Template-Based Runtime Invalidation for Database-Generated Web Contents

Chun Yi Choi Qiong Luo
Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Kowloon, Hong Kong
{stefan, luo}@cs.ust.hk

Abstract

We propose a template-based runtime invalidation approach for maintaining cache consistency in database-generated web contents. In our approach, the invalidator sits between a web cache and a database server and intercepts the query statements as well as the update statements transparently. Moreover, it maintains templates for queries and updates, as well as a mapping between URLs and SQL queries. At runtime, the invalidator checks an update statement against the query statements, whose corresponding HTML fragments have been cached, and decides on if any cached HTML fragments should be invalidated based on an extended satisfiability testing algorithm without sending any polling queries to the backend database. We further improve the efficiency of this checking process by utilizing the semantic information of the templates. We have integrated our invalidator with the Oracle Web Cache and have conducted extensive experiments using the TPC-W benchmark. Our results show that this approach efficiently maintains the consistency of cached HTML fragments with the backend database.

1. Introduction

Large e-commerce sites typically serve many users concurrently with web contents dynamically generated from a backend database. Caching these web contents has been the main solution to scalability and performance problems faced by the e-commerce sites. However, these cached web contents may become obsolete within a short period of time, because their corresponding database contents are constantly changing due to ongoing transactions. Since users usually desire to see up-to-date web contents in their browsing and shopping activities, it is crucial to maintain consistency between the database contents and the cached web contents.

Despite previous research efforts [7, 8, 10], cache consistency remains a challenging problem for database-backed web sites. A major cause is that the sites require several pieces of complicated software – the web server (with a web cache), the application server, the database server, and server-side applications. Moreover, these components speak different languages and run independently from one another. In this paper, we take a holistic approach to address the problem, aiming at making our approach generally applicable to a wide range of applications. Our goal in this work is to invalidate outdated database-generated web contents automatically without putting any extra workload onto the backend database. Figure 1 shows our invalidator in a database-backed web site.

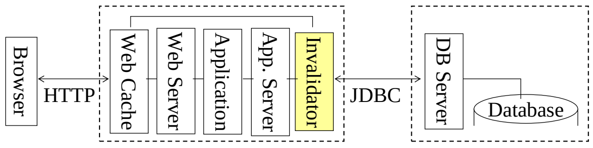


Figure 1. The invalidator in a database-backed web site

Our key observation in this work is that both SQL queries and web pages generated from database-backed web sites have templates. Specifically, server-side applications such as Java servlets, Java Server Pages, Active Server Pages, and Enterprise Java Beans are programmed to contain parameterized SQL statements (both queries and updates) as well as parameterized HTML fragments. Moreover, these parameterized statements and fragments remain visible at application development time, deployment time, or even runtime. Consequently, it is possible to know a priori the expected templates as well as the mapping between the HTML fragments and the SQL statements in an application.

The templates and mapping information reveal the SQL semantics of database-generated HTML fragments, which enables us to connect consistency maintenance of the cached web content with database operations. Subsequently, we need to know the database operations at runtime in order to perform the consistency maintenance of the web contents. Fortunately, the parameterized SQL statements are instantiated with user input or environmental variable values at runtime, and are sent to the database server through the ODBC or JDBC interface. Correspondingly, we chose to intercept SQL statements at the JDBC interface level at runtime in order to perform cache consistency maintenance transparently.

Given an instantiated SQL update and cached query statements at runtime, we have two options for cache consistency maintenance. One is invalidation and the other is update propagation. Update propagation is a more powerful choice in that it refreshes a cached item with new content. However, it requires much more computing and communications than does invalidation – the database server has to re-compute the query results and send them to the applications, while the applications have to regenerate the HTML fragments and update the cache. Therefore, we chose invalidation as our consistency maintenance approach, under which outdated HTML fragments are simply purged from the cache.

To check if a cached HTML fragment (query result) becomes invalid due to an update statement, we have further choices on if we send polling queries to the backend database to confirm the validity of the cached fragment. Recent research [7] has indicated that there is a tradeoff between the degree of over-invalidation and the overhead of polling queries. In this work, we take an approach of invalidating cached fragments based on a satisfiability test of the statement texts only. This eliminates any polling queries to the backend database as well as greatly simplifies the processing in the invalidator. In practice, we find that HTML fragments (for example, product details) are usually generated with key attributes (e.g., the product ID) in the queries and that instantiated update statements often come with key attributes in the condition. Over-invalidation is highly unlikely in such cases.

In order to improve the efficiency of invalidation, we further exploit the use of templates. Specifically, we design a satisfiability matrix with pairs of query templates and update templates to maintain the relationship between the SQL templates. We then organize instantiated queries and updates by their templates, and perform further satisfiability tests if the satisfiability is not yet determined by the matrix. Additionally, we build satisfiability indexes on important attributes referenced in the SQL statements for each template. Finally, we translate an instantiated SQL query to be invalidated into a URL based on the mapping between query templates and HTML fragment templates, and invalidate the HTML fragments identified by that URL.

In addition to designing and implementing our template-based invalidator (TBI), we have integrated it with the Oracle Web Cache [18] that has an Edge Side Includes processor [19]. Furthermore, we evaluated this framework using a Java implementation of the TPC-W benchmark [21]. Our results show that our invalidator enables the web cache to serve fresh HTML pages efficiently even when the update workload is high.

The remainder of this paper is organized as follows. We introduce the background of our work in Section 2 and describe the system architecture of our invalidator in Section 3. We present our template-based invalidation algorithms in Section 4 and our experimental results in Section 5. We discuss related work in Section 6 and draw conclusions in Section 7.

2. Background

This section introduces the theoretical background of our invalidation algorithm as well as the implementation background of our system.

2.1. Theoretical Background

Our invalidation algorithm is based on the satisfiability testing algorithm for conjunctive Boolean expressions [13] by Larson and Yang, and on the results on irrelevant update detection [5]. The following section gives a brief overview of this background and our contributions are further discussed in Section 4.

2.1.1. Satisfiability Testing

The satisfiability testing algorithm for conjunctive Boolean expressions (referred to as CONJUNCTIVE by Larson and Yang [13]) checks if a conjunctive Boolean expression is ***satisfiable*** (i.e., if it is evaluated to be true for some value assignment of its variables). The input conjunctive Boolean expression is a conjunction of multiple atomic conditions, each of which takes the form of *attribute op constant* or *attribute op attribute*. The attributes (or variables) are of an integer data type, and each has a lower bound and an upper bound. The operator is one of the five arithmetic comparison operators (>, <, >=, <=, and =).

