Argos: Efficient Refresh in an XQL-Based Web Caching System

Luping Quan, Li Chen and Elke A. Rundensteiner
Department of Computer Science,
Worcester Polytechnic Institute, Worcester, MA 01609
{lupingq|lichen|rundenst}@cs.wpi.edu

Abstract:
The Web has become a major conduit to information repositories of all kinds. Web caches are employed to store a web view to provide an immediate response to recurring queries. However, the accuracy of the replicates in web caches encounters challenges due to the dynamicity of web data. We are thus developing and evaluating a web caching system equipped with an efficient refresh strategy. With the assistance of a novel index structure, the Aggregation Path Index (APIX), we built Argos, a web cache system based on the GMD XQL query engine. It can achieve a high degree of self-maintenance by diagnosing irrelevant data update cases and hence greatly improve the refresh performance of the materialized web view. We also report experimental results comparing its performance with that of alternative solutions.

Keywords: XQL Query Engine, Web Cache, View Maintenance, Indexing, Data Update.

1 Introduction

The advent of the web has dramatically increased the proliferation of information of all kinds. XML [4] is rapidly becoming popular for representing web data as it brings a finely granulated structure to the web information and exposes the semantics of the web content. In web applications including electronic commerce and intelligent agents, web view mechanisms [13, 15] are recognized as critical and are being widely employed to represent users' specific interests. To compute an integrated web view, web information that resides at sites without query capabilities may need to be shipped over to the web view site to be queried against. Such a transmission of a large volume of data across the network can be costly. Hence the view results are oftenly cached locally for recurring queries instead of being re-computed on the fly.

However, the cached view results may become stale due to the flux of the content or the structure of the underlying web sources. With many applications requiring an up-to-date content of the web view at all times, two alternative web view materialization strategies, namely, materialized or materialized on-demand can be adopted [10]. While they are evaluating the trade-off of whether to materialize a web view and when to refresh it, we aim at improving the refresh efficiency of an materialization strategy. In other words, we present an efficient web view *maintenance* technique designed to minimize the delay of keeping a web view consistent with its updated data sources. This efficiency of adaptation to changes is very critical in the era of electronic commerce. For example, a bid site allows its users to post queries in a watch list for goods of interest to them. For a certain period, users expect to be notified of any changes to the view results in their watch list. The efficiency of the notification will effect the customers' business decisions and hence is critical.

It is observed that not all updates affect cached view results. Therefore recent research efforts [5, 1] study self-maintenance by utilizing auxiliary information piggybacked during the initial view computation phase. Along the same direction, we have developed a materialized web view cache system with an efficient refresh capability by employing a novel index structure [3]. It can be utilized to discover many self-maintenance cases and thus to eliminate unnecessary accesses to base web sources.

In this paper, we describe the web cache system called Argos that adopts this novel refresh technique. Argos is based on the query capability of XQL [14], which is the first XML-based query language under full implemention. We describe the architecture of the Argos system, its index structure and maintenance strategy. We also present the preliminary experimental results of comparing the Argos approach with the naive maintenance approach.

2 Argos System Framework

We assume a distributed environment with multiple web sources registered at the site where the view cache maintained by Argos will reside (see Figure 1). We consider data sources that contain XML documents. They are parsed once and stored in the binary form of persistent DOMs (PDOM) [8, 7], which from then onwards are accessible to DOM operations without the parsing overhead.

