APIX: An Efficient Approach to Maintain Web Views *

Li Chen and Elke A. Rundensteiner

Department of Computer Science
Worcester Polytechnic Institute
Worcester, MA 01609–2280
{lichen|rundenst}@cs.wpi.edu

Abstract

Due to the increase of web data, web view mechanisms are extensively employed to serve customized data views to users. They are usually materialized to achieve fast query response time. However, developing maintenance techniques for materialized web views over dynamic web data sources becomes more challenging than in the conventional database context. To compute web view "patches" for incremental maintenance, a large number of accesses back to base data is usually inevitable. This could cause a severe performance penalty due to the heavy network overhead and the intense contention for base data. In this paper, we address this problem for views defined over XML-based data sources specified in an XQL-like query language. We demonstrate how the requirements of such a query can be represented by means of a *query graph*. We propose a novel index scheme, *aggregation path index (APIX)*, for capturing the complex query pattern and caching maintenance critical data. We also describe a two-phase query evaluation strategy to filter out the maintenance critical information to be stored in APIX. We show that by utilizing APIX, the maintenance of web views becomes more efficient compared to the state-of-art solutions due to cutting down the number of accesses to base data. Our cost analysis illustrates the performance benefits of our approach.

Keywords: Web View Maintenance, Query Graph, Aggregation Path Index, Self-maintainability, Query Performance.

*This work was supported in part by the NSF NYI grant #IRI 94-57609. One of the authors, Li Chen, is the receipient of IBM cooperate fellowship, thus we would like to thank IBM for the IBM Partnership Award and our collaborators at IBM Toronto for their support.

1 Introduction

1.1 Motivation

In conventional database systems, view mechanisms [PGMW95, LMSS95, SDJL96] are popular. View results are generally cached to achieve fast response time for a recurring query over such a view. As more and more applications are becoming web-oriented, views are beginning to also be built over web data, then called web views. Those web views are quickly proliferating due to the popularity of eXtensible Markup Language (XML) [Gro98a] and the web data represented in XML and exchanged in XML. Web view mechanisms require the ability to query data in XML, to integrate data from different sources and to restructure it to provide users with a customized view of the data of interest.

In this paper, we are aiming to tackle the web view maintenance problem due to the dynamicity of the web data sources that the cached (materialized) web view is derived from. The view maintenance techniques in conventional database context have been a well-studied topic [AMR$^+$98, GGMS97, GM95, GMS93, KLMR97, RKRC96, SLT91]. However, developing these techniques in the web data context is more challenging due to the lack of a regular schema restricting the web data and due to the shareability of web data sources enabling a potentially large amount of data re-evaluation caused by an update on a single data object. Specifically, the challenges in maintaining a web view are three-fold. (1) XML data has a hierarchical structure, hence the query against it must be evaluated globally from the root entry, instead of being decomposed into subqueries over small units like separate tables in traditional data warehouses and (2) A Document Type Definition (DTD) [W3C98] allows flexibility of the XML data structure and thus forms a great diversity for potential data instance structure. Thus every possible change in data value or in data structure may cause large accesses of base data for web view maintenance. (3) A single update could cause effects to many different objects due to the existence of not only the inclusion hierarchy structure but also the rich linking mechanism.

A few research efforts [GGMS97, AMR$^+$98] have focussed on developing web view maintenance techniques thus far. The naive approach [GGMS97] re-computes the view from the scratch, which is undesirable since it causes a large amount of accesses back to data sources upon each data update. The self-maintenance approach [ZG98a] analyzes the type of the update and the affected objects and checks their self-maintainability. Using this method, we can avoid the accesses to base data in many data update cases. The relevant-id approach [AMR$^+$98] improves over that by keeping locally a set of object identifiers (oids) for query relevant objects to save the unnecessary accesses to query irrelevant objects. One very important goal of our web view maintenance strategy is also to improve the web view maintenance performance by reducing the number of accesses back to base data sources since that would cause either network transfer delay or a bottleneck of the data server due to the high read/write frequency. Thus the performance may deteriorate or even cause a system breakdown.

1.2 Our Approach and Contributions

For the purpose of web view maintenance, we exploit the query procedure to identify a minimal set of base data that is maintenance critical. We cache it locally in a new index scheme to avoid unnecessary accesses to base data. For this purpose, we analyze the given web view specification and introduce a *query graph* to represent its requirements, which consists of three parts, one for the structural constraints of patterns, one for the value predicates and another one for the data to be returned. A *two-phase-evaluation* strategy is developed first to capture the objects satisfying the required structural patterns indicated by the query graph and then to compute the further restriction of the value predicates on these objects. The maintenance critical base data is identified using this strategy. We propose a novel index scheme, *Aggregation Path IndeX (APIX)*, to cache them and their associated auxiliary information such as statistics of label repetition, truth counts, etc. We present algorithms for efficient view selection and maintenance that exploit APIX to minimize the accesses to base data. Cost analysis and experimental studies are also performed to illustrate the performance benefits of our approach compared to the state-of-art solutions [AMR+98, ZG98a, ZG98b].

In a summary, the contributions of our work are:

- Propose a *query graph* to explicitly represent the query requirements of a web view definition (in our case specified in XQL) over the XML data.
- Develop a *two-phase-evaluation* strategy. Different from the depth-first traversal of the standard evaluation strategy, it decomposes the query evaluation into two separate phases of structure pattern matching and value predicates evaluation.
- Present *APIX* as a *novel index scheme.* Its tree structure represents complex query patterns. It accommodates auxiliary information such as the statistical summaries of the object occurrences of the same structure pattern and the value predicate results precomputed during the query evaluation phase.
- Describe the algorithms for view selection and incremental maintenance that exploit APIX to minimize accesses to base data.
- Analyze the cost of our approach and demonstrate that it wins over alternative state-of-art solutions.

1.3 Outline

In Section 2, we describe the XML web data model and the XQL query language we assume for our web view specification. In Section 3, we present the new concepts of the query graph and the two-phase-evaluation query process. We propose in Section 4 a novel index scheme, APIX, as the data structure for implementing the query graph and the local storage mechanism for the intermediate evaluation results from the query procedure. In Section 5, we introduce the self-maintainability tests as well as the incremental maintenance algorithms based on APIX. We conduct the cost analysis and experimental studies in Section 6 and wrap up our discussion in Section 7.

2 Web Data Modeling and Querying

2.1 Modeling Web Data Using XML

Recently, XML [Gro98a] has been proposed for modeling web data since it is rich in links and naming mechanisms. Given a collection of XML sources, we assume a resource manager that registers all of them by creating a global XML document with a new root element and one external entity referring to each XML source via the registered Universal Resource Identifier (URI). Thus we assume we are always asserting queries against one XML document.

Using a tree structure which is similar to the *directed acyclic graph (DAG)* model of OEM [MAG$^+$97], the elements and attributes of an XML document are denoted as nodes and navigation or structural relationships between elements are represented as directed labels between nodes. However, XML has richer modeling capabilities such as the order and the ID/IDREF mechanism. To model an XML document in this paper, we assign a unique *object identifier (oid)* to each XML node, treat attributes as leaf nodes attached to their element nodes and represent the "referring" links using dotted edges to distinguish them from the normal "parental" edges. This way, an XML tree becomes an interconnected graph due to the "referring" links by the ID/IDREF mechanism. All leaf nodes contain primitive type values and can be evaluated by predicates resulting in boolean values.

We now introduce an example XML document to be used throughout this paper (see Figure 1). It is an E-mall database integrating product information from a large number of shops, each of which has its own information such as the sale categories, its location as well as its products. For each product entry, it includes information such as the names, the makers, the prices and the component items. Figure 2 shows the tree structure model of this XML document.