The CONJUNCTIVE algorithm creates a graph with its nodes being the bounded variables and its edges being the arithmetic comparison relationships between the variables. It manipulates the graph (adding or removing edges and nodes, and changing the bounds of the variables) according to the current bounds of the variables until the graph becomes empty. When the algorithm halts, it returns a true/false answer on the satisfiability of the expression, as well as the final permissible range of each variable.

2.1.2. Detecting Irrelevant Updates

By utilizing the satisfiability test of the CONJUNCTIVE algorithm, the irrelevant update detection algorithms [5] give necessary and sufficient conditions for a UDI (Update/Delete/Insert) operation to be ***irrelevant*** to a query (i.e., the UDI operation on any database does not change the result of the query on that database). The input query (referred to as a ***derived relation*** [5]) is assumed to be a simple PSJ (Projection-Selection-Join) query with no self-join or subquery. The highlight of these results (one for each of the three types of UDI operations) is that ***the update irrelevance is equivalent to the unsatisfiability of the conjunction of the UDI condition and the query condition***. Therefore, we can detect irrelevant UDIs of a query by only checking the statement text, not any of the database content.

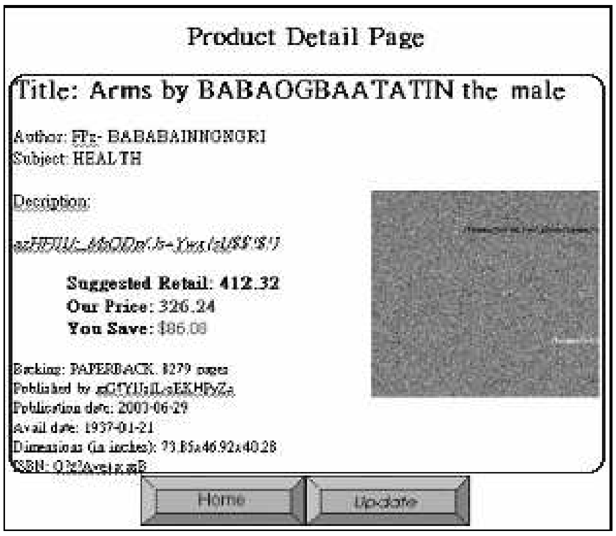
2.2. Implementation Background

2.2.1. Edge Side Includes

ESI (Edge Side Includes) [19] is a simple markup language proposed by Akamai (a major player in the content delivery network business) and Oracle. Both companies have implemented processors [1][18] for the language. This language is used in web pages to define ESI templates as well as ESI page fragments. It can define different caching properties (such as the timeout value) at the page fragment level. Therefore, ESI enables web caching servers to cache individual HTML page fragments and to assemble dynamically the fragments at the edges of networks. As a result, ESI is widely supported by the content delivery network business as well as web content providers.

Let us use a simplified product detail web page as an example to illustrate ESI. As shown in Figures 2 and 3, an ***ESI Template*** defines the static skeleton of a web page. The ESI tags (such as "*<esi: include>*") inside the template define ***ESI fragments*** and direct the assembly and delivery of the fragments. An ESI fragment can be static or dynamic. Static fragments are text or image files, which seldom change over time. Dynamic fragments are contents generated by programs with parameters instantiated at runtime. For instance, the product detail fragment in Figure 2 is generated by *TPCW_product_detail_servlet* and is included using the *<esi:include>* tag in the ESI template. This indicates that the product detail fragment can be cached separately from the template and other fragments (such as the buttons) of the template.

In addition to defining templates and fragments for web pages, ESI supports the use of variables for HTTP request attributes. In particular, the *<esi:vars>* tag indicates that the values of the HTTP request attributes will be extracted from environmental variables and embedded into the URLs at runtime. Consider the home button in the product detail page. Traditionally, the button's embedded URL (*TPCW_home_interaction.html*) has to be generated dynamically to encode the customer ID and shopping ID. With the *<esi:vars>* tag, the value of the *C_ID* parameter in the URL will be replaced by that of the *$(QUERY_STRING_ENCODED{C_ID})* environmental variable. As a result, the product detail template can be reused across users and shopping sessions as opposed to being generated for each user and shopping session.


Product Detail Page

Title: Arms by BABAOGBAATATIN the male

Author: FFz- BABABAINNGNGRI
Subject: HEALTH

Description:

azHFJUJ:_MzODnf.b+Ywx/aU$$'$'J

Suggested Retail: 412.32
Our Price: 326.24
You Save: $86.08

Backing: PAPERBACK. 8279 pages
Published by zGfYIUjLzEKHPyZa
Publication date: 2001-06-29
Avail date: 1937-01-21
Dimensions (in inches): 73.85x46.92x40.28
ISBN: Q?i?Avei.z.mB

Figure 2: The product detail web page

```
<HTML>
<H1>TPC-W Product Detail Page</H1>
<esi:include src="TPCW_product_detail_servlet?
    I_ID=$(QUERY_STRING_ENCODED{I_ID})&
S_ID=$(QUERY_STRING_ENCODED {S_ID})" />
<esi:vars>
<A HREF="TPCW_home_interaction.html?
C_ID=$(QUERY_STRING_ENCODED{C_ID})&
S_ID=$(QUERY_STRING_ENCODED{S_ID})">
<IMG SRC="home_B.gif"></A>
<A HREF = "TPCW_admin_request_servlet.html?
C_ID=$(QUERY_STRING_ENCODED{C_ID})&
S_ID=$(QUERY_STRING_ENCODED{S_ID})">
<IMG SRC="update_B.gif"></A>
</esi:vars>
</HTML>
```

Figure 3: ESI template of the product detail web page

2.2.2. Oracle Web Cache

The Oracle9iAS Web Cache (OWC) [18] performs traditional web caching as well as incorporates an ESI processor to enable ESI templates and fragments caching. OWC uses cache rules to direct caching of ESI templates and fragments. A simplified example of OWC cache rules is shown in Table 1. Since only ESI templates and static HTML files have the .html suffix in our example, the first rule directs OWC to cache all ESI templates and static html files while the subsequent rules direct OWC to cache several dynamic ESI fragments generated by servlets.

Table 1. Examples of simplified OWC cache rules

URL Expression	**Cache?**
.html	Cache
TPCW_product_detail_servlet	Cache
TPCW_execute_search	Cache
TPCW_new_products_servlet	Cache

Moreover, OWC can be configured to ignore specific parameters in a URL. Consider the following two URLs:

TPCW_product_detail_servlet.html?C_ID=123&I_ID=12

TPCW_product_detail_servlet.html?C_ID=124&I_ID=12

These two URLs differ only in their customer ID parameter values. Since the customer ID parameter does not affect the content of this template, we can configure OWC to ignore the *C_ID* parameter in the URL. Therefore, the two URLs correspond to the same cached template.

Finally, OWC purges outdated cache content by either examining the timeout value of a cached unit or serving invalidation requests to the cached unit.