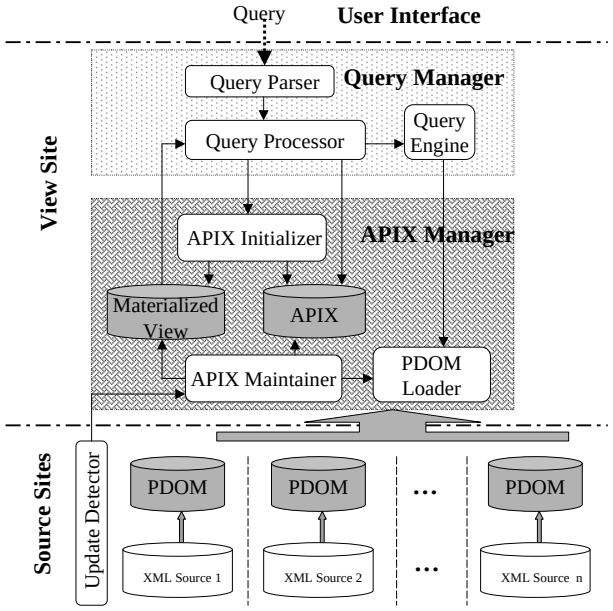


Figure 1: System Framework of Argos

The workflow of the Argos system consists of three phases. First, given a specific XQL query defining the request of a web view, the initial web view results are computed by shipping the remote PDOMs over to the local site and then processing the query against these sets of PDOMs using the GMD XQL query engine [8]. In the Argos system, we built an auxiliary index structure called the Aggregate Path IndeX (APIX) [3] to reflect the structural pattern of the query. During the web view computation, maintenance relevant data is piggybacked to fill in the APIX structure so that it can later be utilized for future self-maintenance. Then after a notification is sent to the view site when an update is detected at any registered web source. The APIX Manager then diagnoses whether it is an irrelevant update by performing self-maintenance tests with the help of the cached objects in the APIX structure. If it is a self-maintenance case, the web view is maintained locally without access to web sources. Otherwise, the third phase of maintenance follows. Namely, the APIX maintainer submit a relevant partial query against the web source to extract relevant information needed for view compensation. In this phase, the APIX Manager is able to limit the web source re-access to its minimal scope. Lastly, the web view is compensated by the additional query results.

As shown in Figure 1, the main part of the Argos system sits in the local web view site consisting of two main modules: Query Manager and APIX Manager. The former is further decomposed into a *Query Parser*, which accepts a specified query in XQL and parses it into an abstract syntax tree, and a *Query Processor* based on the GMD XQL query engine. Within the APIX manager, the *APIX Initializer* module is responsible for the initialization of the APIX structure for a given query and the generation of the materialized web view according to the query processing results. Once the *APIX Maintainer* is notified by the *Update Detector* of an update to a web source, the *APIX Maintainer*, performs view maintenance with the help of *APIX*. In Setction 3, we will describe the APIX structure and its maintenance algorithms in detail.

3 Argos Maintenance Strategy

Given a data update, a *naive* algorithm simply refreshes the web view by re-computing it from scratch. This is as costly as the initial web view computation over all its XML sources [1] since it may involve uploading and processing queries against each of them. Therefore, given a materialized web view at the view site, recent research work [5, 1] is striving to provide an efficient maintenance strategy to minimize the refresh delay of the web view.

The incremental view maintenance approach proposed by Abiteboul et. al [1] utilizes an auxiliary index structure called *RelevantOid* at the view site, which stores all the objects that are relevant to query. For example, if the given XQL query is like "/publisher/book/author", then all the *publisher, book* and *author* elements in the XML sources will be kept in the RelevantOid index. When an update happens to an author element *au1*, this maintenance approach needs to query against all PDOMs and process all *publisher, book* elements before it proceeds to the *author* element type and binds only *au1* to it. Using this approach, an object is kept in the RelevantOid index as long as it is bound to some query variable. Therefore the size of the RelevantOid index tends to be large and the self-maintenance test is not efficient.

We propose a novel index scheme, called Aggregate Path IndeX (APIX) [3], address the indexing

[1] In our experimental study, only the PDOM instead of the XML document is shipped to avoid the parsing overhead.

needs of maintaining the materialized view, besides that of supporting the query process. During the initial web view computation phase, APIX is populated with a collection of qualified objects with respect to the query path or pattern. An object is "qualified" when it has the children elements of the required types. Together with each of these objects also stores the statistics information about the support of the pattern and an indicator (true/false boolean value) about whether it leads to an leaf object satisfying the query predicates. Such information can be utilized during the maintenance phase to determine whether an update would effect the instantiations that fulfill a given query criteria by looking forward the most small necessary steps. APIX can also be used to diagnose many more irrelevant update cases than the RelevantOid approach. Figure 2 illustrates the basic idea of the Argos maintenance strategy using APIX.

