A Fine-Grained Replacement Strategy for XML Query Cache *

Li Chen, Song Wang, Elizabeth Cash, Burke Ryder, Ian Hobbs and Elke A. Rundensteiner
Department of Computer Science
Worcester Polytechnic Institute, Worcester, MA 01609–2280
{lichen|songwang|lizesc|jryder|ianh|rundenst}@cs.wpi.edu

ABSTRACT

Caching popular queries and reusing results of previously computed queries is one important query optimization technique, especially in modern distributed environments such as the WWW. Based on the recent proliferation of XML data and the emergence of the XQuery language, we are thus developing a query-based caching system for XQuery queries, called ACE-XQ. ACE-XQ applies innovative query containment and rewriting strategies to answer incoming user queries based on the cached XQueries, whenever possible, instead of accessing remote XML data sources.

To manage the space of the cache, a straightforward application of traditional replacement strategies would correspond to removing a complete cached query and its derived XML document as a whole when space needs to be freed. This coarse granularity however does not match well with the typical access pattern of web searches where new queries often partially overlap with cached queries.

In this paper, we propose a novel replacement strategy appropriate for such query-based XML caching systems. In particular, we collect user access statistics at the granularity of the XML path structure instead of the complete XML query regions. We then apply a more fine-grained replacement strategy that purges XML fragments off a cached region instead of the whole XML document and accordingly adjusts the query descriptor. This may better capture the user access patterns since more frequently used XML document fragments are likely to remain in the cache while other less beneficial parts are purged. This approach has been implemented in our ACE-XQ System. Preliminary experiment results illustrate the performance improvement achievable by our fine-grained replacement strategy over the one which replaces a whole XML view at a time when the cache size is relatively large.

*This work was supported in part by the NSF NYI grant IIS-979624. Li Chen would like to thank IBM for the IBM corporate fellowship.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
WIDM'02, November 4–9, 2002, McLean, Virginia, USA.
Copyright 2002 ACM 1-58113-492-4/02/0011 ...$5.00.

Categories and subject descriptors: H.2.3 [DATABASE MANAGEMENT]: Languages—Query languages; H.2.4 [DATABASE MANAGEMENT]: Systems—Query processing

General Terms: Algorithms, Languages, Management, Performance

Keywords: XML, XQuery, Semantic Caching, Query Containment, Query Rewriting, Cache Replacement Strategy

1. INTRODUCTION

1.1 Background on Query Caching

Due to the growing demand by web applications for retrieving information from multiple remote XML sources, it becomes more critical to improve the efficiency of current XML query engines by exploiting caching technology to reduce the response latency caused by data transmission over the Internet. Inspired by the semantic caching idea [4], which utilizes cached queries and their results to answer subsequent queries by reasoning about their containment relationships, we propose to build such a caching system to facilitate XML query processing in the Web environment.

One major difference between semantic caching systems [4, 10] and the traditional tuple [12, 5] or page-based [2] caching systems is that the data cached at the client side of the former is logically organized by queries, instead of physical tuple identifications or page numbers. To achieve effective cache management, the access and management of the cached data in a semantic caching system is typically at the level of query descriptions. For example, the decision of whether the answers of a new query can be retrieved from the local cache is based on the query containment analysis of the new query and the cached ones, rather than by looking up each and every tuple or page identification of objects that could possibly answer a current user request. An important responsibility of cache management is to determine which data items should be retained in the cache and which ones should be replaced to make free space for new data, given limited cache space. Most naturally, the data granularity for replacement in a query-based caching system is the query and its associated result.

The semantic caching idea has been extensively studied in the relational context [4], where tuples satisfying the predicates imposed by a relational conjunctive query are stored in a table format in the cache. However, query evaluation and containment dealing with tree-structured XML data differ in their nature and difficulty from those in the relational setting. The essential capabilities of an XML query are *pattern matching* and *result construction* to extract the desired XML fragments from the complex nested XML data tree and then to properly organize the possibly nested view structure of the result XML document.

1.2 Introduction of ACE-XQ

To meet this challenge, we are developing the first XML query-based caching system, named ACE-XQ [8, 9], to realize more sophisticated query containment and cache management issues in the XML context. In ACE-XQ, queries and views are both expressed in XQuery, a quickly thriving XML query language proposed by W3C as the standard [18]. The query descriptions in the ACE-XQ system encode the query constructs necessary for capturing the semantics used in the decision for query containment. While [8] describes the XQuery containment and rewriting techniques in ACE-XQ, in this paper we focus on cache management of ACE-XQ, in particular the replacement strategy.