3. System Architecture

We give an overview of the system architecture and the key components in this section.

3.1. System Overview

Figure 4 shows the components of our Template-based Invalidator (TBI).

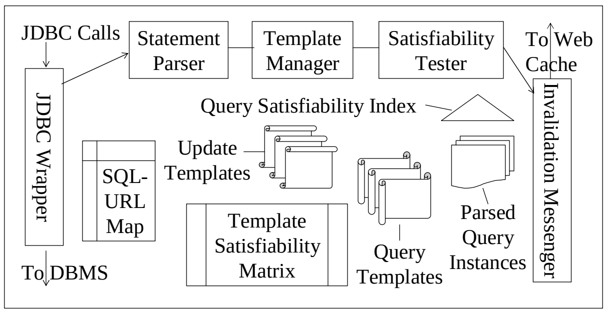


Figure 4. Components of TBI

As shown in Figure 4, the major components of TBI are the *JDBC wrapper*, the *statement parser*, the *template manager*, the *satisifiability tester*, and the *invalidation messenger*. The key data structures in TBI include a *SQL-URL map*, *parsed query instances*, *query templates*, *update templates*, a *template satisfiability matrix*, and *query satisfiability indexes*. We describe these components together with the key data structures in more detail in the following subsections.

3.2. JDBC Wrapper

The interface of TBI to a web application is a wrapper class of a JDBC driver. Vendor-specific JDBC drivers have native implementations for data access to their database servers, but they all conform to the publicly available *java.sql* package interface. Consequently, one implementation of any class defined by the package can be replaced with another implementation without changing the application code. Therefore, we implemented our wrapper to conform to the *java.sql* package interface.

When the web application registers this wrapper package to be its JDBC driver, the application's JDBC calls to the backend database are transparently intercepted by the TBI. After other TBI components check the JDBC calls, the JDBC wrapper sends the JDBC calls to the actual JDBC driver of the backend database server. No modifications of the application code are needed. Note that we chose the JDBC platform for its openness, popularity in the industry and availability of its API documentation. In essence, our technique can also be applied to a non-JDBC compliant platform with additional engineering efforts.

3.3. Statement Parser

A JDBC call is passed to the statement parser in the TBI before it is forwarded to the backend database. If the JDBC call is to prepare a SQL statement, the statement parser will parse the to-be-prepared SQL statement (with or without question marks). If the to-be-prepared statement is a query, it is parsed into a query tree and the WHERE clause is transformed into the Disjunctive Normal Form (DNF). The parser handles Projection-Selection-Join queries with optional Top-N, Order by, and Group by clauses. Subqueries and self-joins are handled as well. If the to-be-prepared statement is a UDI operation, the parser parses it into a UDI tree with update conditions and data (e.g., the tuple to be inserted). The statement parser also handles other JDBC calls such as setting the parameter values in a prepared statement and executing a prepared statement.

3.4. Template Manager

The statement parser passes SQL statement information to the template manager. The template manager maintains the key data structures of TBI.

(1) It maintains the SQL query templates, the SQL UDI templates, and a template satisfiability matrix between the two kinds of SQL templates. ***SQL query templates*** are parameterized SQL queries and ***SQL UDI templates*** are parameterized UDI statements. Together, the two types of templates are referred to as ***SQL templates***. Figures 5 and 6 show examples of SQL templates.

```
SELECT I_ID, I_TITLE FROM ITEM, AUTHOR
WHERE I_A_ID = A_ID AND I_ID = ?
```

Figure 5. The product detail query template

```
UPDATE ITEM SET I_RELATED1 = ? WHERE I_ID = ?
```

Figure 6. An example of a UDI template

The template satisfiability matrix records the satisfiability relationship between a pair of a query template and an update template. We describe the construction and maintenance of the matrix in detail in Section 4.

(2) It maintains a URL-SQL map, which records the mapping between the URL templates of the OWC-cached web contents and the SQL templates that generate the corresponding web contents. ***URL templates*** are URLs with or without uninstantiated parameters. For instance, the URL template "*TPCW_product_detail_servlet?*" is mapped to the SQL template shown in Figure 5. In addition, the map records that the I_ID parameter in the URL template will correspond to the I_ID parameter in the SQL query template.

(3) It maintains the query satisfiability indexes and parsed query instances. The query satisfiability index keeps the values of some attributes referred to in instantiated SQL queries of a query template. The goal is that, for an update operation, TBI can quickly locate the specific instantiated SQL queries of a query template that need to be invalidated.

3.5. Satisfiability Tester

Given an instantiated query statement and an instantiated UDI statement, the satisfiability tester checks if the conjunction of the conditions in the pair of statements is satisfiable. The tester utilizes the template satisfiability matrix as well as the query satisfiability indexes to speed up the process. Details are discussed in Section 4.

3.6. Invalidation Messenger

If the satisfiability tester decides that a cached query should be invalidated due to a UDI operation, it will tell the invalidation messenger to send an invalidation message to the web cache.

The invalidation messenger is given the query template together with the parameter-value pairs. It constructs the URL to be invalidated by looking up the URL-SQL map. Consider the product detail query template in Figure 5 with an I_ID value of 123. The messenger looks up the URL-SQL map and finds the corresponding URL template "*TPCW_product_detail_servlet?*" and the parameter matching rules. It then constructs an HTTP request to invalidate the following URL:

TPCW_product_detail_servlet?I_ID=123

4. Template-Based Invalidation

In this section, we describe our template-based invalidation algorithm.

4.1. Definitions

$T = \{t_1, t_2, \ldots, t_n\}$ is a set of tables.

$A = \{a_1, a_2, \ldots, a_m\}$ is a set of attributes.

QT denotes a query template. UT denotes a UDI template. An instantiated query statement of QT is denoted as q. An instantiated UDI statement of UT is denoted as u.

C is a Boolean expression over attributes $a_1, a_2, \ldots, a_p$ corresponding to the query condition in QT or the UDI condition in UT.

A Boolean expression is ***valid*** if it is always evaluated to be true. A Boolean expression is ***unsatisfiable*** if it is never evaluated to be true. A Boolean expression is ***satisfiable*** if it is evaluated to be true for some values of its attributes.

A Boolean expression, C, can be converted into a disjunctive normal form, $C = B_1 \vee B_2 \vee B_3 \ldots \vee B_x$, where B_k is a conjunction of atomic conditions, $B_k = AC_1 \wedge AC_2 \wedge AC_3 \ldots \wedge AC_y$.

An ***atomic condition*** can take one of the following four forms:

1. a op_C c, where a is an attribute, c is a constant value, and $op_C \in \{<, =, >, \geq, \leq, \text{LIKE}\}$.
2. a_1 op_N a_2, where a_1 and a_2 are attributes and $op_N \in \{<, =, >, \geq, \leq\}$.
3. a op_N sq, where sq is a subquery.
4. a op_N bv, where bv is an uninstantiated variable.