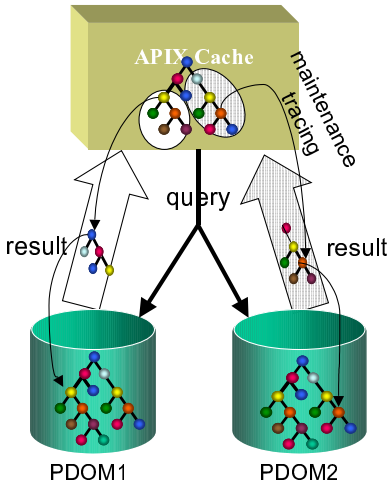


Figure 2: Illustration of the APIX Maintenance

```
<?xml version="1.0"?>
<invoicecollection>
<invoice>
   <customer> Wile E. Coyote, Death Valley, CA </customer>
   <annotation> We guarantee return rights </annotation>
   <entries n=2>
      <entry quantity=2 total_price="134.00">
         <product maker="ACME"
            prod_name="screwdriver" price="80.00"/>
      </entry>
      <entry quantity=1 total_price="20.00">
         <product maker="ACME"
            prod_name="power wrench" price="20.00"/>
      </entry>
   </entries>
</invoice>
<invoice>
   <customer> Camp Mertz </customer>
   <entries n=2>
      <entry quantity=2 total_price="80.00">
         <product maker="BSA"
            prod_name="book" price="40.00"/>
      </entry>
      <entry quantity=2 total_price="26.00">
         <product maker="BSA"
            prod_name="pen" price="13.00"/>
      </entry>
   </entries>
</invoice>
</invoicecollection>
```

Figure 3: An Example XML Document

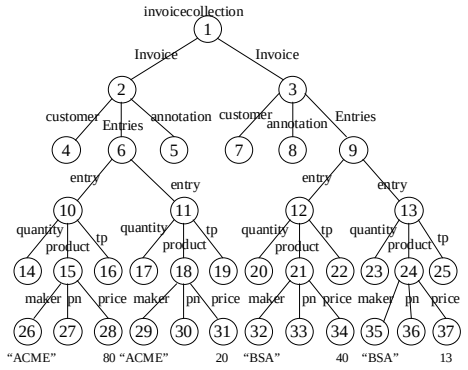


Figure 4: DOM Tree of the Example XML Document

3.1 APIX Structure

Unlike the data structure summary in the Lore [12] system called DataGuide [6], which is subject to change upon updates on the base data, the structure of an empty APIX reflects the structural pattern of a given query and hence is stable.

Given an example XML document (see Figure 3.1 for its content and Figure 4 for its parsed DOM representation), we assume the XQL query below gives the specification of the web view to be maintained.

```
//entry[@quantity=2]/product[@maker="BSA" and @price≤"20"]
```

The semantics of this XQL query is to obtain all products whose immediate parent *entry* has an attribute *quantity* with the value of 2, and the product is made by "BSA" and its price is less than or equal to 20. The notations "/" and "//" denote the path operators for immediate children and for arbitrary depth descendants respectively. The symbol "@" is added as prefix to attribute names. These symbols together with the element or attribute names immediately following them specify the navigation al paths, along each step of which an intermediate search context (a set of objects) is obtained. The partial query within "[ ]" denotes a filter specifying what to further select from the left-side search context for the next step navigation.

By decomposing a given query into parts, each of which is a set of partial queries, we capture the overall navigational pattern in a tree-like structure (see Figure 5). Each node represents a query variable (assigned for an element or attribute name) with its outgoing labels denoting the set of partial queries aggregated. For example, the only partial query related to

the root object of the given example XQL is *"//entry"*. *"./@quantity"* and *"./product"* are the partial queries starting from the variable *entry*.

