Replacement Strategies for XQuery Caching Systems ⋆

Li Chen, Song Wang, and Elke A. Rundensteiner

CS Department, Worcester Polytechnic Institute, Worcester, MA 01609–2280

Abstract

To improve the query performance over XML documents in a distributed environment, we develop a semantic caching system named ACE-XQ for XQuery queries. ACE-XQ applies innovative query containment and rewriting techniques to answer user queries using cached queries. We also design a fine-grained replacement strategy which records user access statistics at a finer granularity than the complete XML query regions. As a result, less frequently used XML view fragments are replaced to maintain a better utilization of the cache space. Extensive experimental results illustrate the performance improvement achieved by this strategy over the traditional one for a variety of situations.

Key words: XML; XQuery; Cache Replacement; Semantic Caching; Query Containment.

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 has become increasingly critical to improve the efficiency of XML query evaluation. One key step towards achieving such an optimization is to exploit caching technology to reduce the response latency caused by data transmission over the Internet. Inspired by the semantic caching idea [14],

⋆ 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.

Email addresses: lichen@cs.wpi.edu (Li Chen), songwang@cs.wpi.edu (Song Wang), rundenst@cs.wpi.edu (Elke A. Rundensteiner).

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 [14,21,6] and the traditional tuple [16] or page-based [6] 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 thus 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 query descriptors themselves, rather than by looking up each and every tuple or page identification of objects that could possibly answer a current user request.

The semantic caching idea has been extensively studied in the relational context [14]. However, query evaluation and containment dealing with XML data differ in their nature and difficulty from those in the relational setting. New challenges are being imposed by the tree-oriented nature of XML and the XQuery language on the tasks of query containment and rewriting, as we will point out in this paper.

1.2 Introduction of ACE-XQ

We have developed the first XQuery-based caching system, named ACE-XQ [9,11], to deploy our proposed query containment and cache management techniques in the XML context. In ACE-XQ, new and cached queries are both expressed in XQuery, a quickly thriving XML query language proposed by W3C as the standard [44]. The query descriptors in the ACE-XQ system help to capture the query semantics which are utilized in the decision for query containment. While [9] describes the XQuery containment and rewriting techniques in ACE-XQ, in this paper we focus on cache management of ACE-XQ, in particular cache replacement issues.

Typically, a cache system utilizes a replacement manager to decide what to retain in the cache and what to discard in case of a full cache. In a query-based caching system, the data granularity for replacement is the query and its associated query result. The cache manager in ACE-XQ maintains a collection of *query regions*, each composed of a *query descriptor* and the corresponding *XML view document*, i.e., *query region = query descriptor + result XML view*. Query descriptors can be utilized for reasoning about the containment relationships between the cached queries and the new query. Also, user access statistics information may be attached to the query descriptors by the deployed replacement strategy to calculate the region utility values. The replacement manager usually picks the cached query with the lowest utility value and purges it to make room for the new query.

1.3 *Drawbacks of Replacement at the Query Level*

Since a new query is often conceptually subsumed by or overlapping with previously cached queries, the query region of the latter can be seen logically segmented into two pieces. One corresponds to the overlapping part which is to be retrieved by the *probe query* for answering the new query. The left-over piece does not contribute to answering the new query. The replacement manager of a traditional query-based caching system may split the containing query region into two regions corresponding to their respective usefulness in this latest query answering process. After the splitting, a uniform utility value is then maintained for each query region. Whenever the cache is full, a complete query region would be the unit for replacement. However, such a region-splitting scheme entails a large decomposition overhead each time when a new query overlaps with the cached queries. Also, it would result in more and more smaller XML view documents over time which are possibly less useful in answering future queries due to their fragmentation.

An alternative solution is to tolerate some redundancy in the cached queries. That is, even if newly incoming queries partially overlap with existing queries, we would opt to not split existing queries in order to avoid fragmentation. Then a straightforward application of replacement would be to replace a complete query region at each iteration. However, the data granularity of a whole query region being deleted each time in such a replacement strategy may be too coarse for "large" XML views. This would impact the cache space utilization. Also, such a replacement strategy doesn't reflect the contribution of different fragments in a cached XML view which may participate in answering different subsequent queries. Replacement at the granularity of complete XML views hence suffers apparent drawbacks.

1.4 *Our Partial Replacement Approach*

We now propose a refined replacement strategy, namely, to record 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 region [12]. To be precise, we attach to each query descriptor a detailed path table listing all paths returned in the query. When a cached query contains or partially overlaps with 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 does not select complete regions but only specific paths with the lowest utility value within such query regions for replacement. It then composes a *filter query* to remove the fragments corresponding to those paths from the cached XML view. The relevant query descriptors are then modified accordingly to be consistent with the changed XML view.

This proposed *partial replacement* strategy utilizes the view structure to maintain utility values at a finer granularity than complete query regions. This way, the replacement helps to maintain in the cache the most likely "hot" query regions. This is because the original cached queries may be refined by future filter queries that remove the less useful fragments within them. It hence forgoes the explicit region splitting upon every new incoming query, avoiding the generation of too many small region fragments with little use for answering future queries.

We have also implemented both the proposed *partial replacement* strategy as well as the complete region replacement strategy (which we now call *total replacement*) within our ACE-XQ caching system. In this paper, we now report upon the extensive experimental study we have conducted to compare the performance of our partial replacement and the alternative total replacement strategies in a variety of scenarios. The results show that in most cases especially when the cache size is medium, the partial replacement strategy outperforms the total replacement strategy in terms of hit count ratio, hit byte ratio and query response delay.

1.5 Organization of the Paper

The rest of the paper is organized as follows. In Section 2, we show the running example queries to motivate the need for a query containment and rewriting solution in the context of XQuery. An overview of our overall XQuery caching solution ACE-XQ is given in Section 3. We then focus on the cache management aspect of the ACE-XQ system. We analyze the advantages and disadvantages of alternative query region managing schemes in Section 4, while in Section 5 we describe a fine-grained replacement strategy (a la *partial replacement*) deployed in ACE-XQ. The experimental studies comparing our partial replacement strategy with the traditional total replacement strategy are given in Section 7. The related work is described in Section 8 and we conclude in Section 9.

2 Running Example of XQuery Containment and Rewriting

The foundation of query-based caching is query containment, i.e., verifying whether one query yields necessarily a subset of the result of another query. In the relational context, the containment problem for conjunctive queries has been extensively studied [27,38,8,29]. Its complexity was shown to be NP-complete in [7].

A query $Q1$ is *contained* in a query $Q2$, denoted $Q1 \sqsubseteq Q2$, if for any database *D*, the answers to $Q1$ form a subset of the answers to $Q2$. The two queries are *equivalent*, denoted $Q1 \equiv Q2$, if $Q1 \sqsubseteq Q2$ and $Q2 \sqsubseteq Q1$. Chandra and Merlin [7] showed that for two conjunctive queries $Q1$ and $Q2$, $Q1 \sqsubseteq Q2$ if and only if there

is a *containment mapping* from variables of $Q2$ to $Q1$.

Conjunctive query containment and rewriting have been extensively studied for the relational algebra and datalog queries. However, the research for XML query containment is still in its infancy with a flurry of recent work focusing on XPath path expression containment [3,1,18]. We are in particular interested in the containment for XQuery [44], which was proposed by W3C as a standard XML query language.

2.1 Running Example of XQueries

In XQuery, the FLWR expression is a major building block. An XQuery composed of nested FLWR expressions is capable of hierarchical "pattern matching" against the tree-structured XML data model and of "restructuring" the result tree. For example, a user may use query $Q1$ (shown in Figure 1) to integrate book information from two remote XML sources, i.e., *bib.xml* and *reviews.xml*. Figure 2 gives a graphical representation of their document structures conforming to *bib.dtd* and *reviews.dtd* respectively.[1] $Q1$ joins these two XML documents based on the value-based equality tests on their *title* element nodes and returns only those books that are published after 1990. Assume the result XML view is *Q1Res.xml*, it contains the *title*, *year*, *author*, *publisher* and reviewer's *rate* information of such books.

```
< booklist >
  {for $b in doc("http://www.bn.com/bib.xml")/bib/book,
       $bk in doc("http://www.xyz.com/review.xml")//book
   where $b/@year > 1990 and $b/title = $bk/title
   return < entryyear = "$b/@year" >
              {$b/title} {$b/author} {$b/publisher} {$bk/*/rate}
          < /entry >}
< /booklist >
```

Fig. 1. An Example XQuery $Q1$

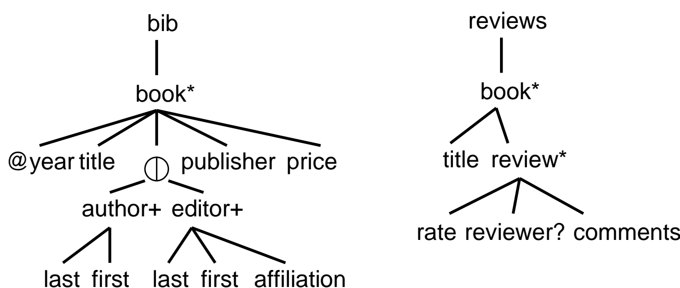


Fig. 2. Graphical Representation of bib.dtd and reviews.dtd

Suppose the user now issues a new query $Q2$ as shown in Figure 3 refining her previous queries. Say, she is interested in finding the books that are published by

[1] The textual Document Type Definition (DTD) for *bib.dtd* is exactly the same as that used for Use Case "XMP" in the W3C working draft "XML Query Use Case" [42].

"Addison-Wesley" more recently (the published year is later than 1995) and highly rated (the reviewer's rate is at least 4 out of 5). Furthermore, she wants particular information about the author's *first* name for these books if the *last* name is "Bernstein". Like $Q1$, $Q2$ also joins the same two XML documents, *bib.xml* and *reviews.xml*. If the requested information would contain only *title*, author's *first* name and reviewer's *rate* of such books, one could easily tell that $Q2$ is totally contained in $Q1$. $Q2$'s answer can fully be retrieved by reusing the result of the cached query $Q1$. In this case, the user however requests some extra information about the book's price, which is not part of the answer for $Q1$. Therefore, $Q2$ needs to be broken into two queries, namely, the probe and the remainder query, the former of which retrieves part of the answer from the local cache and the latter is sent to the remote server to fetch the data used for augmenting the probe query result.

