A Scalable Update Management Mechanism for Query Result Caching Systems at Database-Driven Web Sites

Seunglak Choi[2], Sekyung Huh[1], Su Myeon Kim[1], Junehwa Song[1], and Yoon-Joon Lee[1]

[1] KAIST, 373-1 Kusong-dong Yusong-gu Daejeon 305-701, South Korea
{skhuh, yjlee}@dbserver.kaist.ac.kr,
{smkim, junesong}@nclab.kaist.ac.kr
[2] ETRI, 161 Gajeong-dong Yusong-gu Daejeon 305-350, South Korea
slchoi@etri.re.kr

Abstract. A key problem in using caching technology for dynamic contents lies in update management. An update management scheme should be very efficient without imposing much extra burden to the system, especially to the original database server. We propose an scalable update management mechanism for query result caching in database-backed Web sites. Our mechanism employs a two-phase consistency checking method, which prunes out unaffected queries at the earliest possible moment. The method scales well with a high number of cached instances.

1 Introduction

Caching technology has been frequently used to improve the performance of serving dynamic contents at Web sites. The key problem in using caching technology for dynamic contents lies in update management; that is, cached contents should be ensured consistent to the original data in databases. Thus, an effective update management mechanism is of utmost importance for dynamic content caching. Moreover, an update management scheme should be very efficient without imposing much extra burden to the system, especially to the origin database server. Note that the database server can be easily a bottleneck to overall Web site's performance. Thus, if not efficient, the advantage of using the cache will be significantly impaired due to the extra overhead to keep the freshness of the cached data.

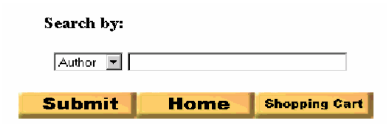
In this paper, we propose an efficient update management mechanism for dynamic content caching, more specifically, for query result caching [5,6,8,2] in database-driven Web sites. The idea of query result caching is to store results of frequently-issued queries and reuse the results to obtain the results of subsequent queries, significantly saving computational cost to process queries and retrieve results. Our method, upon reception of an update request, instantly processes the update and invalidates affected query results in the cache. In doing so, the cache initiates and takes in charge of the update management process, and minimizes the involvement of the database server. In other reported update management schemes [4,3,1], the servers are heavily responsible for the overall update process. In addition, our mechanism employees a two-phase consistency checking method in which the expensive part, *i.e.,* join checking, is performed only once to a group of queries. In this fashion, the method prunes out unaffected queries at the earliest possible moment. Thus, the method scales well with a high number of cached instances. The number of query instances can be very high especially for range queries.

This paper is organized as follows. In section 2, we describe the cache consistency mechanism. In section 3, we evaluate and analyze the performance of the mechanism. Finally in section 4, we present conclusions.

2 Cache Consistency Mechanism

2.1 Query Templates and Query Instances

Before describing our mechanism, we introduce the notions of query templates and instances. In Web-based applications, a user usually submits a request by using a HTML form. Figure 1 shows a simple example. A user types a search keyword and clicks the submit button in the form. Then, a WAS generates a query from the form and sends it to a database server. The generation of the query is done as encoded in the applications. Thus, each HTML form can be translated to a template of queries through the encoding. We call such a template a *query template*. The queries generated from the same HTML form, i.e., from the same query template are of the same form; they share the same projection attributes, base tables, and join conditions. The only difference among the queries lies in their selection regions, which are specified by users. We call the individual queries generated upon user's requests *query instances*.



```
SELECT I_TITLE, I_ID, A_FNAME, A_LNAME
FROM ITEM, AUTHOR
WHERE I_A_ID = A_ID
AND A_LNAME = '@keyword'
```

Fig. 1. HTML form and its corresponding query form. The HTML form is clipped from the Search Request Web page of the TPC-W benchmark [7].

We characterize a query template $QT = (T, PA, JC)$ as follows. T is a set of tables which are specified in a `FROM` clause. PA is a set of projection attributes. JC is a set of join conditions. $T(QT)$, $PA(QT)$, and $JC(QT)$ denotes T, PA, and JC of a query template QT respectively. We define a query group of a template QT as $QG(QT) = \{Q_i\}$ where Q_i is generated from QT. A query instance Q_i is said to be *affected* by an update if Q_i is modified by the update.