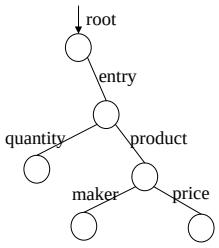
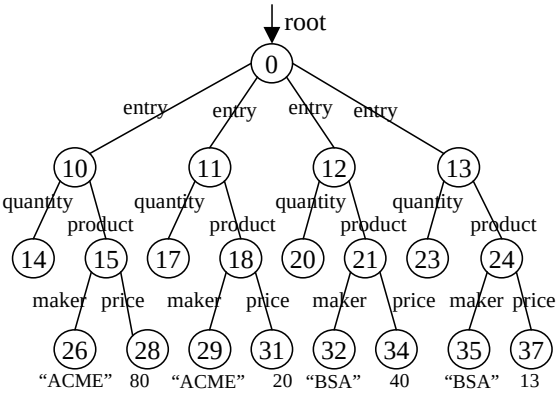


Figure 5: Tree Structure Representation of Query

3.2 APIX Initialization

Argos initially fills in the APIX structure with objects satisfying their query path patterns (the detailed APIX initialization is described in the Path Evaluation procedure in [3]). The resulting content of APIX is depicted in Figure 6.



root	entry	Truth Value
&0	&10, &11, &12, &13	

Partial query: //entry

entry	quantity	product	Truth Value
&10	&14	&15	
&11	&17	&18	
&12	&20	&21	
&13	&23	&24	

Partial query: ./@quantity, ./product

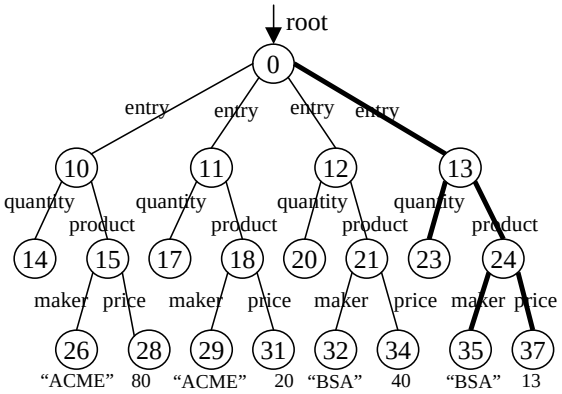
product	maker	price	Truth Value
&15	&26	&28	
&18	&29	&31	
&21	&32	&34	
&24	&35	&37	

Partial query: ./@make, /@price

Figure 6: The Initialized APIX

The size of the objects stored in APIX is much smaller compared to the RelevantOid index structure [1] due to its strict structural requirements on the data to be indexed. These objects are considered critical for maintenance.

We now continue to evaluate the value constraints on the leaf objects in the APIX structure and then propagate the results from bottom up to compute truth values for the non-leaf objects (see Figure 7). The truth value for an object is set to be true iff: *For each set of its children objects that are reached by the same outgoing label, there exists at least one true-valued object.*



root	entry	Truth Value
&0	&10, &11, &12, **&13**	**TRUE**

Partial query: //entry

entry	quantity	product	Truth Value
&10	&14	&15	FALSE
&11	&17	&18	FALSE
&12	&20	&21	FALSE
&13	**&23**	**&24**	**TRUE**

Partial query: ./@quantity, ./product

product	maker	price	Truth Value
&15	&26	&28	FALSE
&18	&29	&31	FALSE
&21	&32	&34	FALSE
&24	**&35**	**&37**	**TRUE**

Partial query: ./@make, ./@price

Figure 7: APIX with Truth Values

View results are derived from APIX by selecting only the true-valued objects along the paths. When an update occurs, it may affect the truth values in APIX and hence the view results. But in many cases, APIX and the view results can be self maintained without requiring any access back to the remote XML sources.

3.3 View Maintenance Using APIX

With the locally cached APIX nodes and their truth values, we can conduct a set of self-maintenance checks before any uploading of remote base data comes necessary. Our self-maintenance tests are more

effective than those proposed by [16]. We define an update to be maintenance irrelevant if it falls into one of the following four cases: 1) The new truth value of a leaf object is the same as that before its content value got modified. 2) For an *insert* or *delete* operation, the variable of the affected object is not an APIX node. 3) For a *delete* case, either the parent object or the object to be deleted is not cached in APIX. 4) The parent object of a value-changed leaf node is not in APIX.

