics, but we attempt to abstract high-level semantics of these functions into a function template with the same query semantics.

Let us continue with the table-valued function fGetNearbyObjEq() called in the Radial search form to illustrate a function template example (Figure 7).

```
<functionTemplate>
    <name>fGetNearByObjEq</name>
    <params>
        <1>$ra</1>
        <2>$dec</2>
        <3>$radius</3>
    </params>
    <shape>hypersphere</shape>
    <numDimensions>3</numDimensions>
    <centerCoordinate>
        <1>cos($ra)*cos($dec)</1>
        <2>sin($ra)*cos($dec)</2>
        <3>sin($dec)</3>
    </centerCoordinate>
    <radius>$radius</radius>
</functionTemplate>
```

Figure 7: The function template of function fGetNearbyObjEq()

In our implementation, a function template is an XML text file specifying high-level query semantics about the user-defined function. As shown in Figure 7, the function template specifies the following information about the function:

(1) The function name is fGetNearByObjEq.

(2) The function has three parameters: $ra, $dec, and $radius.

(3) The returned result tuples are considered as points bounded by a hypersphere.

(4) The hypersphere is three-dimensional, i.e., each point has three Cartesian coordinates.

(5) The center of the hypersphere has three coordinates, each of which is computed from the function input parameters $ra and $dec using the specified formula.

(6) The radius of the hypersphere is given by the function input parameter $radius.

In this template, the query semantics of the function is abstracted as *finding all points that are bounded by a 3-D hypersphere.*

2.3 Implementation

We implemented a Java servlet on top of the Apache Tomcat servlet engine [4]; 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 could be applied to the Apache web server, the Squid proxy, or 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. In our experiments with a real-world bookstore web site (Section 5.2.5), we built such a translator at the proxy for the HTML-serving web site, and the translation time for the query results was negligible compared with the network time. Moreover, some web sites are able to provide their query results in various formats. For example, the SkyServer provides the query results of its HTML forms in XML, HTML and CSV formats.

In our framework, each query template is a text file containing a parameterized query such as those in Figure 4 and Figure 6. In addition, associated with a keyword-based query template, there is a keyword-based query template information file in XML, which specifies the correspondence between the form parameters and the query template keyword parameters.

Figure 8 shows the query template information file for the keyword-based query template in Figure 4. The information file specifies that the query template is for the form queries sent to the URI "/tpcwSearchRequest.xsql", the file name of the query template is BookTitleSearch.sql, 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.

Similarly, each function-embedded query template in our framework is associated with a function-embedded query template information file for the same mapping purpose. The one that corresponds to the query template in Figure 6 is shown in

Figure 9. This template information file specifies that the query template is for the form queries sent to the URI "/en/tools/search/radial.asp", and the file names of the query template and the embedded function template are RadialSearch.sql and fGetNearbyObjEq.xml (Figure 7) correspondingly. The parameters "ra", "dec", and "radius" in the HTTP requests submitted by the search form correspond to the input parameters "$ra", "$dec", and "$radius" of the embedded function, respectively. By specifying query and function 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.

```
<keywordBasedQueryTemplateInfo>
    <URI>/tpcwSearchRequest.xsql</>
    <queryTemplateName>BookTitleSearch.sql</>
    <paramPair>
        <paramName>search_type</><paramValue>i_title</>
    </paramPair>
    <paramNameMapping>
        <requestParam>search_string</><queryParam>$search_string</>
    </paramNameMapping>
</keywordBasedQueryTemplateInfo>
```

Figure 8: The title-search query template information file

```
<functionEmbeddedQueryTemplateInfo>
    <URI>/en/tools/search/radial.asp</>
    <queryTemplateName>RadialSearch.sql</>
    <functionTemplateName>fGetNearbyObjEq.xml</>
    <paramNameMapping>
        <requestParam>ra</><functionParam>$ra</>
    </paramNameMapping>
    <paramNameMapping>
        <requestParam>dec</><functionParam>$dec</>
    </paramNameMapping>
    <paramNameMapping>
        <requestParam>radius</><functionParam>$radius</>
    </paramNameMapping>
            ⋮
</functionEmbeddedQueryTemplateInfo>
```

Figure 9: The SkyServer radial search query template information file

2.4 System Architecture

Having in place the query templates, the function templates, and the information files for the templates, we describe the system architecture of our proxy framework.

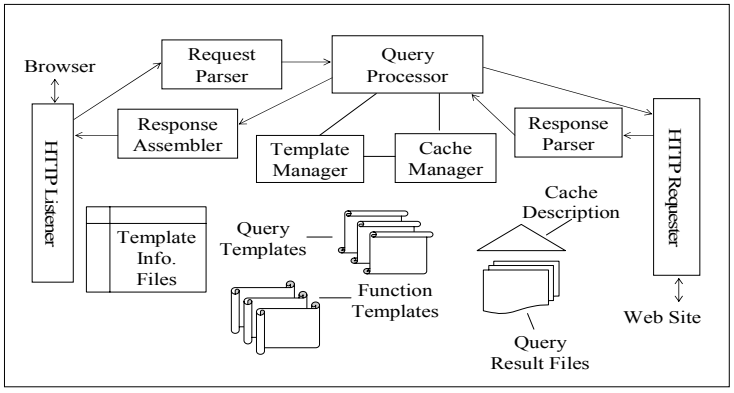

Figure 10: System architecture

Figure 10 illustrates the system architecture of our form-based proxy. The major components of the proxy include an HTTP listener, a request parser, a query processor, a template manager, a cache manager, an HTTP requester, a response parser, and a response assembler. In addition to the templates and the information files, the proxy keeps cached query results as disk files and has in-memory descriptions for the cache content.

In this implementation, the template information for each form is kept separately from one another and the two kinds of queries are handled separately because they have different types of template information. As a result, even if there may be commonality among different forms or different kinds of queries, the system is unaware of this commonality and processes queries for each form separately.

When the HTTP listener receives an HTTP request with a keyword-based or function-embedded query from a web browser, it passes the request to the request parser. The request parser parses the request and passes the information in the request to the query processor. The query processor invokes the template manager to translate the request information into a query with corresponding templates and template information files. It then works with the cache manager and the template manager to check the relationship between the new query and previously cached queries using the templates and the corresponding cache description in memory. Each entry in the cache description corresponds to one cached query, and has a pointer to the query result file stored on disk.

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 to 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 and function templates, although the proxy processing and server interaction is more complex. For instance, if the new query is an exact match to or is contained in some cached query, the query processor will evaluate the new query with the help of the information provided in the template files. Otherwise, the query processor may send the original query or *a remainder query* [10] to the original web site through the HTTP requester. The response parser receives the query result in an HTTP response from the web site, parses it into the format that the query processor can recognize, and passes the parsed result to the query processor. At last, the query processor assembles the result and passes it to the response assembler, which will in turn assemble the query result into an HTTP response and send it back to the browser. We discuss form-based active caching along with the proxy deployment and maintenance issues in detail in Section 4.

3 Two Classes of Queries Handled

Since the efficiency of caching techniques is related to the characteristics of the queries, we focus on two simple but common classes of form-based web queries, which are keyword-based queries and function-embedded queries. In the following, we describe these two classes of queries handled in detail.

3.1 Keyword-Based Queries

The keyword-based queries we study in this paper are *top-n conjunctive keyword-based queries (TCKQ)*. This class of web queries can be expressed in an SQL-like syntax as shown in Figure 11.

The characteristics of this class of queries include:

- Selection-Projection-Join (SPJ)

- A *parameterized* keyword search predicate
- An order by clause
- A top-n operation
- All attributes involved in the query are contained in the selection list.

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

Figure 11: The class of keyword-based queries handled

As simple as it looks, this class of keyword-based queries represents a large number of forms on the web, including those used in online catalog search forms and online 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 [13] log and found only 361 entries used "NOT" and 519 entries used "OR". A report [37] 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 in our study.

While our proxy handles keyword-based query templates that look like the one in Figure 11, it does not mean that our proxy executes joins. Rather, we treat all keyword-based queries from a given form as simple top-n selection queries on a single 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 an advantage of our approach – we cache tuples that may have been generated by complex processing at the server, and avoid that complex processing at the proxy. This approach also applies to function-embedded queries (to be described in Section 3.2); the only difference is that for function-embedded queries we need to consider the function parameters instead of keyword predicate parameters.

In the remainder of this section, we discuss *keyword-based queries from the same form*. Whenever appropriate, we omit the n value of the top-n operation, the fields in the SELECT 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 in addition to set semantics. These definitions and facts are not new; we repeat them here to make this paper self-contained.

3.1.1 Definitions

A *list* is an ordered set. A list L_1 is a *sub-list* of another list L_2 if and only if the elements in L_1 all appear in L_2, and in the same order ignoring absent elements. L_2 is then a *super-list* of L_1. 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 follow the standard definitions of *query containment and equivalence* for general SQL queries [17] and extend them to lists. A query Q_1 is *contained* in another query Q_2, denoted $Q_1 \subseteq Q_2$, 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, $Q_2(D)$. Q_1 and Q_2 are *equivalent* (exact match) if and only if $Q_1 \subseteq Q_2$ and $Q_2 \subseteq Q_1$. Two queries Q_1 and Q_2 are *disjoint* if and only if for any databases D, $Q_1(D) \cap Q_2(D) = \varnothing$. Q_1 and Q_2 *overlap* if and only if $Q_1 \not\subset Q_2$, $Q_2 \not\subset Q_1$, and Q_1 and Q_2 are not disjoint.