We aim to find an automated technique for such a task. Clearly, the query containment reasoning between XQuery queries cannot directly utilize the traditional containment mapping [7] technique developed in the relational context, since the mapping atoms in relational queries are flat relations and attributes.

```
< result >
  {for $b in doc("http://www.bn.com/bib.xml")/bib/book,
       $bk in doc("http://www.xyz.com/review.xml")//book
    where $b/@year > 1995 and $b/publisher = "Addison-Wesley"
      and $b/title = $bk/title and $bk/review/rate > 4
    return < goodbook >
              {$b/title} {$b/price} {$bk/review/rate}
              {for $a in $b/author[last = "Bernstein"]
               return < author >
                          {$a/first}
                      < /author >}
           < /goodbook >}
< /result >
```

Fig. 3. A New XQuery $Q2$

2.2 A Quick Review of Our XQuery Containment Solution

We hence have proposed a containment and rewriting framework for XQueries [9]. Below we identify the challenges imposed by XQuery for the tasks of query containment and rewriting. Correspondingly, we have proposed techniques, as briefly stated below, for tackling the relevant problems. We refer the interested readers to [10] for details.

- The flexibility of the XQuery syntax makes it possible to compose an XQuery expression using arbitrarily nested FLWR expressions. Our first task is hence to normalize the input XQuery, which is restricted to consist of only the negation-

free, disjunction-free and loop-free XPath path expressions and nested FLWR constructs. Also, we currently do not consider queries involving XML constructs such as Comment, PI data or mixed content.
- The semantics of an XQuery are composed of two essential parts: pattern matching and result restructuring. We propose to cleanly separate them using a pattern tree and a tagging template to capture both semantics respectively. Different from the pattern tree representation used in the literature [45] which basically captures the navigation pattern purely based on navigation steps, our pattern tree is composed of the defined variables and hence called *VarTree*. The variable dependencies are captured via the expressions-based tree edges in VarTree and the return path expressions form the leaf nodes attached to their prefixing variable nodes. In addition, another tree-like structure called TagTree is used to provide the restructuring template and to preserve the correspondence mappings between element types in the view and the source document structures.
- In order to tackle the XQuery containment problem, we minimize the *VarTree* structure to contain only the essential variables needed for result construction. Our containment mapping process then incorporates type inference and subtyping mechanisms for regular expression types [19,41] to check the subsumption relationships between variable types.
- XQuery rewriting needs to consider the possibly restructured view schema. Based on the containment mappings established during the containment checking procedure, we rewrite the navigation paths for defining variables in the new query according to the tagging template of the cached query.

Here, we briefly explain our containment mapping and rewriting techniques using the running example queries, while more details can be found in [9]. In order to showcase how we utilize the XML type related theory [19,41,40,30] to facilitate our query containment process, we first declare some types derived from *bib.dtd* following the style used for the example in [19]. [2]

```
type Bib       = bib[Book*];
type Book      = book[Title, Year, (Author+|Editor+), Publisher, Price];
...
type Author    = author[Alast, Afirst];
type Alast     = last[PCDATA];
type Editor    = editor[Elast, Efirst, Affiliation];
type Elast     = last[PCDATA];
...
```

[2] For simplification, the name convention for the type of an element is to capitalize the first letter of its label name. In case of name conflicts, we make the type specification dependent on the context, along the lines shown in [19]. In this example, the Alast and Elast types have the same label name and content model but they differ in their context element types, i.e., Author and Editor respectively.

First, we build VarTrees for $Q1$ and $Q2$ respectively for capturing the corresponding variable dependencies in them. For example, $\$b^{Q2}$ is a variable node in the VarTrees of $Q2$ ($\$b^{Q2}$ denotes a variable $\$b$ in $Q2$ to be distinguished from $\$b^{Q1}$ for $\$b$ in $Q1$). $\$b^{Q2}$ has a child variable node $\$a^{Q2}$, because $\$a$ is defined based on $\$b^{Q2}$ in the inner FLWR block. Also, we associate with $\$b^{Q2}$, $\$a^{Q2}$ and $\$b^{Q1}$ their affiliated return path expressions .

Based on our type-enhanced containment mapping criteria in [10], since variable $\$b$ in $Q2$ (denoted by $\$b^{Q2}$) is defined using the same path expression */bib/book* as variable $\$b$ in $Q1$ (denoted by $\$b^{Q1}$) and the former is associated with more restricted conditions than the latter, we can hence set up a containment mapping $\$b^{Q1} \rightarrow \$b^{Q2}$. Furthermore, we annotate with this mapping the remaining condition *($b/@year>1995 and $b/publisher="Addison-Wesley" and $b/rate>4).* Following the same procedure, we set up another containment mapping $\$bk^{Q1} \rightarrow \$bk^{Q2}$. We then check whether the return expressions associated with $\$b^{Q2}$ and $\$bk^{Q2}$, i.e., $\$b^{Q2}/title$, $\$b^{Q2}/price$ and $\$bk^{Q2}/review/rate$, have their corresponding return expression mappings in $Q1$. It turns out that $\$b^{Q1}/title \rightarrow \$b^{Q2}/title$, $\$bk^{Q1}/*/rate \rightarrow \$bk^{Q2}/review/rate$ based on the mappings established between their corresponding prefixing variables. [3] However, $\$b^{Q2}/price$ has no match and it is thus marked to reflect the need for compensation via a remainder query.

The variable $\$a$ specified in the inner block of $Q2$ is mappable to $\$b^{Q1}/author$, i.e., $\$b^{Q1}/author \rightarrow \$a^{Q2}$, based on our containment mapping rule allowing a variable in a new query to be mappable to a return path expression in a previous query. [4] In such a case, we consider all the return path expressions associated with $\$a^{Q2}$ ($\$a^{Q2}/first$ is the only return path in this example) as matched due to the implicit deep-copying semantics for returning $\$a^{Q1}$.

After this top-down progressive containment mapping process, we formulate for $Q2$ a *probe query* for retrieving the part of the answer available from the result view of $Q1$, and a *remainder query* for fetching the *price* information in particular from the remote server. Figure 4 shows the probe and remainder queries produced by our ACE-XQ. The probe query rewrites the original path expressions in $Q2$ with respect to the view structure of "Q1Res.xml" based on the established containment mappings. For example, the bound expression */bib/book* for variable $\$b^{Q2}$ is rewritten as */booklist/entry* because $\$b^{Q1} \rightarrow \$b^{Q2}$ and a new tag $< entry >$ is constructed for each $\$b^{Q1}$ binding within the root element scope labeled by $< booklist >$. The remainder query is constructed based on the by-product yield from the containment mapping procedure, i.e., the remaining conditions annotated on the established containment mapping pairs and the marked missing return path expressions.

---

[3] The type inference and subtyping theory has been utilized to identify the mapping $\$bk^{Q1}/*/rate \rightarrow \$bk^{Q2}/review/rate$.

[4] However, the other direction, namely, mapping from a variable specified in a previous query to a return path expression in a new query is not allowed. Details are skipped here due to space limitations.

```
<ProbeQueryResult>
  {for $bb in doc("Q1XMLView.xml")/booklist/entry
   where $bb/@year > 1995 and $bb/publisher = "Addison-Wesley"
     and $bb/rate > 4
   return <bb>
            {$bb/title} {$bb/rate}
            {for $aa in $bb/author[last = "Bernstein"]
             return <author>{$aa/first}</author>}
          </bb>}
</ProbeQueryResult>                                   Probe Query
```

```
<RemainderQueryResult>
  {for $bb in doc("http://www.bn.com/bib.xml")//book
   where $bb/@year > 1995 and $bb/publisher = "Addison-Wesley"
   return <bb>
            {$bb/title} {$bb/price}
          </bb>}
</RemainderQueryResult>                           Remainder Query
```

```
<Result>
  {for $r1 in doc("pqRes.xml")//bb, $r2 in doc("rqRes.xml")//bb
   where $r1/title = r2/title
   return <goodbook>
            {$r1/title} {$r2/price} {$r1/rate} {$r1/author}
          </goodbook>}
</Result>                                          Combining Query
```

Fig. 4. The Probe, Remainder and Combining Queries for Answering $Q2$

The results returned from the *probe* and *remainder* queries need to be combined at last to provide the user with the complete answers. ACE-XQ hence deals with the issues such as "data correspondence" in order to merge the results derived from the same source by different queries using a *combining query*. Unlike other XML merging work [31] which either assume or impose object identifiers throughout the source documents to make it possible merging two pieces, we utilize primarily the DTD knowledge and user designated key constraints for identifying node equivalency. The concept of relative key path proposed by [2] is exploited to allow the source subscriber to specify the key constraints, e.g., "/bib[book[title]]" means that "under the context of the whole file, each book can be uniquely identified by its title child". A heuristic rule for deriving key constraints is to use the required attribute(s) or non-descendant singleton child element(s) of an element as its key constraint. By non-descendant, it implies that the comparison of key values would not involve a deep equality test. The inferred key constraints, however, need to be confirmed by the source subscriber or validated by the underlying documents.

In our running example, the remainder query for $Q2$ needs to fetch the missing *price* information to be appropriately merged with the other information obtained by the probe query for the identical book. As shown in Figure 4, we hence also piggyback

title in the remainder query and employ a join condition *$r1/title=$r2/title* in the combining query for matching up the *book* elements derived from the probe query and from the remainder queries (i.e., $r1 and $r2 bindings respectively).

In case when an element is not provided with a key constraint, the default key is the composition value of all its children elements, whose key values are recursively defined in the same way. We assume that such a key generation function is affiliated with the query evaluation engines at both the cache and the remote server sites. At the remote site upon the arrival of a remainder query augmented with the request for joinable keys, our key generation function is called from the query engine to walk through the subtree of a specific element and return the generated key value. This key generation function guarantees to produce the same key value for the same element regardless of the different queries. [5] Surely, a remainder query is augmented only if the probe query can be modified in the same manner for retrieving the corresponding key values available in the XML view document. We request that even the warm-up queries are augmented properly to have the cached XML view documents equipped with keys.

Key generation on-the-fly may be time-consuming. However, the complexity has a linear upper bound in terms of the subtree size. It can be optimized for situations where some subtree elements have designated key constraints available to allow for early returns from tree walking. When this approach of utilizing keys and query augmentation is found to be too costly in some scenarios, we may choose to simply flag a non-rewritable case and let the query bypass the cache. In contrast to the approach assuming an object identifier for each XML element, this is a rather unrealistic expectation in practice for many autonomous XML data servers. Furthermore, it is typically not possible for a cache to take charge of the remote data source so to insert the generated ids back into the XML sources. Hence, the design choices sketched out above are favorable.

