An Efficient Update Management Mechanism for Clustered Query Result Caching Systems at Database-driven Web sites

Seunglak Choi Sekyung Huh Su Myeon Kim
Junehwa Song Yoon-Joon Lee

CS/TR-CS-TR-2005-238

May 24, 2005

K A I S T
Department of Computer Science

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 efficient update management mechanism for query result caching in database-backed Web sites. Our method instantly processes each update and invalidates affected query results in the cache. The cache takes in charge of the update management process, and minimizes the overhead imposed to the database server. In addition, the mechanism employees 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. As it minimizes the overhead to database servers, it also scales well up with the number of cache nodes in a system. Thus, it provides a scalable framework of update management for a multi-node or clustered cache architecture.

1 Introduction

With the rapid expansion of the Internet, lots of interesting and value-generating services, *e.g.*, e-commerce, are now provided through WWW. Mostly, those services are provided through Web contents which are dynamically constructed upon users' requests, and the systems providing such services are in many cases composed of multi-tier system components, *e.g.*, Web servers, Web application servers (WASs), and database servers, etc. Thus, the generation and delivery of a Web content involves a long costly sequence of operations through multiple system components, resulting in serious overhead to the system. To ease performance problem of such Web sites, many attempts have been made, and among them, caching [17] and clustering [9] have been quite successful in serving pre-generated html contents. However, related technologies are still limited in serving dynamically generated contents.

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 [7, 14, 20, 3] 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 [5, 4, 2], 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. In addition, as it minimizes the overhead to database servers, it also scales well up with the number of cache nodes in a system. Thus, it provides a scalable framework of update management for a multi-node or clustered cache architecture.

The two-phase method is effective because in many Web-based applications a lot of query results share the same query format [14], and hence, have the same join condition. This two-phase method is applied in a multi-node or clustered architecture as well. In that case, the join check is performed in only one node, which broadcasts the result to the whole system. Thus, the overhead of consistency check incurred to the database server is identical regardless of the number of cache nodes in the system.

The proposed mechanism ensures *strong consistency*. It is done by invalidating affected query results before the completion (*i.e.*, commit) of a database update. After the completion, the cache serves up-to-date data by obtaining them from the origin database. It is often critical to serve up-to-date contents (*e.g.* the price of products in on-line shopping sites).

Additionally, our method supports transparent deployment; as we design and build an invalidation module in a cache system, it is not necessary to install additional modules in application/database servers. Our invalidation module requires that update queries go though a cache. Recently, Web contents in many large-scale sites are managed by Web-based content management systems. In these sites, the updates to Web contents are generated in a WAS and thus are delivered to a cache. Therefore, our mechanism is practically valid.

This paper is organized as follows. In section 2, we introduce background knowledge. In section 3, we describe the cache consistency mechanism. In section 4, we suggest the issues occurred in clustered cache nodes and their solutions. In section 5, we evaluate and analyze the performance of the mechanism. In section 6, we describe related work. Finally in section 7, we present conclusions.

2 Preliminaries

In this section, we describe the typical configuration of query result caching systems on which the proposed mechanism is deployed. We then introduce a characteristic of the database queries frequently used in most Web-based applications. The characteristic is utilized by our mechanism.

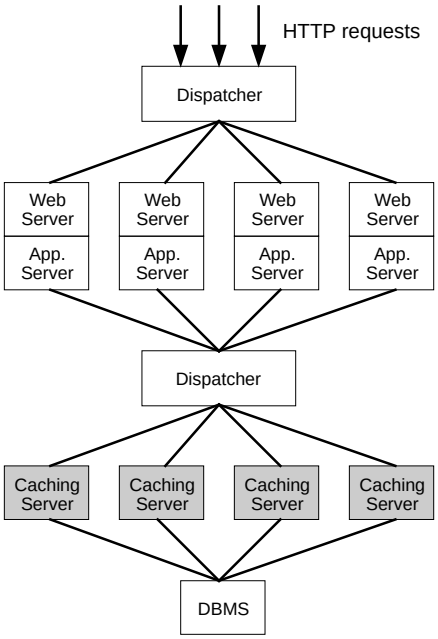

Fig. 1. The configuration of the Web sites adopting the query result caching systems