2.2 Querying XML Data

Several query languages [Gro98b] have been proposed for web data, such as Lore [AQM$^+$97], YAT [SC98], XML-QL [DFF$^+$99], XQL [RLS98]. They vary in terms of their input, output, processing model, and the desired form in which results are returned. However, they have something in common in the sense that their query conditions are all composed of two parts: 1) a pattern matching model that filters elements in the input document; 2) a content assertion mechanism that imposes constraints on the values of objects. Among these query languages, XML-QL and XQL are especially designed for XML and are the two most promising candidates. In this paper, we adopt XQL since it features a concise syntax, a capability to preserve document order and relative hierarchy in the query result. Furthermore, a working implementation of its core query functionalities already exists [HM99, Eis99]. Efforts are being taken to extend it with abilities such as variable binding, processing joins across multiple documents, creating of new elements and restructuring of documents. In this paper, we focus on the web view specifications with the core query functionalities of XQL for simplification of discussion. The the evaluation of our approach could be applied later to XQL

```
<?xml version=“1.0”?>
<!ELEMENT E-mall (shop)*>
<!ELEMENT shop (entry)*>
<!ATTLIST shop name CDATA #IMPLIED
                  category CDATA #IMPLIED
                  location CDATA #IMPLIED>
<!ELEMENT entry (price?,item*)>
<!ATTLIST entry name CDATA #IMPLIED
                  maker CDATA #IMPLIED
                  quantity CDATA #IMPLIED>
<!ELEMENT price (#PCDATA)>
<!ELEMENT item (#PCDATA)>
   <E-mall>
      <shop name=“J.C. Penny” category=“fashion”/>
      <shop name=“Toys R us” category=“toy” location=“NYC”>
         <entry name=“little mermaid” maker=“ASE” quantity=12>
            <price>$18.00</price>
            <item>video</item>
         </entry>
         <entry name=“antz” maker=“ASE” quantity=20>
            <price>$15.00</price>
            <item>book</item>
            <item>puzzle</item>
         </entry>
      </shop>
      <shop name=“Sam’s” category=“toy”>
         <entry name=“batman” quantity=8>
            <price>$40.00</price>
            <item>game</item>
         </entry>
      </shop>
      <shop name=“Michael” location=“Boston”>
         <entry name=“pokemon”>
            <item>book</item>
         </entry>
      </shop>
      ...
   </E-mall>
```

Figure 1: Example XML Document

extensions.

We show in Figure 3 via an XQL example query the semantics of a web view definition over the example XML document in Figure 1. We assume that this web view is defined for some customer group and it will select out a collection of favorite product entries, each found in *toy* shops of this E-mall, its maker is *ASE*, still in stock (*quantity*>*0*), costs less than *$20* and contains at least one item of *book*.

In XQL syntax [RLS98], the symbols “/”, “//” and “@” denote path operators. Each path operator is immediately followed by a string indicating the path label via which a collection of data objects are accessed from the current object set. Assume each such object collection is bound to a variable type and thus each path label connects two variables. Given a query, the beginning variable is the root variable that corresponds to the root object of a data tree. “/” indicates a single-step path navigation, “//” denotes a navigation that can go arbitrarily deep, and “@” selects from attribute nodes of the current object collection. The symbol “[ ]″ is a filter containing subqueries that usually end with value predicates. The logical operators “and” and “or” stand for the conjunction and disjunctive operations on the results of multiple subqueries respectively.

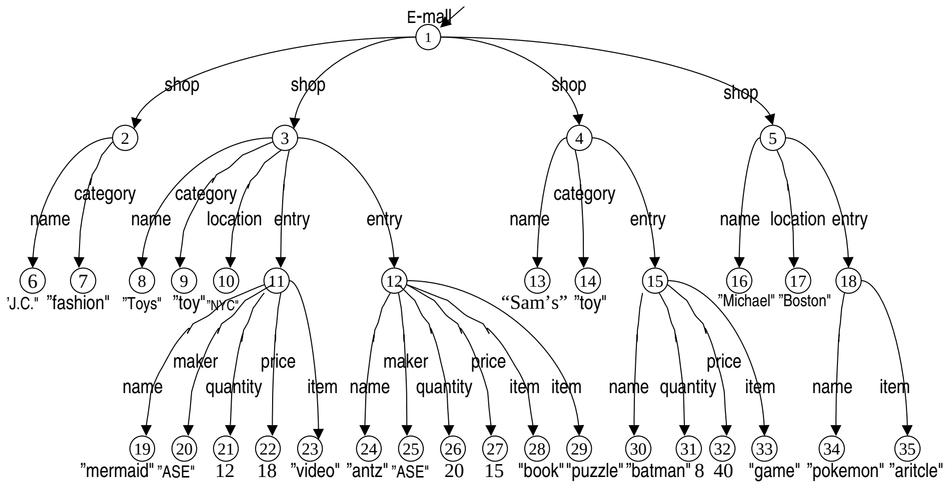


Figure 2: The Tree Structured Model of XML Document

Define web view *favorite_entries* as
/E-mall/shop[@category="toy"]/entry[@maker="ASE" *and* @quantity>0 *and* price<$20 *and* item="book"]

Figure 3: Example Web View Specification

We limit in this paper the output model of an XQL query to be that it can only return a flat collection of data objects with no hierarchy. Given an XQL query, all the labels outside of the filters can be concatenated to form a "straight" path via which the objects of request can be selected. Complex structural patterns and value predicates can be asserted via the subqueries within the filters. In this paper, we focus on exact path expressions that specify each single-step path and ignore the *regular path expression (RPE)* and the "//" operator which allow for subqueries with wildcards or of arbitrary depth. We believe that the results and conclusion from our APIX approach and its maintenance technology can be extended for RPE queries. We also assume the existential quantification of predicates.

2.3 XML Data Updates

We assume that each XML data source cooperates with the integration site and submits any change by sending up an update message. For example, the update message in Figure 4 adds a new piece of location information as the last child element to an existing shop named "J.C. Penny". This update fragment written in XML format envelops together a query selection statement with the update operation to specify the exact location where the update (in this case, it is an insertion of a subtree of data elements) takes place.

Updates on XML sources may affect the materialized web view and thus the need to re-evaluate the base data sources or compute "patches" for incremental web view maintenance. However, a large amount of accesses back to base data takes time and would impact the query performance greatly. Thus it is undesirable

```
<?xml version="1.0"?>
  <xmlupdate version="1.0">
    <insert select=/E-mall/shop[name="J.C. Penny"]>
      <element location=lastchild>
        <location>
          <Street>Piazza Mazzini 1</Street>
          <State>Italy</State>
          <City>Portofino</City>
        </location>
      </element>
    </insert>
  </xmlupdate>
```

Figure 4: An Update Fragment Message

and means of minimizing it has been explored by web view maintenance techniques.

3 Two-Phase-Evaluation Query Strategy

An efficient web view maintenance technique should be able to identify and cache the maintenance critical base data at the view site to avoid unnecessary accesses to base data stored at distributed site. For this purpose, we analyze a web view query by separating its requirement of the search pattern from that of the value predicates. We then present a corresponding two-phase-evaluation strategy to identify and pre-evaluate the maintenance critical data.

3.1 Standard Query Processing Strategy

A variety of query languages proposed for querying XML data have the capability to combine structural pattern constraints with value predicate assertions. Assume we have a web view specified by the XQL query in Figure 3 over the example XML document from Figure 1 with its tree representation illustrated in Figure 2. A standard query evaluation corresponds to a depth-first traversal of the XML data tree. The nodes are visited in the order of:

1, 2, 7, 3, 9, 11, 20, 21, 22, 23, 12, 25, 26, 27, 28, 29, 4, 14, 15, 31, 32, 33, 5, 18, 35

We indicate the traversal order through the tree in Figure 5 using the bold line the navigation track. We find that sometimes a long deep traversal from a non-leaf node is unnecessary if the node doesn't satisfied the path requirements on it. Take node *5* for example. Its descent node *35* is visited via the path */entry/item*. However, we could save this access if we were to check beforehand whether node *5* has any child object of type "category". Thus a potentially large search scope can be pruned by evaluating the aggregated path assertions on each accessed object.