Next, we explore conjunctive keyword-based queries.

Definition 1. [Conjunctive keyword predicate] An n-ary conjunctive keyword predicate is of the form contains(e, $\{k_1, k_2, \ldots, k_n\}$), where e is a field name, and $\{k_1, k_2, \ldots, k_n\}$ is a set of distinct words. The predicate contains(e, $\{k_1, k_2, \ldots, k_n\}$) is true if and only if all of the keywords $k_1, k_2, \ldots, k_n$ (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-based query (CKQ) is of the form $Q_e(\{k_1, k_2, \ldots, k_n\})$ where Q_e is a query with a keyword predicate contains(e, $\{k_1, k_2, \ldots, k_n\}$). The query returns a set of tuples. For simplicity, we will use $Q(\{k_1, k_2, \ldots, k_n\})$ when appropriate.■

Definition 5. [**OCKQ**] An order-by conjunctive keyword-based query (OCKQ), denoted $OQ(\{k_1, k_2, \ldots, k_n\})$, is defined as $SORT(Q\{k_1, k_2, \ldots, k_n\})$ where Q is a CKQ. The query returns a list of tuples.■

Definition 6. [**TCKQ**] A top-n conjunctive keyword-based query (TCKQ), denoted $TQ(\{k_1, k_2, \ldots, k_n\})$, is defined as $TOP(OQ\{k_1, k_2, \ldots, k_n\})$. The query returns a list of tuples.■

3.1.2 Properties of Keyword-Based Queries

From definitions in Section 3.1.1, we have the following simple but useful facts and properties about the keyword-based queries that we are caching.

Fact 1. $Q(\{k_1, k_2, \ldots, k_n\} \cup \{j_1, j_2, \ldots, j_m\}) = \sigma_{contains(e, \{k_1, k_2, \ldots, k_n\})}(Q(\{j_1, j_2, \ldots, j_m\}))$■

Fact 2. $Q(\{k_1, k_2, \ldots, k_n\} \cup \{j_1, j_2, \ldots, j_m\}) = Q(\{k_1, k_2, \ldots, k_n\}) \cap Q(\{j_1, j_2, \ldots, j_m\})$■

These two facts tell us how to answer more restrictive conjunctive keyword-based 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. $TQ(\{k_1, k_2, \ldots, k_n\} \cup \{j_1, j_2, \ldots, j_m\}) \supseteq \sigma_{contains(e, \{k_1, k_2, \ldots, k_n\})}(TQ(\{j_1, j_2, \ldots, j_m\}))$■

Fact 4. $TQ(\{k_1, k_2, \ldots, k_n\} \cup \{j_1, j_2, \ldots, j_m\}) \supseteq TQ(\{k_1, k_2, \ldots, k_n\}) \cap TQ(\{j_1, j_2, \ldots, j_m\})$■

Next we show that CKQs and OCKQs have similar properties on containment and equivalence, but TCKQs do not.

Proposition 1. A CKQ $Q_1 = Q(\{k_1, k_2, \ldots, k_n\})$ is contained in a CKQ $Q_2 = Q(\{j_1, j_2, \ldots, j_m\})$ if and only if $\{k_1, k_2, \ldots, k_n\}$ is a superset of $\{j_1, j_2, \ldots, j_m\}$. This also holds for OCKQ.■

Proposition 2. A TCKQ $TQ_1 = TQ(\{k_1, k_2, \ldots, k_n\})$ is contained in a TCKQ $TQ_2 = TQ(\{j_1, j_2, \ldots, j_m\})$ implies $\{k_1, k_2, \ldots, k_n\}$ is a superset of $\{j_1, j_2, \ldots, j_m\}$, but not vice versa.■

For TCKQs the following stronger proposition holds.

Proposition 3. A TCKQ $TQ_1 = TQ(\{k_1, k_2, \ldots, k_n\})$ is contained in a TCKQ $TQ_2 = TQ(\{j_1, j_2, \ldots, j_m\})$ if and only if $\{k_1, k_2, \ldots, k_n\} = \{j_1, j_2, \ldots, j_m\}$.■

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

Proposition 4. A CKQ $Q_1 = Q(\{k_1, k_2, \ldots, k_n\})$ is equivalent to a CKQ $Q_2 = Q(\{j_1, j_2, \ldots, j_m\})$ if and only if $\{k_1, k_2, \ldots, k_n\} = \{j_1, j2, \ldots, j_m\}$. 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 $Q_1 = Q(\{k_1, k_2, \ldots, k_n\})$ is contained in a union of other CKQ's $Q_2 \cup Q_3 \cup \ldots \cup Q_x$, if and only if for some Q_y, $2 <= y <= x$, Q_1 is contained in Q_y.■

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

Proposition 6. For any two CKQs $Q_1 = Q(\{k_1, k_2, \ldots, k_n\})$, $Q_2 = Q(\{j_1, j_2, \ldots, j_m\})$, Q_1 and Q_2 are not disjoint. The same holds for OCKQs and TCKQs.■

3.2 Function-Embedded Queries

In addition to the keyword-based queries, our proxy also handles a class of function-embedded queries as shown in Figure 12.

```
SELECT   [top n] selection_list
FROM     table-valued_function(parameter1, parameter2, …,parametern) [,other_ relations]
[WHERE   predicate_list]
```

Figure 12: The class of function-embedded queries handled

The characteristics of this class of queries include:

- Selection-Projection-Join (SPJ)
- A *parameterized* table-valued function in the FROM clause
- An optional WHERE clause
- An optional top-n operation
- All attributes involved in the query are contained in the selection list.

In addition to these syntactical characteristics, the class of function-embedded queries that can be handled by our proxy cache also has the following properties:

(1) **Spatial Selection Query Semantics.** The tuples in the result table of the table-valued function are abstracted as points in a multidimensional region. The function is equivalent to a spatial selection query, which returns all points in the multidimensional region. The shape of the region can be a hypercube (most common), a hypersphere, or even a polytope (more complex) [31]. As examples, the fGetNearbyObjEq() function of the SkyServer returns all objects within a 3-D sphere, and the fGetObjFromRect() function returns all objects within a 2-D rectangular region.

(2) **Semantics-Preserving Join.** If the table-valued function is joined with other tables in a function-embedded query, such joins should preserve the query semantics of the table-valued function. For example, the queries for the Radial search form

preserve the query semantics of the fGetNearbyObjEq() function, and return objects within a 3-D sphere with some additional predicates.

(3) **Result Attribute Availability.** Some attributes are required for checking the spatial relationship between a new function-embedded query and previously cached queries as well as for evaluating a new query result over previously cached ones. An important case is the attributes that serve as the Cartesian coordinates of the point that a result tuple represents in the multidimensional space. Therefore, we require that the attributes involved in the query relationship checking and local query evaluation are all contained in the cached result tuples.

If a function-embedded query has these three properties, we can transform the problem of checking the relationship between two queries (query exact match, containment, overlap, or disjoint) into that of checking the spatial relationship between the two corresponding regions. This is the basic idea that we use to implement our active caching schemes for function-embedded queries in our framework.

4 Form-Based Active Caching

In this section, we present the active caching schemes that we have designed and implemented in our form-based proxy for the two classes of queries. Our active caching schemes are all based on the semantic caching [10] and are tailored for the class of queries and the proxy-caching environment. The main idea of the original semantic caching is to treat cached query results as semantic regions and to evaluate a new query by utilizing the cache content and contacting the database server for the non-cached content. Each new query is compared with the description of the cached query results (which we call relationship checking) and is split into two queries: the *probing query* that is evaluated over the cache and the *remainder query* that is sent to the database server for the remaining result.

Even though the two classes of queries have similar active caching schemes, the implementation considerations for the two classes are different. For the clarity of presentation, we describe active caching for each class separately.

4.1 Active Caching for Keyword-Based Queries

4.1.1 Design Decisions

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

From the properties we studied in Section 3.1, we know that limiting the result size with top-n operation 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 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 operator at the proxy before returning results to the user.

Given a cache of the union of results from order-by conjunctive queries, when a new query comes in, there are three possibilities: the result of the new query could be contained in the cache (including exact match), it could overlap 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 the cached queries, but only need to check the 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 query 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 [10], the query evaluated over the cache is the *probing query*, whereas the difference query sent to the web server DBMS is the *remainder query*. In this context, the remainder query is easy to specify.

Consider a new query Q, with keywords $k_1, k_2, \ldots, k_m$. Furthermore, let $Q_1(c_1), Q_2(c_2), \ldots, Q_n(c_n)$ be the queries that currently appear in the cache, where c_i is the conjunct of keywords that appear in query Q_i. Then the remainder query Q_R is just $Q_R(k_1, k_2, \ldots, k_m, not\ c_1, not\ c_2, \ldots, not\ c_n)$. We refer to the *not* c_i as *remainder predicates*.

Clearly, with a large cache Q_R 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 query 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 use 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 cache-overlapping query results should be allowed in the cache. For keyword-based queries, 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.

4.1.2 Query Evaluation and Cache Organization

If a new query presented to the cache is contained in a previous query, we simply execute the conjunctive keyword-based 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 query to

the server. When the web server responds with the result of the remainder query, the proxy 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 with the existing cache contents, and adds the original query to the list of cached queries.