1.3 Drawbacks of Replacement at Query Level

Typically, the replacement manager helps to decide what to retain in the cache and what to discard in case of a full cache. Since subsequent queries are often conceptually subsumed by or overlapped with previously cached queries, the semantic regions may be managed in two ways. The first solution is to decompose the containing query into an overlapping part which corresponds to the *probe query* and a non-used part which is the difference between the original query and the probe query. In this scheme, a uniform utility value is maintained for each query. When the cache space is full, a victim query is picked by the replacement manager to be discarded from the cache to free room for the new query. However, such a solution entails a large decomposition overhead each time when a new query is overlapped with the cached queries. Also, it would result in more and more smaller region fragments over time which are less possibly useful in answering future queries.

An alternative solution is to tolerate reasonable redundancy in cached queries. Then a straightforward application of replacement would be to replace a complete query (the same one as originally cached) and its associated XML document at each iteration. However, the data granularity of a whole XML document handled in such a replacement strategy is too coarse for "large" size XML documents. This would impact the cache space utilization. Also, such a replacement strategy doesn't reflect the contribution of different XML regions in a cached XML document which may participate in answering different subsequent queries. The replacement in the granularity of complete XML documents hence suffers apparent drawbacks.

1.4 Our Partial Replacement Approach

We now propose a refined replacement strategy that records utility values for finer regions of existing cached views in terms of their internal structure rather than assigning a uniform value for the whole cached query. To be precise, we attach to each query region a detailed path table listing all paths returned in the query. When a cached query contains or partially overlaps a new query, the utility statistics of those paths requested by the probe query are updated, however without splitting the cached query. When the cache is full, the replacement manager chooses paths of query regions with the lowest utility value. It then composes a *filter query* to remove the XML fragments corresponding to those paths from the cached view document. The relevant query descriptors are also modified accordingly to be consistent with the changed result XML documents.

In this proposed replacement scheme, we utilize the view structure to maintain utility values at a finer granularity than complete query descriptions. This way, the replacement helps to maintain real "hot" queries (although in their refined descriptions) within the cache while it avoids query splitting which may cause too many small region fragments with little use for answering future queries.

1.5 Organization of the Paper

In the rest of the paper, we will explain the fine-grained replacement strategy (also called *partial replacement*) deployed in our XML query-based caching system – ACE-XQ. We have also implemented the complete region replacement strategy (called *total replacement*) and compare both strategies. Our experiments show that when the cache is relatively large, the *partial strategy* results in a better cache performance in terms of query response delay and hit ratio than the *total replacement*.

2. RUNNING EXAMPLE OF XQUERY CONTAINMENT AND REWRITING

Previous work on query containment has considered queries expressed in either relational algebra or datalog. We say that $Q_1 \subseteq Q_2$ if and only if there is a *containment mapping* from Variables (Q_1) to Variables (Q_2) [3]. However, the research for XML query containment is still in its infancy due to that the complexity of the problem comes with the extra expressiveness of pattern matching based on XML hierarchy and result construction. For example, a user may want to use Q_1 (shown in Figure 1) to find in the `bib.xml` document the books that are highly rated (4.5 out of 5) by another resource `reviews.xml`. Suppose the returning result XML document is stored as *Q1Res.xml* in the cache. It contains a list of such `goodbooks` with their `title`, `author` and `price` information.

```
FOR $b IN doc("http://bib.com/bib.xml")//book,
    $b' IN doc("http://reviews.com/reviews.xml")//book
WHERE $b/title = $b'/title AND $b'/review/rate > 4.5
RETURN <goodbook>$b/title, $b/price
          </goodbook>
```

Figure 1: An Example XQuery Q_1

Q_1 performs a "join" of two XML data sources conforming to different DTDs. The traditional containment mapping [3] is not directly applicable to such tree structure-based XML queries since mapping atoms in relational queries are flat relations and attributes.

Due to the tree structure-based XML queries, we have proposed a framework for XQuery containment [8]. The main idea is to first normalize all queries into a format which explicitly reveals the variable dependencies, and then to perform a progressive containment mapping on the variables of a new query and a cached query. Since variables are specified using regular path expressions, the containment mapping process incorporates type inference and subtyping mechanisms for regular expression types to check the subsumption relationships between variable types. Here, we briefly explain our containment mapping technique using an example, while more details can be found in [8].

```
FOR $a IN doc("http://bib.com/bib.xml")//author[last = "Wang"]
RETURN <book_byauthor>$a,
     <books>
     FOR $b IN doc("http://bib.com/bib.xml")//book[author = $a],
          $b' IN doc("http://reviews.com/reviews.xml")//book
     WHERE $b/title = $b'/title AND $b'/review/rate > 4.5
     RETURN $b/title, $b/price
     </books>
</book_byauthor>
```

Figure 2: A New XQuery Q_2

Suppose the user now issues a new query Q_2 as shown in Figure 2 to find among the highly rated books the ones authored by the persons with the last name "Wang". The `title` information of such books is returned, and then grouped by the authors. There are two joins involved in Q_2. One join is between the `book` elements in `bib.xml` and those in `reviews.xml`, and the other is a self-join between the `author` elements bound to $a in one pass of `bib.xml` and the `author` elements related to $b (`book` elements) bound in another pass of the same document. Obviously, we can re-use the cached query Q_1 to save the computation cost for the first join of Q_2.