In the maintenance strategy [AMR+98] proposed by Abiteboul et. al., the base data objects being accessed by the standard query evaluation are captured in an auxiliary index structure called the *relevant-*

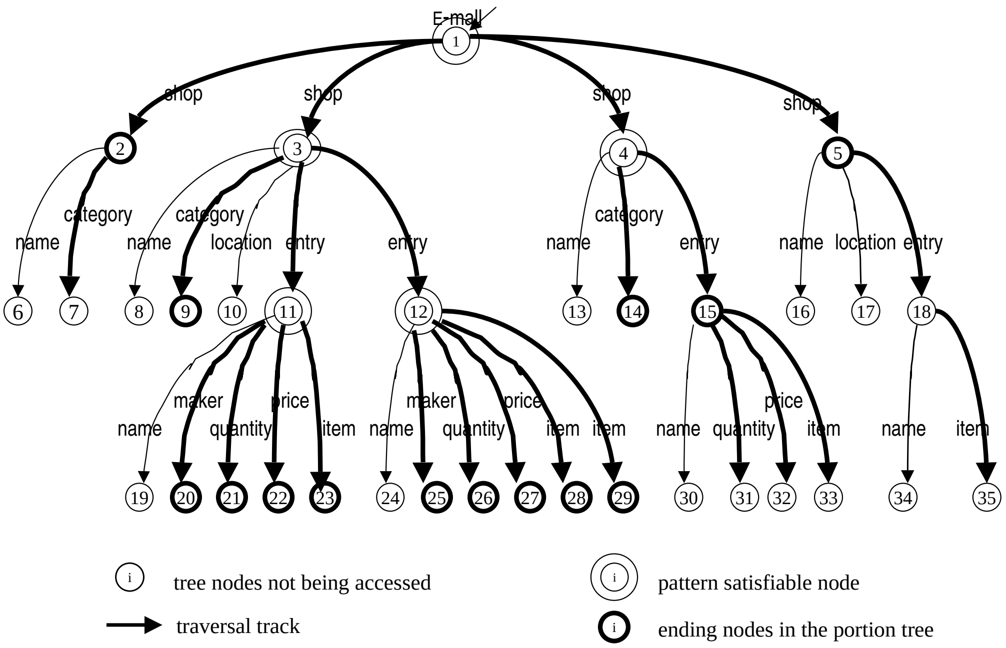


Figure 5: Query Track on The Example XML Tree

oid structure. From now onwards we refer to this method as the *RelOid* approach. When an update occurs to an object indexed by the relevant-oid structure, it is considered to be maintenance relevant and hence incurs accesses to base data. Thus the more objects are indexed in the relevant-oid structure, the less efficient the maintenance process would be. Therefore, we now introduce strategies to prune the traversal space to achieve evaluation efficiency as well as to improve maintenance performance by caching a reduced set of maintenance relevant objects. For this purpose, we propose a new query strategy, which is divided into two phases based on the two separated query requirements of path patterns and value predicates. We call it *two-phase-evaluation*. First we introduce how we propose to capture these two kinds of query requirements by a so called *query graph*.

3.2 Query Graph

A query in the XQL syntax can be decomposed into a list of query segments, each in the form of $v_l.p\ v_r$, with all types of path operators and filters are abstracted as a dot, indicating the navigation from the object collection on the left-side to that of the right-side via the path label p. We explicitly refer to the data collections that p is associated with by binding them to the variables v_l and v_r respectively. v_l of each query segment is termed a *head* and v_r is termed a *tail*. Starting from the root variable, which is bound to the root object of the data tree, each tail can be uniquely defined based on the head and the attached path label. Each *head* may appear in several query segments with multiple *tails*. For example, assume we are in the process of an XQL query *a[c]/b*. After the first single-step path evaluation, we obtain a runtime stack of objects of type "a", which are bound to the variable v_a and the evaluation of the remaining partial query *[c]/b* is to be continued. Analyzing the semantics of such a partial query, we observe that, besides the main path indicated by the immediate following label b outside of the filter "[ ]", the subquery within the filter introduces a branching path labeled c stemming from the current object collection. We thus would have query segments $v_a.b\ v_b$ and $v_a.c\ v_c$.

Definition 1 (Aggregated Path Pattern) *Given an XQL query Q, each head v_l on the left-hand of all its query segments is attached via a set of paths $\{p_1,\ p_2,\ ...,\ p_n\}$ to the corresponding tail set $\{v_{r1},\ v_{r1},\ ...,\ v_{rn}\}$. We call each such compound path $\vec{P} = \{p_1, p_2, ..., p_n\}$ starting at v_l the aggregate path pattern of v_l.*

In Figure 6 we show the decomposed query segments of the example XQL query from Figure 3. For example, $\vec{P}$ of the head e is the path set of $\{$*maker, quantity, price, item*$\}$ with the corresponding tail set of $\{m,\ q,\ p,\ i\}$.

r.E-mall E, E.shop s, s.category c, s.entry e, e.maker m, e.quantity q, e.price p and e.item i

Figure 6: Decomposed Partial Queries of the Example Web View Specification

Definition 2 (Query Graph) *Let G be a graph composed of two types of nodes. Each internal node $N_i = < v_l, \vec{P} >$ contains a head v_l and its aggregate path pattern $\vec{P}$. It is also linked to all the nodes containing its tails. Each leaf node is represented as $N_e = < v_e, pred >$ with pred representing the value predicate on v_e. We call G the query graph of a given XQL query Q and each variable appearing in G a query variable v of G.*

Figure 7 shows the query graph of the example XQL query (see Figure 3). Its tree-like structure graphically reflects the overall structural constraints on the XML data tree to be evaluated. Each tree node summarizes for each head v_l its aggregate path pattern $\vec{P}$ and the pointers along which its tails can be chased. "And" and "or" are the logic operators connecting the path members of a $\vec{P}$. In this paper, we only focus on the "and" semantics for the aggregated paths, which means that we project out a data object of type v_l if it lacks any of the outgoing labels as indicated in $\vec{P}$.

A query graph can thus be exploited to represent the evaluation criteria of a web view query. Its visual form can be utilized for helping users to formulate queries and to understand the requirements of a given query. For a set of web view queries whose structural constraints are the same but whose value predicates or return models are different, we can reuse the same query graph structure to capture the common path patterns of the heads.

In a summary, the benefits obtained by modeling queries using query graphs are described below:

- It is an intuitive way to represent the aggregate path pattern on each query variable.
- It is a canonical visual form of representing query evaluation criteria without ambiguity.
- It can facilitate users with the process of query formulation and query understanding.
- It would help with the query analysis and query rewriting strategy.

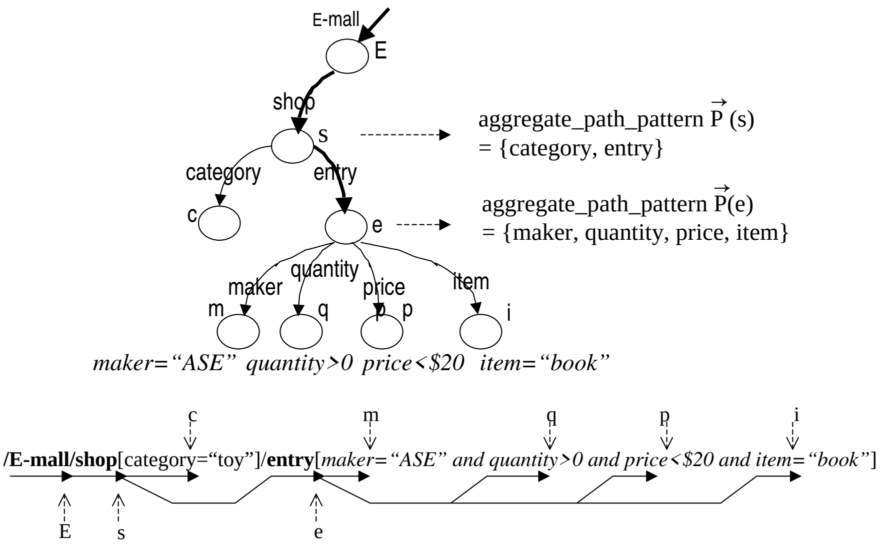


Figure 7: Query Graph for the Example Qeury

3.3 Two-Phase-Evaluation Strategy

Based on the two types of query requirements, the structural constraints and the value predicates, we develop a new query evaluation strategy that handles the evaluation in separate phases of **Path Pattern Evaluation** (PPE) and **Predicate Value Evaluation** (PVE). PPE assesses the pattern satisfiability of the data tree objects, while VPE evaluates the value predicates to determine what objects to return.