An important special case occurs if our proxy decides to handle only containment relationships and to ignore query overlap. In this case, the proxy never sends a remainder query; rather, it always passes on the original query to the web server, and merges the result of that query 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 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. If we consider that a user would examine only a few top results in practice, we could progressively get the next-n results instead of getting the full result set at once. The danger of this progressive next-n alternative is its incomplete result set and potential inefficiency if many results are needed for query processing anyway.

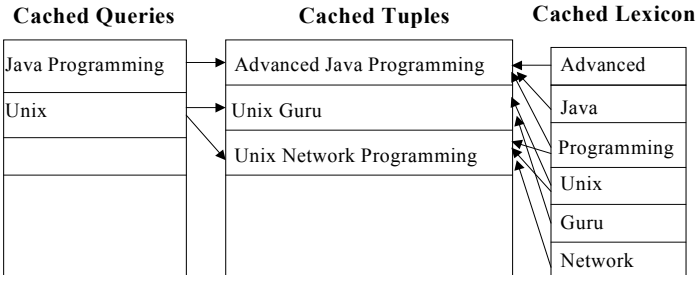

Figure 13: Cache organization for keyword-based queries

As we see from Figure 13, each cache for keyword-based queries consists of an array 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 use LRU (Least Frequently Used) for cache replacement.

The list of cached queries is an in-memory cache description that is used to check the relationship between a new query and previously cached queries. If a new query is neither an exact match nor a cache-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.

The result pointers of a cached query are ordered by the order-by field in the query template so that for an exact match or a cache-contained query, the top-n result tuples can be returned efficiently. However, the tuple pointers of a lexicon word are not ordered by the query template's order-by field, but by the pointer values themselves. This is to enable fast set intersection between the tuple pointers of two lexicon words. In our current implementation, we use a linear search on the cached queries because the queries are simple and cache replacement will limit the number of them. For a large number of complex cached queries, techniques such as those proposed by Altinel and Franklin [2] may be applicable.

Finally, a special case of query overlap is that the new query contains (subsumes) some cached queries. Previous work [9] terms this case *region containment*, in that the new query subsumes a region of the cache (one or more cached queries). For example, a new query containing the search keyword "Programming" subsumes the cached queries with the keywords "Java Programming" and "C++ Programming". In this special case, we remove the subsumed cached queries from the list of cached queries in order to limit the number of queries in the cache. Note that all tuples of the cached queries contained in the region are the results for the new query, so in region containment the cached tuples do not need to be checked one by one as in the query containment cases. This special case deserves attention, because it reduces the number of cached queries and improves cache utilization. To differentiate region containment with other cases of query overlap, we call those cases of query overlap that are not region containment *query intersection*.

4.2 Active Caching for Function-Embedded Queries

4.2.1 Algorithms for Relationship Checking

As described in Section 3.2, we transform the problem of checking the relationship between two function-embedded queries into that of checking the spatial relationship between the two corresponding regions for the class of queries we handle.

Agorithm 1: Hypersphere Relationship Checking
Input: the Cartesian coordinates of the centers and the radii $radius_1$ and $radius_2$ of two hyperspheres
Output: status (*DS*, *EM*, *FCS*, *SCF*, or *IS*)

```
1.  Compute the Euclidean distance d between the two centers
    using their Cartesian coordinates
2.  if ( d > (radius1 + radius2) )
3.     return DS;
4.  else if ( (d = = 0) && ( radius1 = = radius2) )
5.        return EM;
6.  else if ( (d + radius2) <= radius1 )
7.        return FCS;
8.  else if ( (d + radius1) <= radius2 )
9.        return SCF;
10. else return IS;
```

Figure 14: The hypersphere relationship checking algorithm

Algorithm 2: Hypercube Relationship Checking
Input: the lower bound, upper bound pairs (lb_{1i}, ub_{1i}), (lb_{2i}, ub_{2i}) for each dimension *i* of two hypercubes
Output: status (*DS*, *EM*, *FCS*, *SCF*, or *IS*)

```
1.  for each dimension i
2.     if ( (lb1i >= ub2i) || (lb2i >= ub1i) )
3.        return DS;
4.  equivalent = true;
5.  for each dimension i
6.     if ( (lb1i != lb2i) || (ub1i != ub2i) )
7.        { equivalent = false; break; }
8.  if (equivalent) return EM;
9.  contained = true;
10. for each dimension i
11.    if ((lb1i > lb2i) || (ub1i < ub2i) )
12.       { contained = false; break; }
13. if (contained) return FCS;
14. contained = true;
15. for each dimension i
16.    if (lb1i < lb2i) || (ub1i > ub2i) )
17.       { contained = false; break; }
18. if (contained) return SCF;
19. return IS;
```

Figure 15: The hypercube relationship checking algorithm

We have implemented active caching for the hypersphere and hypercube cases. The algorithms for checking the relationship between two regions (hyperspheres or hypercubes) in a multidimensional space are given in Figures 14 and 15 correspondingly. In these algorithms, the output DS, EM, FCS, SCF, and IS are pre-defined flags, which represent "the two regions are disjoint", "the two regions are equivalent (exact match)", "the first region contains the second one", "the second region contains the first one", and "the two regions intersect", respectively.

Figures 16 and 17 illustrate the relationships that are checked by the two algorithms respectively (both figures show 2-D cases only).

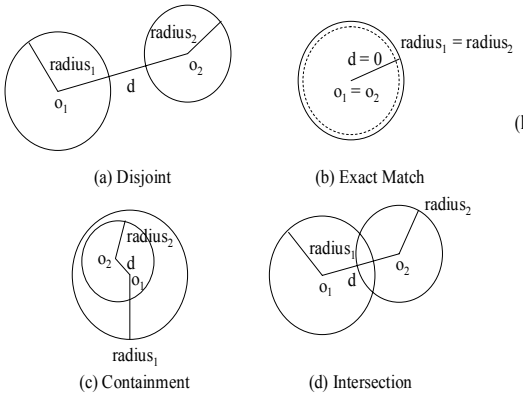

Figure 16: Hypersphere relationship illustration

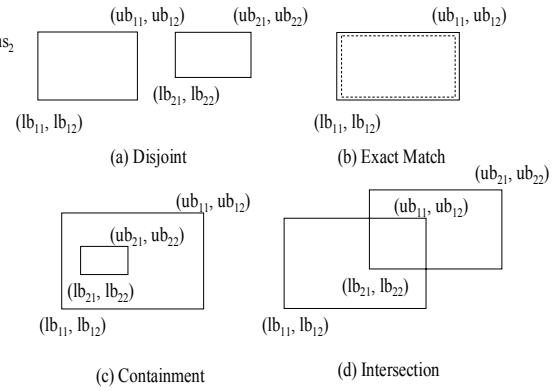

Figure 17: Hypercube relationship illustration

4.2.2 Query Evaluation

Given a new function-embedded query and the previously cached queries, the proxy uses the relationship checking algorithm (without looking at the query results) and returns a status for this new query. The status can be one of the five cases: (1) exact match, (2) query containment, (3) region containment, (4) query intersection, and (5) disjoint. After the status is identified, the proxy will evaluate the new query at the proxy correspondingly.

The handling of the two extreme cases (cases 1 and 5) is simple. On one extreme, if the new query is an exact match to a cached query, the proxy will read the cached result and return it to the user. On the other extreme, if the new query is disjoint

from all of the cached queries, the proxy will forward the query to and get a response from the web server, cache this result, and return it to the user.

If the new query is contained in a cached query (case 2), the proxy evaluates the new query over the result of that cached query and returns the new result to the user. The result of the new query is not cached, since it is already contained in the result of the cached query. The local evaluation process of the cache-contained query is as follows.

As described in Section 3.2, some attributes of a cached result tuple serve as the Cartesian coordinates of the point that the tuple represents in a multi-dimensional space. Therefore, the proxy evaluates the new query by checking the cached result tuples and selecting those that represent points falling into the multidimensional region of the new query. In essence, the evaluation of a subsumed query becomes that of a spatial region selection query over the cached results.

Note that the local evaluation procedure is determined by the spatial semantics (e.g., the shape and dimensions of region) of the function template and the function-embedded query template. Since we assume that the abstraction of the semantics of an application is correct, the local evaluation procedure will produce correct results for the application. Moreover, this local evaluation implementation is much simpler than the evaluation procedure at the web server, since the web server has to deal with a large amount of base data and executes its own (probably more application-specific) implementation of functions.

If there are multiple cached queries that subsume the new query, we choose the one that has the least number of result tuples. It is also possible that the new query is not subsumed by any single cached query but is contained in the union of some cached queries. We have not yet handled this case, since the complexity for such relationship checking is high and the performance gain is unclear.

Cases 3 and 4 both belong to query overlap, in which the cache can serve only a portion but not all of the answers to the new query. As in caching for keyword-based queries, the major decision is whether the proxy should send the original query or a remainder query to the web server. If the web server does not support modified queries at all, i.e., it does not have a remainder query facility, the proxy has no choice

but always sends the original query to the web site. In contrast, if the web site has a facility to handle remainder queries, we need to further consider the performance tradeoffs between the proxy, the original web server, and the network.

In the region containment case (case 3), our proxy merges the results of all of subsumed cached queries with the result of the remainder query to form the final result of the new query. After answering this new query, the result of the new query is cached and all those subsumed cached queries and their results are removed.