To retrieve the answers contained in Q_1, the remaining part from the remote XML document and to combine them to form the final result for Q_2, ACE-XQ generates the *probe, remainder* and *result combination* queries respectively, as shown in Figure 3.

```
FOR $b IN doc("Q1Res.xml")//goodbook
RETURN ⟨result⟩$b/title, $b/price
⟨/result⟩
                                    Probe Query
```

```
FOR $a IN doc("http://bib.com/bib.xml")//author[last = "Wang"]
RETURN ⟨result⟩$a, ⟨entrys⟩
    FOR $b IN doc("http://bib.com/bib.xml")//book[author = $a]
    RETURN $b/title ⟨/entrys⟩
⟨/result⟩
                                    Remainder Query
```

```
FOR $r IN doc("remainderRes.xml")//result, $a IN $r/author,
RETURN ⟨book_byauthor⟩$a, ⟨books⟩
    FOR $t IN $r/entrys/title, $b IN doc("probeRes.xml")//result
    WHERE $b/title = $t'
    RETURN $b/title, $b/price ⟨/books⟩
⟨/book_byauthor⟩
                                    Result Combination Query
```

Figure 3: The Probe, Remainder and Combining Queries for Answering Q_2 Using Q_1

3. THE ACE-XQ SYSTEM OVERVIEW

In this section, we briefly describe the ACE-XQ system. The framework of the ACE-XQ system is depicted in Figure 4. It mainly consists of two subsystems, a *Query Matcher* and a *Cache Manager*. The *Query Matcher* subsystem implements the query containment and rewriting, and the *Cache Manager* manages the cache space and applies replacement and coalescing techniques.

When a new user query comes in, the **Query Decomposer** applies normalization rules [7, 8] to derive its nesting format, revealing the variable dependency hierarchy specified in the query's matching patterns. It further re-groups the conditions and return expressions centering around their referring variables to form variable-specific sub-queries. The **Query Pattern Register** encodes the semantics of a query and registers them as the query descriptor. For a pair of new and cached queries, the **Query Containment Mapper** explores containment mappings between their variables. It makes the query containment decision depending on whether one-to-one containment mappings can be established. Type inference and sub-typing mechanisms are utilized for this containment mapping. Based on the established containment mappings, the **Query Rewriter** rewrites the new query with respect to the view structures of the cached queries. Thus the user's new XQuery is divided into a *probe query* to retrieve answers from the cached local views, and a *remainder query* to obtain the remaining answers from remote sources.

The *Cache Manager* of ACE-XQ manages a collection of semantic regions, each identified by an encoded *query descriptor* that captures the semantics of the cached view. Each region also has a pointer referring to its cached *view XML document*. Query descriptors are utilized by the containment mapper to determine query containment relationships between the new query and cached queries. The corresponding XML document is then accessed by the probe query to retrieve relevant answers. Besides the query descriptor, a semantic region usually also has some associated user access statistics. This allows the **Replacement Manager** to calculate the region utility values in order to pick victim queries when there is no room for caching a new query.

The remainder of this paper will focus on the description of a *partial replacement* strategy utilizing the user access statistics recorded at the path level to perform a fine-grained region purging, as opposed to a *total replacement* strategy which replaces a complete query and its associated XML document at each iteration. As our preliminary experiments illustrate, this *partial replacement* helps to improve the cache performance over time.

The **Region Coalescer** uses some semantic locality algorithm to discover "adjacent" queries so to merge them into a combined region. In contrast to the replacement manager, coalescing results in no deletion. It is more a heuristic to optimize the cached regions by reasoning about the semantic distance between queries.

4. CACHE REGION MANAGEMENT IN ACE-XQ

4.1 Region-preserving vs. Region-splitting Replacement

When a new query arrives, the containment mapper decides if it is contained or partially overlapping with a cached query. If yes, a probe query (`PQ`) is formulated to access the cached data which satisfies the new query and thus will contribute to the answer. If not all the desired data is stored in the cache, a remainder query (`RQ`) will also be sent out to remote servers to fetch the rest of the answers. Obviously, cached queries may logically be segmented by probe queries upon the arrival of new queries. Below we describe alternative solutions for maintaining the regions.

Some semantic caching systems allow redundancy between cached queries and do not adjust cache regions. In this scenario, queries are kept the same as when they first come to the cache system and they do not split even when subsequently cached queries overlap with them. For each such cached query, one uniform utility value is maintained.