3.3.1 Path Pattern Evaluation

Once an XQL query is decomposed into query segments and the query graph G is formed, the first phase of the query evaluation, PPE, starts from the root object in a top-down traversal along the data tree T. This data tree traversal is synchronized with a traversal of G also from the root variable. At any point of the PPE phase, the current collection of objects $\{o\}$ is evaluated by the aggregate path pattern $\vec{P}$ of its binding variable v. The evaluation on each object o within the data collection employs a one-step *look-ahead* strategy to check on its direct children nodes for the pattern satisfiability of $\vec{P}$. If there exists for o at least one outgoing path instance for each required p_i within $\vec{P}$, then o is **pattern satisfiable**. In this case, we can further compute the duplicates of each path label p_i stemming from o, to which we refer as **Label Duplicates (Dup)**. o is not pattern satisfiable if any of these Dup values is 0. In this case, the subtree scope rooted from o is pruned to terminate any further traversal.

We show in Figure 8 the PPE procedure by providing four input parameters, a query graph G, a data tree T, a query variable v in G from where the query starts and its object mapping {o} from where the data tree navigation is conducted.

Definition 3 (Portion Tree) *A node N of a given data tree T is called* **query relevant** *for a given query Q, if it is accessed by the path pattern evaluation phase to have its pattern satisfiability evaluated. A portion*

```
Procedure PPE(G, T, v, {o})
Input: G is the query graph used to express the query constraints;
        T is the data tree to be evaluated by G;
        v is a query variable in G from where the query starts;
        {o} is a collection of objects bound to v;
Output: all query_relevant data objects that are accessed by PPE;
// {vars} is a queue of all children query variables of v in G;
// objects in {o} are to be evaluated by the P⃗ of v;
if v is a leaf node in the query graph
    if {vars} is empty
        return all objects that are marked query_relevant;
else
    foreach object o in the object set of {o}
        o.pattern_satisfiable = true;
        o.query_relevant = true;
        foreach path p_i in the P⃗ of v
            if there is no direct child object attaching to o via p_i
                o.pattern_satisfiable = false;
                break;
            else
                foreach child object o_i bound to v_i via p_i
                    o_i.query_relevant = true;
get from {var} the next variable v_j and its data collection o'
PPE (G, T, v_j, {o'});
```

Figure 8: Path Pattern Evaluation Procedure

tree PT of T for the query Q is then defined to be a subtree of the data tree T that is composed of all nodes that are query relevant with respect to Q.

We show the *query relevant* nodes of the data tree from Figure 2 as the nodes being pointed to by bold edges in Figure 5. Later in Section 5, we will see that our web view maintenance technique treats these nodes within the *portion tree* as maintenance critical data objects. A *direct* child node of a *pattern satisfiable* node (marked by a double circle) is *query relevant* since it is always accessed by PPE. However, it is not necessarily *pattern satisfiable* if it fails its *aggregated path pattern* constraints. Thus a *query relevant* but not *pattern satisfiable* node stops the traversal from further probing its children nodes, which is guaranteed to never become query relevant. Compared to the scope traversed by the previous standard query procedure (see Section 3.1), PPE can save accesses to some data nodes. For example, node *2* and node *5*, both of which are bound to the variable s in the query graph, are evaluated against their $\vec{P}$ of {*category, entry*}. Node *2* fails the *entry* type children nodes and node *5* has no *category* type children. Hence they are *query relevant* but not *pattern satisfiable* nodes and there is no need to access their children including nodes *7*, *18* and *35*. They would however be accessed by the standard query evaluation procedure.

3.3.2 Value Predicate Evaluation

After the path pattern matching phase of PPE, VPE starts a bottom-up traversal of the *portion tree* obtained by PPE. The traversal begins with the value evaluation on the leaf nodes of a *portion tree* and propagates the results upwards for each node in the portion tree. It is noticed that a leaf node of a portion tree, which we name as an *ending node*, is not necessary a leaf node of the data tree since it is decided by PPE.

For each *ending node* of a *portion tree*, it is assigned a boolean truth value indicating whether it is satisfied the value predicate asserted. We call such a truth value a **seed** since it is computed at the beginning of VPE and to needs to be propagated upwards. However, an *ending node* does not always have a value predicate. In this case, if it is a *pattern satisfiable* node, then the seed is a *true/1* value by default. Otherwise, its seed value is *false/0*. For example, the seed for node *2* is *false/0*. We summarize the seed assignment formula in Figure 3.3.2.

$$seed(o) = \begin{cases} \textit{the resultant truth by value predicate} & \textit{if any value predicate} \\ false/0 & \textit{if no value predicate nor pattern satisfiable} \\ true/1 & \textit{if no value predicate nor pattern requirement} \end{cases}$$

Once we compute all *seed* values, they are propagated upwards and converge at other *query relevant* nodes in the *portion tree* to determine their effects on the view results to be returned. We refer to the boolean effects of these nodes as **Evaluation Results (ER)**. We now explain the VPE procedure in Figure 9 by describing how to obtain the ER value of an object o in the portion tree by propagating their children ER values. Suppose o is bound to the query variable v in the query graph, which has a set of tail variables v_1, v_2, v_i, ..., v_n, the children nodes of o are correspondingly partitioned into n clusters. For the cluster of objects that are bound to v_i, we compute the **Cluster Truth (CT)** value according to the semantics of the existential quantifier, i.e., if there *exists* any child object with a *true* ER value, the CT value is *true*. Otherwise, it is *false* and would result in a *false* ER value of o. By representing a *truth* value in its integer format, i.e., using $i \geq 1$ to represent a *true* value and $i = o$ a *false* value, we can obtain the CT of a cluster by adding all its children ER values (shown by the function of f_{CT} in Figure 9). Then the ER value of o can be obtained by a "multiple" function on all CT values of its clusters (shown by the function of f_{ER}) due to the "and" semantics among clusters.

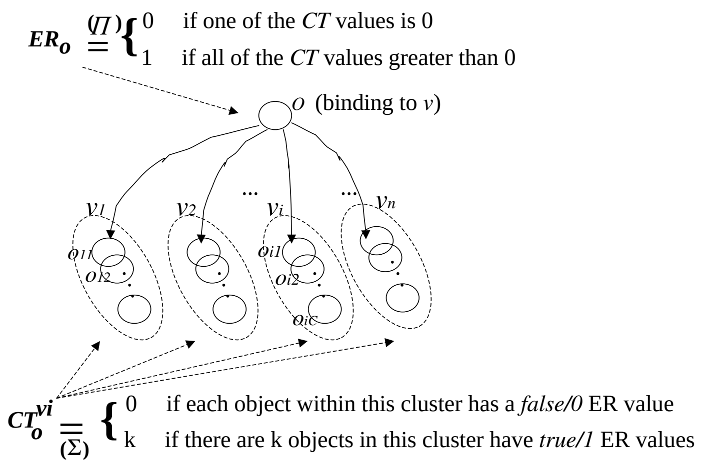


Figure 9: CT and ER Values Computation

For example, assume we are computing the ER for node *12* in Figure 5 by propagating its children ERs. It has clusters of *maker, quantity, price* and *item* with their object sets {*25*} {*26*} {*27*} and {*28, 29*} and their Dup values of *1, 1, 1* and *2* respectively. The corresponding CT values are *1, 1, 1* and *1* and therefore the ER value for node *12* is *1*. While the CT values for node *11* are *1, 1, 1* and *0* and hence its ER value is *false/0*.

So far we have introduced three types of values, Dup, CT and ER, for the portion tree nodes. Dup values can be obtained from the PPE phase while the computations of CT and ER values interleave with each other during the VPE phase. CT and ER values reflect the relative strictness of value predicates or the diversity of value domains. Dup value asserts an upper bound for the corresponding CT value while ER values are node specific.

3.4 View Object Selection Using APIX

After our two-phase-evaluation query processing, we obtain a portion tree of nodes with computed ER values. This portion tree indicated the minimized accessed scope of the data tree. For each pattern satisfiable (double circled) node on the portion tree, its children nodes are partitioned into clusters according to the path labels leading to them. Dup and CT values are computed for each such cluster and they are aggregated via a conjunction logic to obtain the ER value for the converged node. Thus, the nodes with *true* ER values in the portion tree indicate the effective trials to select desired view objects.