In the query intersection case (case 4), there may be multiple cached queries intersecting the new query. Correspondingly, the proxy needs to exclude these queries when formulating the remainder query. When there are many cached queries, the number of queries that overlap with the new query may be large as well. We used heuristics in selecting a fixed number of cached queries to be used in probing query evaluation and remainder query formulation. For queries representing hypercubes, we select candidates by the volume of the region that a cached query overlaps with the new query; the larger the volume, the better. However, for queries representing hyperspheres, computing the volume of the overlapping region is computationally expensive due to the shape. Therefore, we used approximation for hyperspheres. Let d be the Euclidian distance between the centers of the new query and a cached query (this distance is already computed during query relationship checking), and r be the radius of the cached query, we select candidates who have a small d / r value. The intuition is that the larger volume (r) a cached query has and the closer it is to the new query (d), the better it would be for answering the new query.

4.2.3 Cache Organization

Similar to cached keyword-based queries, cached function-embedded queries and their results are organized by query templates. However, the result of each function-embedded query is stored in a separate file. Therefore, there might be redundancy in the cached results of overlapping queries. This redundancy is tolerated so that there is no need to maintain the descriptions of the overlapping regions, whose shapes may be complex. Instead, duplicate elimination is performed in the query evaluation

process, specifically, in the merging process. We have implemented both hashing-based merging and sorting-based merging, and adopted the former for performance reasons.

For the cached results on disk, there are in-memory cache descriptions used for relationship checking between a new query and the previously cached queries. Each entry of the cache description describes the corresponding spatial region of a function-embedded query and contains a pointer to the corresponding cached result. We have implemented the cache description with both linear arrays and R-Trees [16] (since the cached queries have spatial semantics), and have compared their performance in our experiments. We used LRU as the cache replacement policy.

4.3 Proxy Deployment and Maintenance Issues

In summary, we have implemented three active caching schemes for the two classes of queries in our form-based proxy framework: (1) the full semantic caching; (2) a variant of the full semantic caching that checks exact match, query containment, and region containment relationships between the new query and the cached queries; and (3) the containment-based active caching that only checks query containment (including exact matches). These three active caching schemes in addition to the traditional passive query caching have different requirements on the proxy deployment and maintenance. In the following, we discuss issues on the proxy deployment and maintenance for the four caching schemes.

First, if our proxy handles only traditional passive query caching, it becomes a regular proxy and has no need for any template information. Rather, the proxy administrator specifies rules for cacheable URLs at the proxy configuration time. By default, regular proxies disable caching dynamic URLs that embed parameters or use the POST method. To enable caching web pages generated from form-based queries, the rules need to be set to allow cacheable URLs containing parameters and POST requests. The proxy maintains its cache content as disk files and communicates with any web server as a regular proxy.

Next, if our proxy handles active caching schemes in addition to the passive caching, the only difference between deploying our form-based proxy and deploying

a regular proxy is that for active caching our proxy needs to know the query semantics of the forms. This requirement is because the application at the web server can perform arbitrary computation based upon the user input. Thus, to enable active caching, we require the web site to provide the text of the SQL query corresponding to each form and the mapping information between the form parameters and query template parameters. These information are provided in the form of query templates and template information files. For function-embedded queries, we further require the web site to provide high-level semantic information about functions through function templates. We assume that these templates provide correct semantic information, as this assumption is the basis for the correctness of the query processing in the proxy.

In our experiments, we generated the template files ourselves for the web sites being proxied by analyzing the HTML source of the forms and the publicly available documentation, e.g., those at the SkyServer web site. Our effort was minimal as it involved only a couple of web sites and these web sites had simple form interfaces and sufficient documentation. Moreover, our template files require only high-level semantic information and therefore it is safe to ignore the internal processing logic of the forms at the web sites, which is unknown to us. Nevertheless, it is an interesting question whether we should automate the template generation process, partially or fully at the proxy, rather than requiring the web sites to provide them. Additionally, it is possible to identify and generate query templates if the SQL queries are visible at the proxy, as done in DBProxy [3], which intercepts SQL statements in the JDBC driver.

After the templates and template information files are generated, they are usually registered at the proxy at configuration time. In the configuration step of a regular proxy, the proxy administrator specifies in the configuration file which URLs the proxy is allowed to cache. When configuring our form-based proxy, the proxy administrator specifies which forms that the proxy may cache and adds the template and template information files to the appropriate cache directories at the proxy. At the proxy startup time, these template information are loaded into memory and are checked at runtime upon an incoming HTTP request. Furthermore, after the proxy

starts up, new templates can be added and loaded dynamically and there is no need to restart the proxy.

After the proxy is deployed, the pure containment based active caching requires no further collaboration from the original web site for query processing whereas the other two, full semantic caching and its variant that handles region containment, requires the original web site to support remainder queries. While some web sites, such as the SkyServer, provide SQL-based facility that can handle remainder queries, pure containment based caching is more practical for other real-world database-backed web sites.

Consistency is always an issue in caching. In the current form of our proxy implementation, the changes of the forms or the data at the original web site can only be detected by the proxy administrator manually, and the cache consistency maintenance is to clear all cached data from the affected templates. Nevertheless, 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 relaxed attitude toward consistency. It is possible that many form-based applications will be well served by simply providing a facility for the web site to invalidate its data and/or templates stored at a proxy. Furthermore, web sites such as the SkyServer update their data in batches off-line [41]. Consequently, the invalidation at the proxy can be done infrequently when the proxy administrator detects such batch updates at the original web site.

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 [7] 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 [39] 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 and function 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.

5 Experiments

In this section, we examine the feasibility of our form-based proxy and the performance of active caching schemes using extensive experiments.

For keyword-based queries, we first exercised the proxy caching framework using the TPC-W book title search query traces. We then used modified workloads to investigate properties of active caching not revealed by the simple TPC-W traces. Finally, we played a user trace through our proxy to an online bookseller's web site. For function-embedded queries, we only used real-word user traces extracted from the SkyServer web logs in the experiment. This is because the usefulness of synthetic workloads for performance evaluation is less definitive for this class of queries due to the application-specific nature of the embedded functions. Additionally, for both classes of queries our proxy recorded various timing information in each step of query processing for the purpose of detailed analysis.

5.1 Experimental Setup

At first, we used four computers for the TPC-W synthetic workload experiments. The four machines all had a Pentium III 800MHz CPU and 256MB memory. The machine for the database server had 20GB disk space, while the other three machines each had a 9GB disk.

Table 1: The software deployment in the experiments on synthetic TPC-W traces

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

All four machines used the Red Hat Linux 6.2 operating system. The RBE program (Remote Browser Emulator) and proxy servlet were homegrown. The RBE program ran the query traces in a batch and stored all the returned query results on disk for analysis. The servlet engine was the Apache Tomcat Servlet Engine. The database server was Oracle 8.1.6 Enterprise Edition with the InterMedia Text 8.1.6 index server. We used 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.

Next, we conducted further experiments for keyword-based queries with an online bookseller's web site. In this set of experiments, the RBE and proxy configuration remained the same whereas our own web server and database server were replaced by the real-world website.

Finally, we used a different set of hardware and software configuration for function-embedded queries using real-world SkyServer traces. The RBE machine had 256MB memory and 60GB disk space. The proxy machine had 1GB memory and 100GB disk space. Both machines had a Pentium IV 1.8GHz CPU and were running the Windows XP Professional operating system. As the setup for the two classes of queries are different due to implementation considerations, the performance results are compared within each class of queries only.

All the machines involved in our experiments (except for the real web servers) were on a 100Mbit/second Ethernet. In the following, we present in detail our experimental results for keyword-based queries and function-embedded queries respectively.

5.2 Experimental Results for Keyword-Based Queries

As the result size of a keyword-based query was usually small in our setting, in all experiments on keyword-based queries we assume an unlimited cache size, i.e., no cache replacement was triggered.

5.2.1 On TPC-W Query Traces

To measure the effects of proxy caching for keyword-based queries on response times, we set up the TPC-W databases at three scales: 10K, 100K, and 1M (in terms of the cardinality of the *item* table) in Oracle. The cardinality of the *author* table was ¼ of that of the *item* table. The ASCII data files of the two tables were 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 (HTML 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. Table 2 shows a few examples.

Table 2: The example titles in the TPC-W databases

i_id	i_title
3	Years will break BABABABABABARI pleasant, free terms--
4	Final, previous feet can want BABABABABABARE more?
5	Visitors should result. Public, BABABABABABASElikely

In this dataset each title got one "signature word" (as shown in Table 2), and each signature word was inserted into an average of five titles. The search string in a TPC-W query is a signature word. This caused each query to return an average of five books, and two queries in the trace were either identical (if they have the same search string) or had 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 contained two thousand distinct queries, and the caches reached a hit ratio of 80%. At the end of the experiment, both caches contained nearly 10K items.

Table 3: The TPC-W average response time (in milliseconds)

Database scale		10K	100K	1M
DIRECT	OVERALL	74	384	4144
PC	HIT	11	11	12
	MISS	110	442	4215
	OVERALL	31	98	853
AC0	HIT	11	13	12
	MISS	262	539	4499
	OVERALL	61	118	905

We compared timings in four cases: RBE directly to the server (DIRECT), RBE through the proxy without any cache (NC), RBE through the proxy with passive query caching (PC), and RBE through the proxy with active query caching sending no remainder predicates (AC0). 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 PC setting, we only show the other three cases in Table 3.