3 The ACE-XQ System Overview

The framework of the ACE-XQ system is depicted in Figure 5. It consists of two subsystems, a **Query Matcher** which implements the query containment and rewriting techniques and a **Cache Manager** which manages the cache space and applies replacement and coalescing techniques.

When a new user query comes in, the *Query Decomposer* (shown in the **Query Matcher** subsystem on the left hand side of Figure 5) applies normalization rules [32,9] to derive its nesting format, revealing the variable dependency hierarchy specified in the query's matching patterns. It further re-groups the conditions and

[5] We assume no updates occur to the source documents.

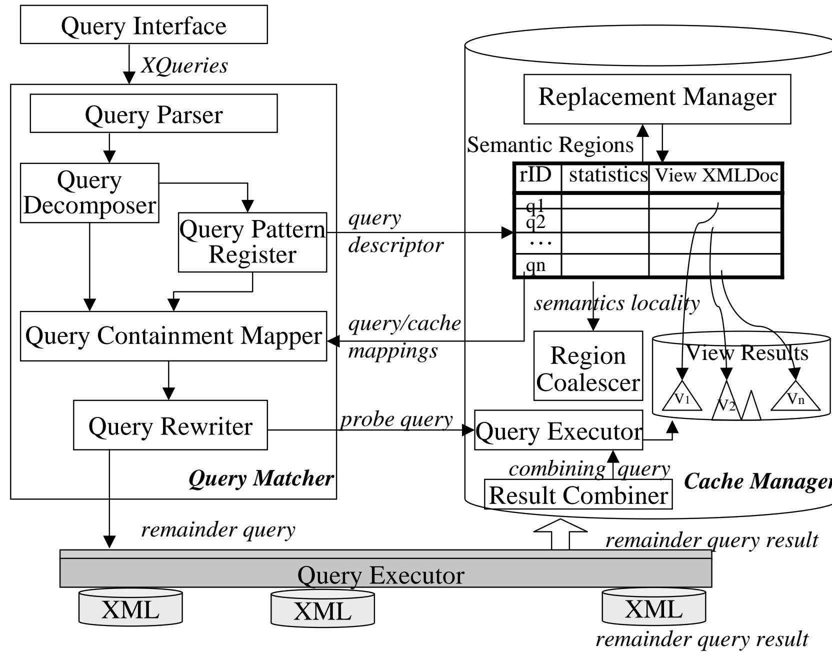


Fig. 5. The ACE-XQ System Architecture

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, and a *combining query* to make one complete answer.

As shown on the right hand side of Figure 5, the **Cache Manager** of ACE-XQ manages a collection of query regions, each composed of a semantic *query descriptor* of a cached query and the result XML view. The former part is used for reasoning about query containment while the latter can be queried by the probe query to provide any answer available in the cache to the new query as quickly as possible.

Due to the limited cache space, we propose a novel replacement strategy and deploy it in the *Replacement Manager* of ACE-XQ. When a new query comes and there is no cache space left, victim queries are chosen to be evicted from the cache to make room for the new query. The user access statistics are necessary since they can be used for calculating the region utility values, based on which the replacement decision is made. The remainder of this paper will focus on the description of the proposed replacement strategy, which utilizes the user access statistics recorded at

the path level to perform a fine-grained region purging, as opposed to the strategy of completely replacing a cached query and its associated XML view.

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 will not cause any data to be removed from the cache. Instead, it is a heuristic to optimize the cached regions by reasoning about the semantic distance between queries.

4 Design Choices for Alternative Cache Region Management Schemes

In a traditional query-based caching system [14], a query region is the minimal granularity managed in the cache. A query region consists of an encoded query descriptor and a pointer to access the associated result XML view. In this section, we will take a look at existing alternative schemes and compare them in their ways of managing the query regions in the presence of replacement activities.

When a new query arrives, the containment mapper will first determine 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 requested by the new query is available in the cache, a remainder query *RQ* will also be sent to the remote servers to fetch the rest of the answer. In this sense, query regions may logically be segmented by probe queries upon the arrival of new queries. Below we describe several possible schemes proposed by [14,24] for maintaining such query regions.

One region management scheme is to allow redundancy between query regions. In such a scheme, query regions are never adjusted once they have been formed. They are not split even if the subsequently cached queries overlap with them. For each such cached query, one uniform utility value is maintained that assesses the perceived usefulness of the query region to the users of the system. We refer to this as the *region-preserving* scheme.

Another way of managing the query regions is to split a cached query $Q1$ into two regions upon the arrival of a new query $Q2$. One corresponds to the part utilized by the probe query $PQ2$ for answering the new query, and the other, represented by $Q1 - PQ2$, corresponds to the part not usable for answering the new query. The region $Q1 - PQ2$ inherits its utility value from its parent region from which it was split off. The region $PQ2$ is marked with an increased utility value compared to the original cached query to indicate its contribution in answering this current query. This process is shown in Figure 6.

Alternatively, the query regions for the earlier cached queries may be preserved as a

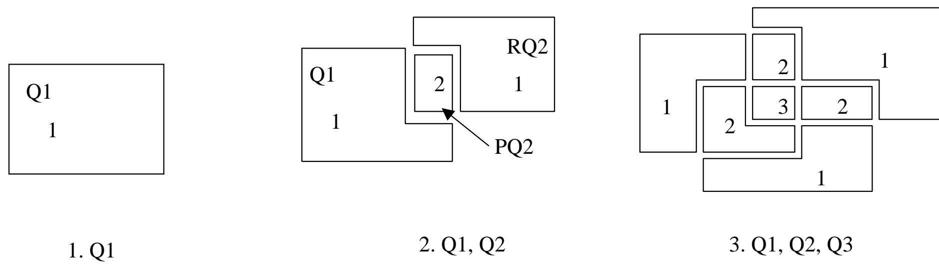


Fig. 6. Pictorial Illustration of the First Region-splitting Technique

whole. In this scheme, a new region is allocated to capture only the remainder query $RQ2$ since $PQ2$ is already contained in the existing query region $Q1$. The XML view content of this new region $RQ2$ represents the "net" increase of information obtained by the new query. This can equivalently be seen as a process of splitting the newly incoming query region first and then caching only the non-redundant portion of the region as a new region. This process is shown in Figure 7. The utility values of each region will be increased every time a hit occurs.

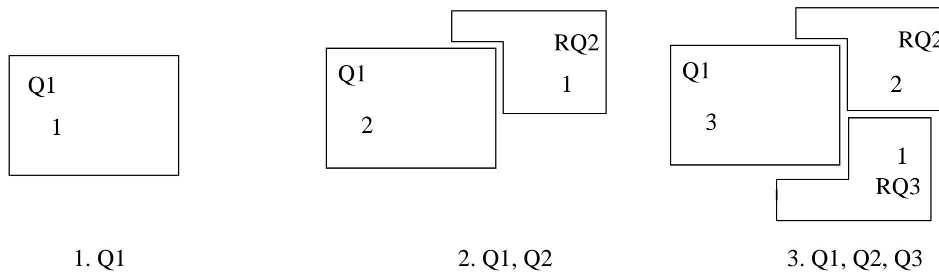


Fig. 7. Pictorial Illustration of the Second Region-splitting Technique

In the latter two scenarios, in effect the region-splitting scheme is applied. This helps to reduce the cache redundancy. However, the first region-splitting scheme of managing regions tends to result in too many small region fragments over time which tend to be less useful in answering future queries. Also, such a scheme entails the overhead of query region splitting each time when a new query is launched. Hence, it may have to resort to frequent coalescing to make up for the fact that the cache space has been severely fragmented over time.

The second region-splitting scheme avoids the coalescing computation overhead compared to the first splitting schema. 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 region fragments in answering subsequent queries. Second, a straightforward application of a replacement strategy would replace a complete region at a time. Such a replacement unit may likely be too coarse grained, requiring us to remove potentially huge sets of XML elements even when only a small space is needed in the cache. This would result in less efficient cache space utilization.

5 The Partial Replacement Strategy

5.1 Query Descriptor Hierarchy

To overcome the drawbacks identified above of naive region-splitting replacement strategies, we instead suggest here that different utility values may be maintained for finer parts within a given region to account for different levels of accesses by users. This then should be done independently from the final decision of splitting the region. When a new query overlaps with a cached query region, the overlapped portion in the cached region is accessed by the return expressions of the probe query. They correspond to the data objects associated with certain paths in the region. To record such accesses of certain parts of a cached region, we extend each query descriptor with a utility tracking table containing all complete path expressions in the corresponding XML view document. Each path expression corresponds to a row in this *path table*, referred to as *XPathRow*.

The XPathRows of such a path table can be easily constructed based on the return expressions in a query. We simply enumerate all the complete paths from the root of the view document to the leaf element types in the view schema and use them as XPathRows. For example, the type inferred for a return path expression *$b/author* in $Q1$ is *Author* which contains two leaf element types *Last* and *First*. Therefore, two XPathRows */booklist/entry/author/last* and */booklist/entry/author/first* corresponding to these types are listed in our XPathRow table. All the other XPathRows in the path table are complete paths appearing in the view schema of $Q1$. The statistics related to the user access information are now no longer associated with the complete region, but more precisely with the specific XPathRows. The different types of statistics such as hit frequency, last access timestamp and etc., are all associated with each XPathRow. With the utilization of the path table, we maintain the user access statistics at the granularity level of *XPathRow*s for each cached query. Figure 8 displays a snapshot of the extended query descriptor of $Q1$.

$Q1$	XPathRow	hits	last_access_time	obj_bytes	...
	/booklist/entry/@year	1	12:33pm May 30	1600	
	/booklist/entry/title	1	12:33pm May 30	2100	
	/booklist/entry/author/last	1	12:33pm May 30	2860	
	/booklist/entry/author/first	1	12:33pm May 30	2620	
	/booklist/entry/publisher	1	12:33pm May 30	1840	
	/booklist/entry/rate	1	12:33pm May 30	1980	

Fig. 8. Path Table with Initial Statistics for $Q1$

When a new query overlaps with a cached query, the probe query *PQ* is formulated to retrieve the relevant data in the XML view via the path expressions specified in

the return clauses. These path expressions correspond to the XPathRows in the path table. We can hence correspondingly update the statistics for these XPathRows that are involved in the probe query. For example, Figure 9 shows the path table constructed for $Q2$'s query region and the updated statistics in the cached query region $Q1$. As we explained before, the answer of $Q2$ has utilized the cached data from region $Q1$. The replacement manager hence correspondingly modifies the statistics of the highlighted XPathRows in $Q1$'s path table that are involved in the probe query for answering $Q2$. The fragments along two paths */booklist/entry/@year*, */booklist/entry/title* and */booklist/entry/author/first* in $Q1$'s XML view contribute to answering $Q2$.