2.1 Configuration of Query Result Caching Systems

The caching system processes queries delivered from front-side WASs on behalf of a database server. It is located between WASs and a database server. For large-scale Web sites, the clustering technology is frequently applied to the cache systems as shown in Figure 1. The clustering improves total uptime (availability) and ability of adding resources to support more queries (scalability). In the clustering, it is important to equally distribute the amount of load to all cache nodes. A *dispatcher* (*e.g.*, L4 switch) ensures the uniform distribution among caches. It is installed between WASs and cache nodes. The dispatcher periodically monitors the current loads of the cache nodes. When receiving a query from a WAS, it redirects the query to the cache node with the lowest load. The dispatcher is also used to distribute HTML requests among Web servers and WASs.

2.2 Query Templates and Query Instances

In Web-based applications, a user usually submits a request by using a HTML form. Figure 2 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


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

Fig. 2. HTML form and its corresponding query form. The HTML form is clipped from the Search Request Web page of the TPC-W benchmark [19].

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*.

We characterize a query template QT as follows:

$$QT = (T, PA, JC)$$

- 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. Given a query template QT, we define a query group of the template 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.

3 Cache Consistency Mechanism

3.1 Architectural Overview

Under the proposed mechanism, a caching system conceptually consists of the *Consistency Maintainer* (CM) and the *Read-Query Processor* (RQP) (see Figure 3). 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. RPQ maintains meta information about query templates and instances stored in a cache.

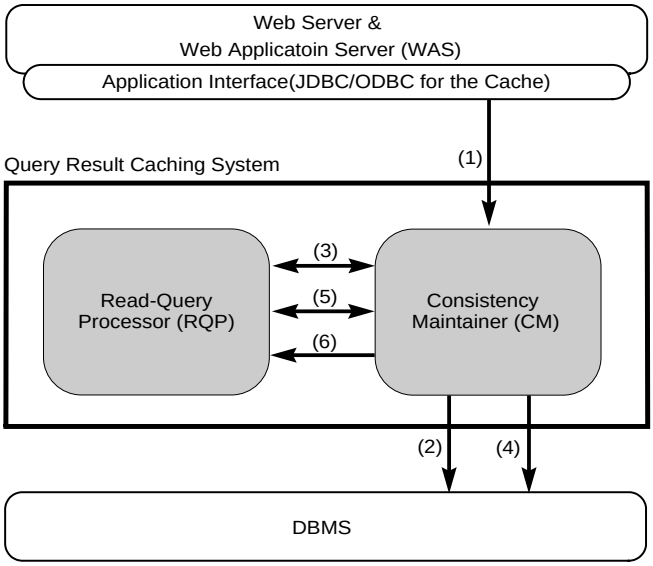

Fig. 3. Processing Flow

RPQ incrementally adds meta information on a new query template whenever encountering a new kind of a query. The meta information is used by CM for the consistency check.

Figure 3 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 3.2. Once the templates are determined, CM finds affected query instances in the templates (5) and removes them from the cache (6).

The following is the characteristics of the proposed mechanism.

- *Cache-driven Consistency Maintenance.* One of our design goals is to provide a consistency mechanism which can be easily deployed without much modification to existing Web sites. Thus, we would like to have the entire process of the consistency maintenance performed by the cache. As shown in Figure 3, the cache detects update queries, performs the consistency check, and invalidates affected query results.
- *Invalidation-every-update.* Upon receiving an update query, a cache immediately invalidates query results affected by the update. The invalidation is performed before the commit of the update in a database server. This policy ensures the strong consistency between cached query results and their corresponding origin data. Other alternative to ensure the strong consistency

is *polling-every-read*. Upon receiving a read query, this scheme performs the consistency check over the corresponding query results of the query. The scheme is not efficient in most Web sites where read queries are much more frequently issued than updates.

3.2 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.

The template check determines whether a given update may affect a set of query instances generated from a query template. This test checks a necessary condition; if a query template QT fails the test, none of the query instances generated from QT are affected. The template check examines projection attributes, base tables, and join conditions of templates. If a template passes the test, it is possible that some of the query instances are affected by the update. Thus, we match each instance to the update. In this step, only the selection region of each instance is matched against that of the update.