From Table 3, we see that the database server processing time dominated (comparing PC cache misses with the direct-to-server case) and this got worse when the scale of the database increased. Passive query caching achieved an overall average response time ¼ of that of the direct-to-server case. On a miss, passive

caching added less than 70 milliseconds of overhead when compared to the direct-to-server case. The active cache added another 100-280 milliseconds overhead on miss because of its more sophisticated query cache management. As the scale of the database increased, this overhead was dominated by the server time. As a result, the slight increase in the proxy load achieved a large gain in the overall response time.

5.2.2 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 caching performs when a new query is contained in a cached query or overlaps with some data in the cache.

Table 4: The composition of NounPhrase trace

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%

The four NounPhrase traces we experimented with were Noun40, Noun60, Noun80, and Noun100. Each trace contained two thousand queries; which could 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 4). 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 cache-contained queries were 12%, 33%, 52%, and 71%.

Figure 18 shows the average response time of the four NounPhrase traces on the 100K-scale TPC-W database running directly to the server (DIRECT), through a passive caching proxy (PC), or through an active caching proxy with no remainder

predicates (AC0). Recall that this AC0 is the case that does not require close collaboration between the web server and the proxy cache. When the number of noun queries on the fixed vocabulary increased, the ratio of exact matches did not change much and the passive caching had a limited performance, but the ratio of cache-contained queries increased and benefited active caching to a larger extent.

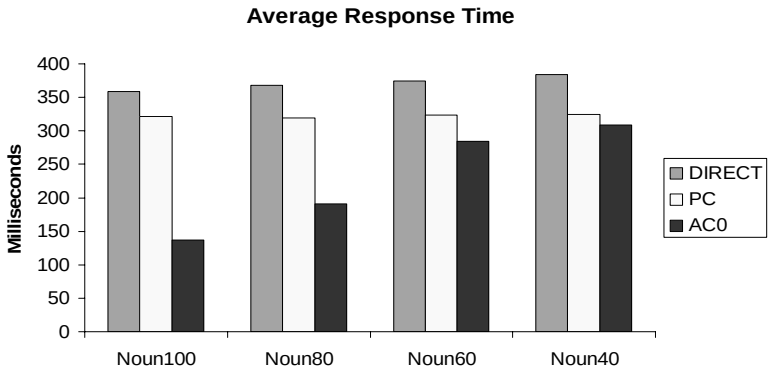

Figure 18: The average response time of NounPhrase traces

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

We compared four cases at the active cache: an exact match (EM), a query containment (QC), a query intersection (IS), and a disjoint query (DS). In this set of synthetic traces we generated, the region-containment queries had similar response times as cache-intersecting queries on average, so we only show four cases here. For the passive query cache, the cases are simply cache 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 showed their response times averaged from the three runs.

Table 5: The response time (in milliseconds) of four cases in the Noun40 trace

Query ID		515	511	510	514
AC0	Status	EM	QC	IS	DS
	Time	17	18	2683	472
PC	Status	HIT	MISS	MISS	MISS
	Time	18	361	664	376

From Table 5 we see that both caches had similar response times on an exact match query (Query 515). A cache-contained query (Query 511) also had similar response time to an exact match in the active cache, which was much better than a

miss in the passive query cache. Query 514 was a dummy query returning no answers, and an active cache miss on it was 27% more expensive than a passive cache miss. Query 510 was a 2-noun query returning 50 tuples (top 50), and an active cache intersection was three times slower than a passive cache miss. This was because in the passive query cache, only the top 50 tuples were obtained from the server, returned to the user, and saved into the cache while in the active cache case the active cache got 62 result tuples from the cache, got 510 result tuples (the whole answer set) from the server, merged these two parts of answers to eliminate duplicates, returned the top 50 to the user, and cached the un-cached answers.

We conducted further experiments on the Noun40 trace and found that increasing the number of remainder predicates had a very limited effect on limiting the number of remainder tuples (as an example, we show this for Query 510 in Table 6). This was because in the TPC-W database there is very little overlap among titles.

Table 6: Numbers of remainder tuples of Query 510

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

5.2.3 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 $\{w_0, w_1, w_2, \ldots, w_9\}$ for the title field. This data set was tailor-made to benefit remainder query processing.

In this dataset, each title field had three words: the id, w_i, and w_j, where $0 =< i, j < 9$. There were 100 distinct combinations of the (w_i, w_j) pairs, but the id field was unique so that each title was unique. We generated 1000 tuples with each combination of (w_i, w_j) appearing in 10 titles and appended these 1000 tuples to the 100K TPC-W database. We then ran the ten queries $w_0, w_1, \ldots, w_9$, and compared the performance of the 10^{th} 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 7 shows the number of remainder tuples of Query 10 and Figure 19 shows the time breakdown, averaged over three runs.

Table 7: Numbers of remainder tuples of Query 10

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

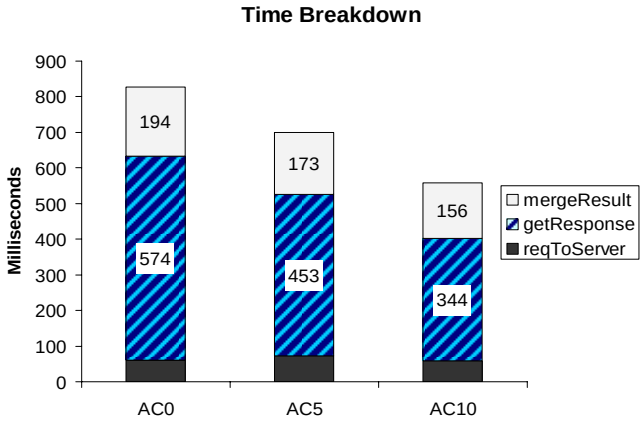

Figure 19: Time breakdown of Query 10

The legends from bottom to up in Figure 19 correspond to the portions bottom up in the bars. The time spent on checking the query relationship (checkRelationship) and probing the cache (probeCache) were very small and were negligible in comparison with other times, so they are omitted in the figure. The time spent on sending the remainder query to the server (reqToServer) was also small. The time taken waiting for the server response (getResponse) and merging the probe query results and the remainder query 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.

5.2.4 Passive Caching and Combinatorial Cache Overload

Passive caching does no duplicate elimination in the cache. This is especially bad for keyword-based queries because different combinations of keywords can all return the same data. We conducted a simple experiment to illustrate this point.

We chose 5 words that were not in the TPC-W vocabulary and inserted 50 tuples with the title field all containing these five words to the *item* table in the 100K-scale

TPC-W database. We then ran a 325-query trace that consisted of all distinct combinations of one to five of these words. At the end of the run, the passive query cache held 325 copies of these 50 tuples, or 16250 tuples, while the active query cache just kept one copy of the 50 tuples. Clearly, if there are a lot of containment patterns among queries in a trace, passive caching runs the risk of a combinatorial blow-up. More importantly, the passive cache has no cache hits, while the active cache has almost all cache hits.

5.2.5 Proxy Caching for Real Web Sites

In this section, we further explore the performance of our proxy framework for keyword-based queries with real user traces over real-world web sites. We obtained a server log from an online comparison-shopping bookstore and extracted a query trace on book titles. We call this query trace BBQ (Best Book Queries). The BBQ trace had 2416 queries on book titles, representing book title searches from users in the U.S. and Europe over a period of one week in October 2000. They all were conjunctive keyword-based queries with 1 to 18 words in each query. The median length (in words) of the queries was 3 and the average length was 3.3. The 1-word queries, 2-word queries, …, and 5-word queries made up 13%, 25%, 24%, 15%, and 11% of the trace.

Table 8: The BBQ trace on two bookstore web sites

Category	AM	BM
Ratio of requests with less than 1 page result	74%	87%
Total number of books in the 1st page	42.2K	14.3K
Total number of distinct books in the 1st page	24.4K	6.7K
Duplicate ratio of books in the 1st page	42%	53%

To investigate the properties of the query trace and its interaction with real-world data sets, we replayed the BBQ trace to two large operational online bookstore web sites in November 2000. (In this study we did not use any caching.) We call them site AM and site BM. From Table 8 we see that the query result size was usually very small in comparison with the backend database sizes. 74% and 87% of the requests

got results of less than one page. Moreover, the duplicate book ratios across requests were as high as 42% and 53%, which represents significant opportunities for duplicate elimination at the cache.

To investigate the performance of our form-based proxy framework in a real-world setting, we built a wrapper server for the BM site and replayed the BBQ trace through our proxy. Notice that there was no change for the proxy code; we just added a dummy book title search query template for the BM site to the query template directory. The wrapper server was used to wrap the results into XML before giving it back to the proxy. The setup is as shown in Figure 20.

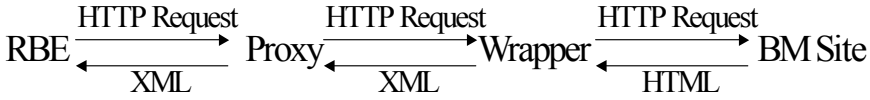

Figure 20: The setup of proxy caching for the BM site

Recall that in such a situation, where there is no collaboration between the web site and the proxy, we can do passive caching or containment-based active caching, but not the full semantic caching.

We compared the RBE response times of three cases: NC, PC, and AC0. The hit ratio of the passive cache was 30% while that of the active cache was 47% (22% exact matches plus 25% cache-contained queries). The ratio of exact matches was lower in the active cache than the passive cache because we did not cache subsumed queries. There were few cache-overlapping queries (2 out of the entire BBQ trace) so there is essentially no room for remainder predicates to improve performance. Because the response time of a query depends on the locality, the result size, and the cache status, we report average response times for several portions of the trace in Table 9. Not surprisingly, passive caching outperformed the no cache case, and active caching outperformed passive caching. More interestingly, active caching warmed up the proxy cache faster than passive caching.