$Q2$	XPathRow	hits	last_access_time	obj_bytes	...
	/goodbook/title	1	12:47pm May 30	220	
	/goodbook/price	1	12:47pm May 30	150	
	/goodbook/rate	1	12:47pm May 30	260	
	/goodbook/author/first	1	12:47pm May 30	380	

$Q1$	XPathRow	hits	last_access_time	obj_bytes	...
	/booklist/entry/@year	1	12:33pm May 30	1600	
	/booklist/entry/title	**2**	**12:47pm May 30**	**2100**	
	/booklist/entry/author/last	1	12:33pm May 30	2860	
	/booklist/entry/author/first	**2**	**12:47pm May 30**	**2620**	
	/booklist/entry/publisher	1	12:33pm May 30	1840	
	/booklist/entry/rate	**2**	**12:47pm May 30**	1980	

Fig. 9. $Q2$'s Path Table and $Q1$'s Path Table with Updated Statistics

5.2 Utility Value and Replacement Function

The *utility value* is considered to be 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 over other ones. A *replacement function* is used to reflect the replacement preference of a caching system. It calculates the utility values of cached objects, based on which the replacement manager chooses the victim to be purged to make room for new objects.

Cache replacement policies have been extensively studied in different scenarios, such as page-based [6] and tuple-based [16] caches. Various replacement schemes [37,33,4,35,28] have been investigated. 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 simplic-

ity while still being effective when recently referenced objects are likely to be re-referenced in the near future. The LFU policy [37] uses reference frequency instead of recency as the parameter for the replacement function.

In this paper, we propose to utilize the detailed path tables [6] to perform a finer granularity replacement than replacing a complete query region at a time. That is, the input to the replacement function are 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 an XPathRow is used in the latest probe query. *hits* is the number of times an XPathRow has been used for answering subsequent queries. *obj_bytes* is a size estimation of the data collected along a particular path. If a path were to be selected as the next victim, this number gives a hint about how large a fragment would be purged from the XML view. We also keep track of more global statistics at the query level such as *xml_doc_size*, which is the overall document size in bytes, and *fetching_delay_cost*, the original query evaluation time.

The considered statistics can be classified into several categories. One category concerns the user access reference pattern, such as the recency value *last_access_time* and frequency value *hits*. The *hits* measure on an XPathRow is increased by one each time when it is requested by a probe query. Its *last_access_time* is updated to the current time upon such an update. The second category is related to the data size that would be freed upon a purge, i.e., the *obj_bytes* on a particular XPathRow and the *xml_doc_size*. If two groups of XPathRows have a tie in their frequency values, the one associated with a larger *obj_bytes* 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.

The distance involved in the data transmission and network delay fall into the third category. We consider the benefits brought by preserving a region by measuring the loss caused by not caching it, which can be represented by the initial response delay for answering a query, denoted as *fetching_delay_cost*. By retaining regions with longer initial fetching delay, large fetching cost for such regions in the future could be avoided. Furthermore, we roughly measure the *byte_fetching_delay_cost* for a particular query by dividing *fetching_delay_cost* by *xml_doc_size* (in bytes). Therefore, the benefits of retaining a particular XPathRow in the query region can be measured by $byte_fetching_delay_cost \times obj_bytes$. Below, we propose a replacement function to calculate the comprehensive utility value of each XPathRow based on the collected statistics.

$$rp_fun = \frac{hits \times fetching_delay_cost \times obj_bytes}{xml_doc_size} \quad (1)$$

[6] Concerning the enlarged descriptor size caused by this path table structure, we consider to minimize the space overhead by adopting some indexing or compression techniques.

Other functions in favor of different scenarios can be easily plugged into our system. We have indeed experimentally studied several such functions when designing this formula.

5.3 *The Partial Replacement Algorithm*

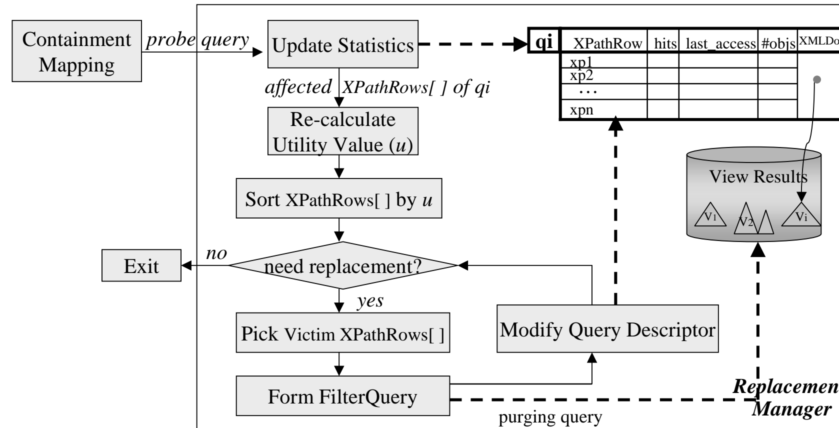


Fig. 10. The Replacement Control Flow

Figure 10 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 re-calculates their utility values. Only when there is a need for replacement due to exhausting of the cache capacity, the replacement manager chooses those XPathRows with the lowest utility value as the victim XPathRows. It then composes a *filter query* to remove the fragments corresponding to these paths from the relevant XML view(s). The query descriptors of those affected cached queries are also modified accordingly to be consistent with their changed XML views. The detailed replacement algorithm is described in Algorithm 1.

Filter Query. Suppose ten queries $Q1$ to $Q10$ 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 $Q11$ arrives and there is not enough space in the cache for it. Based on the lowest utility value, the replacement manager decides that $victimQ$ and $victimXPathRows[\,]$ are $Q1$ and *[*$/booklist/entry/@year$*,* $/booklist/entry/author/last$*,* $/booklist/entry/publisher$*]* respectively (the statistics of $Q1$ and $Q2$ may not be the same as illustrated in Figure 9 any more). To remove XML fragments corresponding to $victimXPathRows[\,]$ from $Q1Res.xml$, XML update statements can be adopted [39]. An alternative method is to extract the parts that are of interest and discard the remainder from the view content with the help of some filtering mechanism such as the filter query shown in Figure 11. [7]

[7] Note that the filter query returns the complement path set of $victimXPathRows[\,]$. The construction of such a filter query utilizes the view DTD knowledge for preserving the

Algorithm 1 **The Replacement Algorithm**

```
Initialize cache capacity cap, replacement function rp_fun.
Buffer incoming queries in a query queue QQ.
loop
  while QQ is not empty do
    q ← pop(QQ)
    if q overlaps with a cached query cq then
      Update statistics of overlapped XPathRows[ ] in cq's region.
    end if
    while Not enough space for holding q do
      for all CachedQueries do
        for all XPathRows do
          Invoke rp_fun to calculate the utility value.
        end for
      end for
      victimXPathRows[ ] ← victimXPathRows with lowest utility value.
      victimQ ← cached query containing victimXPathRows[ ].
      Formulate filter query to purge fragments related to victimXPathRows[].
      Modify query descriptor of victimQ.
    end while
    Construct a query region for q.
  end while
end loop
```

```
<booklist>
  {for $b in doc("Q1Res.xml")/booklist/entry
   return <entry>
            {$b/title} {$b/author/first} {$bk/rate}
          </entry>}
</booklist>
```

Fig. 11. An Example Filter Query

6 The Analysis of Cache Performance

We have discussed earlier the methodologies adopted by our partial replacement approach versus the alternative approaches. Here, we attempt to give an analytical model for better understanding how the caching system interacts with various factors such as cache size and query access pattern. Based on this model, we analyze how the cache would behave in the face of different replacement strategies. This may help us to gain insights into the reason why the cache equipped with our partial replacement strategy can achieve a higher cache hit ratio than the alternative one.

structural hierarchy in the updated view content.

6.1 Query Trace Model

We describe the semantic nature of a query trace in terms of query selectivity, locality and skewness, etc. Suppose there is a sequence of query access requests for one of $N \gg 1$ documents $D1, \ldots, DN$. Visually, we may imagine the whole semantic space is set by the source document DTDs. A query can then be seen as an object with certain spatial expansion like a rectangle region. Query selectivity, denoted as $\mathcal{S}$, is the size ratio of the query result over the source documents. Assume the semantic space is normalized, the selectivity of Qi, namely $\mathcal{S}(Qi)$, can then be used for representing its query region size. $\mathcal{S}(Qi)$ is closely related to the query condition strictness and the projection scope, which can be viewed as the length and the width of Qi. The query locality is measured by the distance $\mathcal{D}$ between a given query pair. Suppose Qi, Qj are two queries with Qi preceding Qj in a given query trace. Below we show three scenarios where the computing of $\mathcal{D}(Qi, Qj)$ (i.e., the distance between Qi and Qj) takes different approaches.

$$\mathcal{D}(Qi, Qj) = \begin{cases} \infty & \text{when Qi and Qj are semantically disjoint;} \\ \frac{\mathcal{S}(Qj)}{\mathcal{S}(Qj \cap Qi)} & \text{when Qj overlaps with Qi;} \\ 1 & \text{when Qj is totally contained within Qi.} \end{cases}$$

The query skewness $\mathcal{K}$ is an overall characteristic of a query trace indicating how the query regions are distributed in the semantic space. Corresponding to our two-tier query descriptor scheme, we group queries into a two-tier cluster hierarchy. Each top-tier cluster (called TTC) corresponds to an individual document. Queries that involve the same document are collected into one TTC. One query may appear in more than one TTC. The low-tier clusters (LTC) represent the local "hot spots" within a specific document. The measurement of $\mathcal{K}$ occurs at both tiers. On the TTC tier, $\mathcal{K}t$ indicates the document access distribution pattern. Certain documents may have more query accesses than others. Whereas $\mathcal{K}l$ on the LTC tier hints how unevenly queries are spread over a particular document.