Given an XQL query, its evaluation criteria part is extracted and represented in a query graph. The output model is actually a path, termed a *view path*, via which the data objects bound to the ending variable are to be returned. We call this ending variable a *view variable* and its binding objects *view objects*. This view path can be extracted from the given XQL query by omitting all the branching paths within the filter. We highlight the view path in the query graph as shown in Figure 7. By navigating along the portion tree via the view path, we pick up the ones with true ER values to obtain the desired view objects.

Theorem 1 (View Object Computation) *An object o of the view variable vv is a view object iff*

1. *It is accessible via the view path vp in the data tree.*
2. *Every o_i along vp to o is in the portion tree PT and its ER value is true.*

4 Aggregate Path IndeX

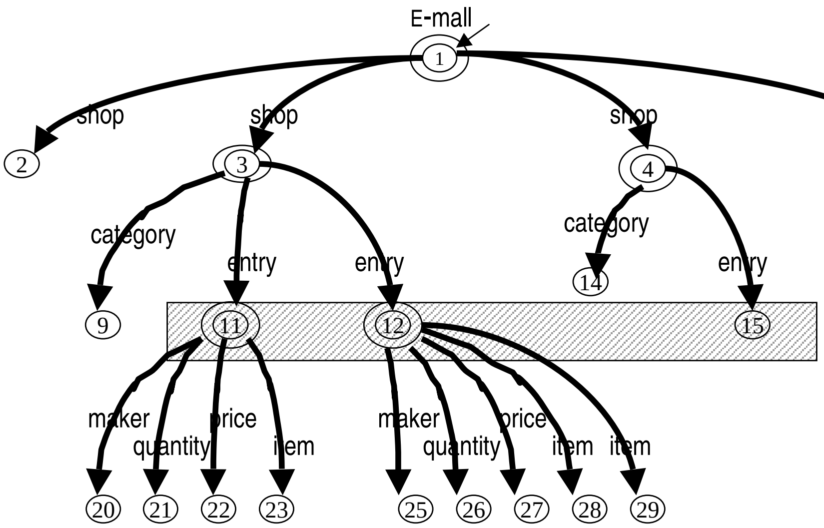
To help with the implementation of the proposed query evaluation strategy and to capture maintenance critical information for improving maintenance performance, we propose a new index scheme, Aggregate Path IndeX (APIX).

Given a query Q, its query graph G captures the query constraints, which are asserted on an XML data tree T to obtain the portion tree PT that are composed of the query relevant nodes. G abstracts the structure of the portion tree PT, we thus can establish the mapping between each internal node N_i of G

and a collection of pattern satisfiable objects $\{d\}$ in PT that are bound to it. We represent such mappings established during the PPE procedure (shown in Figure 8) as $\mathcal{E}(G, PT)$.

Definition 4 (Aggregate Path IndeX — APIX) *Let $I_{\mathcal{E}}$ $(G, PT) = I\{e(v, \{d\}) \mid v$ in G, $\{d\}$ in PT and e is the mapping between v and $\{d\}\}$ denote the structure index of G on the portion tree PT. We call $I_{\mathcal{E}}(G, PT)$ an aggregate path index and each $I\{e(v, \{d\})\}$ an index table with columns for all paths in the $\vec{P}$ of v.*

APIX can be established and initialized along with the PPE procedure (shown in Figure 8) as described in Subsection 3.3.1. If a node o is bound to v and evaluated by the PPE procedure to be a pattern satisfiable node, we then put it into the index table $I(v)$ and its children objects into their corresponding children variable columns. We now illustrate in Figure 10 an index table $I(e)$ (for variable "entry") of the APIX for the example query graph (see Figure 7). We see that the $I(e)$ has four columes for each of its required children variables, m, q, p and i. Pattern satisfiable nodes are stored in it. For each column v_i of an object tuple keyed by its oid o, it stores the set of the v_i type children nodes of o, which would not be null. However, we can also cache the auxiliary information such as Dup (always greater than 0), CT and ER values in additional columns. Figure 10 illustrates the APIX initialization when PPE navigates to "entry" nodes. According to the PPE algorithm, node *15* is not a pattern satisfiable node, hence it is not in $I(e)$. In contract, pattern satisfiable nodes *11* and *12* are in the "entry" table but their CT and ER values remain empty until the VPE phase.



"entry" index table I(e):

entry	maker			quantity			price			item			ER
	Oids	Dup	CT	Oids	Dup	CT	Oids	Dup	CT	Oids	Dup	CT	
11	{20}	1		{21}	1		{22}	1		{23}	1		
12	{25}	1		{26}	1		{27}	1		{28,29}	2		

Figure 10: PPE and APIX Initialization

Once APIX is established and initialized during PPE, we can start the propagation phase of VPE to compute the CT and ER values for objects stored in the APIX. We show the propagation algorithm in Figure 11. The computed results for objects in I(e) are shown in Figure 12.

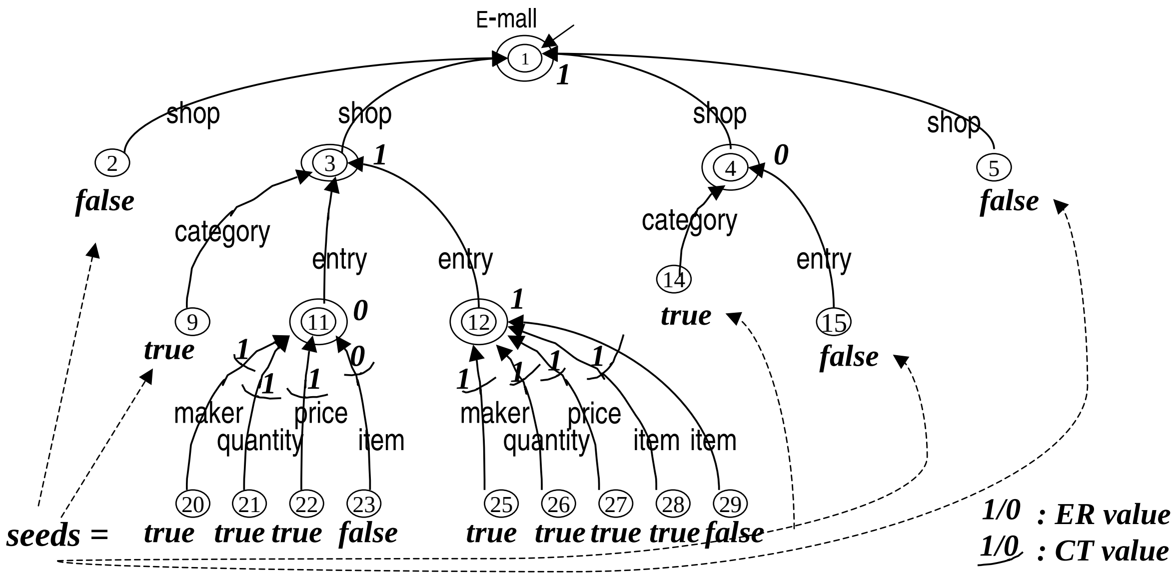
```
Procedure VPE(G, I_ε, v)
Input: G is the query graph;
        I_ε is APIX established and initialized after the PPE;
        v is a query variable in G from where the VPE to be continued;
Output: the computed ER values for objects in I;
// {vars} is a reversed list of children query variables of v in G;
// {d} is a collection of pattern satisfiable nodes in I(v);
if v is a leaf node in the query graph
    foreach object o in the object set of {d}
        if there is a pred on v
            seed = pred_eval(o);
        else
            seed = true;
else
    foreach object o in {d}
        foreach child variable v_i in I(v_i)
            // initialize the ER value
            o.ER[v_i] = true;
            foreach child object o_i in the column of v_i
                if o_i.ER == true
                    o.CT[v_i] ++;
            if o.CT[v_i] == 0
                o.ER = false;
// VPE propagates up to another index table
pick from {var} the next variable v_j
VPE (G, I_ε, v_j);
```

Figure 11: ER Values Computation in VPE

An APIX structure is established based on PPE. It in return provides an implementation mechanism for VPE since it can accommodate the intermediate values produced by each propagation step. Thus our strategy of separating the whole evaluation process into two phases fits well with the APIX structure. APIX can be in particular exploited when there are multiple similar questions asked to the APIX view that only differ in terms of their value predicates. For example, "[price<20]" is changed to "[price>70]".