Table 9: The response time of BBQ trace

Avg. time(ms)	NC	PC	AC0
Entire Trace	1827	1216	992
Query 1-1000	1827	1196	1171
Query 1001-2000	1827	1163	795
Query 1001-1500	1853	1322	923
Query 1501-2000	1540	1004	666

5.3 Experimental Results for Function-Embedded Queries

After evaluating the performance of our active caching schemes for keyword-based queries with both synthetic and real-world workloads, we continue to investigate its effects on function-embedded queries.

We conducted our experiments on caching the query results of the Radial web search form of the SkyServer. We extracted real-world query traces for this form from the SkyServer web logs. We chose this form because it is representative of function-embedded queries, and the fGetNearbyObjEq() function embedded in the query template is the building block of many other functions at the SkyServer. We varied the cache size for the query trace, and an unlimited cache size reached 500MB for caching the entire trace.

5.3.1 Analysis of the Radial Search Form Query Trace

Before using the Radial query trace for experiments, we analyzed its characteristics with respect to caching.

The query trace had a total of 11,323 queries from June 2001 to May 2003. A large portion of the queries were not directly issued from the Radial search form at the SkyServer; instead, they were issued from the Radial search form at another astronomy web site MAST [30]. The Radial search form at MAST forwards the queries it receives to its counterpart at the SkyServer, gets the result and returns it to the user. The query templates of the two Radial search forms are almost identical, except a few differences in the optional parameterized predicates. After confirming the validity of this query trace, we used them in our experiments.

To study the characteristics of the trace with respect to active caching, we ran the trace through our active caching module with the full semantic caching scheme and an unlimited cache size. Table 10 summarizes the numbers of occurrences of exact match, query containment and query overlap. It shows that nearly 51% (17% exact matches and 34% cache-contained queries) of the Radial search form queries can be completely answered by the cache if the query results are cached. Additionally, about 9% of the queries overlap. Combining with the analytical results of real-world

keyword-based query traces in Section 5.2.5, this relatively small number further convinces us that in real-world scenarios there are not much performance improvement opportunities for handling query overlap at a proxy.

Table 10: Characteristics of the Radial query trace

Total # queries	# exact match	# query containment	# query overlap
11323	1958	3829	1022

Note that since our active query cache does not store the results of subsumed function-embedded queries, the number of exact matches shown in Table 10 is fewer than that of exact matches in a passive query cache, which is 3628.

5.3.2 Proxy Caching with the Radial Query Trace

We ran the Radial query trace through our proxy in various configurations to the SkyServer. A configuration includes the caching scheme, the cache description implementation, and the cache size. The caching scheme can be no cache, a passive cache, or an active cache. An active cache can be one of the three alternatives we implemented: (1) the full semantic caching, (2) the variant of full semantic caching handling exact match, query containment, and region containment, and (3) the containment-based active caching. The cache description implementation can be an array or an R-tree.

In addition to response time, we used cache efficiency as another performance metric in the experiments for function-embedded queries. The *cache efficiency* of a query is defined as the percentage of the result tuples that are served from the proxy cache to the total number of result tuples of the query. The average cache efficiency of a query trace is the arithmetic average of the cache efficiency values of all queries. By trial experiments, we found that for function-embedded queries the cache efficiency reveals the cache utilization more accurately than a cache hit ratio.

We first examined the performance impact of active caching for function-embedded queries in comparison with a proxy without any cache (NC) and a proxy with passive caching (PC). We picked the full semantic caching as the active caching scheme since it exercises all aspects of active caching. We further compared the performance impact of the cache description implementation of the active caching

– ACR is the active caching with an R-tree cache description and ACNR is the active caching with a linear array cache description. In addition, the cache size was varied from 1/6 of the total result size of the query trace to the total result size (nearly 600MB XML files).

Figure 21 illustrates the average response times of the first 10,000 queries in the trace under various proxy configurations. Table 11 summarizes the average cache efficiencies of active caching (which was the same for ACNR and ACR) and passive caching on the trace. Apparently, the cache efficiency of active caching, 53-59%, was much higher than that of passive caching, 30%. Moreover, the increase in cache size improved cache efficiency better for active caching than for passive caching.

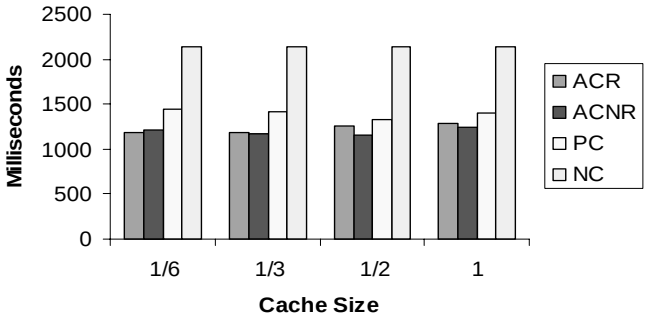

Figure 21: The average response time of four cases in the Radial trace

Table 11: The average cache efficiency of AC and PC

Cache Size	1/6	1/3	1/2	1
AC	0.531	0.565	0.582	0.593
PC	0.290	0.305	0.311	0.313

Associating Table 11 with Figure 21, we see that the general trend in response time corresponded to that in cache efficiency. Without any cache (NC), the average response time was more than 2 seconds. With a passive cache, it went down to around 1.4 seconds, resulting in 30% improvement over NC. With an active cache, it went down further to around 1.2 seconds. However, the average response time of either caching scheme did not improve much when the cache size increased. This is because response time is not only affected by the cache efficiency, but also by the cache maintenance cost. Therefore, in the remainder of this section, we show the results on the unlimited cache size; results on other cache sizes are similar.

Interestingly, Figure 21 shows that the R-tree index on the cache description did not accelerate the active caching scheme and in some cases even slowed it down

slightly. The main reason was that the size of the cache description was not large in our experiment so that a linear search and a tree search had similar main memory performance. A more detailed study on the time breakdown of individual queries revealed that the cache checking time with or without the R-tree index was always under 100 milliseconds. Finally, the maintenance of the R-tree index is more costly than that of an array.

We then picked some individual queries from the query trace to investigate their response times in ACNR and PC with an unlimited cache size. Table 12 shows the response times of five representative queries at about halfway in the trace. The cache checking status of these queries in active caching were exact match (EM), query containment (QC), region containment (RC), query intersection (IS), and disjoint (DS), respectively.

Table 12: The response time of five queries with a large result size (in milliseconds)

Query ID		4365	4057	4093	4062	4046
ACNR	Status	EM	QC	RC	IS	DS
	Time	312	531	5203	3172	6735
PC	Status	HIT	MISS	MISS	MISS	MISS
	Time	312	6718	6391	6468	6812

The five queries shown in Table 12 all had a large result size (nearly 1000 tuples). In the exact match and disjoint cases, the two caching schemes had very similar response times. In the query containment case, active caching outperformed passive caching greatly (531 milliseconds versus 6718 milliseconds). Query 4062 intersected with a cached query and almost 86% of its result was served locally from the cache in active caching. Because sending a remainder query rather than the original query greatly reduced the transmitted data volume in this case, the response time of active caching was less than half of that of passive caching. Finally, Query 4093 subsumed a cached query. In this case, the active cache served some of the result tuples from the cache and fetched the others from the server. Consequently, it outperformed the passive query cache by 20%.

Figure 22 shows the time breakdown of these five queries recorded in the proxy with the active caching. Here “checkRelationship” was the time spent on checking the cache upon a new query, “getResponse” was the time spent on receiving results from the original web site, and “handleResult” was the time spent generating the

query result as well as updating the cache, including probing the cache (probeCache) and merge the probe and remainder (if any) query results (mergeResult). In the figure, the relationship checking time was too small to be visible (at most 100 milliseconds). The time spent in sending the remainder query to server (reqToServer) was even smaller so we omit it in the figure. In comparison, the time spent for getting the result from the server dominated and the time for result handling and cache maintenance was considerable.

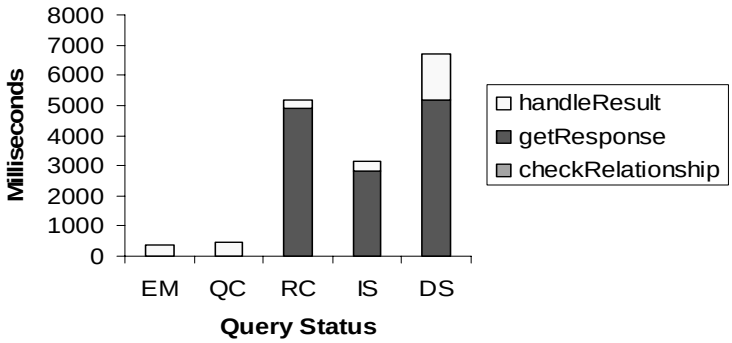

Figure 22: Time breakdown of five queries with a large result size

Table 13: Response time (milliseconds) of five queries with a small result size

Query ID		4000	4030	4034	4162	4006
ACNR	Status	EM	QC	RC	IS	DS
	Time	15	47	657	656	610
PC	Status	HIT	MISS	MISS	MISS	MISS
	Time	15	640	640	609	609

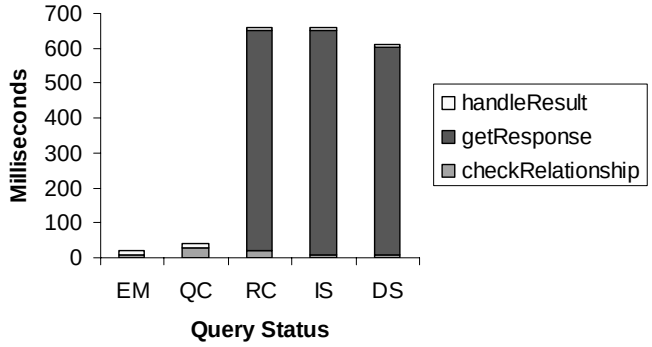

Figure 23: Time breakdown of five queries with a small result size

We also show in Table 13 the response times of five representative queries of a small result size (less than five result tuples) at about halfway in the trace. For these queries, the active caching still outperformed passive caching by over a factor of ten in the case of query containment while it performed similarly to or slightly worse than

passive caching in other cases. Figure 23 shows the time breakdown of these queries in active caching. Due to the small result size, the time for result handling was negligible.