4.2. Class of Queries Handled

TBI handles selection-projection-join queries with optional order-by and group-by clauses. Subqueries and self-joins in these queries are also handled. Figure 7 shows a generic form of the class of queries handled.

```
SELECT TOP n          A(S(QT))
FROM                  T(QT)
WHERE                 C(QT)
ORDER BY              A(O(QT))
GROUP BY              A(G(QT))
```

Figure 7. The class of queries handled

In this generic form of a query template QT, A(S(QT)) denotes the attributes in the select clause. T(QT) denotes the tables involved in this query template. C(QT) denotes the where condition (note that it may contain subqueries). We will use A(C(QT)) to denote the attributes in the where condition. A(O(QT)) denotes the attributes in the order-by clause. A(G(QT)) denotes the attributes in the group-by clause.

Finally, we define $A(QT) = A(S(QT)) \cup A(C(QT)) \cup A(O(QT)) \cup A(G(QT))$, which denotes all attributes involved in QT. If C(QT) has any subquery, sq, A(C(QT)) includes A(sq) as well.

4.3. Class of UDIs Handled

Given a UDI template, UT, S(UT) denotes the clause containing the modification action, and A(S(UT)) denotes the attributes that a UDI modifies. S(UT) varies by the type (insertion, deletion, or update) of UDI. S(UT) of an insert statement explicitly denotes the list of all the attributes in the insertion table and the corresponding insertion values. S(UT) of a delete statement implicitly denotes all the attributes in the deletion table. For an update statement, S(UT) denotes the SET clause.

TBI handles the following class of UDIs.

(1) INS(T(INS), C(INS)):

INSERT INTO T(INS) A(T(INS)) VALUES(? ? … ?)

INS(T(INS), C(INS)) denotes an insertion into an insertion table, T(INS), where the insertion tuple is defined over attributes A(T(INS)). C(INS) denotes the Boolean expression in the form of a conjunction of equality predicates formed on all pairs of attributes and insertion values.

(2) DEL(T(DEL), C(DEL)):

DELETE FROM T(DEL) WHERE C(DEL)

DEL(T(DEL), C(DEL)) denotes a deletion from a table, T(DEL), with the deletion condition C(DEL).

(3) UPD(T(UPD), C(S(UPD)), C(W(UPD))):

UPDATE T(UPD) SET S(UPD) WHERE W(UPD)

UPD(T(UPD), C(S(UPD)), C(W(UPD))) denotes an update of a table, T(UPD), where C(S(UPD)) denotes the conjunction of the equality predicates formed on all pairs of attributes and update values from the update statement's SET clause. C(W(UPD)) denotes the Boolean expression in the WHERE clause of the update statement.

Consequently, A(S(UT) for an insert or a delete statement includes all attributes of a tuple and for an update statement includes only attributes in the set clause.

4.4. Common Attributes Test

Formally, we say that QT and UT have *common attributes* if $A(S(UT)) \cap A(QT) \neq \Phi$. Given that q belongs to QT and that u belongs to UT, the first step of invalidation is to test if QT and UT have common attributes. If $A(S(UT)) \cap A(QT) = \Phi$, u does not modify the result of q and UT is *irrelevant* to QT. If $A(S(UT)) \cap A(QT) \neq \Phi$, UT may be relevant to QT and u may modify the result of q. This test is mainly for an update statement rather than for an insert or delete statement.

Proposition 1. If A(S(UT)) ∩ A(QT) = Φ, UT is irrelevant to QT.

Proof: Given A(S(UT)) ∩ A(QT) = Φ, no attributes of the query condition are modified by S(UT). Therefore, for any result tuple, t, of q, it is not added, removed or modified when a UDI, u, of UT is applied to the database. In other words, the result of q is unaffected by u. Thus, UT is irrelevant to QT. ∎

Let us consider the examples of templates in Figures 5 and 6. Both the QT and UT involve the ITEM table, and the I_ID attribute is involved in the WHERE clause of both templates. However, I_ID is not involved in the SET clause of the update template. Correspondingly, the two attribute sets are as follows

A(S(UT)) = {I_RELATED1};

A(QT) = {I_ID, I_TITLE, I_A_ID, A_ID}.

Since A(S(UT)) ∩ A(QT) = Φ, these two templates do not share common attributes. By Proposition 1, UT is irrelevant to QT.

4.5. Construction of Boolean Expressions

If QT and UT share common attributes, we will construct a Boolean expression of q and u for satisfiability testing. The Boolean expression of q, C(q), is simply the q's where condition. The construction of the Boolean expression of u, C(u), is more complex. As an insert statement, INS, only adds new tuples to a table, C(INS) describes the new tuples being added. As a delete statement, DEL, only removes tuples that satisfy C(DEL) from a table, C(DEL) describes the removed data that satisfy C(DEL). As an update statement, UPD, modifies tuples in a table, C(UPD) is constructed to describe both the old values and the new values of the tuples being updated. Therefore, C(UPD) = C(S(u)) OR C(W(u)), the disjunction of the set clause condition and the where condition.

Given q and u, the Boolean expressions constructed for them are summarized in Table 2.

Table 2. Boolean expressions constructed for q and u

UDI Type of u	**Boolean Expression**
INS(T(INS), C(INS))	C(q) AND C(INS)
DEL(T(DEL), C(DEL))	C(q) AND C(DEL)
UPD(T(UPD), C(S(UPD)), C(W(UPD)))	C(q) AND (C(S(UPD)) OR C(W(UPD)))

4.6. Satisfiability Testing Algorithm

Larson and Yang's [13] CONJUNCTIVE algorithm processes conjunctive Boolean expressions of numeric comparison predicates on integer-valued attributes with predefined ranges. We have made the following extensions in our satisfiability testing algorithm.

(1) Handling more data types and predicates. This extension includes the string data type and "LIKE" predicates and the float point data type.

(2) Automatic discovery of attribute data type. This extension ensures that the invalidator does not need to query the backend database for metadata.

(3) Handling uninstantiated variables. This extension is for testing satisfiability between query templates and UDI templates. Because uninstantiated variables may assume any possible value, we assume that an atomic condition involving an uninstantiated variable is always true in the satisifiability testing. This assumption defers further satisfiability testing to instantiated SQL statements.

(4) Handling subqueries that return an atomic value. For a query, q, with a subquery, sq, and a UDI, u, we first test the satisfiability of *C(sq) AND C(u).* If u is relevant to sq, it is also relevant to q. Otherwise, we replace the atomic condition on the subquery to be true and further test the satisfiability of *C(q) AND C(u).*

(5) Handling self-joins. Since a table is referenced multiple times in a self-join, we associate all aliases of the table with the table itself and identify attributes of the aliased table accordingly.