6.2 Cache Hit Probability

Numerous studies in the web caching field have concluded that web access follows a Zipf-like distribution [26,25]. That is, the relative probability of a request for a document is inversely proportional to its popularity rank i ($i = 1..N$). The probability $Pd(i)$ of a request for the i'th popular document is proportional to $1/i^{\alpha}$ ($0 < \alpha \leq 1$). In our context, we think it is appropriate to model the XML document access pattern using this Zipf-like distribution. Since α in the distribution

model implies document access skewness [8] , it is closely related to $\mathcal{K}t$.

Independent of the document access pattern, we also observe that the average distance of query pairs within a given document likely indicates the skewness of query region access distribution. That is, the smaller the average distance, the more intensive is the concentration of query accesses to hot regions. Hence, the indicator of query region access skewness $\mathcal{K}l$ is $\mathcal{K}l = \frac{\sum(1/\mathcal{D}(Qi,Qj))^\tau}{m(m-1)/2}$. In this formula, the numerator is the sum of the inverse of the distances (of all query pairs within a certain document cluster) adjusted by a parameter τ. The denominator is the number of all possible combinations of query pairs, assuming m is the number of queries accessing the given document). Suppose the region access distribution follows a similar Zipf-like model, we then compute the probability $Pr(j)$ of a request for the k'th popular region is proportional to $1/j^\beta$ ($0 < \beta \leq 1$ and β closely related to $\mathcal{K}l$).

For Zipf-like distributions, the cumulative probability that one of the top k documents (among the total N documents) is accessed is given asymptotically by:

$$\phi(k) = \sum_{i=1}^{k} \frac{\Omega}{i^\alpha}, \text{ where } \Omega = (\sum_{i=1}^{N} 1/i^\alpha)^{-1} \approx (1-\alpha)/N^{1-\alpha}.$$

Thus $\phi(k) \approx (k/N)^{1-\alpha}$ (when $\alpha = 1$, $\phi(k) \approx ln(k/N)$). Because $k/N < 1$, a larger α increases $\phi(k)$, meaning more queries focus on a few hot documents. The probability $Pd(i)$ of an access to the top i'th popular document is $Pd(i) = \frac{\Omega}{i^\alpha} \approx \frac{1-\alpha}{N}(\frac{N}{i})^\alpha$. Similarly, if considering the probability $Pr(j)$ of a query request for the top j'th popular region *within* a particular document, we have $Pr(j) \approx \frac{1-\beta}{M}(\frac{M}{j})^\beta$, where M is the number of query regions in a document, and β is the parameter suits the region access distribution in a particular document.

In a query-based caching environment, we are concerned about the popularity ranking of query regions across documents. First, we look at the overall probability $P(i,j)$ of a query request for the j'th popular region within the i'th popular document. Suppose $Pd(i)$ and $Pr(j)$ are independent of each other, we obtain $P(i,j) = Pd(i) \times Pr(j) \approx \frac{(1-\alpha)(1-\beta)}{M \times N}(\frac{N}{i})^\alpha(\frac{M}{j})^\beta$. If the situation is simpler and a uniform α suits both the document and all the query region access distributions, $P(i,j) \approx \frac{(1-\alpha)^2}{M \times N}(\frac{M \times N}{i \times j})^\alpha$, which implies a multivariate Zipf-like distribution. We infer from this equation that if two query regions have the same $i \times j$ production value (i.e., the document popularity rank times the local query region popularity rank), then they have the same overall popularity. For such a multivariate distribution, we have the cumulative probability $\phi(k) = \sum_{i=1}^{t}\sum_{j=1}^{u} P(i,j)$, where $t \times u \leq k$.

This model assumes that the query requests are independent and both the document and query region access patterns follow the Zipf-like distribution with the same

[8] When α is close to 1, the top 1 popular document gets twice query accesses than the next most popular one. If α is close to 0, likely every document is evenly accessed.

parameter. It may be not very realistic, but the model is tractable and it is sufficient to help us understand how the hit ratio can be influenced by various factors.

Correlation between Hit Ratio and Cache Size. Studies of web caching have found that when the cache size is infinitely large, the correlation between the access frequency and document size, if any, is weak in general and can be ignored [26]. We believe this finding is valid in our context as well. However, if the cache source is limited, the Zipf-like distribution will be "cut-off" and eventually the top c most popular query region groups [9] will fill the cache to its size limit ideally. However, it is hard to derive c from the cache size C due to the factoring problem. If we assume that the query region sizes are the same and the factoring can be continuous (not a realistic assumption though), c is approximated as $\sqrt{2C}$ due to $\sum_{i=1}^{c}\sum_{j=1}^{c/i} = C$. Thus the cumulative probability

$$\phi(C) \approx \sum_{i=1}^{\sqrt{2C}}\sum_{j=1}^{\sqrt{2C}/i} P(i,j) \approx \frac{(1-\alpha)^2}{(MN)^{(1-\alpha)}}\sum_{i=1}^{\sqrt{2C}}\sum_{j=1}^{\sqrt{2C}/i}(\frac{1}{i\times j})^{\alpha}.$$

The asymptotic hit ratio $H(C)$ is closely related to $\phi(C)$. If α is very close to 1, $H(C)$ grows with the cache size C logarithmically, i.e., $H(C) \approx ln\frac{C}{MN}$. Otherwise, $H(C)$ cannot easily be approximated by a particular function. However, it is bounded by some polynomial function with a small power, e.g, $H(C) < (\frac{C}{MN})^{1-\alpha}$.

Correlation between Hit Ratio and Query Pattern. From the hit ratio function, we can see that the parameter α plays a role in controlling the slope steepness of the curve. Since $\frac{C}{MN} < 1$, the closer α is to 1, the smaller $1-\alpha$ is and consequently the larger $H(C)$ gets. As we discussed before, α is related to the document access skewness $\mathcal{K}t$ and the query region access skewness $\mathcal{K}l$. Therefore, the more query requests are concentrated on a few hot spots, the higher hit ratios can be achieved, ideally. We also observe that, if the overall document size is fixed, N and M will increase when the average individual document size and query region sizes decrease. With the same cache size and query skewness, $H(C)$ will become smaller. Due to the close relationship between the average query region size and the query selectivity $\mathcal{S}$ as we have discussed earlier, a larger $\mathcal{S}$ implies a larger region and thus a smaller $H(C)$.

6.3 Hit Ratio and Different Replacement Strategies

We have analyzed the cache hit ratio with varied cache sizes, document sizes, query selectivities and skewnesses. The assumption is that the cache replacement strategy is nearly ideal and it replaces query regions by *strictly* following the popularity order. It is hence unlikely for a real cache to achieve this high expectation. Also, the concept of query regions used in the model is oversimplified. For example, all

[9] If multiple query regions have the same overall popularity, e.g, $P(1,6) = P(2,3) = P(3,2) = P(6,1)$, we consider them as one query region group.

queries would have to have the same selectivity to result in the uniform region size. Furthermore, a query region is the smallest unit and one either exact matches or it is disjoint with another. There is no notion of partial overlap between query regions. In this sense, the model cannot be directly applied to our query-based caching context. However, we could make some adjustment and still use this model to analyze how the cache would behave differently in the face of different replacement strategies, in particular for recurring partial overlapping query cases.

Suppose a new query arrives at the cache and it partially overlaps with a cached one. We now consider a query region consists of only *one* return path expression. That is, we associate the concept of query region with our concept of XPathRow. For an incoming query Qn, it is then viewed as being broken down into a series of n smaller query regions. Among them, p regions that belong to the overlapping part with a cached query Qc can be seen as cache hits while the rest as misses.

Imagine $Qnew$ enters a cache where the region-preserving strategy is deployed. As a result, all n query regions that $Qnew$ is composed of together with those m composing $Qnew$ are marked with one more hit, even though only p regions really hit the cache. This is because this region-preserving does not split queries to indicate different uses of different query regions. This means a unfaithful recording of the utility values (i.e., the popularity ranks in the model) and would likely impair the cache performance due to its failing to separate hot-pots from cold-pots.

If the region-splitting strategy is applied, $Qnew$ would be physically divided into *PQ* (containing p regions) and *Q-PQ* (containing $n - p$ regions). Having overcome the aforementioned shortcomings of the region-preserving strategy, the region-splitting however raises a new issue that too many small queries are likely produced from splitting. Over time, queries in the cache may have been split to be as small as just a query region unit. In this scenario, the size difference between a new query (which has not experienced splitting) and a cached query is expected to be big. This is not desirable when query containment and rewriting is considered. Due to the problem of query fragmentation, we need to find possibly up to p cached queries for matching them with $Qnew$'s p regions. This exhaustive search within the cache space is usually costly, let alone the cost for combining results from p different views. Therefore, we prefer the cached queries not being overly split as this may possibly induce extraneous efforts for containment mapping and rewriting [10] .

In our partial replacement strategy, we set up for a query its descriptor in two tiers, namely corresponding to the query itself and its component query regions. Our replacement approach overcomes both disadvantages of the region-preserving and the region-splitting strategy due to its strategy of separating the notion of physical splitting from that of statistics value recording. This way, the utility values can

[10] Even if we can adopt the "first-found" policy to randomly picks *one* candidate cached query for query rewriting, the cache utilization would be very inefficient since the small cache query size further restricts the possibly overlapping region size

be computed at the finer query region granularity without necessarily splitting the query physically. The coarser-level query splitting only occurs when replacement is inevitable. When it happens, the statistics recorded at the finer-level regions help to replace fairly the less popular regions and preserve the more popular ones. A by-product of this replacement process is an "optimized" query in the sense that the " fluff" (i.e., less useful portions of query regions) is removed from the query to result in an overall more popular query. In summary, the cache utility is improved due to the fine-grained optimization of each individual query in the cache.

7 Experimental Studies

7.1 System Setup

We have implemented our ACE-XQ [9] 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 [20] in ACE-XQ.

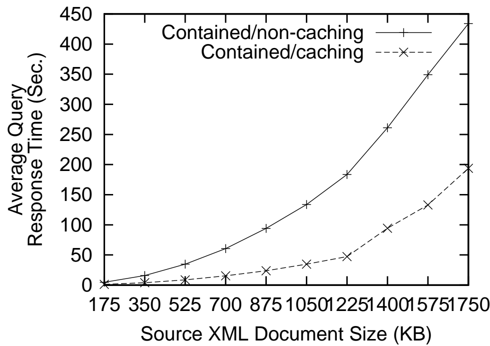
We installed the Kweelt query engine on a local UNIX machine where the ACE-XQ system resides, and another one on a remote web server where a set of XML documents is hosted. We have validated the correctness of our XQueries rewritten by ACE-XQ by comparing their results with those produced by directly evaluating the original query against the remote documents. ACE-XQ has been deployed as a testbed for various experimental studies. Our experiments investigate the query performance gain achieved by answering queries using cached views. They also compare our partial replacement strategy with the total replacement strategy.