Next, we examined the worst cases for query overlap in active caching. We have shown that when the cache efficiency is high for cache-intersecting queries with a large result size, the performance can be improved greatly. However, when the cache efficiency is low and the number of cached queries is large, cache-intersecting function-embedded queries will have a much worse performance in active caching than in passive caching. We picked four such queries towards the end of the query trace (Table 14) to demonstrate this problem.

Table 14: The worst cases of cache-intersecting queries

Query ID		9469	9472	9505	9516
ACNR	Status	IS	IS	IS	IS
	Time	12734	11156	641	641
PC	Status	MISS	MISS	MISS	MISS
	Time	10688	10625	547	532

The first two cache-intersecting queries shown in Table 14 had a large result size and the other two had a small result size. Each of them intersected with a number of cached queries, but the cache efficiency of each query was low (less than 1%). After checking the time breakdown of the queries recorded in the proxy, we found that the difference in response times between the two caching schemes was mainly resulted from the difference in the time spent on getting results from the server. This was because the handling efforts at the cache were fruitless and the complicated remainder queries increased the query processing time at the web site.

Since cache-intersecting queries may not always be helpful for active caching, we continued to compare the full semantic caching with the other two active caching alternatives that have no handling of cache-intersecting queries. In Figure 24, the "First" is full semantic aching, the "Second" is active caching without handling query overlap other than region containment, and the "Third" is pure query containment based active caching. The results shown in the figure were for the first 10,000 queries in the Radial query trace with an unlimited cache sizes and an array-based cache description. The latter two schemes had slightly worse cache efficiency than that of the first (0.544 and 0.511 respectively as opposed to 0.593), but they

outperformed the full semantic caching in response time. This highlights the usefulness of containment-based active caching, which does not require a high degree of cooperation with the web sites.

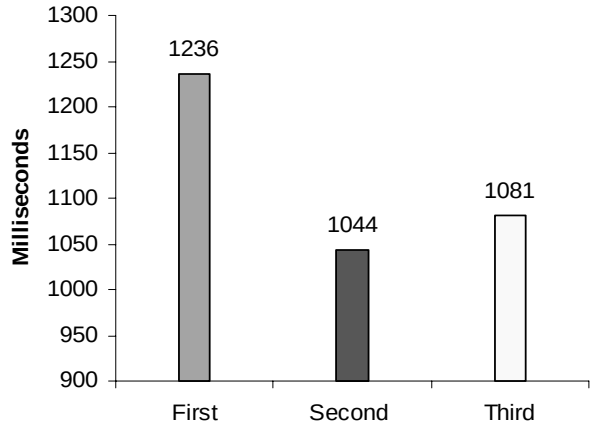

Figure 24: The average response time of three active caching schemes

5.4 Summary

The experimental results show that our form-based proxy works well with both synthetic benchmark workloads and real-world user traces and that active caching generally achieves a better response time and cache hit ratio (or cache efficiency) than passive caching. For both classes of queries, the time of cache operations is small in the total query time; rather, the server response time dominates in all cases. Consequently, the slight increase on the proxy workload in the caching schemes generates a large gain in the overall system response time. Nevertheless, the usefulness of handling cache-intersecting queries depends on the cache efficiency of such queries. In general, pure query containment based active caching is sufficient for performance improvement in addition to its advantage of low requirement on server collaboration. Additionally, active caching schemes, especially those sending no remainder queries, reduce server workload, which is highly desirable when the database server is under a heavy workload and becomes the bottleneck.

6 Related Work

There has been a large body of work in the literature on data caching and query caching. Some of them [10][12][19][36] dealt with relational queries while others

[1][9][22] focused on caching for heterogeneous sources. Semantic caching [10] and predicate caching [19] were initially proposed in traditional client-server database architectures. Lee and Chu [22] focused on algorithms for choosing the best matching query in the context of semantic caching for range queries. While Chidlovskii et al. [9] studied semantic caching for keyword-based queries over meta-searchers, we focus on using templates to enable active caching for database-backed web sites.

Query templates have previously proven useful in other contexts, e.g. in information integration systems [34] and declarative specification of data-intensive web sites [5][14]. Amiri et al. [3] proposed query containment checking algorithms for general predicate-based queries based on query templates. Their DBProxy processes a full SQL processing capability locally. The OLAP view caching [24] is a simulation study in the context of an enterprise LAN to minimize the cost of OLAP processing. It assumes the proxy has a query processing capability for OLAP aggregations. In comparison, we designed and implemented various active caching schemes with simple but efficient query processing logic in our form-based proxy for both keyword-based queries and function-embedded queries using query and function templates.

Caching and materialization for databases on the web has received a lot of attention recently [6][8][20][43]. These studies all consider passive caching of the HTML or XML pages generated from DBMS-resident data. In contrast, our major focus is active proxy caching at the query level.

There have been many publications in web caching that are closely related to our work [11][29][39]. All of these studies did not consider database queries. Our previous work [26] 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 or function-embedded queries over the web.

User-defined functions have been widely supported in commercial DBMS products, such as IBM DB2 and Microsoft SQL Server. Hellerstein and Naughton

proposed a query execution technique called Hybrid Cache [18] for caching the results of expensive methods in a full-fledge DBMS. The MOCHA system [35] implemented a migration paradigm for shipping application-specific Java code around distributed data sources. In comparison, our work focuses on caching for table-valued functions in a web proxy, as opposed to migration or execution of scalar functions.

Our work is also related to answering queries using views [15][17][21][23][33][40]. Although table-valued functions can be regarded as a kind of parameterized views, they usually have non-SQL application-specific semantics (as an evidence, they are often implemented in programming languages other than SQL). Consequently, known view selection and view matching algorithms are not directly applicable to the active caching of table-valued functions. Moreover, our proxy has only a limited query processing capability such that we chose to answer new function-embedded queries fully at the proxy based on query-containment checking.

Recently, there is an increasing commercial interest in caching for database web servers, for example, the Oracle 9i Application Server [32] and the IBM DBcache project [25]. The Oracle 9i Application Server includes the Oracle Database Cache and the Oracle Web Cache. The Oracle Web Cache does passive caching. The Oracle Database Cache and the IBM DBCache mainly cache full tables; and features such as caching selected rows, columns and 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 an SQL processor can answer arbitrary queries on those tables. In general, using full-fledged databases for the cache provides powerful query processing capabilities but requires more schema information and administrative effort, which seems to be overkill in our proxy caching scenario.

An alternative to our lightweight proxy-caching framework approach is to develop customized caching solutions at the home web sites. This alternative has no collaboration issues and is easy to control. However, it is unable to take advantage of the widely deployed proxy servers on the Internet to save the wide-area network latency. If this alternative is pushed to the network edges, it seems to be cost-inefficient as it only works for the home web site.

The form-based proxy caching framework presented in this paper is an integration and extension of our previous work on form-based active proxy caching for keyword-based queries [27] and function-embedded queries [28]. In comparison with our previous work, we have added real-world query trace evaluation for keyword-based queries, and have significantly extended our active caching techniques for function-embedded queries and conducted more extensive performance evaluation for them.

7 Conclusion

We have proposed a form-based proxy caching framework for database-backed web sites with two common classes of web queries, which are keyword-based queries and function-embedded queries. Our form-based proxy operates around query and function templates that describe high-level query semantics, and query template information files that describe the parameter mapping between the HTML forms and the templates. We study the traditional passive query caching and various active caching schemes for both classes of queries using a full system implementation and evaluation. We show that while passive caching is sufficient for some synthetic benchmark workloads, active caching is more promising for other generated traces and real-world workloads. More specifically, answering cache-contained queries results in a significant performance gain, but answering cache-intersecting queries is probably not worthwhile for the top-n conjunctive keyword-based queries and queries calling table-valued functions. Finally, different caching schemes rely on different degrees of collaboration from servers. Passive query caching does not need query semantics information from the server whereas the contain-based active caching needs; full semantic caching further needs the server to handle remainder queries.

There are several lines of future work that we intend to consider. One challenge is to use our framework to study other classes of web queries. Another is to do further empirical studies over real traces and web sites. Still another challenge to tackle is consistency. The Web, as it currently exists, works with relaxed consistency in its proxy caches; we intend to explore how this can be used in conjunction with active caching for database-backed web sites.