Some systems split a cached query Q into two regions. One corresponds to `PQ` and the other represents `Q−PQ`. The latter region inherits its utility value from its parent from where it was split off, while the former is marked with an increased utility value compared to the original cached query. The cache manager combines `PQ` and `RQ` both used for answering the new query and forms one new region.

Some other systems prefer to keep the earlier cached query as a whole while only allocating one new region for `RQ` (since `PQ` is already contained in the existing cached query). Although the region maintaining schemes used in the latter two scenarios can help to reduce redundancy, they often result in too many smaller region fragments over time which tend to be less useful in answering further queries. Also, such strategies entail query region splitting overhead each time when a new query is launched, and later frequent coalescing to make up for the fact that the cache space has been severely fragmented.

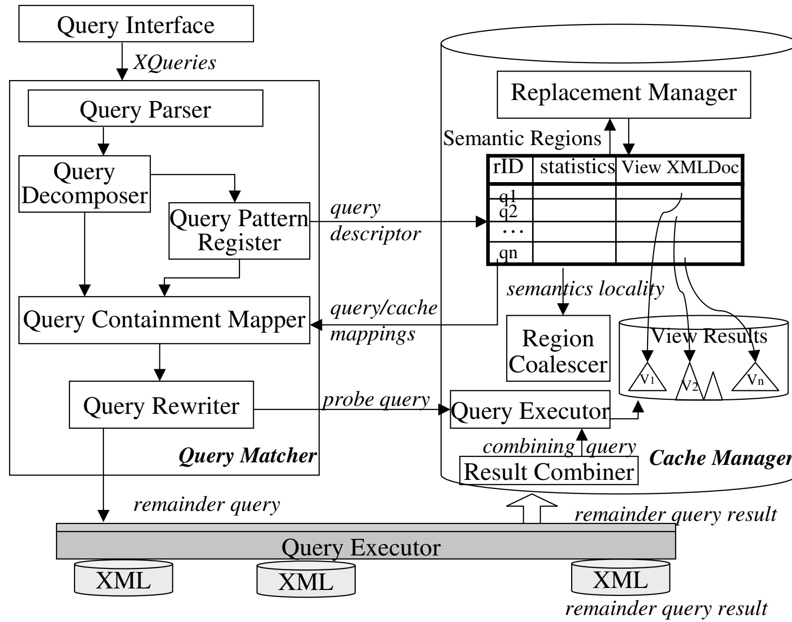

Figure 4: The ACE-XQ System Architecture

Q_1

XPathRow	hit_count	last_access_time	#_of_objects	...
/goodbook/title	1	12:33pm May 30	40	
/goodbook/author/last	1	12:33pm May 30	86	
/goodbook/author/first	1	12:33pm May 30	86	
/goodbook/price	1	12:33pm May 30	132	

Figure 5: Path Table with Initial Statistics for Q_1

Within the limit of tolerant redundancy, the region-preserving strategy avoids the computation overhead compared to the region-splitting strategy. However, it has its own drawbacks as well. First, the uniform utility value assigned for the whole region does not precisely indicate the various contributions made by different parts in answering subsequent queries. Second, a straightforward application of replacement would replace a complete region at a time. Such a replacement unit would be too coarse for "large" size XML documents. This would result in less efficient cache space utilization.

4.2 Cache Region with Associated Path Table

To overcome the drawbacks implied by a naive region-preserving replacement strategy, we suggest that despite the query region still being preserved, different utility values may be maintained for finer parts within a region by utilizing the internal view structure. In other words, we attach with each region descriptor a detailed *path table* containing all complete XPaths in the corresponding result XML document. Each such XPath corresponds to a row in the path table, hence referred to as *XPathRow*. The statistics related to the user access information (as explained in the following section) form the columns of the path table. The XPathRows of such a path table can be easily constructed based on the *return* clause of a query, which reveals the result view document structure and thus how it is composed of XML fragments extracted from the original XML document.

When a new query overlaps with a cached query, the probe query `PQ` is formulated to retrieve the relevant data via navigating along XPaths in the cached XML document. We maintain the utility statistics at the level of XPathRows, and update those involved in the probe query. Figure 5 shows the path table structure for the query descriptor Q_1 in Figure 1. We use complete XPaths of leaf objects in an XML document, thus avoiding a potential size explosion of the path table.

When Q_2 is answered by ACE-XQ, its region is also constructed in the cache (see bottom of Figure 6). The replacement manager modifies the statistics for Q_1 since the current probe query has been posed against its view structure. Figure 6 shows the cache region status after Q_2 is cached. The XML fragments along two paths, `/goodbook/title` and `/goodbook/price`, within Q_1's result document contribute to answering Q_2.