4.7. Template Satisfiability Matrix

We implement the satisfiability testing between query and update templates by building a template satisfiability matrix. For each pair of query and UDI templates, we first test if they share common attributes. If they do not share common attributes, they are already irrelevant and the entry for this pair in the matrix is set to FALSE. Otherwise, we test the relevance of the UDI template to the query template. Table 3 shows an example of a satisfiability matrix between three query templates and four UDI templates.

Consider a pair of a query template QT3 and a UDI template UT4 in Table 3:

QT3: "SELECT I_ID, I_COST, A_FNAME, A_LNAME FROM ITEM, AUTHOR WHERE I_A_ID = A_ID AND I_ID = ?"

A(QT3) = {I_ID, I_COST, A_FNAME, A_LNAME, I_A_ID, A_ID}

C(QT3) = {I_A_ID = A_ID AND I_ID = ?}

UT4:"UPDATE ITEM SET I_COST = ? WHERE I_ID = ?"

C(UT4) ={I_ID = ? OR I_COST = ?}

A(S(UT4)) = {I_COST}.

This pair of templates shares common attributes as A(S(UT4)) ∩ A(QT3) is not empty. Therefore, we further test the Boolean expression C(QT3) AND C(UT4).

C(QT3) AND C(UT4)

= {I_A_ID = A_ID AND I_ID = ?} AND {I_ID = ? OR I_COST = ?}

Table 3. An example of a template satisfiability matrix

	QT1	QT2	QT3
UT1	TRUE	TRUE	TRUE
UT2	TRUE	FALSE	TRUE
UT3	TRUE	TRUE	FALSE
UT4	TRUE	FALSE	TRUE

As described in the previous sections, the predicates with uninstantiated variables will be assumed to be true. The satisfiability algorithm evaluates the Boolean expression of this pair of templates to be satisfiable and therefore the entry for this template pair in the template satisfiability matrix is TRUE. At runtime, for each instantiated u belonging to UT, any cached query, q, belonging to QT must be further checked for satisifabiliy with instantiated values.

In contrast, if a pair of query and UDI templates is irrelevant, the entry for this template pair in the template satisfiability matrix is FALSE. Subsequently, any UDI of that UDI template will not invalidate any query of the query template. For example, with UT2 and QT2, a UDI belonging to UT2 does not need to check against any cached query of QT2 as they are always unsatisfiable regardless of the values of the instantiated variables.

In short, the template satisfiability matrix reduces the satsifiability test of individual queries and updates to the satisfiability test of the corresponding templates; only when the pair of templates is relevant, the individual statements of the templates are tested further. The size of the template satisfiability matrix is linear to the number of relevant pairs of query templates and UDI templates.

4.8. Query Satisfiability Indexes

For each relevant pair of query and UDI templates, we build a satisfiability index on the important attributes of the instantiated queries if such attributes exist. By important attributes, we refer to those whose instantiated values affect the answer of the satisfiability test on instantiated statements. Intuitively, these important attributes should be in the predicates of the common attributes shared by the two templates.

Consider the following Boolean expression:

C(QT) AND C(UT)

= {I_ID = ? AND I_A_ID = A_ID} AND {I_ID = ?}.

The common attributes of the two templates are I_ID and the I_ID attribute has uninstantiated variables in the predicate on it (*I_ID = ?*). Moreover, the variable value in this predicate in a query of QT determines if an update of UT will invalidate the query. For instance, if the parameter has the value 8 in the query and has the value 9 in the update, the update is irrelevant to the query. In comparison, if the parameter has the value 8 in both the query and the update, we will need to invalidate the query upon the update.

In this example, I_ID is the important attribute and we build a query satisfiability index on the values of the parameter in the *I_ID = ?* predicate of the instantiated queries. When an update of QT arrives, we can use the parameter value in the update statement to look up relevant queries of the given query template.

4.9. Summary

In summary, we extend the satisfiability testing to query templates and build a template satisfiability matrix for filtering out irrelevant updates early on. We then build query satisfiability indexes for queries of each template to speed up the checking process. A summary of the control flow is shown in Figure 8.

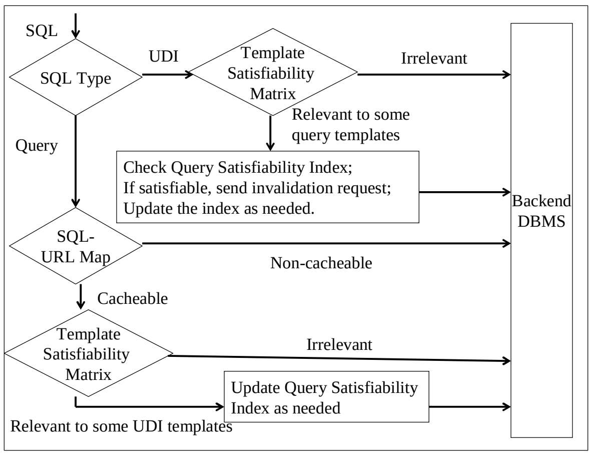

SQL
SQL Type
UDI
Template Satisfiability Matrix
Irrelevant
Relevant to some query templates
Query
Check Query Satisfiability Index;
If satisfiable, send invalidation request;
Update the index as needed.
Backend DBMS
SQL-URL Map
Non-cacheable
Cacheable
Template Satisfiability Matrix
Irrelevant
Update Query Satisfiability Index as needed
Relevant to some UDI templates

Figure 8. TBI control flow

5. Experimental Evaluation

We deployed the original Java implementation [14] of the TPC-W benchmark [21] as well as our modifications (to incorporate ESI tags) in a local area network. We ran remote browser emulators to study the performance of the web site under test. We compared the performance of the web site without a web cache, with the Oracle Web Cache, and with the Oracle Web Cache and our TBI.

5.1. System Configuration

There were three computers involved in the experiments. All of them ran the Red Hat Linux 7.3 operating system. The database server resided on a Dell PowerEdge server machine with a Pentium-III 1.26GHz processor and 512MB memory. The other two machines had a Pentium-IV 1.8GHz CPU with 1GB memory each.

5.1.1. Client Tier

The Remote Browser Emulator (RBE) program provided by the original TPC-W implementation acted as an emulated browser client interacting with the web site under test. We used 50 RBEs with no think time in all experiments to simulate an intensive, real-world web load.

5.1.2. Middle Tier

The middle tier included the web server, the application server, the web cache, and the TPC-W application. We used the Oracle9iAS application server, which was bundled with the Apache Web Server and the Oracle Web Cache Release 2 v9.0.3 (OWC). The OWC was configured to cache ESI templates and dynamic fragments. We also used the Tomcat 4.0 application server as a servlet container for the TPCW servlets.