This two-phase consistency check is a scalable solution saving much of computation overhead. It performs a join condition check only once over each template. Note that join check is the most expensive step. It requires communication with and some computation in a database server.

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 in the example 2. 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.

Example 2. Let us reconsider the update u and the template QT from the example 1. In order to determine whether u satisfies the join condition of QT in the example 1, the following join check query is sent to a database server.

```
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.

We further reduce the overhead by observing the type of each update query. This is an advantage over [4], which finds queries affected by a *set* of updates. Thus, it cannot examine each update query. There are several cases where the join check can be omitted. First, if an update is to delete tuples, the join check can be skipped. All query results stored in a cache definitely satisfy their join conditions. Thus, in the instance check, when the deleted tuples do not satisfy the join conditions, there is no possibility to find any query results.

Second, if a referential integrity rule is applied to a join condition, a database server examines whether the newly inserted tuples are joined. In the example 1, assume that the referential integrity has been applied to the join condition `I_A_ID = A_ID`. If an update query tries to insert into `ITEM` the tuples that are not joined with tuples of `AUTHOR`, a database server denies the update. A cache terminates the process of the consistency check to the denied update.

Third, if an update is to modify the values of non-join attributes, it is intuitively obvious that the join check is needless. If an update is to modify the

values of join attributes, a cache treats the update as deleting the tuples with the old values and inserting the tuples with the new values. As mentioned above, for the deletion, the join check is not needed. Thus, a cache performs the join check only over the new values.

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.

A cache can quickly find affected query instances by using an index. The query result caching system usually includes an index to speed up searching for the query results matching an incoming query. It indexes query results by their selection regions. For more detailed information on the index, refer to [7].

If the domain of an update selection region and that of a query selection region are different, the selection region check is impossible. In the example 3, the domain of the update U (*i.e.*, `I_COST`) is different from that of Q (*i.e.*, `I_PUBLISHER`). Our solution to this problem is to retrieve the values of `I_PUBLISHER` of the deleted tuples before sending the update u. Now, a cache can compare the retrieved values to the selection regions of query instances.

Example 3. Given a query instance Q and an update U as follows,