5. THE PARTIAL REPLACEMENT STRATEGY

5.1 Utility Value and Replacement Function

Utility value is usually considered as the indicator for the replacement likelihood of cached objects. Based on the collected statistics, a caching system may adopt a particular replacement policy in favor of purging some cached objects with certain characteristics illustrated via their statistics over the others. A *replacement function* is often used to reflect the replacement preference of a caching system. It calculates the utility values of cached objects,

Q_1

XPathRow	hit_count	last_access_time	#_of_objects	...
/goodbook/title	**2**	**12:47pm May 30**	40	
/goodbook/author/last	1	12:33pm May 30	86	
/goodbook/author/first	1	12:33pm May 30	86	
/goodbook/price	**2**	**12:47pm May 30**	132	

Q_2

XPathRow	hit_count	last_access_time	#_of_objects	...
/book_byauthor/author	1	12:47pm May 30	7	
/book_byauthor/books/title	1	12:47pm May 30	15	
/book_byauthor/books/price	1	12:47pm May 30	21	

Figure 6: Q_1's Path Table with Updated Statistics and Q_2's Path Table

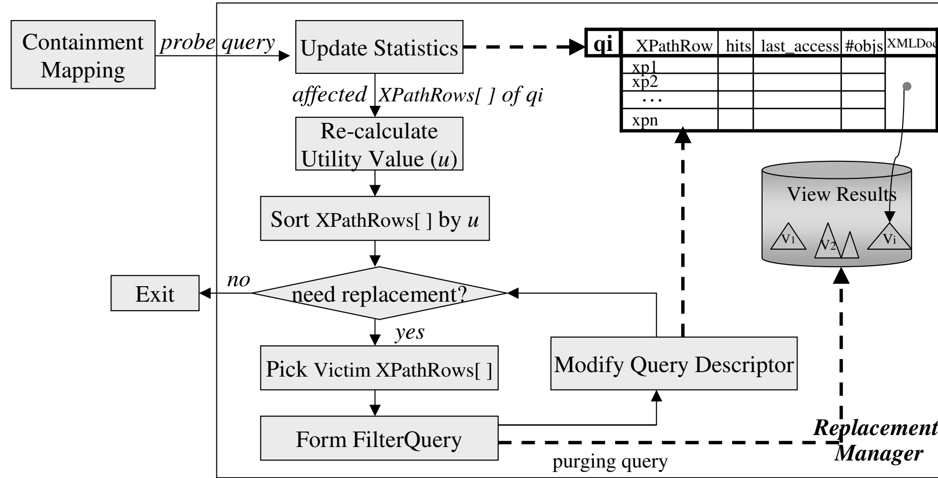

Figure 7: The Replacement Control Flow

based on which the replacement manager chooses the victim to be purged to leave room for new objects.

Cache replacement policies have been extensively studied in different scenarios, such as page-based [2] and tuple-based [5] caches. Various replacement schemes [17, 14, 1, 16, 11] have been investigated and applied based on different heuristics. Among them, the well-known replacement schemes are the Least Recently Used (LRU), the Least Frequently Used (LFU) schemes and their varieties. The LRU scheme is widely used due to its simplicity while still being effective when recently referenced objects are likely to be re-referenced in the near future. The LFU policy [17] uses reference frequency instead of recency as the parameter for the replacement function.

For the Web context, other replacement functions have been proposed to address the size and latency concerns. Among them, GreedyDual [13] is a simple yet popular algorithm which handles variable-cost cache replacement. One of its extended versions GreedyDual-Size [15] combines locality, size and latency cost concerns to achieve a better performance in terms of hit ratio and latency reduction.

For a semantic cache in the relational model (eg. [4]), cache hits are typically recorded on the basis of query regions. Some further approaches adopt region-splitting strategies to decompose the original cache region and then update the hits or recency value on different smaller regions after the split.

The goal of our effort is not to propose new replacement functions for calculating utility values. Rather, we adapt the existing replacement functions for the ACE-XQ system. However, we utilize the detailed path tables to perform a finer granularity replacement than replacing a complete query region at a time. That is, the input to the replacement function is not the statistics recorded at the whole query level, but those at the level of the internal path structure of view documents. For example, **last_access_time** is a timestamp recorded when a path is used in the latest probe query. The **hit_count** is the number of times a path has been used for answering subsequent queries. The **#_of_objects** is an estimation of the number of leaf objects along that complete XPath. If a path were to be selected as the next victim, this number gives a hint about how large an XML fragment would be purged from the XML document. We also keep track of more global statistics at the query level, such as **xml_doc_size**, **initial_create_time**, **fetching_delay_cost**, etc.