5.1.3. Database Tier

We used Oracle9i Release 2 (9.2.0.1) as our database server. The TPC-W databases of different scales were created and populated according to the TPC-W specification.

5.2. TPC-W Workload

The TPC-W benchmark specification [21] is published by the Transaction Processing Council (TPC) to model an e-commerce web site. The TPC-W workload consists of two categories of web interactions, “Browse” and “Order”. The “Order” web interactions are centered on the ordering process and generate a large number of SQL UDIs. In contrast, the “Browse” web interactions primarily supply informational web contents about the products to the users, generating SQL queries to the backend.

We experimented with the three transaction mixes defined by the TPC-W specification, namely the Browsing mix, the Shopping mix, and the Ordering mix. The Browsing mix consists of 95% “Browse” web interactions and 5% “Order” web interactions. The Shopping mix comprises of 80% “Browse” and 20% “Order” web interactions. The Ordering mix is composed of 50% “Browse” and 50% “Order” web interactions. As TBI is an invalidation framework, we focused on measuring the performance with the Ordering mix to stress test the UDIs.

The UDIs in the TPC-W workload operate on the ORDERS and ORDER_LINE tables. Other tables such as ITEM, AUTHOR and ADDRESS are relatively static. Among the 14 types of web interactions, no updates in the workload are relevant to the Search Results page fragment, and one or more UDIs in the workload are relevant to page fragments of other web interactions.

5.3. Framework Construction

We constructed three frameworks for the TPC-W Java implementation. They were the no-cache baseline framework, the Oracle Web Cache with ESI framework, and the Oracle Web Cache with ESI and our TBI framework.

5.3.1. No-Cache Baseline Framework (NC)

The No-Cache Baseline Framework (NC) uses the original Java implementation [14] of the TPC-W benchmark, which is purely servlet based. Regardless of the types of the requested web contents, all requests are answered dynamically by Java servlets running on top of the Apache Tomcat Servlet Engine [4]. The Oracle Web Cache is configured to cache nothing. Although the contents are always served fresh, the performance (in terms of response time and throughput) may suffer due to the heavy workload on the servlets.

5.3.2. OWC ESI Framework (OWC)

The OWC ESI Framework uses ESI templates and fragments as caching units. We constructed this framework by transforming the servlet-based TPC-W implementation into an ESI-enabled implementation and configuring the OWC to cache the ESI templates and fragments. We configured the OWC to cache all the templates and dynamic fragments in the "Browse" web interactions except the promotion banner fragment that had to be generated randomly at runtime. Dynamic fragments in the "Order" web interactions are mostly un-cached due to their transactional semantics and privacy-related issues. In particular, the Shopping Cart and Buy Request pages are not cached.

We constructed the OWC ESI Framework mainly for the purpose of comparing it with the OWC-TBI framework. Without invalidation, the OWC ESI Framework caches the templates and dynamic fragments but never purges them. Therefore, the cached dynamic fragments may be inconsistent with the backend database. This framework may have excellent performance, but it is unrealistic due to the poor data consistency.

5.3.3. OWC ESI with TBI Framework (TBI)

The OWC-TBI framework, or the TBI framework in short, uses the ESI-enabled TPC-W implementation but compiled with our TBI package. The OWC in this framework is configured to cache the same contents as in the OWC-ESI framework. In addition, the OWC is configured to accept the invalidation requests from our TBI module. This approach may have a slightly worse performance than the OWC-ESI framework due to the invalidation overhead and the resulting higher cache miss ratio, but it delivers fresh web content.

5.4. Measurement Methodology

We measured both response time and throughput (in terms of WIPS, Web Interaction Per Second) at the Remote Browser Emulator (RBE). Each experiment was executed for 5000 seconds and measurements were only reported for the last 1000 seconds. The first 4000 seconds are the system warm-up time, because the warm-up process is essential for complex systems, especially systems with caching components.

Before executing each experiment, several key procedures were carried out to avoid interference from previous runs. The goal was to provide a clean start. First, the data in the databases were refreshed to ensure approximately the same database content for each experiment. Second, the OWC was restarted to purge the cached contents left from the previous run. Last, a random seed conforming to the TPC-W specification was used for each experiment.

5.5. Experimental Results

5.5.1. General Metrics

We first present the throughput (Table 4) and response time (Table 5) of different transaction mixes on the three frameworks. We experimented with different database sizes and the results we present in this section are from the 100K database (there are 100,000 records in the ITEM table). Both OWC and TBI outperformed the baseline framework on different transaction mixes. The results of these experiments confirm that our TBI is sufficiently generic to be deployed at database-backed web sites with arbitrary database sizes and transaction mixes.

Not surprisingly, TBI performed worse than OWC, because the invalidation component needs processing time at the web site and to reduce number of cache hits at the web cache. However, TBI ensures the freshness of the web content, which is highly desirable in e-commerce.

Table 4. Throughput (WIPS) with a 100K database

Throughput	NC	TBI	OWC
Browsing	0.3	1.1	3.4
Shopping	0.5	0.8	1.9
Ordering	2.1	2.6	3.8

Table 5. Average response time (in seconds) with a 100K database

Response time	NC	TBI	OWC
Browsing	38.3	28.9	7.2
Shopping	45.1	36.8	18.6
Ordering	15.8	11.1	6.0

An additional note on these results is that the Ordering mix has a better performance than have the Shopping and the Browsing mixes for NC and TBI. This is because the execution cost of the TPC-W queries is more expensive than that of the UDI operations. The majority of TPC-W queries involves aggregations, order-by clauses and group-by clauses whereas the UDIs are much simpler.

5.5.2. Response Time by Web Interaction Category

We further investigate the average response time by web interaction categories. This is because, from the web users' point of view, the time spent waiting for web pages to be returned in individual web interactions is most important. Based on the results on general metrics, we chose the 100K database size and the ordering mix since this combination exercises TBI the most.

There are four web interaction categories by the web pages they generate: (A) a template-only page, (B) a page with cacheable templates and fragments, (C) a page with cacheable templates and non-cacheable fragments, and (D) a servlet page.

A template-only page does not contain any fragments and is readily cached by OWC. As shown in Figure 9, a template-only page (Category A) took 8 seconds to be generated in the baseline framework but zero seconds in both TBI and OWC. The sub-second response time indicates that the template-only page is directly served out of the web cache. Moreover, the cached templates have no dynamic contents. No invalidation overhead is incurred by TBI. Therefore, both TBI and OWC achieved identical degrees of performance improvement.