```
Q : SELECT I_TITLE, I_COST, A_FNAME
    FROM ITEM, AUTHOR
    WHERE I_A_ID = A_ID
    AND I_PUBLISHER = 'McGrowHill'
U : DELETE ITEM WHERE I_COST < 1000
```

the instance check cannot be directly performed due to the different domains of their selection regions.

4 Issues in Clustered Cache Nodes

In this section, we discuss a couple of issues occurred in clustering caches and show our solutions.

4.1 Overhead of Join Checks

With multiple cache nodes, an origin database server would be easily overloaded due to multiple issues of join check queries. Different cache nodes may have different template information. Imagine that one cache node has encountered a query generated from a new query template QT, and the other node has not

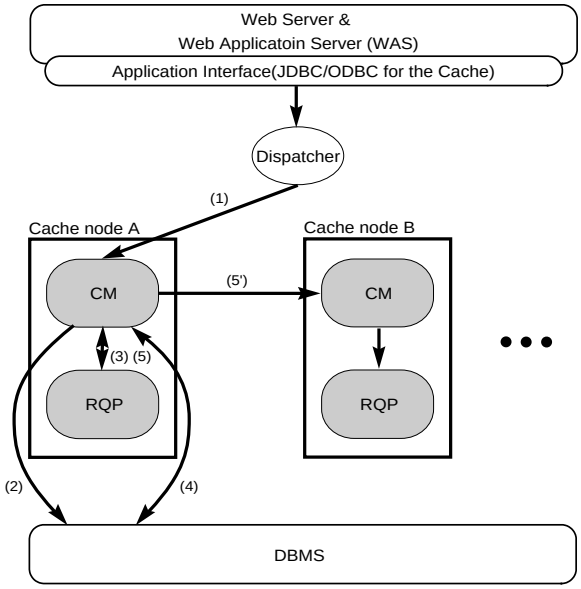

Fig. 4. Template check at a single node

received a query generated from QT. Therefore, nodes must issue their own join check queries to the database server. The overhead to the database server restricts the maximum number of nodes, *i.e.*, the scalability of a cache cluster.

To address this problem, we designed cache nodes to share the same template information. Under this situation, only one cache node which has received an update query performs the join check. Then, other nodes utilize the result. To share the same template information with other nodes, each node broadcasts the template information whenever encountering a new template. Other nodes then insert the new template to its own information.

Figure 4 shows the processing flow of an update query when all caches have the same template information. The cache node A receives an update query (1) and forwards the update to the database server (2). Then, the node A investigates local template information (3) and sends join check queries to the database server and receives the results (4). If the result is not null, it performs the instance check over its local cache (5) and simultaneously broadcasts the *instance check request* to other cache nodes (5'). Other cache nodes will perform the instance check over their own caches. We refer to this approach as the *template check at a single node* (TCS).

4.2 Temporal Inconsistency of Template Information

Even with the broadcast mechanism, the template information may become temporarily inconsistent among cache nodes. Figure 5 shows the situation. The

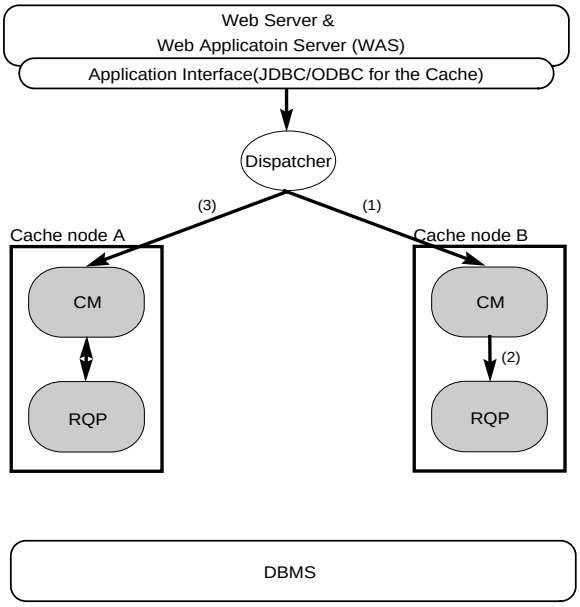

Fig. 5. A Case of template inconsistency among caches

node B receives a new kind of a read query (1), adds a new template information and stores the result of the read query sent from a database server (2). Before receiving the new template information from the node B, the node A receives an update query which affects the read query (3). Because the node A does not know the existence of the new template, it does not perform the template check for the new template. Consequently, the result of the read query in the node B becomes staled.

To resolve this problem, the node B invalidates all the query instances in the templates that have not been checked by the node A. The node B determines the unchecked templates by comparing local template information to that of the node A. If a template of the node B is not included in the template information of the node A, the template has not been checked. The node A broadcasts its template information with the instance check request. Note that the invalidation will rarely happen due to the small number of templates. The template inconsistency occurs only when a new template is added.

5 Performance Evaluation

5.1 Experimental Setup

Performance Metrics We measured the update throughputs of the query result caching system adopting the proposed mechanism. We sent update queries

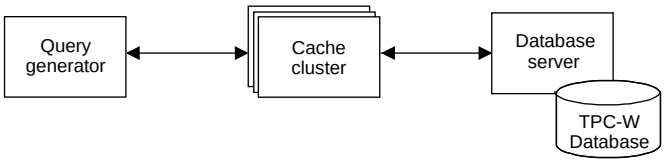

Fig. 6. Experimental setup

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.

Setup Figure 6 shows the setup for evaluating the proposed mechanism. We used three kinds of machines: one for *Query Generator*, another for the caching system and the third for the database server. 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 systems in the cluster run on the machine with a Pentium III 1GHz, 512M RAM. We implemented the proposed mechanism as a part of WDBAccel query result caching system [7]. All machines run Linux and are connected through a 100Mbps Ethernet.

Workloads We populated the database of the TPC-W benchmark in the database server at 100K scale [1]. The TPC-W benchmark [19] is an industrial standard benchmark to evaluate the performance of database-driven Web sites. It models an e-commerce site (specifically, an online bookstore) that is a representative, common Web application. We used the update query modified from one used in Admin Confirm Web page of TPC-W. The page includes a query which updates non-join attributes of the table `ITEM`. As mentioned in section 3.2, for such an update query, the join check is needless. Thus, we modified the query as follows. This query inserts information on a book into `ITEM`. The values of the attribute `I_ID` follow the random distribution.