After the two evaluation phases, we can then pick up the view objects from the objects stored in the APIX based on the theorem described in Section 3.4. We build the APIX to accommodate the query relevant objects of an XML data tree. APIX is a novel index scheme in the sense that it is not an ordinary index structure indexing on values or labels or nesting paths. Rather its index structure is implemented as a set of table structure and thus it possesses the capability of performing relational operations such as select, project and join. The view object computation is based on these operations on the index tables. Dup, CT and ER values are computed by conducting simple aggregation and other logic operations on the data stored in the index tables. Selection is based on CT or ER value conditions. For an index table $I(v)$, projection can project on the variable attribute that are along the view path and then join it with another index table. We show the algorithm for view object computation in Figure 13.

For example, given the example XQL query in Figure 3, its *view variable* is *entry* and its view path is



"entry" index table I(e):

maker			quantity			price			item			ER
Oids	Dup	CT	Oids	Dup	CT	Oids	Dup	CT	Oids	Dup	CT	
{20}	1	1	{21}	1	1	{22}	1	1	{23}	1	0	0
{25}	1	1	{26}	1	1	{27}	1	1	{28,29}	2	1	1

Figure 12: Computated Values in APIX

E-mall/shop/entry. Within the portion tree shown in Figure 12, a successful path leading to a *view object 12* is the path passing through nodes *1, 3* and *12*, all of whose ER values are *true*.

5 Web View Maintenance Using APIX

A web view maintenance technique using APIX is developed to improve the maintenance performance by three means. 1) Test the maintenance irrelevancy of an update by checking in APIX the existence of the affected objects. 2) Further check out the maintenance irrelevant case by exploiting the auxiliary values provided by APIX. 3) Reduce the amount of accesses to base data by utilizing the structural pattern of the

Procedure Sele_ViewObj
select the view variable
from a list of decomposed single-step paths v_i.p v_{i+1} of the view path vp
foreach $< v_i, v_{i+1} >$ pair
    project on the v_{i+1} column of I(v_i);
    get pair set $\{< o_i, o_{i+1} >\}$, each o_i in $\{< o_{i-1}, o_i >\}$, and o_{i+1} has true ER value

Figure 13: View Object Selection

objects maintained by APIX.

First, we classify all the possible updates (such as that written in XML in Figure 4) on XML sources, similar to previous work [AMR+98], into three basic types: insertion, deletion of a segment of an XML tree and update on the value of a leaf node of the XML tree. The former two operations can be represented in a more concise format (compared to the update fragment message in XML format in Figure 4) as <*Ins,* o_1*, p,* o_2> and <*Del,* o_1*, p,* o_2> respectively. o_1 represents the object that results from the selection clause of the update message (see Figure 4), and we assume it is bound to v_1. o_2 is the inserted or attached subtree root and p is the path from o_1 to o_2 and we use v_2 instead as the bound variable for o_2. The update operation is represented as <*Chg, o, OldVal, NewVal*> denoting the change of the value of the leaf node o from its *OldVal* to the *NewVal*.

5.1 Self-Maintainability Test

Updates are said to be irrelevant to the materialized web view if they would not cause any effect on it. With the help of the locally cached APIX nodes, we can conduct a set of self-maintenance checks before having to access remote base data. We illustrate a list of the cases in which irrelevant updates are discovered by our algorithm.

- **Query Irrelevant Update.** For an *Ins* or *Del* or *Chg* operation, if any of their variables is not a query variable in the query graph, then the update is irrelevant. For example, the operations <*Ins,* &2, *"rank",* &36> or <*Chg,* &6, *"J.C. Penny", "Macy"*> are not relevant to the query, thus there is no effect on the materialized web view.

- **Value Irrelevant Change.** For a <*Chg, o, OldVal, NewVal*> update, if the seed value of *OldVal* is the same as that of *NewVal*, then this update would not cause any change propagation of ER values and thus is irrelevant. For example, given the example XML tree shown in Figure 2, <*Chg,* &22, *$18, $16*> is an irrelevant update since the evaluation of predicate *price*<*$20* on either of *OldVal* or *NewVal* is true.

- **Delete Irrelevant Update.** In a <*Del,* o_1*, p,* o_2 > case, if either o_1 or o_2 is not cached in Iv_1 or Iv_2, then the deletion operation is irrelevant. An example is <*Del,* &18, "item", &35>. &18 wasn't cached in its Ie before the update operation, thus its deletion would have no effect on the materialized view.

- **Parent Irrelevant Change.** For a <*Chg, o, OldVal, NewVal*> transaction, if the seed of o is changed from *true* to *false* but there is no parent or referee object of o cached in the APIX (assume we store in APIX bi-direction links), then this update is irrelevant.

The above checks are enumerated in an increasing order of complexity. The first test only requires to check whether the variable types that the affected object pair is bound to variables that are within the query graph. The second one needs to evaluate the old and new seed values of the updated object. These first two kinds of self-maintenance checks also appear in other work [ZG98b]. The later two kinds of update irrelevant cases are discovered by the self-maintenance checks based on whether the affected nodes are cached in the APIX structure. Similarly, the relevant-oid auxiliary structure in the RelOid approach [AMR$^+$98] is also capable of capturing complicated irrelevant update cases of the last two types. However, more maintenance irrelevant cases can be checked out by our APIX approach than by theirs.

Claim 1 *Given a variable v in a query graph G, let $\{d_r\}$ and $\{d_a\}$ be the object sets of v that are kept in the relevant-oid structure and in APIX structure respectively. $\{d_r\}$ is a superset of $\{d_a\}$.*

Both approaches consider an update irrelevant if the nodes being affected were not cached by their auxiliary index structures. Therefore, the less the cached objects, the more update cases are checked to be irrelevant. We will prove this in Section 6. To illustrate such a case, we see that in Figure 10, object &18 would not be captured in *Ie* but it would appear in the relevant-oid structure. Thus a *Del* operation involving object &18 would be treated as an irrelevant update but would cause accesses back to base objects by other strategies. Using our APIX approach, the reduction factor of the cached objects to those in RelOid is 2^k with k as the average fanout of variable nodes in the query graph.

5.2 Further Irrelevant Case Checks

An update will not affect the materialized view if it doesn't change the ER values of the objects along the view path. We thus can rule out more update irrelevant cases by exploiting APIX for further information.

Case 1 *For an insertion update case, if neither o_2 was in APIX before it is to be referred to, nor it is a newly inserted node to be evaluated as pattern satisfiable, then the update would not cause any insertion of the APIX tuple. Thus it is irrelevant.*

For example, assume we have an insertion operation <*Ins,* &5, *"entry",* &15>. o_2 &15 is an existing object to be linked from o_1 &5. Since &15 does not exist in *Ie* and thus this insertion of reference would not affect the materialized web view. We see that this kind of link insertion only involves one single access of the o_2 tuple in APIX. However, a subtree insertion would usually cost an evaluation of the corresponding partial query.

Now assume another *Ins* case, in which o_2 was or is evaluated to be a pattern satisfiable node. However, if the ER value of o_2 is *false* after the VPE propagation on the subtree of o_2, the insertion is irrelevant. If the ER value of o_2 is *true* and so was that of o_1, then it would not effect any old ER values. Thus it would not affect the web view except it possibly introduces new view objects within the new inserted subtree.