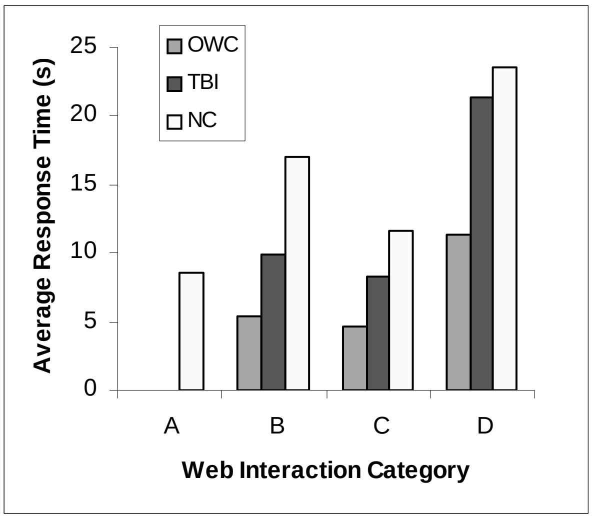


Figure 9. Average response time (in seconds) by web interaction category

For pages with cacheable templates and fragments (Category B), both the templates and the cacheable fragments are cached. The average response time of the baseline framework was 17 seconds, while that of the two caching frameworks was under 10 seconds. Both TBI and OWC achieved significant improvement on pages with cacheable templates and cacheable fragments.

For pages with cacheable templates but non-cacheable fragments (Category C), the template is cached but the dynamic fragments are not. Hence, the improvement on the response time in the caching frameworks was not as significant as for Category B. Nevertheless, the ESI-enabled caching frameworks were still more efficient than the servlet-based baseline framework.

For servlet pages (Category D), the two caching frameworks also outperformed the baseline framework as these serlvet pages are cacheable.

5.5.3. Distribution of Response Time

While the average response time gives a general idea about the performance, the distribution of the response time allows us to see the percentage of web pages that are accelerated by caching (Figure 10).

All of the web contents in either caching framework can be served in a shorter response time than in the baseline framework. In particular, almost 100% of the template-only pages were served in zero-second response time (not shown in Figure 10). For pages with cacheable templates and fragments (Category B), OWC and TBI respectively served around 80% and 60% of the pages within eight seconds while the baseline framework served only around 25% in the same time. Results were similar for the pages with cacheable templates but non-cacheable fragments (Category C). Finally, the servlet pages (Category D) are the most expensive in terms of time among the four categories of pages to generate.

5.5.4. Overhead of Template-Based Invalidator

After the measurements from the remote browser emulator indicated the overall performance of TBI, we further examined the time breakdown of the internal processing of TBI.

Query We investigated the amount of time spent in each step of the invalidation cycle, including parsing the statement, checking the template satisfiability matrix, updating the indexes, and executing the statement. We measured these steps (if applicable) in four cases of an HTTP request invoking a query: a cache hit, a cache miss with a UDI template relevant to the query, a cache miss without any UDI template relevant to the query, and a non-cacheable request.

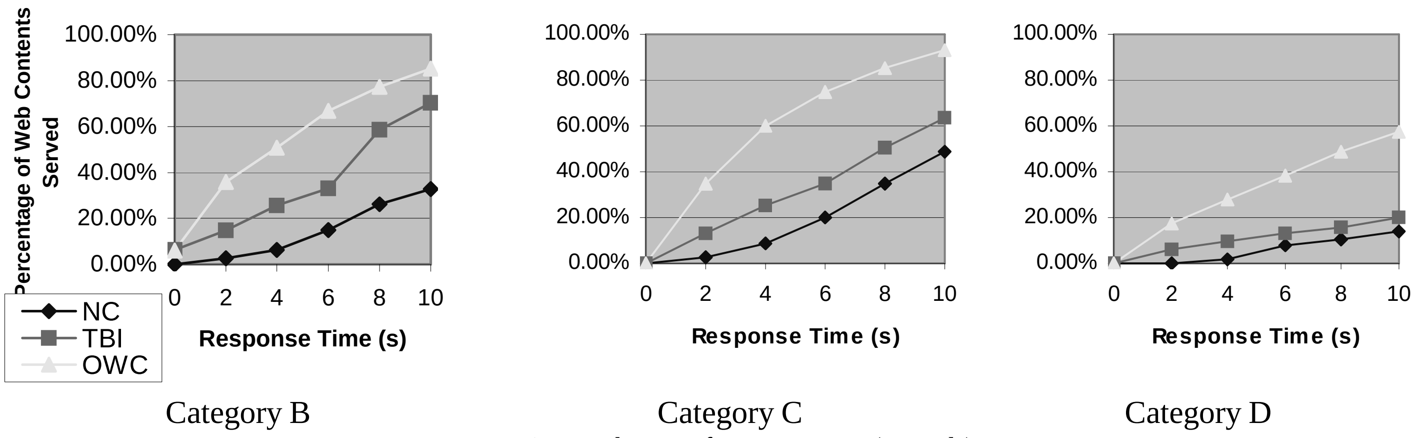


Figure 10. Distribution of response time (seconds)

Table 6. Query time breakdown (in milliseconds) for Ordering Mix with a 100K database

HTTP Requests	Statement Parsing	Matrix Checking	Indexes Updating	JDBC Execution
Non-Cacheable	0.0	0.0	0.0	126.0
Cache Hit	0.0	0.0	0.0	0.0
Cache Miss - Irrelevant UDI	6.1	0.0	0.0	4835.0
Cache Miss - Relevant UDI	40.1	0.1	0.1	5600.0

Table 7. UDI time breakdown (in milliseconds) for Ordering Mix with a 100K database

UDIs	Statement Parsing	Matrix Checking	Indexes Checking & Updating	JDBC Execution	Invali-dation
Query Template Irrelevant	1.0	0.0	0.0	6.3	0.0
Query Irrelevant	3.3	0.0	0.0	6.1	0.0
Query Relevant	3.6	0.0	467.0	3.2	55.7

A non-cacheable request does not induce additional cost in TBI; it is directly passed to the underlying JDBC driver. With cacheable fragments, there may be misses and hits. A cacheable fragment hit does not induce additional cost in TBI, as the fragment is served by OWC. TBI is invoked only when the fragment is not found in OWC. This is a cache miss.

With a cache miss, we need to update the data structures. If the query template is not indexed, the query instance has to be inserted into the corresponding query instance vector of all relevant UDI templates. Otherwise, we can simply update the query satisfiability index and there is no need to store the entire query instance in the query instance vector.

Table 6 shows that the amounts of time spent on statement parsing, checking the template satisfiability matrix, and updating the data structures were negligible. The query time was dominated by the JDBC execution at the backend database.

UDI We also investigated the time breakdown of a UDI. A UDI may fall into one of the three cases – irrelevant to any query templates (by checking the template satisfiability matrix), relevant to some query templates but irrelevant to the cached queries according to the data structures, or relevant to some cached query. We show these three cases in Table 7 as "Query Template Irrelevant", "Query Irrelevant", and "Query Relevant".