In the experiments, we implemented our own XML data generator for generating data sets conforming to the bibliography DTD *bib.dtd*, which is the example XML document DTD used for the "XML Query Use Case" [42]. Our XML data generator enables us to produce source XML documents satisfying certain characteristics by tuning the input parameters. This way, we have a full control of the data value range, the number of elements of a certain type, etc.

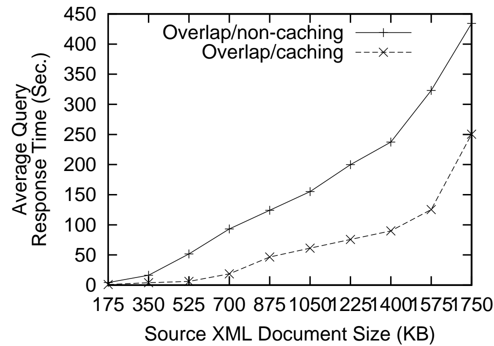
7.2 Experiments on Caching versus Non-Caching

In this experiment, we set the cache size to unlimited, which means there will not be any replacement for any query workload. We design three types of query workloads, each containing 40 XQueries. In the first query workload, by shrinking

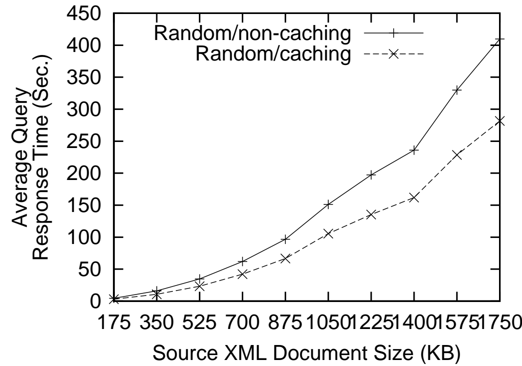
of predicate data range and removing returned XPaths, the subsequent incoming queries are designed to be totally contained in one of the previous queries in the trace. Therefore, each time the new query can be answered by a previously cached query. The second query trace has new queries partially overlapping with the previous ones. Some of them have overlapping value predicates, others have overlapping return expressions. For the third query workload, we randomly select path expressions to be returned and predicate data values to be applied according to the source XML structure and data value domain. This way we generate a set of random XQueries. We vary the source document size for 10 times and run these three traces on them. We test the average query response time for each query workload to compare the performance when using the ACE-XQ system versus when directly fetching the result from the remote server.



(a) Contained Query Workload



(b) Overlap Query Workload



(c) Random Query Workload

Fig. 12. Query Response Delays for a Variety of User Traces: Caching versus Non-Caching Architecture.

We show in Figure 12 the query response time for these three different query workloads in two scenarios, i.e., when using the ACE-XQ system versus when directly fetching the result from the remote server without using the cache. In both scenarios, the Kweelt engine is deployed locally but the source XML file is on a remote

web server. Consistent with our expectation, the results show that the average response time is reduced when utilizing the ACE-XQ caching system for all three workloads. In particular for the query workload with only contained queries in Figure 12(a), the reduction in response time delay is significant. We also observe that the response times in both scenarios get longer when the sizes of the source XML documents increase.

For the remainder of our experimental study, we fix the size of the source XML documents and perform the experiments focusing on the comparison of the performance of partial replacement and total replacement strategies.

7.3 Experiments on Replacement Strategies

We now compare the performance of the two different replacement strategies, i.e., partial replacement versus total replacement, that are employed in ACE-XQ when the cache space is limited. To be concrete, we generate two query traces to compare the query performance of alternative replacement strategies when the cache size varies. Each query trace includes 40 queries that are specified against 10 different, although about equally sized, XML documents located on remote web servers. Each XML document is around 180K bytes (the minimum size of the XML documents tested in the first experiment), and the size of all XML documents combined in total is about 1.8M bytes. We expect that our experimental results such as hit count ratio and hit byte ratio obtained for small XML source documents would scale in proportion to the XML document size given static selectivity of each XPath in the source documents.

In the first query trace, queries are randomly selected from all possible valid user queries against source XML documents. The second query trace contains only refining queries on different XML source files, i.e., most subsequent queries are contained in some previous queries requesting the same documents. We refer to the first query trace as a *random trace* and the second one as a *refining trace* [11] . Practical scenarios for both query traces can easily be found. For example, in a web search, a user may issue a query with conditions expressing his/her main concerns before he/she has enough knowledge about the queried web source. The user may then refine the query conditions over time based on the information gained from previous query results. This would form a query trace of the refining type with the queries having more and more refined conditions and thus smaller return results overtime.

In both query traces, we control the query selectivity to range from 15% to 70%.

[11] Actually in the case when the partial replacement strategy is applied, we cannot guarantee that the subsequent queries are totally contained in previous ones, since part of a cached query result may have already been replaced.

That is, for a query imposed against a source document of 180K bytes, the returning result size varies from 30K bytes to 130K bytes. Initially, we set the cache space to be 150K bytes and then increase it each time by 50K bytes until it reaches 1.2M bytes. That is, for each designed query trace, our smallest cache size is still large enough to hold at least one query result, while the largest cache may approximately hold the most frequently queried fragments of all the source XML documents in the most ideal scenario. In the following experiments, we employ three metrics as shown in Figure 13 to measure the query performance.

For partial replacement experiments, we keep track of all the access statistics on the XPathRow table and use statistic of each XPath in replacement decision making. We also keep track of statistics for each query result and use them for total replacement.

Metric (acronym)	Meaning	Formula
Hit Count Ratio (HCR)	Number of queries that re-use cached query results compared to total number of asked queries	$\frac{\#CacheHits}{\#TotalQueries}$
Hit Byte Ratio (HBR)	Average bytes of cached query results that are re-used by a subsequent query compared to cache size	$\frac{\sum Reused_Bytes}{\#Total_Queries \times Cache_Size}$
Response Time (RT)	Average response time per query in a query trace	$\frac{\sum Response_Time}{\#Total_Queries}$

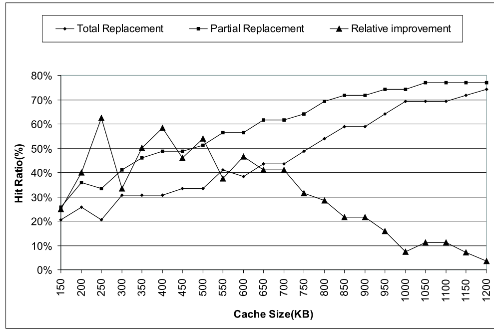
Fig. 13. Query Performance Metrics

As shown in the replacement function rp_fun in Section 4, the usage information *hits* (hit counts) recorded in the path table of a cached query is one of our main statistics in determining the victim *XPathRows* in our partial replacement strategy. In other words, the LFU policy is incorporated into our cache replacement strategy for the selection of the victim objects, being *XPathRows* if the partial replacement strategy is employed in ACE-XQ or complete cached query regions if the total replacement strategy is deployed instead.

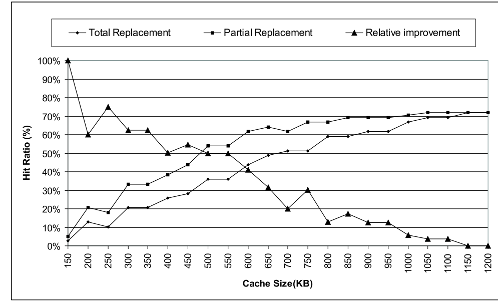
7.3.1 Hit Count Ratio Comparison of the Two Replacement Strategies

In this set of experiments, we compare the query performance of the two replacement strategies in terms of the *Hit Count Ratio (HCR)*. Each time when a new query is contained or overlapping with one of the queries in the cache and hence a probe query is generated, we consider this a *cache hit*. HCR refers to the percentage of such hits over the total number of queries in the given workload (see Figure 13).

Figures 14(a) and 14(b) show the HCR for the refining and random query traces



(a) Refining trace



(b) Random trace

Fig. 14. HCR for Two Replacement Strategies With Varying Cache Sizes

respectively. The two different replacement strategies we employed are partial and total replacement. For both strategies and for both query traces, HCR increases with the growth of the cache size. Overall, we see that HCR of partial replacement is always higher than that of total replacement. From both Figures 14(a) and 14(b), we see that the relative HCR improvement of the partial replacement compared to the total replacement is declining when enlarging the cache size. The "zigzag" for the small cache mainly is due to the "thrashing" activity of the cache when it is rather small and not stable. This also explains the unstable curves for the relative HCR improvement of our partial replacement over the total replacement during these stages. However, the relative HCR improvement is in general positive, indicating that our partial replacement strategy always wins over the total replacement strategy.

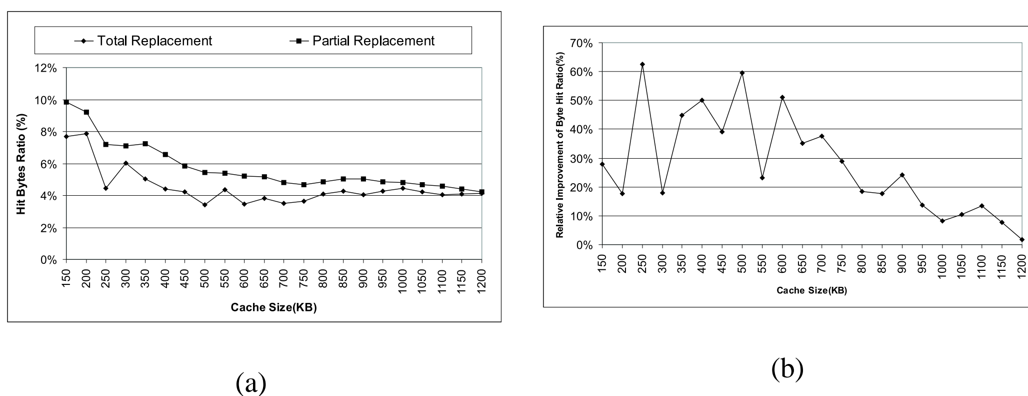
As shown in Figures 14(a) and 14(b), our partial replacement outperforms the total replacement with a relative HCR improvement of 40%-50% for both query traces when the absolute HCRs for both strategies range from 30% to 50%. This would happen for a medium size cache, when replacements may not occur that frequently for both strategies. In this case, the cache may hold more valuable queries when the partial replacement is applied than when the total replacement is used. The reason is that the partial replacement can adjust the cached queries and preserve only the more useful partial queries. This way, a given size cache holds likely more but smaller queries for the partial replacement than for the alternative total replacement. While for a very large cache, the cache space resource is not precious any more. The query region refinement pursued by partial replacement becomes less critical.