For example, assume an insertion as indicated in message 14 occurs. It inserts an "entry" object with an associated subtree to node &5 and then is evaluated by *[@maker="ASE" and @quantity>0 and price<$20 and item="book"].* It turns out that this subtree possesses a small portion tree shown as depicted in Figure 15. However, after applying VPE on this small portion tree, the ER value of &5 is *false.* Therefore, this is an irrelevant insertion operation.

```
<?xml version=“1.0”?>
  <xmlupdate version=“1.0”>
    <insert select=/E-mall/shop[name=“Michael”]>
      <element location=“lastchild”>
        <entry name=“pokemon” maker=“GSW” quantity=25>
          <price>$15.00</price>
          <item>book</item>
          <item>puzzle</item>
        </entry>
      </element>
    </insert>
  </xmlupdate>
```

Figure 14: An Update Fragment Message

After the self-maintenability tests (see Section 5.1), there remains only the cases that both o_1 and o_2 were cached in APIX in a deletion scenario or node o bears different seed values before and after a value change operation.

Case 2 *For a deletion operation <Del, o_1, p, o_2 >, if the CT value of o_1's cluster v_2 is greater than 0, then the ER value of o_1 won't be affected and it is thus an irrelevant deletion except that the view objects within the subtree of o_2 (if any) would be removed.*

For example, if we delete object &4 from the root object &1, &1.CT[“shop”] is reduced from 4 to 3. However, it won't affect the successful view object selection path from object &1 via object &3 to &12.

Case 3 *For an update operation, let v_2, v_1 be the variables for o, and its parent node o' respectively. If the ER value of o is changed from false to true, then the change operation equals to <Ins, o', “v_2”, o>. If the ER value is changed from ture to false, then this change is actually a deletion <Del, o', “v_2”, o>.*

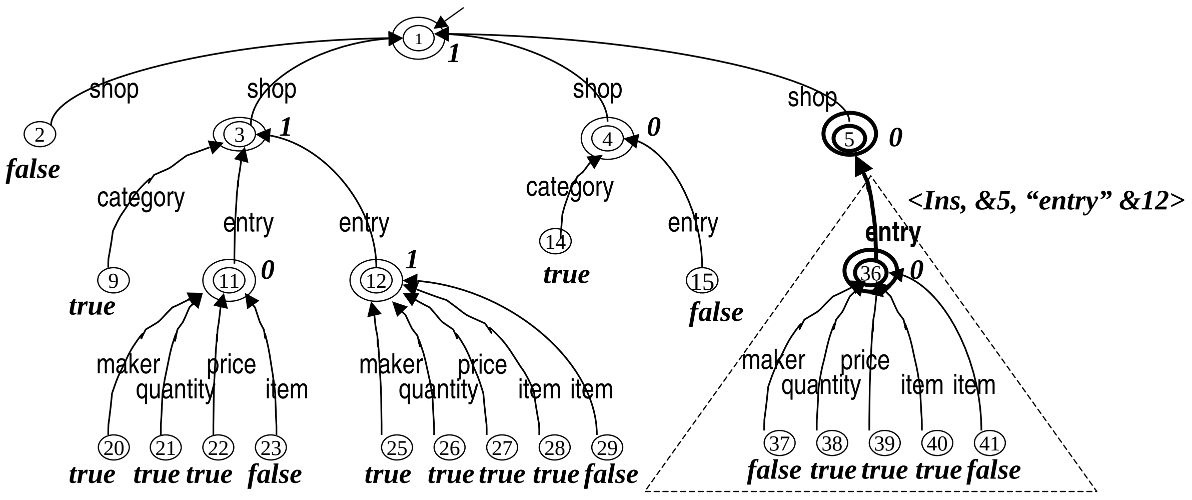
Thus, if $o.ER$ is changed from *false* to *true* and o' was already in APIX with a *true* ER value, then this change is an irrelevant update. Otherwise, if $o.ER$ is changed from *true* to *false* and $o'.CT[v_2] > 1$, then this is an irrelevant change operation as well.

For example, an update <Chg, &29, “puzzle”, “book”> causes the change in the seed value of &29 from *false* to *true*. Since the parent node &12 had a *true* ER value before the change, this is an irrelevant operation. To see another example of the opposite case, assume node &12 had two “book” items &28 and &29, the value change of either node is an irrelevant update.

5.3 Web View Maintenance Using APIX

The third mechanism for improving the performance of web view maintenance is to maintain the materialized web view with minimized accesses to base data by exploiting the maintenance critical objects that are captured in APIX.

Updates on objects may cause some query relevant nodes to be introduced or to be removed. In other words, we need to maintain the nodes kept in APIX along with their respective Dup, CT and ER values. Since the web view objects are selected via a traversal along the view path pattern to the view objects by choosing



entry	maker			quantity			price			item			ER
	Oids	Dup	CT	Oids	Dup	CT	Oids	Dup	CT	Oids	Dup	CT	
11	{20}	1	1	{21}	1	1	{22}	1	1	{23}	1	0	0
12	{25}	1	1	{26}	1	1	{27}	1	1	{28,29}	2	1	1
36	{37}	1	0	{38}	1	1	{39}	1	1	{40,41}	2	1	0

new inserted APIX tuple due to the insertion of subtree rooted from &36

Figure 15: An Subtree Insertion

the true-valued ER objects, the same rule can be applied to the computation of web view "patches". That is, the newly appearing true-valued ER objects along the view path would trigger a maintenance request. The algorithm of generating the patch insertion statements is shown in Figure 16.

Procedure ADD_MaintStat
// Among the objects with their ER values changed to true, let v_i be the topmost variable;
if v_i is not one of the variables along the view path vp
    return;
project on the v_j column of I(v_i); // $<v_i$.p $v_j>$ along vp
get $\{<o_i, o_{i+1}>\}$, o_i has new true-valued ER;
View_Object_Collection $+=$
    select view variable
    from a list of v_k.p v_{k+1} starting from v_j along vp
    foreach variable pair $<v_k, v_{k+1}>$
        apply their true-valued ER object pairs

Figure 16: Maintenance Statement for Adding Web View Patches

Using the maintenance requests generated by the algorithms in Figure 16, we could compute the patches of view objects quickly by utilizing the pattern satisfiable objects that are kept in APIX instead of accessing base objects for re-evaluation. For example, if by changing the value of node &23 from "video" to "book", then its new evaluated ER value become *true* and it is propagated up to the node &11. Since the ER value of the parent of &11 was already *true*, &11 thus becomes the top most new true-valued ER node. According to the Procedure ADD_MaintStat in Figure 16, &11 is easily computed to become the view object to be inserted into the materialized web view object list. In this case, the computation involves no access to remote base node, but only to nodes &23 and &11 in APIX. It thus saves a large amount of access costs as compared to the alternative method of relevant-oid, which needs to re-evaluate the query starting from the root object.

On the other hand, the removed true-valued ER objects, either due to the deletion of APIX tuples or the changed ER values that are propagated up from descendent nodes, would trigger a deletion maintenance request. The procedure of computing the view objects to be removed from the materialized web view is shown in Figure 17 .

Procedure DEL_MaintStat
// Among the objects with their ER values changed to false, let v_i be the topmost variable;
if v_i is not one of the variables along the view path vp
    return;
project on the v_j column of I(v_i); // $< v_i$.p $v_j >$ along vp
get $\{< o_i, o_{i+1} >\}$, o_i has new false-valued ER;
View_Object_Collection $- =$
    select view variable
    from a list of v_k.p v_{k+1} starting from v_j along vp
    foreach variable pair $< v_k, v_{k+1} >$
        apply their true-valued ER object pairs

Figure 17: Maintenance Statement for Deleting Web View Patches

5.4 Maintenance Approach: "Putting It All Together"

Web view maintenance involves the APIX maintenance, which includes adding and deleting object tuples and keeping their auxiliary values up-to-date with the updates on the XML data tree. In the insertion scenario, if o_1 wasn't materialized in APIX before, the newly inserted o_2 may cause o_1 to be "pattern satisfiable". Then the partial query evaluation starting from o_1 needs to be carried out again to keep its APIX up-to-date with the new situation. If the ER value of o_1 after the evaluation is changed to be *true*, then it will result in a upwards propagation to re-evaluate the Dup, CT and ER values. We now give an update case to illustrate the effect on the index tables (see Figure 18).

change of "entry" index table after <Ins, &11, "item" &36> and value for &36 is "book"

entry	maker			quantity			price			item			ER
	Oids	Dup	CT	Oids	Dup	CT	Oids	Dup	CT	Oids	Dup	CT	
11	{20}	1	1	{21}	1	1	{22}	1	1	{23,36}	2	1	1
12	{25}	1	1	{26}	1	1	{27}	1	1	{28,29}	2	1	1