The considered statistics can be classified into several categories. One category concerns the recency and frequency values, such as the *last_access_time* and *hit_count* respectively. The *hit_count* on an XPathRow is increased by one each time when the XPath is requested by a probe query. Its *last_access_time* is updated to the current time in such a case. The second category is related to the data size that would be free upon a purge, i.e., the *#_of_objects* on a particular XPath and the *xml_doc_size*. If two groups of XPathRows have a tie in their recency and frequency values but one has a larger XML fragment associated with it than the other, the former group with the larger *#_of_objects* is replaced. This is because our fine-grained partial replacement strategy may need to perform path-related-region subtraction several times to free enough space for a new region. Hence our replacement function is in favor of purging a larger piece at a time for efficiency. Instead of considering the benefits brought by preserving a region, the *fetching_delay_cost*

is a factor indicating the loss caused by not caching a region. By retaining regions with longer initial fetching delay, large fetching cost for such regions could be avoided.

Based on the collected statistics, we propose to calculate the comprehensive utility value using the following replacement function: $rp_fun = (\frac{(hit_count \times fetching_delay_cost)}{\#_of_objects})$. Other functions in favor of different scenarios can be easily plugged into our system. We have indeed experimentally studied several such functions when designing the just described formula.

5.2 The Partial Replacement Algorithm

Figure 7 shows the control flow of the replacement manager in ACE-XQ. After the statistics information has been updated for those XPathRows involved in a probe query, the pre-defined replacement function is called to re-calculate their utility values. Then the replacement manager chooses those XPathRows with the lowest utility value as the victim XPathRows and composes a *filter query* to remove the XML fragments corresponding to these paths from the relevant XML document(s). The query descriptors of those affected cached queries are also modified accordingly to be consistent with their changed view XML documents.

Filter Query. Suppose ten queries Q_1 to Q_{10} are in the cache after the cache has been in use for a while. Different utility values are recorded in these queries' path tables. Now a query Q_{11} arrives and there is not enough space in the cache for it. Based on the lowest utility value, the replacement manager decides $victimQ$ and $victimXPathRows[\,]$ are Q_1 and *[*$//goodbook/author/first$, $/goodbook/price$*]* respectively (assuming the statistics of Q_1 and Q_2 are not the same as illustrated in Figure 6 any more). To remove $victimXPathRows[\,]$ from $victimQ$, we instead use a query to keep the remaining paths $remainingPaths[\,]$ which are the paths in Q_1's path table except those included in $victimXPathRows[\,]$. Since the $remainingPaths[\,]$ of Q_1 in this case are *[*$//goodbook/title$, $/goodbook/author/last$*]*, a filter query is formulated as in Figure 8.

6. EXPERIMENTAL RESULTS

We have implemented our ACE-XQ [8] in Java 1.3. We utilize the Quilt parser and Kweelt query engine available at: *http://cheops.cis.upenn.edu/Kweelt* to analyze and evaluate the input XQuery. To realize the type-enhanced query containment and rewriting algorithm, we deploy the type inference and subtyping mechanisms provided by the XDuce system [6] in ACE-XQ.

We install the Kweelt query engine on a local UNIX machine which holds the ACE-XQ system, and another one on a remote server where a set of XML documents are also hosted. We have validated the correctness of our queries rewritten by ACE-XQ via comparing their results with those produced by directly evaluating the original query against the remote documents. ACE-XQ also serves as a testbed for various experimental studies investigating the query performance gain achieved by answering queries using cached views.

6.1 Experiment on Caching versus Non-Caching

First, we use a number of fixed queries to "warm up" the cache and design the new incoming queries to be either totally contained in the cached queries or partially overlapping with them. The cache capacity is assumed infinitely large so that no replacement occurs. We compare the query performance in terms of the response delay when using the ACE-XQ system to answer queries versus when directly fetching the result from the remote server. Figure 9, consistent with our expectation, shows that for such contained cases, the query performance improvement using ACE-XQ is significant, by an order of magnitude in a distributed environment.