```
INSERT INTO ITEM
   (I_ID, I_A_ID, I_COST, I_PUBLISHER)
VALUES
   (@I_ID, @I_A_ID, @I_COST, '@I_PUBLISHER')
```

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

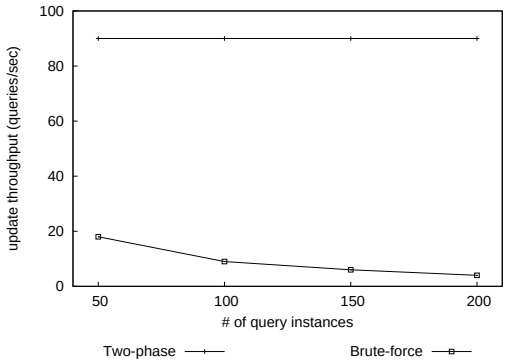

Fig. 7. Update throughputs of a single cache node

5.2 Experimental Results

Throughput of Two-phase Consistency Check 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 at a single cache node. 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.)

Figure 7 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 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.

Throughput of TCS Next, in order to identify the performance improvement by the TCS, we measured the update throughputs of TCS and those of *template check at every node* (TCE) at a cache cluster. In TCE, all cache nodes send a join check query. Both approaches use the two-phase consistency check within each cache node.

Figure 8 shows that the throughput of TCS does not change over the number of cache nodes. This is due to the fact that only one join check query is sent to a database server regardless of the number of nodes. On the contrary, the throughput of TCE significantly degrades as the number of nodes increases. When the cluster size is less than 5, throughputs of TCE is flat. This is because the disk I/O cost of a database server dominates the throughput. The CPU cost of a database server increases as the number of cache nodes. In the cluster sizes more than 4, the CPU limits the throughput. The disk cost does not increase due to a disk buffer.

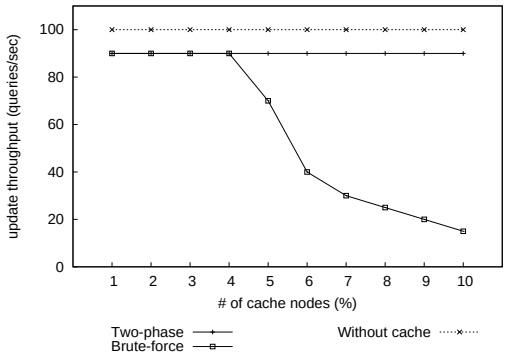

Fig. 8. Update throughput of a cache cluster

The throughputs without the cache consistency check are also shown in the figure. The degradation due to adopting the consistency check is about 10%. This overhead comes from the processing of join check queries. The overhead is inherent to the consistency maintenance. In TCS, the overhead stays fixed, while the overhead becomes larger in TCE as the cluster size increases.

6 Related Work

Recently, several consistency mechanisms have been proposed. Challenger *et al.* [6] proposed a solution in which application programs explicitly specify the mapping between an underlying data item and cached results affected by its update. At the time of writing, there exist various commercial caching products such as XCache [18] and SpiderCache [16]. They also provide consistency frameworks, however, still require the manual mapping.

Candan *et al.* [5] proposed an automatic solution. It finds without the application specifications (1) the mapping between cached results and queries used for generating these results and (2) the mapping between the queries and the underlying data that affect these queries. The invalidation-based approach was proposed by Candan *et al.* [4]. This approach tried to invalidate queries affected by updates as we did. The main difference from our mechanism is that it performs the selection region check on a database server as well as the join check. Thus, it incurs more overhead to a database server and the maintenance cost of keeping the selection regions of cached query results in a database server. Another difference is that this approach focused on invalidating queries affected by a *set* of updates. Our mechanism finds the queries affected by a *single* update. Depending the type of each update, our approach omits the join check, thus further reducing the overhead of consistency check.

Replication-based caching systems [3, 13, 15, 8] improve the performance of database processing by distributing the workload into multiple cache nodes. Those systems replicate a database into multiple nodes and distribute queries