As we have expected, both replacement strategies are especially beneficial for the refining query trace. The sequence of queries in this trace are designed to have more contained query cases than the random trace and hence more cache hits likely happen in the refining trace. We also see that the relative improvement of the partial replacement over the total replacement for the refining query trace is somewhat better than that for the random query trace. In this sense, partial replacement works

more precisely to discover the "hot portions" of queries that are more likely to be used by the subsequent queries in a refining query trace.

7.3.2 *Hit Byte Ratio Comparison of the Two Replacement Strategies*

We now use the metric of *Hit Byte Ratio (HBR)* to compare the performance of the two different replacement strategies. The two curves in Figure 15 represent the HBR for the partial and total replacement strategies respectively. When the cache size is small, the HBR is shown to be about 10% and 8%, with the partial strategy wining over the total one by 2%. With an increase of the cache size, both HBRs slowly decline to about 4% when the cache size reaches 1.2M bytes. The second chart in Figure 15 illustrates the relative HBR improvement of the partial replacement over the total replacement for the refining query trace. This is shown to be above 30% when the cache is roughly of medium size.

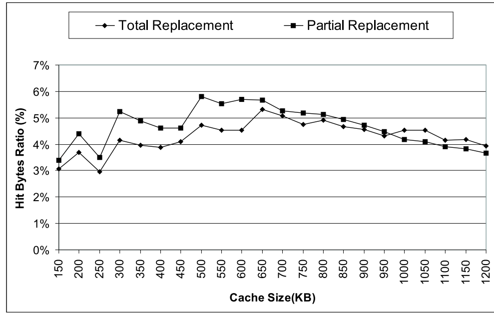


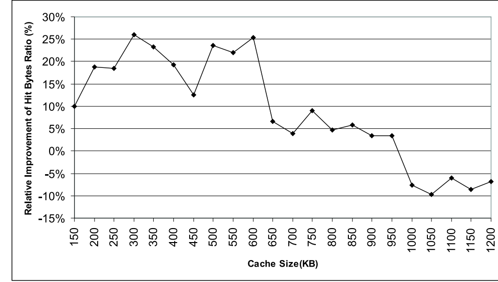
(a) (b)

Fig. 15. Refining trace: HBR for Two Replacement Strategies With Varying Cache Sizes

We then repeat this experiment with the random query trace. The HBR results are shown in Figure 16. The HBRs for both replacement strategies follow the same trend as for the refining trace, decreasing when the cache size becomes gradually larger. Although the partial replacement still outperforms the total replacement in most cases, the relative HBR improvement is shown to be less significant than that for the refining trace.

We can also observe that in some cases when the cache size is very large, the partial replacement may not always work better than the total replacement. We interpret this phenomenon as below. When applying the partial replacement strategy for a large cache, the cached queries tend to become smaller due to the partial replacement even though this is not necessarily needed given the availability of cache space. Therefore, even if the hit count ratio (HCR) of the partial replacement may be slightly higher than that of the total replacement, the average bytes used by each cache hit in a partial replacement scenario are likely smaller than those used in the total replacement.





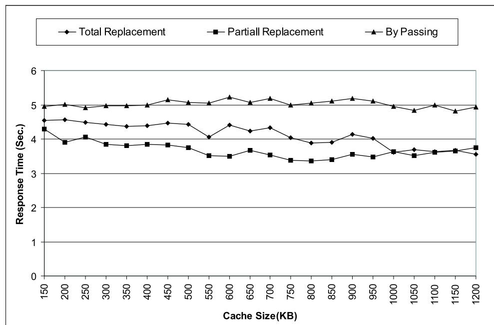
(a) (b)

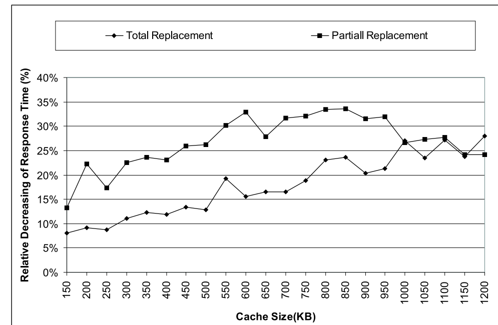
Fig. 16. Random trace: HBR for Two Replacement Strategies With Varying Cache Sizes

7.3.3 Response Time Comparison of the Two Replacement Strategies

In the last set of experiments, we study the effects of the two replacement strategies on the query performance in terms of response time. This intuitively has a close relationship with the HCR and HBR measures. Since the response time for a local probe query is usually much smaller than that for a query fetching results across the Internet, the higher the HCR or the HBR is, the larger a reduction in the average query response time is likely achieved.

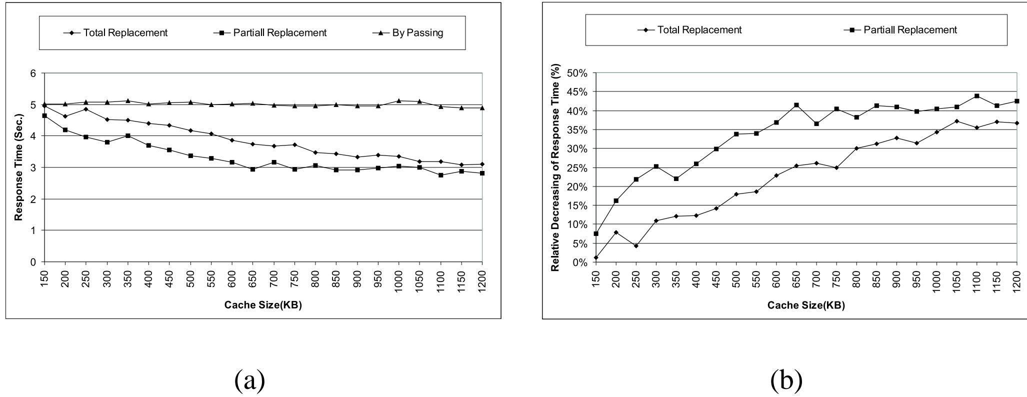
In both Figures 17 and 18, we compare the response times under three scenarios: 1) retrieving the query answers directly from the remote data server while bypassing the ACE-XQ system; 2) utilizing our ACE-XQ caching system supported by the total replacement strategy; 3) utilizing our caching system supported by the partial replacement strategy. Figure 17 shows that the response time for the bypassing scenario (without using the caching system) is roughly the same, i.e., about 5 seconds. The partial replacement strategy helps to improve the query performance by about 25% to 35% in terms of the average reduction in response time, while the total replacement achieves about 15% to 25% improvement on average.





(a) (b)

Fig. 17. Refining trace: RT for Two Replacement Strategies With Varying Cache Sizes



(a) (b)

Fig. 18. Random trace: RT for Two Replacement Strategies With Varying Cache Sizes

These experiments show that our partial replacement strategy outperforms the total replacement in most of the tested scenarios. The former strategy will perform especially better than the latter for a medium sized cache, neither too small to hold the result for a single query, nor too huge to make a replacement of necessity since all the query results can be held in the cache. We can have over 40% relative improvement on hit count ratio and 15%-30% on relative improvement on hit byte ratio under this scenario. It is also clear that the partial replacement will be more useful when the user query trace is in the favor of refining queries. So when there are some queries totally contained in previous ones in the user access history, the partial replacement will win.

8 The Related Work

8.1 XML Query Containment

The problem of query containment is fundamental to query evaluation and optimization in database systems. This problem was first studied by Chandra and Merlin [7] for conjunctive queries, whose expressive power is equivalent to that of the Select-Project-Join (SPJ) queries in relational algebra. A flurry of extensive research efforts have followed to investigate all the relevant aspects ranging from the complexity theory to its practical applications in optimizing queries, answering queries using views and detecting update relevancy.

The emerging tree-oriented XML data model and its compatible query languages has stirred a re-newed interest in studying the query containment problem in the XML setting. Several research projects have started studying containment for the core query mechanism over semistructured data, namely regular path queries. Such regular path expressions also serve as the navigation facility commonly used in

many recently proposed XML query languages such as XPath [43], XML-QL [15], XQL [36] and XQuery [44]. In general, the complexity of containment is NP-complete for the relational conjunctive query and even goes up to ExpSpace for conjunctive regular path queries even with inverse operators [3]. Significant contributions have been made [1,18] to identify the fragments of regular path queries with their corresponding containment complexities.

In ACE-XQ, we hence base our XQuery containment work on the state-of-the-art theoretical research in query containment for XPath and regular path expressions. To trade-off the expressiveness and tractability, we consider only the negation-free, disjunction-free and loop-free XPath path expression in XQuery. We allow wildcards "*", branches "[ ]" and descendant-or-self axis "//" in XPath path expression. Such a fragment of XPath path expression is identified to form a tractable class and sometimes is even PTIME efficient [18].

There is a focused interest in the containment problem for XPath path expression [18], regular path expressions [3], regular-path-expression based query languages such as StruQL [17] and TSL [34] over semi-structured databases and the tree pattern queries in XCacher [22]. However, regular path expressions alone are not sufficient to be used as an XML view specification language. Simple tree pattern queries [22] are incapable of restructuring the results and hence not appropriate to serve as a view specification language either. In contrast, our semantic caching system ACE-XQ targets an XML query language in the real-world, i.e., the standard XML query language XQuery composed of nested block-based structures. To our best knowledge, no work so far has tackled this issue for XQuery, which is obviously of practical interest due to its increasing popularity.