Compared with handling a query, handling a UDI involves two more steps, namely, checking the data structures and possibly sending invalidation messages. Different from the performance of handling a query, Table 7 shows that the times of all steps are insignificant (under 0.5 seconds). The largest two items are the invalidation time of 50 milliseconds and the data structure update time of 467 milliseconds.

5.5.5. Summary

Our experimental results confirm that our TBI is sufficiently generic to be deployed by database-backed web sites with arbitrary database sizes and transaction mixes. Furthermore, both OWC and OWC-TBI caching frameworks outperformed the baseline framework with different database sizes and transaction mixes. The OWC-TBI framework represents a tradeoff between two extremes (no caching versus caching but no cache-consistency). It incurs slight processing overhead at the web site and fewer cache hits at the web cache, but it delivers fresh web content with improved performance.

6. Related Work

Recent work has studied cache consistency and freshness in various contexts. Bright and Raschid proposed latency-recency profiles to accommodate user preferences [6]. Cho and Garcia-Molina studied crawling scheduling for maintaining the freshness of a crawled web page repository [9]. Labrinidis and Roussopoulos presented an update propagation policy for data-intensive web sites [12]. Olston and Widom addressed the tradeoff between data freshness and transfer cost [17]. In comparison to treating the cached data units as opaque objects in these studies, we considered the SQL semantics of updates as well as the semantics of cached HTML page fragments.

There has been a rich body of research on materialized view maintenance [11], which exploits SQL semantics of queries and updates. These techniques have been recently applied to update propagation in DBCache [15], DBProxy [2], and TimesTen Front-end Cache [20] for database-backed web sites. In comparison, our work focuses on lightweight invalidation of generated HTML fragments as opposed to full update propagation to cached relational data tuples.

Invalidation has been previously proposed for database-backed web sites. While the Oracle Web Cache [3] and the Dynamic Page Cache [10] provided primitives for applications to specify their invalidation policies (such as a timeout value), the DUP (Data Update Propagation) algorithm [8] and the view invalidation algorithms [7] facilitate automatic invalidation at the application level. Our work argues for a lightweight, automatic invalidation policy in that we neither install triggers at nor send polling queries to the backend database server.

Finally, query templates have been presented in caching for database-backed web sites [2][7][16]. In this work, we extensively utilize templates to improve the efficiency of our invalidation. Specifically, we exploit not only SQL query templates, but also SQL update templates, URL templates, as well as ESI templates [19]. This enables the invalidator to handle a large number of updates and queries efficiently at runtime.

7. Conclusions

We have presented a template-based invalidator (TBI) for cached database-generated web contents. The invalidator works by checking the satisfiability relationship between a SQL query and a UDI statement without checking the database content. It maintains a mapping between URLs and SQL queries so that it can send invalidation requests to the web cache when a UDI is relevant to a cached query. It further improves efficiency by building a template satisfiability matrix and query satisfiability indexes. We have integrated TBI into the Oracle Web Cache (including an ESI processor) and conducted extensive experiments using the TPC-W benchmark. Our experimental results show that OWC-TBI delivers fresh web content efficiently.

Acknowledgements

We would like to thank Henry Wong from the Computer Science Department Systems Lab at HKUST for his timely technical support on the computers used in the experiments. Funding for this research was from the RGC grant HKUST6158/03E.

References

[1] Akamai Technologies Inc. Akamai EdgeSuite. http://www.akamai.com/html/en/tc/core tech.html.

[2] K. Amiri, S. Park, R. Tewari, and S. Padmanabhan. DBProxy: A Self-Managing Edge-of-Network Data Cache. Technical Report RC22419, IBM Research, 2002.

[3] J. Anton, L. Jacobs, X. Liu, J. Parker, Z. Zeng, and T. Zhong. Web Caching for Database Applications with Oracle Web Cache. Proc. ACM SIGMOD, 2002.

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

[5] J. Blakeley, N. Coburn, and P.-A. Larson. Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates. Proc. VLDB, 1986.

[6] L. Bright and L. Raschid. Using Latency-Recency Profiles for Data Delivery on the Web. Proc. VLDB, 2002.

[7] K.S. Candan, D. Agrawal, W.-S. Li, O. Po, W.-P. Hsiung. View Invalidation for Dynamic Content Caching in Multitiered Architectures. Proc. VLDB, 2002.

[8] J. Challenger, A. Iyengar, and P. Dantzig. A Scalable System for Consistently Caching Dynamic Web Data. Proc. IEEE INFOCOM, 1999.

[9] J. Cho and H. Garcia-Molina. Synchronizing a Database to Improve Freshness. Proc. SIGMOD, 2000.

[10] A. Datta, K. Dutta, Suresha, K. Ramamritham, H. Thomas, and D. VanderMeer. Proxy-Based Acceleration of Dynamically Generated Content on the World Wide Web: An Approach and Implementation. Proc. ACM SIGMOD, 2002.

[11] A. Gupta and I. S. Mumick (Editors). Materialized Views: Techniques, Implementations, and Applications. The MIT Press, 1999.

[12] A. Labrinidis and N. Roussopoulos. Update Propagation Strategies for Improving the Quality of Data on the Web. Proc. VLDB, 2001.

[13] P. A. Larson and H. Z. Yang. Computing Queries from Derived Relation: Theorectical Foundation. Technical Report CS-87-35, Department of Computer Science, University of Waterloo, 1987.

[14] M. H. Lipasti (University of Wisconsin). Java TPC-W Implementation. http://www.ece.wisc.edu/~pharm/tpcw.shtml.

[15] Q. Luo, S. Krishnamurthy, C. Mohan, H. Pirahesh, H. Woo, B. G. Lindsay, and J. F. Naughton. Middle-Tier Database Caching for e-Business. Proc. SIGMOD, 2002

[16] Q. Luo and J. F. Naughton. Form-Based Proxy Caching for Database-Backed Web Sites. Proc. VLDB, 2001.

[17] C. Olston and J. Widom. Best-Effort Cache Synchronization with Source Cooperaton. Proc. ACM SIGMOD, 2002.

[18] Oracle Corporation. Oracle9iAS Web Cache. http://otn.oracle.com/products/ias/web_cache/content.html

[19] Oracle Corporation and Akamai Technologies, Inc. Edge Side Includes (ESI). http://www.esi.org/index.html

[20] Times-Ten Team. Mid-Tier Caching: the TimesTen Approach. Proc. ACM SIGMOD, 2002.

[21] Transaction Processing Performance Council (TPC). TPC Benchmark™ W (Web Commerce) Specification Version 1.8, 2002.