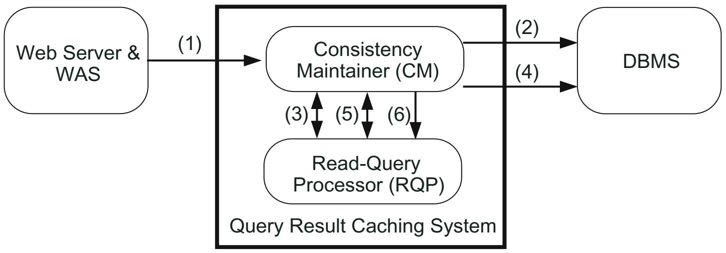

Web Server & WAS
(1)
Consistency Maintainer (CM)
(2)
(4)
DBMS
(3)
(5)
(6)
Read-Query Processor (RQP)
Query Result Caching System

Fig. 2. Processing Flow

2.2 Architectural Overview

Under the proposed mechanism, a caching system conceptually consists of the *Consistency Maintainer* (CM) and the *Read-Query Processor* (RQP) (see Figure 2). CM performs the consistency check to identify query results affected by a given update and invalidates affected results. RQP is a main component of the caching system, which stores query results and serves read queries.

Figure 2 depicts the processing flow of the consistency check. A WAS sends an update query to CM (1). CM forwards the update to the origin database server (2). In order to find the templates which can include the query instances affected by the update, CM investigates query template information kept in RPQ (3) and sends to the database server a *join check query* (4), which will be discussed in detail in section 2.3. Once the templates are determined, CM finds affected query instances in the templates (5) and removes them from the cache (6).

2.3 Two-Phase Consistency Check

Consistency check is to test if there exist any query instances which are affected by an update. This involves repeated matching of each query instance against a given update, and thus costs serious computation overhead. We identify that there are many computation steps which are repeated in testing different instances. To avoid such repetition, we propose a two-phase consistency checking mechanism. We note that different query instances generated from the same query template differ only in their selection regions. Thus, during the first step called *template check*, we match the query template against the update and identifies if it is possible that any of the query instances from the template are affected by the update. Then, during the second step, called *instance check*, individual instances are matched against the update.

Template Check. The following three conditions are satisfied, if U affects any query instances in $QG(QT)$.

1. If a set of attributes modified by U intersects $PA(QT)$. Note that, for `INSERT` and `DELETE`, this condition is always true. These queries insert or delete an entire tuple.

2. If a table on which U is performed is included in $T(QT)$.
3. If one or more newly inserted tuples by U satisfy $JC(QT)$. If QT has join conditions, query instances generated from QT include only joined tuples. Thus, only when the inserted tuples are joined, U can affects the query instances.

Example 1. Let us consider a query template QT, a query instance Q, and an update U as follows. Q is generated from QT.

$QT : T = \{$`ITEM, AUTHOR`$\}, PA = \{$`I_TITLE, I_COST, A_FNAME`$\}$,
$JC = \{$`I_A_ID = A_ID`$\}$

Q :
```
SELECT I_TITLE, I_COST, A_FNAME FROM ITEM, AUTHOR
WHERE I_A_ID = A_ID AND I_PUBLISHER = 'McGrowHill'
```
U :
```
INSERT INTO ITEM (I_ID, I_A_ID, I_TITLE, I_PUBLISHER)
VALUES (30, 100, 'XML', 'McGrowHill')
```

The conditions (1) and (2) are easily evaluated as trues. U modifies the projection attribute `I_TITLE` and the table `ITEM`. For checking the condition (3), a cache sends to a database server a join check query as shown below. This query examines whether the table `AUTHOR` has the tuples which can be joined with the tuple inserted by U. Because the join attribute value of the inserted tuple is 100, the join check query finds the tuples with `A_ID` $= 100$. If the result of the query is not null, we know that the inserted tuple is joined.

```
SELECT A_ID FROM AUTHOR WHERE AUTHOR.A_ID = 100
```

As described in the example 1, the join check requires the query processing of a database server. The two-phase consistency check performs the join check over each query template, not each query instance. Thus, it dramatically decreases the overhead to a database server.

Instance Check. Once a template passed the template check, the query instance check is applied to the template. It finds the affected query instances by comparing the selection region of an update query to those of query instances. If the selection region of a query instance overlaps that of an update query, we know that the query instance is affected by the update. In the example 1, the query instance Q is affected by the update U because the selection region of Q, `I_PUBLISHER='McGrowHill'`, is equal to that of U.

3 Performance Evaluation

Experimental Setup. We measured the update throughputs of the query result caching system adopting the proposed mechanism. We sent update queries to the caching system. For each update, the caching system forwards the update query and sends join check queries to a database server. Under this situation, the throughput is limited by the amount of processing these queries in the database server.