among them. In these systems, the view maintenance techniques [12, 1, 10, 11] are used for maintaining cache consistency. It propagates updates to cache nodes immediately or periodically. Amiri *et al.* [2] proposed a filtering scheme that reduces update propagations to edge database caches. The filtering scheme exploits the template-rich nature of Web applications, however, does not consider join conditions of templates. For the query result caching systems, the view maintenance is avoided because it requires higher cost than invalidation. The view maintenance should build the same consequences as an underlying data. On the contrary, the invalidation just discards staled data.

7 Conclusions

In this paper, we proposed a update management mechanism for the query result caching systems. We addressed three important requirements for the mechanism: supporting strong consistency, enabling transparent deployment, and minimizing the overhead of a database server for the scalability. The proposed mechanism ensures strong consistency by invalidating affected query results before the completion of a database update. The mechanism is designed to be transparently deployed by performing an entire process of consistency maintenance at a cache. For the third requirement, we divided a consistency check to two phases, template check and instance check. The template check is performed over a query template, not individual instance. We also applied the mechanism to a cluster of caches. We presented the experimental results that verify a high level of the scalability of the mechanism.

References

1. D. Agrawal, A. El Abbadi, A. Singh, and T. Yurek. Efficient view maintenance at data warehouses. In *Proceedings of ACM SIGMOD Conference*, 1997.
2. 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.
3. 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.
4. 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.
5. 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.
6. Jim Challenger, Arun Iyengar, Karen Witting, Cameron Ferstat, and Paul Reed. A publishing system for efficiently creating dynamic web content. In *Proceedings of IEEE INFOCOM*, 2000.
7. Seunglak Choi, Jinwon Lee, Su Myeon Kim, Sungjae Jo, Junehwa Song, and Yoon-Joon Lee. Wdbaccel: A high-performance database server accelerator for database-driven web sites. *Under submission.*

8. Oracle Corporation. Oracle9ias cache.
http://www.oracle.com/ip/deploy/ias/index.html?cache.html.
9. Om P. Damani, P. Emerald Chung, Yennun Huang, Chandra Kintala, and Yi-Min Wang. ONE-IP: Techniques for hosting a service on a cluster of machines. In *Proceedings of the International World Wide Web Conference*, 1997.
10. Ashish Gupta and Inderpal Singh Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Bulletin of the Technical Committee on Data Engineering*, 18(2), June 1995.
11. Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proceedings of ACM SIGMOD Conference*, 1993.
12. Ki Yong Lee, Jin Hyun Son, and Myoung Ho Kim. Efficient incremental view maintenance in data warehouses. In *Conference on Information and Knowledge Management*, 2001.
13. Qiong Luo, Sailesh Krishnamurthy, C. Mohan, Hamid Pirahesh, Honguk Woo, Bruce G. Lindsay, and Jeffrey F. Naughton. Middle-tier database caching for e-business. In *Proceedings of ACM SIGMOD Conference*, 2002.
14. 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.
15. TimesTen Performance Software. Timesten library.
http://www.timesten.com/library/index.html.
16. Warp Solutions. SpiderCache technologies
http://www.warpsolutions.com/products/enterprise/indexent.htm.
17. Junehwa Song, Arun K. Iyengar, E. Levy-Abegnoli, and Daniel M. Dias. Architecture of a web server accelerator. *Computer Networks*, 38, January 2002.
18. XCache Technologies. XCache overview
http://www.xcache.com/home/default.asp?c=45&p=352.
19. Transaction Processing Performance Council (TPC). TPC benchmark^{TM}W (web commerce) specification version 1.4. February 7, 2001.
20. Khaled Yagoub, Daniela Florescu, Valerie Issarny, and Patrick Valduriez. Caching strategies for data-intensive web sites. In *Proceedings of the 26th VLDB Conference*, 2000.