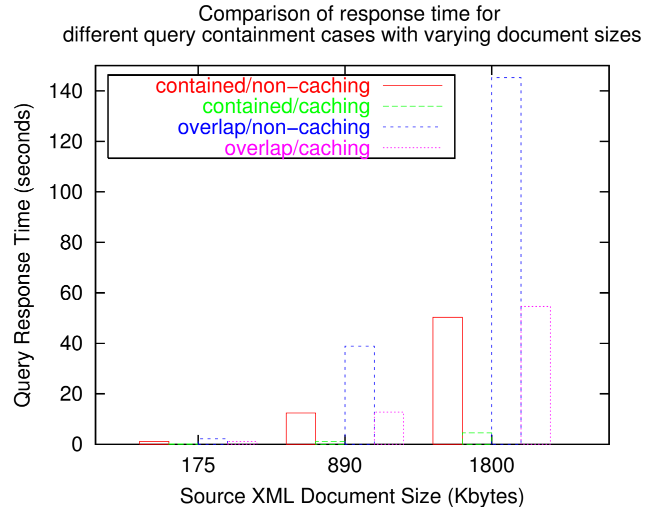

Figure 9: Query Response Delay for Totally Contained Queries

6.2 Experiments on Replacement Strategies

We now study the impact on the performance of ACE-XQ when the partial versus total replacement strategies are used for limited cache space. We generate two query traces. Each query trace has 80 queries that may query against 10 different XML documents (in total about 2M bytes) located on remote web servers. Queries in the first trace are randomly generated. The second query trace contains only refining queries, i.e., subsequent queries that are usually contained in the previous queries requesting the same documents. Initially, we set the cache space as large as 200K bytes and then increase it each time by 180K bytes until it reaches 1.2M bytes.

In the experiments, we consider the performance in terms of not only the *response delay*, but also the *hit ratio*. Each time when a new query is contained or overlapping with the cache and hence a probe query is generated, we consider it a cache *hit* and the percentage of hits over the total number of queries *hit ratio*. Since a probe query is a local query and the response delay is relatively smaller than queries across the Internet, the higher the hit ratio is, the more improvement on the average initial response to the user's query can be achieved.

Figure 10 shows the average response time for the random and refining query traces under two different replacement strategies: partial versus total. Overall, we see the average response time for the refining query trace is smaller than that for the random query trace. The response times for both traces are decreasing when the cache size becomes large. Within each query trace, two replacement strategies have roughly very close query response time, although the partial replacement does slightly better (by 5%) when the cache size is medium (about 614.4Mbytes).

As shown in Figure 11, the hit ratio increases with the growth of the cache size for both replacement strategies and for both query traces. Although the partial replacement strategy does not show obvious advantage over the total replacement in either traces, it does win a little over the latter when the cache size is not too big nor too small. We interpret this phenomenon as below. In the case of a very small cache, fewer queries are held in the cache which may

```
FOR $r IN document("Q1Res.xml")
FILTER (./self :: goodbook | ./self :: title || ./self :: text()[./parent :: title]
        ./self :: goodbook | ./self :: author | ./self :: last | ./self :: text()[./parent :: last])
RETURN $r
```

Figure 8: An Example Filter XQuery

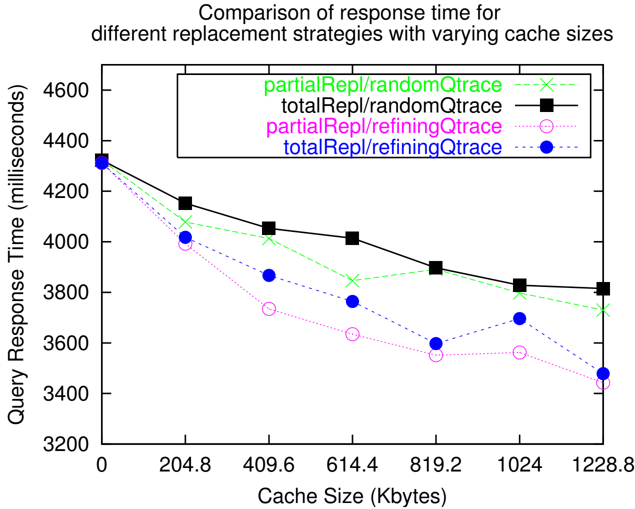

Figure 10: Response Time for Different Replacement Strategies With Varying Cache Sizes

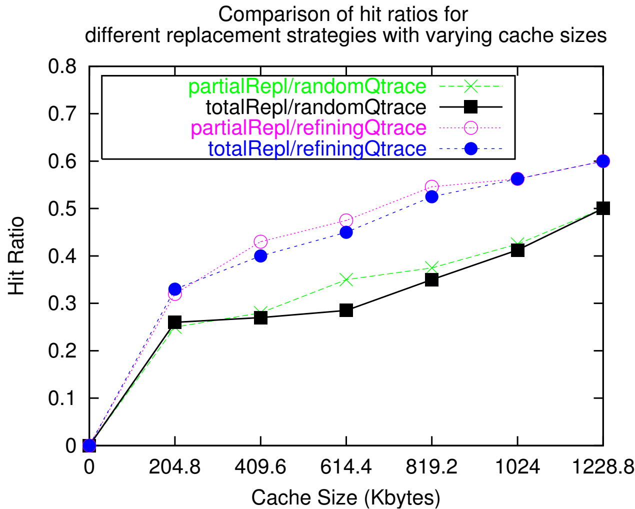

Figure 11: Hit Ratio for Different Replacement Strategies With Varying Cache Sizes

cause the replacement to occur more frequently. When a partial replacement happens, it attempts several rounds of partially purging less-useful XML fragments from the cached XML documents, which may still leave not enough space for the new query. More likely, some XML document may need to be removed as a whole to make enough space for the new query. This behaves similar to total replacement, except that more replacement efforts are wasted at the beginning. For a medium cache, such frequent replacement may not happen and the cache may hold more queries for the partial replacement case than for the total replacement case since the queries in the former case may be "0" by filter queries and hence relatively smaller than those in the latter case. While for the very large cache, cache space resource is not precious any more. Query region refinement pursued by partial replacement becomes less critical.