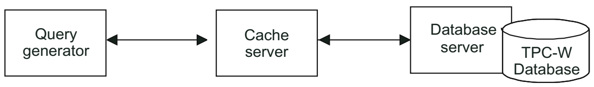


Fig. 3. Experimental setup

Figure 3 shows the setup for evaluating the proposed mechanism. The *Query Generator* emulates a group of Web application servers, generating queries. It runs on a machine with a Pentium III 800MHz, 256MB RAM. For the database server, we used Oracle 8i with the default buffer pool size of 16MB. The database server runs on a machine with a Pentium IV 1.5GHz, 512M RAM. The caching system runs on the machine with a Pentium III 1GHz, 512M RAM. We implemented the proposed mechanism as a part of WDBAccel query result caching system [5]. All machines run Linux and are connected through a 100Mbps Ethernet.

We populated the database of the TPC-W benchmark in the database server at 100K scale [1]. The TPC-W benchmark [7] is an industrial standard benchmark to evaluate the performance of database-driven Web sites. We used the update query modified from one used in Admin Confirm Web page of TPC-W: `INSERT INTO ITEM (I_ID, I_A_ID, I_COST, I_PUBLISHER) VALUES (@I_ID, @I_A_ID, @I_COST, '@I_PUBLISHER')`. This query inserts information on a book into `ITEM`. The values of the attribute `I_ID` follow the random distribution.

Experimental Results. In order to determine the performance improvement by the two-phase consistency check, we measured the update throughputs of the two-phase check and the brute-force approach. In the brute-force approach, the join check is performed against individual query instances. (Note that in the two-phase check, the join check is performed against a query template.)

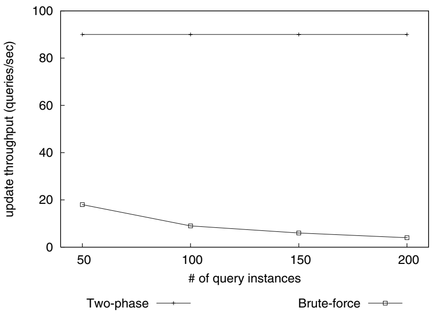


Fig. 4. Update throughputs of a single cache node

Figure 4 shows the update throughputs as the number of query instances ranging from 50 to 200. The figure shows that the throughputs of the two-phase

[1] In TPC-W, the scale of database is determined by the cardinality of `ITEM` table.

check are equal. This means that the two-phase check imposes the same overhead on a database server regardless of how many query instances are. The amount of overhead of the two-phase check will depend on only the number of query templates. The two-phase check generates a join check query for each query template.

4 Conclusions

In this paper, we proposed a scalable update management mechanism for the query result caching systems. We divided a consistency check to two phases, template check and instance check. The template check is performed over a query template, not an individual instance. We presented the experimental results that verify a high level of the scalability of the mechanism.

References

1. Khalil Amiri, Sara Sprenkle, Renu Tewari, and Sriram Padmanabhan. Exploiting templates to scale consistency maintenance in edge database caches. In *Proceedings of the International Workshop on Web Caching and Content Distribution*, 2003.
2. Khalil S. Amiri, Sanghyun Park, Renu Tewari, and Sriram Padmanabhan. DBProxy: A self-managing edge-of-network data cache. In *19th IEEE International Conference on Data Engineering*, 2003.
3. K. Selcuk Candan, Divyakant Agrawal, Wen-Syan Li, Oliver Po, and Wang-Pin Hsiung. View invalidation for dynamic content caching in multitiered architectures. In *Proceedings of the 28th VLDB Conference*, 2002.
4. K. Selcuk Candan, Wen-Syan Li, Qiong Luo, Wang-Pin Hsiung, and Divyakant Agrawal. Enabling dynamic content caching for database-driven web sites. In *Proceedings of ACM SIGMOD Conference*, Santa Barbara, USA, 2001.
5. Seunglak Choi, Jinwon Lee, Su Myeon Kim, Junehwa Song, and Yoon-Joon Lee. Accelerating database processing at e-commerce sites. In *Proceedings of 5th International Conference on Electronic Commerce and Web Technologies (EC-Web 2004)*, 2004.
6. Qiong Luo and Jeffrey F. Naughton. Form-based proxy caching for database-backed web sites. In *Proceedings of the 27th VLDB Conference*, Roma, Italy, 2001.
7. Transaction Processing Performance Council (TPC). TPC benchmark^{TM}W (web commerce) specification version 1.4. February 7, 2001.
8. Khaled Yagoub, Daniela Florescu, Valerie Issarny, and Patrick Valduriez. Caching strategies for data-intensive web sites. In *Proceedings of the 26th VLDB Conference*, 2000.