Acknowledgements. We would like to thank Wenfeng Cai for providing the online bookstore query trace. We would also like to thank Jim Gray for making the personal SkyServer code and data available and answering our questions about the SkyServer. Funding for this work was provided in part by HKUST6158/03E from the Hong Kong Research Grants Council, and NSF Awards CDA-9623632 and ITR 00860.

References

1. Sibel Adali, K. Selçuk Candan, Yannis Papakonstantinou, V. S. Subrahmanian (1996) Query caching and optimization in distributed mediator systems. In: Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Canada, 4-6 June 1996, pp 137-148
2. Mehmet Altinel, Michael J. Franklin (2000) Efficient filtering of XML documents for selective dissemination of information. In: Proceedings of the 26th International Conference on Very Large Data Bases (VLDB), Cairo, Egypt, 10-14 September 2000, pp 53-64
3. Khalil Amiri, Sanghyun Park, Renu Tewari, Sriram Padmanabhan (2003) Scalable template-based query containment checking for web semantic caches. In: Proceedings of the 19th International Conference on Data Engineering (ICDE), Bangalore, India, 5-8 March 2003, pp 493-504
4. Apache Tomcat Servlet Engine. http://jakarta.apache.org/tomcat/index.html
5. Paolo Atzeni, Giansalvatore Mecca, Paolo Merialdo (1997) To weave the web. In: Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB), Athens, Greece, 25-29 August 1997, pp 206-215
6. K. Selçuk Candan, Wen-Syan Li, Qiong Luo, Wang-Pin Hsiung, Divyakant Agrawal (2001) Enabling dynamic content caching for database-driven web sites. In: Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data, Santa Barbara, CA, USA, 21-24 May 2001, pp 532-543
7. Pei Cao, Jin Zhang, Kevin Beach (1998) Active cache: Caching dynamic contents on the web. In: Proceedings of IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98),The Lake District, England, 15-18 September 1998, pp 373-388
8. Jim Challenger, Arun Iyengar, Paul Dantzig (1999) A scalable system for consistently caching dynamic web data. In: Proceedings of IEEE INFOCOM '99, The Conference on Computer Communications, 18th Annual

Joint Conference of the IEEE Computer and Communications Societies, New York, NY, USA, 21-25 March 1999, Vol. 1, pp 294-303

9. Boris Chidlovskii, Claudia Roncancio, Marie-Luise Schneider (1999) Semantic cache mechanism for heterogeneous web querying. In: Computer Networks 31(11-16): 1347-1360
10. Shaul Dar, Michael J. Franklin, Björn Þór Jónsson, Divesh Srivastava, Michael Tan (1996) Semantic data caching and replacement. In: Proceedings of 22nd International Conference on Very Large Data Bases (VLDB), Mumbai (Bombay), India, 3-6 September 1996, pp 330-341
11. Anindya Datta, Kaushik Dutta, Helen Thomas, Debra Vandermeer, Suresha, Krithi Ramamritham (2002) Proxy-based acceleration of dynamically generated content on the world wide web: An Approach and Implementation. In: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, WI, USA, 3-6 June 2002, pp 97-108
12. Prasad Deshpande, Karthikeyan Ramasamy, Amit Shukla, Jeffrey F. Naughton (1998) Caching multidimensional queries using chunks. In: Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data, Seattle, WI, USA, 2-4 June 1998, pp 259-270
13. Excite Search Engine. http://www.excite.com
14. Mary F. Fernandez, Daniela Florescu, Jaewoo Kang, Alon Y. Levy, Dan Suciu (1998) Catching the boat with strudel: Experiences with a web-site management system. In: Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data, Seattle, WI, USA, 2-4 June 1998, pp 414-425
15. Jonathan Goldstein, Per-Åke Larson (2001) Optimizing queries using materialized views: A practical, scalable solution. In: Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data, Santa Barbara, CA, USA, 21-24 May 2001, pp 331-342
16. Antonin Guttman (1984) R-trees: A dynamic index structure for spatial searching. In: Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data, Boston, MA, USA, 18-21 June 1984, pp 47-57
17. Alon Y. Halevy (2000) Theory of answering queries using views. SIGMOD Record 29(4): 40-47
18. Joseph M. Hellerstein, Jeffrey F. Naughton (1996) Query execution techniques for caching expensive methods. In: Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Canada, 4-6 June 1996, pp 423-434
19. Arthur M. Keller, Julie Basu (1996) A predicate-based caching scheme for client-server database architectures. In: VLDB Journal 5(1): 35-47
20. Alexandros Labrinidis, Nick Roussopoulos (2000) WebView materialization. In: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, Dallas, TX, USA, 16-18 May 2000, pp 367-378
21. Per-Åke Larson, H. Z. Yang (1985) Computing queries from derived relations. In: Proceedings of the 11th International Conference on Very Large Data Bases (VLDB), Stockholm, Sweden, 21-23 August 1985, pp 259-269
22. Dongwon Lee, Wesley W. Chu (1999) Semantic caching via query matching for web sources. In: Proceedings of the 8th International Conference on Information and Knowledge Management (CIKM), Kansas City, MO, USA, 2-6 November 1999, pp 77-85

23. Alon Y. Levy, Alberto O. Mendelzon, Yehoshua Sagiv, Divesh Srivastava (1995) Answering queries using views. In: Proceedings of the 14th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS), San Jose, CA, USA, 22-25 May 1995, pp 95-104
24. Thanasis Loukopoulos, Panos Kalnis, Ishfaq Ahmad, Dimitris Papadias (2001) Active caching of on-line-analytical-processing queries in WWW proxies. In: Proceedings of the 2001 International Conference on Parallel Processing (ICPP), Valencia, Spain, 3-7 September 2001, pp 419-426
25. Qiong Luo, Sailesh Krishnamurthy, C. Mohan, Hamid Pirahesh, Honguk Woo, Bruce G. Lindsay, Jeffrey F. Naughton (2002) Middle-tier database caching for e-business. In: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, WI, USA, 3-6 June 2002, pp 600-611
26. Qiong Luo, Jeffrey F. Naughton, Rajasekar Krishnamurthy, Pei Cao, Yunrui Li (2000) Active query caching for database web servers. In: Proceedings of the 3rd International Workshop on Web and Databases (WebDB), Dallas, TX, USA, 18-19 May 2000, pp 92-104
27. Qiong Luo, Jeffery F. Naughton (2001) Form-based proxy caching for database-backed web sites. In: Proceedings of the 27th International Conference on Very Large Data Bases (VLDB), Roma, Italy, 11-14 September 2001, pp 191-200
28. Qiong Luo, Wenwei Xue (2004) Template-based proxy caching for table-valued functions. In: Proceedings of 9th International Conference on Database Systems for Advances Applications (DASFAA), Jeju Island, Korea, 17-19 March 2004, pp 339-351
29. Evangelos P. Markatos (2000) On caching search engine query results. In: Proceedings of the 5th International Web Caching and Content Delivery Workshop, Lisbon, Portugal, 22-24 May 2000
30. MAST SDSS Query Interface. http://archive.stsci.edu/cgi-bin/sdss/catalog.
31. Multidimensional Geometry from MathWorld. http://mathworld.wolfram.com/topics/MultidimensionalGeometry.html
32. Oracle Application Server. http://www.oracle.com/appserver/
33. Rachel Pottinger, Alon Halevy (2000) A scalable algorithm for answering queries using views. In: Proceedings of the 26th International Conference on Very Large Data Bases (VLDB), Cairo, Egypt, 10-14 September 2000, pp 484-495
34. Anand Rajaraman, Yeshoshua Sagiv, Jeffrey D. Ullman (1995) Answering queries using templates with binding patterns. In: Proceedings of the 14th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS), San Jose, CA, USA, 22-25 May 1995, pp 105-112
35. Manuel Rodriguez-Martinez, Nick Roussopoulos (2000) MOCHA: A self-extensible database middleware system for distributed data sources. In: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, Dallas, TX, USA, 16-18 May 2000, pp 213-224
36. Timos K. Sellis (1988) Intelligent caching and indexing techniques for relational database systems. In: Information Systems 13(2): 175-185
37. Craig Silverstein, Monika Henzinger, Hannes Marais, Michael Moicz (1998) Analysis of a very large AltaVista query log. In: Compaq SRC Technical Note 1998-014, October 1998.
38. SkyServer. http://skyserver.sdss.org/
39. Ben Smith, Anurag Acharya, Tao Yang, Huican Zhu (1999) Exploiting result equivalence in caching dynamic web content. In: Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems, Boulder, CO, USA, 11-14 October 1999, pp 209-220

40. Divesh Srivastava, Shaul Dar, H. V. Jagadish, Alon Y. Levy (1996) Answering queries with aggregations using views. In: Proceedings of 22th International Conference on Very Large Data Bases (VLDB), Mumbai (Bombay), India, 3-6 September 1996, pp 318-329
41. Alexander S. Szalay, Jim Gray, Ani R. Thakar, Peter Z. Kunszt, Tanu Malik, Jordan Raddick, Christopher Stoughton, Jan vandenBerg (2002) The SDSS skyserver – Public access to the sloan digital sky server data. In: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, WI, USA, 3-6 June 2002, pp 570-581
42. Transaction Processing Performance Council (TPC). TPC Benchmark™ W (Web Commerce) Specification Version 1.1., 27 June 2000
43. Khaled Yagoub, Daniela Florescu, Valérie Issarny, Patrick Valduriez (2000) Building and customizing data-intensive web sites using weave. In: Proceedings of the 26th International Conference on Very Large Data Bases (VLDB), Cairo, Egypt, 10-14 September 2000, pp 607-610