Empowered by nested FLWR clauses, the expressiveness of XQuery is beyond pure tree pattern matching and other navigation-based languages mentioned above in that it is a strongly typed query language with a hybrid of features from both logic and functional programming languages. For example, variable-based XQuery expressions induce variable dependencies, and the nested FLWR structure implies functional control flow. Our containment checking algorithm hence takes into consideration both variable dependencies and control flow dependencies. In XCacher [22], containment mapping is based on the graph homomorphism by simply matching labels between the input tree pattern queries. The containment checking of two simple queries is mainly based on the logical implications between their *value conditions*. Whereas the containment mapping module in our ACE-XQ system incorporates the type inference and subtyping mechanisms to facilitate our containment checking process. It can handle complex queries with nested structures and the restructuring capability. In this sense, our ACE-XQ provides a more comprehensive XQuery-based caching system than XCacher.

8.2 Cache Space Management and Replacement Strategies

As the cache space is a limited resource, the cache may need to discard some data to free space for new data. To this end, various replacement schemes have been studied [37,33,23,4,35,28]. Yet, the Least Recently Used (LRU) replacement scheme is still widely used due to its simplicity. While simple, it adapts well to the changes of the workload, and has been shown to be effective when recently referenced objects are likely to be re-referenced in the near future [13]. A main drawback of the LRU scheme, however, is that it cannot exploit regularities in region accesses such as sequential and looping references and thus yields degraded performance [23] in such cases. Instead of using recency as the parameter for replacement, the Least Frequently Used (LFU) replacement scheme [37] uses reference frequency. LFU assumes that the more *often* a query is being used to answer sequential queries, the more likely it is to be used to answer a future query. However, a potential hole of LFU is that some data may accumulate its use to a high number and then is never used again. When applying the LFU scheme, such data tend to be more difficult to purge to free space for other useful data.

The varieties of the LRU and LFU schemes include the LRU-K scheme [33], the IRG scheme [35] and the LRFU scheme [28]. The LRU-K scheme bases its replacement decision on the regions' kth-to-last reference. The IRG scheme [35] considers the inter-reference gap factor and the LRFU scheme [28] considers both the recency and frequency factors. These schemes, however, show limited performance improvements because they do not consider regular references such as sequential and looping references. There are other proposed replacement schemes oriented to the reference regularities. For example, the 2Q scheme [23] can quickly remove from the cache sequentially-referenced blocks and looping-referenced blocks with long loop periods. The SEQ scheme [35] detects long sequences of page faults and applies the Most Recently Used (MRU) scheme to those pages.

In the Web context, other replacement functions have been proposed to address the size and latency concerns. Among them, GreedyDual [46] is a simple yet popular algorithm which handles variable-cost cache replacement. One of its extended versions GreedyDual-Size [5] combines locality, size and latency cost concerns to achieve a better performance in terms of hit ratio and latency reduction.

Different from the traditional replacement schemes designed for replacing buffer pages [6] or data tuples [16] in the database systems, a semantic caching system based on the relational model replaces query-based regions. That is, the *hits* and *recency* values are recorded for each query and the replacement unit is the query descriptor and the associated query results.

The partial replacement strategy utilized in our semantic caching system ACE-XQ maintains user access statistics for finer-grained fragments within a query region

at the path level. Our replacement function is based on several parameters such as reference frequency, initial fetching cost and the number of objects involved. For example, since the number of objects along different paths may differ significantly, our replacement function favors removing a larger fragment in case of a tie of other statistics values of two XPathRows for replacement efficiency.

In XCacher [22], all cached queries are integrated to be represented by one *modified incomplete tree (MIT)*. Corresponding to the different value domains specified in consecutive queries, a set of disjunctive conditional tree types are specified in MIT. The concept of specialized tree type in XCacher is similar to that of query region in ACE-XQ. Hence their replacement by utilizing the update statement is analogous to our partial replacement strategy in a sense. However, when a new query enters XCacher, only the incremental value differences caused by it with respect to MIT are specialized into new disjunctive types to be added into MIT. This means that they do not identify the overlapped region. Thus there is no natural way to record the relative high usage of the overlapped fragment in XCacher. In this sense, their replacement strategy is not as fine-grained as our replacement, which may incorporate a variety of replacement policies.

9 Conclusion

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

In this paper, we also report on extensive experiments which are conducted to compare the performance of the partial replacement strategy and the total replacement strategy. The experimental results clearly demonstrate the advantages of the partial replacement over the total replacement in most scenarios, in terms of the query performance measured by hit count ratio, hit byte ratio and the response time. These experimental results may help us in determining the deployment of favorable replacement strategies given the particulars of the scenarios.

References

[1] A. Deutsch and V. Tannen. Containment of Regular Path Expressions under Integrity Constraints. In *8th Int. Workshop on Knowledge Representation Meets Databases (KRDB), Rome, Italy*, pages 1–11, June 2001.

[2] P. Buneman andS. Davidsonand W. FanandC. Hara. Keys for XML. In *World Wide Web Conference (WWW10), Hong Kong, China*, pages 21–36, 2001.

[3] D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Y. Vardi. View-Based Query Processing for Regular Path Queries with Inverse. In *Symposium on Principles of Database Systems (PODS), Dallas, Texas*, pages 58–66, May 2000.

[4] 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.

[5] 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.

[6] M. J. Carey, M. J. Franklin, and M. Zaharioudakis. Fine-Grained Sharing in a Page Server OODBMS. In *SIGMOD, Minneapolis, Minnesota*, pages 359–370, 1994.

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

[8] C. M. Chen and N. Roussopoulos. The Implementation and Performance Evaluation of the ADMS Query Optimizer: Integrating Query Result Caching and Matching. In *EDBT, Cambridge, United Kingdom*, pages 323–336, Mar. 1994.

[9] 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.

[10] L. Chen and E. A. Rundensteiner. A Semantic Caching System for XQueries. Technical report, Computer Science Department, WPI, 2003. in progress.

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

[12] L. Chen, S. Wang, and E. A. Rundensteiner. A Fine-Grained Replacement Strategy for XML Query Cache. In *4th Intl. Workshop on Web Information and Data Management (WIDM'02), McLean, Virginia*, pages 76–83, November 2002.

[13] E. G. Coffman and P. J. Denning. Operating Systems Theory. Prentice-Hall International Editions, 1973.

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

[15] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A Query Language for XML. In *Proceedings of the 8th International World Wide Web Conference (WWW-8), Toronto, Canada*, volume 31, pages 1155–1169, 1999.

[16] 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.

[17] D. Florescu, A. Levy, and D. Suciu. Query Containment for Conjunctive Queries With Regular Expressions. In *Symposium on Principles of Database Systems (PODS), Seattle, Washington*, pages 139–148, June 1998.

[18] G. Miklau and D. Suciu. Containment and Equivalence for an XPath Fragment. In *Symposium on Principles of Database Systems (PODS), Madison, Wisconsin*, pages 65–76, June 2002.

[19] H. Hosoya and B. C. Pierce. XDuce: A Typed XML Processing Language. In *WebDB, Dallas, Texas*, pages 111–116, May 2000.

[20] H. Hosoya and J. Vouillon and B. C. Pierce. Regular Expression Types for XML, Montreal, Canada. In *International Conference on Functional Programming (ICFP)*, pages 11–22, 2000.

[21] L. M. Haas, D. Kossmann, and I. Ursu. Loading a Cache With Query Results. In *Proceedings of the 25th VLDB Conference, Edinburgh, Scotland*, pages 351–362, 1999.

[22] V. Hristidis and M. Petropoulos. Semantic Caching of XML Databases. In *5th International Workshop on the Web and Databases (WebDB), Madison, Wisconsin*, pages 25–30, June 2002.

[23] T. Johnson and D. Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of the 20th International Conference on VLDB*, pages 439–450, 1994.

[24] A. M. Keller and J. Basu. A Predicate-based Caching Scheme for Client-Server Database Architectures. *The VLDB Journal*, 5(1):35–47, 1996.

[25] L. Breslau and P. Cao and L. Fan and G. Phillips. the Implications of Zipf's Law for Web Caching. In *WWW Caching Workshop, Wisconsin, MI*, June 1998.

[26] L. Breslau and P. Cao and L. Fan and G. Phillips. Web Caching and Zipf-like Distributions: Evidence and Implications. In *INFOCOM (1), New York, NY*, pages 126–134, 1999.

[27] P. A. Larson and H. Z. Yang. Computing Queries from Derived Relation. In *VLDB, Stockholm, Sweden*, pages 259–269, Aug. 1985.

[28] 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.

[29] A. Levy, A. Mendelzon, Y. Sagiv, and D. Srivastava. Answering Queries Using Views. In *PODS, San Jose, CA*, pages 95–104, June 1995.

[30] M. Fernandez and J. Simeon and P. Wadler. A Semi-monad for Semi-structured Data. In *ICDT, London, UK*, pages 263–300, January 2001.

[31] G. W. Manger. A Generic Algorithm for Merging SGML/XML-Instances. In *XMLEurope 2001, Berlin, Germany*, 2001.

[32] I. Manolescu, 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.

[33] 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.

[34] Y. Papakonstantinou and V. Vassalos. Query Rewriting for Semistructured Views. In *SIGMOD, Philadelphia, USA*, pages 455–466, 1999.

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

[36] J. Robie, J. Lapp, and D. Schach. XML Query Language (XQL). http://www.w3.org/TandS/QL/QL98/pp/xql.html.

[37] 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.

[38] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On Rules, Procedures, Caching and Views in DataBase Systems. In *SIGMOD, Atlantic City, NJ*, pages 281–290, May 1990.

[39] I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S Weld. Updating XML. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Santa Barbara, CA*, pages 413–424, May 2001.

[40] V. Benzaken and G. Castagna and A. Frisch. Regular Language Description for XML (RELAX). In *Technical Report, ISO/IEC DTR 22250-1*, 2001.

[41] V. Benzaken and G. Castagna and A. Frisch. Semantic Subtyping. In *IEEE Symposium on Logic in Computer Science (LICS'2002), Copenhagen, Denmark*, pages 137–146, July 2002.

[42] W3C. XML Query Use Cases, W3C Working Draft 02, May, 2003. http://www.w3.org/TR/xquery-use-cases.

[43] W3C. XML Path Language (XPath)Version 1.0. W3C Recommendation. http://www.w3.org/TR/xpath.html, March 2000.

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

[45] S. A. Yahia, S. Cho, L. V. Lakshmanan, and D. Srivastava. Minimization of Tree Pattern Queries. In *SIGMOD, Santa Barbara, California*, pages 497–508, June 2001.

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