Even if the self-maintenance test fails, APIX can compute the view patch either on APIX itself at the view site or by accessing a restricted scope of source data. For example, if the truth value of an APIX object is changed from *true* to *false* due to the deletion of one of its children objects, it may or may not affect the truth value of its parent object in APIX. APIX can help to propagate such an effect proper levels upwards till no further propagation is needed. On the other hand, an *insertion* may result in the remote access to the source data, however, the scope of data to be re-evaluated is restricted to the subtree of the highest ascender object that is in APIX yet.

We now illustrate how to maintain APIX and update the materialized view. Assume three value change operations occur sequentially: a) The value of object &17 is changed from 1 to 2, b) The value of object &19 is changed from 20 to 40, c) The value of object &29 is changed from "ACME" to "BSA".

For the first case, the truth value associated with object &17 is changed to be *true*. Thus it triggers a recomputing of the truth value of the label associated with object &17 and a propagation up to object &11. There the propagation stops since its truth value won't change because of the *false* of the other child object &18. For the second case, object &19 is not cached in APIX and it thus is an irrelevant update. For the third case, the truth value of object &29 is changed and it further causes that of its parent object &18 and grandparent &11 to be examined via propagation.

By computing the delta view objects based on the changed truth values in APIX, the materialized view can be easily maintained.

4 System Implementation

The Argos system has been developed on the Windows/NT platform using java. The implementation of the XQL parser is based on the Antlr [9] compiler toolsuite. Its query facility is based on the GMD-IPSI XQL engine with persistent DOM capability.

Argos is implemented as a client-server system. XML sources at multiple remote servers are registered at the client site where the user specifies the web view. Argos implements an id mechanism for each PDOM object to be referred by APIX. The APIX index and its manager module co-exist with the materialized web view at the local view site. Partial queries, filtering mechanisms and truth value propagation functions associated with APIX nodes collaborate to maintain the materialized web view and to minimize the search scope if loading of base data is inevitable.

5 Experimental Evaluation

Experimental tests are conducted to compare the refresh delay time of Argos to the naive recomputation approach. We use a large test dataset distributed at different sites. These test data are in either XML document or XML PDOM. We keep the overall data size and the query fixed and vary the type of updates. We are experimenting on a collection of Shakespeare plays [2] (the overall data size is about 7.5M bytes and the maximum document depth is 7) and the width of the query structure is 2.

In the first experiment, we test for each type of update 50 different cases, run each case 10 times, and take the average time (in millisecond) as the maintenance cost for each type of update. For the naive approach, it re-computes the view every time when an update occur. Its refresh time is always about 260 ms since it is irrelevant to the type of updates. We observe from the experimental result (Figure 8) that the APIX approach wins significantly over the naive approach especially in the *delete* and *update* situations. For the *insert* update cases, APIX can diagnose a few to be irrelevant. However, there remain most of the cases that re-evaluating the query is still inevitable and hence the overall average maintenance costs by the APIX approach is just slightly better than the naive approach. A *deletion* or a *change* involves the propagation of the changed value evaluation result or the dropping of object tuples at the local APIX site and thus tend to be much less costly (fast in refresh).

In the second experiment, We modify the documents to increase their depth up to 10 levels deep. We then design various update cases occuring at 10 different levels. Figure 9 shows that Argos achieves significant reductions in the refresh delay of the materialized web view for all the deletion and value change cases (consistent with the observation of the first experiment). Especially, when an update occurs closer to the root object, the maintenance cost

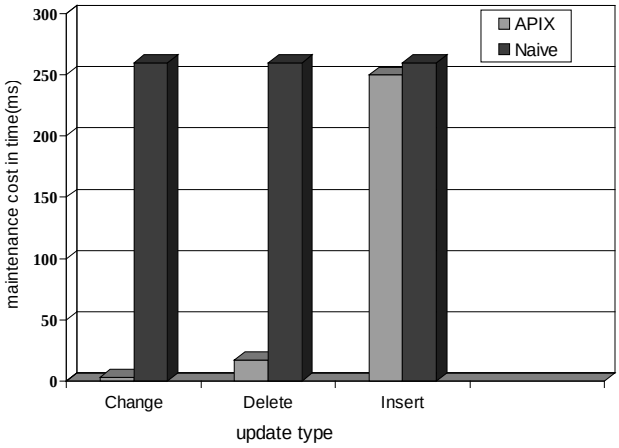


Figure 8: Different Types of Updates

is slightly less. This is because the higher the objects are (opposed to the lowest atomic objects), the shorter paths they may go through to propagate up their dropped/changed value evaluation information to maintain APIX.