7. CONCLUSION

We have proposed a fine granularity replacement strategy and deployed it in our ACE-XQ XML query caching system. As opposed to the *total replacement* at the query level, this strategy maintains utility values for the XPaths in a cached view and performs the *partial replacement* that discards the non-beneficial XML fragments while retaining the useful portions within the view XML document.

We have also conducted experiments to compare the performance of the partial replacement strategy and the total replacement strategy, when varying the cache size. The experiment results have revealed some interesting patterns between the performance of different replacement strategies and the cache sizes. Extensive experiments need to be conducted in the future work to result in a more comprehensive performance analysis.

8. REFERENCES

[1] P. Cao, E. W. Felten, , and K. Li. Application-Controlled File Caching Policies. In *Proceedings of the USENIX Summer 1994 Technical Conference*, pages 171–182, 1994.

[2] M. Carey, M. Franklin, and M. Zaharioudakis. Fine-grained Sharing in Page Server Database System. In *Proceedings of 1994 ACM SIGMOD, Bombay, India*, pages 359–370, June 1994.

[3] A. K. Chandra and P. M. Merlin. Optimal Implementations of Conjunctive Queries in Relational Data Bases. In *STOC*, pages 77–90, 1977.

[4] S. Dar, M. J. Franklin, and B. Jonsson. Semantic Data Caching and Replacement. In *VLDB, Bombay, India*, pages 330–341, 1996.

[5] D. DeWitt, P. Futtersack, D. Maier, and F. Velez. A Study of Three Alternative Workstation-Server Architectures For Object-Oriented Database Systems. In *VLDB, Queensland, Australia*, pages 107–121, Aug. 1990.

[6] H. Hosoya and J. Vouillon and B. C. Pierce. Regular Expression Types for XML, Montreal, Canada. In *ICFP*, pages 11–22, 2000.

[7] I. Manolescu and D. Florescu and D. Kossmann. Answering XML Queries on Heterogeneous Data Sources. In *Proceedings of the 27th VLDB Conference, Edinburgh, Scotland*, pages 241–250, 2001.

[8] L. Chen and E. A. Rundensteiner. ACE-XQ: A CachE-aware XQuery Answering System. In *Proceedings of the 5th International Workshop on the Web and Databases (WebDB), Madison, WI*, pages 31–36, 2002.

[9] L. Chen and E. A. Rundensteiner and S. Wang. XCache - A Semantic Caching System for XML Queries. In *SIGMOD demonstration paper, Madison, WI*, page 618, 2002.

[10] L. M. Haas and D. Kossmann and I. Ursu. Loading a Cache With Query Results. In *Proceedings of the 25th VLDB Conference, Edinburgh, Scotland*, 1999.

[11] D. Lee, J. Choi, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. On the Existence of a Spectrum of Policies that Subsumes the Least Recently Used (LRU) and Least Frequently Used (LFU) Policies. In *Proceedings of the 1999 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 134–143, 1999.

[12] M. J. Carey and M. J. Franklin and M. Zaharioudakis. Fine-Grained Sharing in a Page Server OODBMS. In *Proceedings of the 13rd SIGMOD Conference, Minneapolis, Minnesota*, pages 359–370, 1994.

[13] N. Yong. On-line caching as cache size varies. In *In the 2nd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 241–250, 1991.

[14] E. J. O. Neil, P. E. O. Neil, and G. Weikum. The LRU-K Page Replacement Algorithm for Database Disk Buffering. In *SIGMOD*, pages 297–306, 1993.

[15] P. Cao and S. Irani. Cost Aware WWW Proxy Caching Algorithms. In *Proceedings of USENIX Symposium on Internet Technologies and Systems (USITS)*, pages 193–206, 1997.

[16] V. Phalke and B. Gopinath. An Inter-Reference Gap Model for Temporal Locality in Program Behavior. In *Proceedings of the 1995 ACMSIGMET-RICS Conference on Measurement and Modeling of Computer Systems*, pages 291–300, 1995.

[17] J. T. Robinson and M. V. Devarakonda. Data Cache Management Using Frequency-Based Replacement. In *SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 134–142, 1990.

[18] W3C. XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/, December 2001.