Template-Based Runtime Invalidation for Database-Generated Web Contents

Chun Yi Choi and Qiong Luo

Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Kowloon
Hong Kong, China
Contact: luo@cs.ust.hk

Abstract. We propose a template based runtime invalidation approach for maintaining cache consistency of database generated web contents. In our approach, the invalidator sits between a web cache and a database server, and intercepts query statements as well as 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

J.X. Yu, X. Lin, H. Lu, and Y. Zhang (Eds.): APWeb 2004, LNCS 3007, pp. 755–764, 2004.

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.

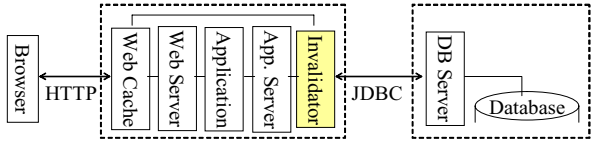

Fig. 1. The invalidator in a database-backed web site

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 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.

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

We use the Oracle9iAS Web Cache (OWC) [18] and its ESI (Edge Side Includes) [19] processor to enable ESI templates and fragments caching. ESI is a simple markup language proposed by Akamai (a major player in the content delivery network business) and Oracle. 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.

The OWC uses cache rules to direct caching of ESI templates and fragments. It 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

Figure 2 shows the components of our Template Based Invalidator (TBI). 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. The following describes the components and data structures in order.

The JDBC wrapper is the interface of TBI to a web application. It conforms to the java.sql package interface. After 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 finish their corresponding tasks, 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.

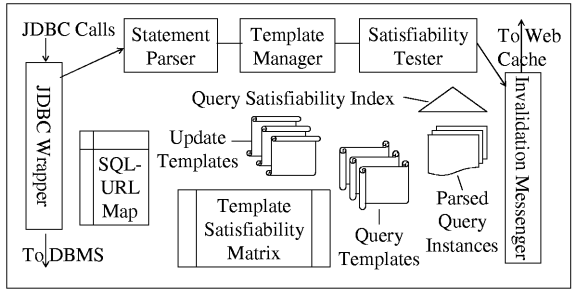

Fig. 2. Components of TBI

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.

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 3 and 4 show examples of SQL templates.

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

Fig. 3. An example of a query template

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

Fig. 4. 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.

(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 3. 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.

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. 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.

4 Template-Based Invalidation

Due to the space constraints, we omit the formal description of our template-based invalidation algorithms. Instead, we show the control flow of the TBI in Figure 5.

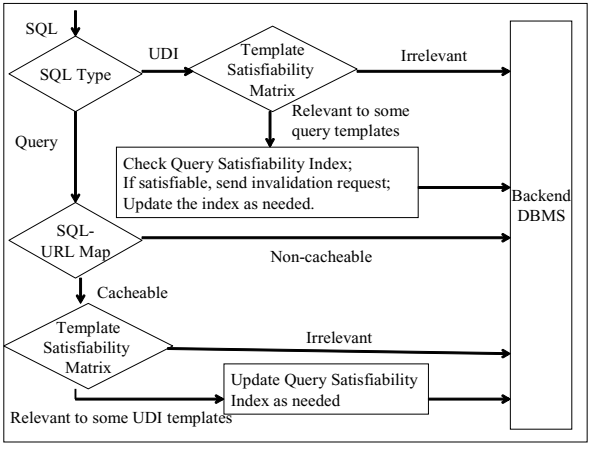

Fig. 5. TBI control flow

In brief, we extend the satifiability testing algorithm to handling SQL templates. The class of queries handled is selection-projection-join queries with optional subqueries, order-by and group-by clauses. The extensions to the algorithm includes string and float data types, "LIKE" predicates, self-joins, and uninstantiated variables.

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. Due to space constraints, we report only a small portion of the results.

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.

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.

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.

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 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.

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.

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 without invalidation. 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.

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.

5.3 Experimental Results

We present the throughput (Table 1) in terms of WIPS (Web Interaction Per Second) and response time (Table 2) of different transaction mixes (Browsing, Shopping, or Ordering) 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 reduces 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 1. 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 2. 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

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. An extended version of this paper is available at http://www.cs.ust.hk/~luo.

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.