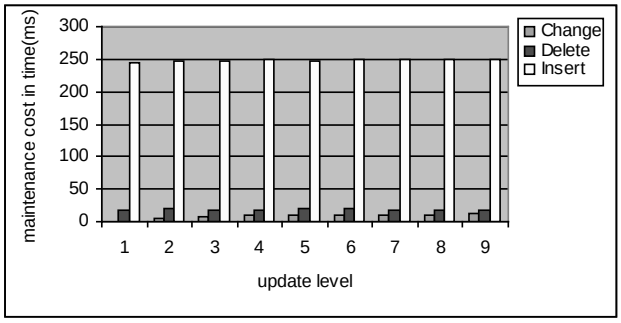


Figure 9: Different Levels of Updates

6 Conclusion and Future Work

We present the Argos client/server framework aiming to minimize the refresh delay of the materialized web view with the help of a locally cached index structure – APIX. We designed and implemented the Argos caching system based on the GMD-IPSI XQL engine, and also present preliminary experimental results.

References

[1] Document Object Model (DOM). http://www.w3.org/TR/REC-DOM-Level-1/.

[2] XML^{TM} . http://www.w3.org/XML, 1998.

[3] S. Abiteboul, J. McHugh, M. Rys, V. Vassalos, and J. Wiener. Incremental Maintenance for Materialized Views over Semistructured Data. In *The 24th International Conference on Very Large Databases, New York*, pages 38–49, August 1998.

[4] J. Bosak. Shakespeare 2.00. http://metalab.unc.edu/bosak/xml/eg/shaks200.zip.

[5] L. Chen and E. A. Rundensteiner. Aggregation Path Index for Incremetnal Web View Maintenance. In *The 2nd Int. Workshop on Advanced Issues of E-Commerce and Web-based Information Systems, San Jose, to appear*, June 2000.

[6] D. Gluche, T. Grust, C. Mainberger, and M. H. Scholl. Incremental Updates for Materialized OQL Views. In *The 5th International Conference on Deductive and ObjectOriented Databases (DOOD), Switzerland*, pages 52–66, Dec. 1997.

[7] R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *the 23rd Int. Conf. on Very Large Databases (VLDB), Athens, Greece*, pages 436–445, Aug. 1997.

[8] G. Huck and I. Macherius. GMD-IPSI XQL Engine. http://xml.darmstadt.gmd.de/xql/, 1999.

[9] J. Inc. ANTLR: Complete Language Translation Solutions. http://www.antlr.org/.

[10] A. Labrinidis and N. Roussopoulos. On the Materialization of WebViews. In *The Workshop on the Web and Databases (WebDB'99), Philadelphia, USA*, pages 79–84, June 1999.

[11] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A Database Management System for Semistructured Data. In *SIGMOD Record 26(3)*, pages 54–66, Sep. 1997.

[12] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object Exchange Across Heterogeneous Information Sources. In *IEEE Int. Conf. on Data Engineering*, pages 251–260, 1995.

[13] J. Robie, J. Lapp, and D. Schach. XML Query Language (XQL). http://www.w3.org/TandS/QL/QL98/pp/xql.html, Sep. 1998.

[14] D. Srivastava, S. Dar, H. V. Jagadish, and A. Y. Levy. Answering sql queries using materialized views. In *Proceedings of VLDB*, 1996.

[15] Y. Zhuge and H. GarciaMolina. Self-Maintainability of Graph Structured Views. In *Technical Report, Computer Science Department, Stanford University*, pages 116–125, October 1998.