Figure 18: Effects on APIX by An Chg Update

Procedure *Ins(o_1, l, o_2)* (shown in Figure 19) puts all checking and maintenance actions together for an insertion <*Ins*, o_1, *p*, o_2 >. From the algorithm, we can see that, in the worst case, the maintenance access costs is proximately the subtree size of o_1, which is much reduced compared to the alternative solutions.

```
Procedure Ins (o1, p, o2)
// v1, v2 are the variables for o1, o2
If o2 is not in I(v2)
    ppe_subtree (o2);
while o1 is not in I(v1)
    ppe_subtree (o1);
    //propagate ppe if o1 becomes a pattern satisfiable object
    assign o1 to its parent, v1 the variable for o1;
o1.Dup[v2] ++;
while o2.ER == 1
    o1.CT[v2] ++;
    if o1.CT[v2] > 0 and o1.ER == 0
        re-compute o1.ER;
        //propagate changed ER value upwards
        assign o2 and o1 to their parents respectively;
    else
        exit;

Sub_Procedure ppe_subtree (o)
// v is the variables for o
    PPE(G, T, v, {o}) //partial PPE from o
    If o is not a pattern satisfiable object
        Judged to be an Irrelevant Update;
        exit;
    Compute the ER and CT values of o;
    if o.ER = 0
        Judged to be an Irrelevant Update;
```

Figure 19: Insertion Maintenance on APIX

```
Procedure Del (o1, p, o2)
//v1, v2 are variables for o1, o2
If o2 is not in I(v2) or o1 is not in I(v1)
    Judged to be an Irrelevant Update;
    exit;
loop1:
    o1.Dup[v2] --;
    if o1.Dup[v2] == 0 ;
        remove o1 from APIX;
        assign o1 and o2 to their parents respectively;
        //propagate changed Dup value upwards
        go to loop1;
    else
        if o2.ER == 1
        loop2:
            o1.CT[v2] --;
            if o1.ER == 1 and o1.CT[v2] == 0 ;
                o1.ER = 0;
                assign o1 and o2 to their parents respectively
                //propagate changed ER value upwards
                go to loop2;
```

Figure 20: Deletion Maintenance on APIX

Upon a deletion case <*Del*, o_1, p, o_2 >, the irrelevancy of the update is first tested by the self-maintainability checks. If o_1 has the only child element o_2 via path p, this deletion would cause the removal of an APIX tuple and possible the propagation of its effect. The deletion procedure *Del(*o_1, p, o_2*)* is depicted in Figure 20. Given the maintenance algorithms for the insertion and the deletion case, the maintenance of a <*Chg, o, OldVal, NewVal*> update is tackled already, since it can be simulated as either of the former two cases.

6 Evaluation of Costs for Web View Maintenance

Like others' work [AMR+98, ZG98a, ZG98b], we assume that the main cost for the maintenance of a web view can be estimated in terms of the numbers of base objects being fetched. This is based on the fact that each web object could be far away from the view site and its retrieval takes time. Naive algorithm re-evaluate the base data and compute the web view from scratch, its maintenance cost is always as much as the size of the base data. Auxiliary index mechanism at the view site is thus employed to help the maintenance procedures reduce the accesses to base data. It is exploited by both the RelOid approach proposed by Abiteboul et. al. [AMR+98] and our APIX approach. There is a trade-off between the index overhead and the performance gained. The more objects are indexed, the more space the index will require, and the more performance improvements may be possible by saving remote accesses. The maintenance of APIX itself may require more complex strategy and higher computability, but these can be achieved easily in the local view site. However, space is always limited and remote accesses is the dominant cost. Here we show that our APIX wins over the RelOid approach in that it not only reduces the accesses to base data, but also saves index space.

6.1 Cost Analysis for Index Space

In RelOid approach, relevant-oid index stores for the objects for all query relevant variables while APIX uses much reduced space since its asserts strict structural requirements on the data objects to be indexed. Assume a source XML document with the DTD as shown in Figure 21, we give an estimation on the space costs of the APIX structure compared to the alternative relevant-oid index.

<!Element A (B_1?, B_2?, $\cdots$, B_n?)*>
<!Element B_1 (C_1?, C_2?, $\cdots$, C_n?)*>
...
<!Element V_m (#PCDATA)>

Figure 21: A DTD Fragment

If we use a tree to represent this DTD, its depth is m and each of its element nodes has a fan-out of n optional subelements. Assume each non-leaf element in an XML tree, there is an even possibility of any variable combination. For instance, an object of A type may have no child object, or children objects of a single type, or of any two types etc., till of all n types. Thus the space of all possible combinations of its n children variables is 2^n. The query graph of a given query has the same height and each of its query variables has an average fan-out of k. For example, A is a query variable in the query graph and it has an aggregate path pattern of $B_1, ..., B_k$, then the possiblity of collecting an A type tree node o into the relevant-oid index would be $1 - \frac{2^{n-k}}{2^n} = 1 - 2^{-k}$. However, the possibility of put o into APIX as a pattern satisfiable node by our PPE evaluation strategy would only be $\frac{2^{n-k}}{2^n} = 2^{-k}$ due to the requirement of the aggregated path set (every object needs to have all these k variable children objects). Thus the reduction factor of the objects being indexed by our APIX approach compared to RelevantOid method is $2^{-k}/1 - 2^{-k} = 2^{k-1}$. It is a fairly big factor affecting the object selection of each level, and accumulating exponentially with the depth of the tree. Even though APIX index tables cost extra space for the k columns of children variables and a few more for the auxiliary values, it is a fixed factor and can be ignored compared to the reduction factor 2^{km}.

Due to the reduced index space by our APIX approach compared to that of the RelOid approach, we can check out more maintenance irrelevant cases as described in Section 5. Intuitively, in the cases if elements in an XML data tree are diverse in the types of subelements or attributes and most of them are optional, and if the query is deep and wide at each level with a conjunctive requirement of multiple paths on each variable, then the APIX index scheme and its associated query evaluation strategy would be favorable and may save a good deal of index space.

6.2 Access Cost during Maintenance Phase

Given the example web view specification (see Figure 3) over the example XML database (see Figure 2), we compare the maintenance access costs for insertion situation by the naive algorithm, the RelOid algorithm and our APIX algorithm.

Algorithm Naive: The maintenance by the naive approach (without any auxiliary structure) involves the total re-computation of the view from the base databases. Hence the cost is (we use C_v to denote the number of object instantiates bound to variable v):

$$Cost^{maint}_{naive} = C_E + C_s + C_c + C_e + C_m + C_q + C_p + C_i$$

Algorithm RelOid: The RelOid maintenance approach also needs to go back to the root object and re-evaluate the whole base data except that it applies the affected objects directly to the corresponding query variables while saving the accesses to other objects of the same types. For example, if an update <*Ins*, &15, *"item"*, &39> occurs, RelOid algorithm accesses all the objects that are bound to E, s and c till it reaches object &15 and uses it to instantiate variable e while ignoring all the other $(C_e - 1)$ objects. The objects bound to variables m, q, p and i (descendant variables of variable e) are also restricted to only those descendant nodes of &15.

Let the object occurrences of variables m, q, p and i stemming from the object &15 be C'_m, C'_q, C'_p, and C'_i respectively, then the number of total objects accessed using RelOid algorithm is:

$$Cost^{maint}_{RelOid} = C_E + C_s + C_c + C_e + 1 + C'_m + C'_q + C'_p + C'_i$$

Algorithm APIX: As stated before, APIX algorithm can avoid a large number of accesses to base databases by detecting irrelevant updates. Even in the worst case when referring back to base databases is inevitable, whose possibility is the reduction factor of that of RelOid approach, we can restrict the accesses within a smaller scope starting from the affected node (i.e., &15). Thus in our example, the accesses to the objects of higher-level variables E, s, c and c are saved. The number of total base objects evaluated by our algorithm under the same situation as RelOid is:

$$Cost^{maint}_{APIX} = 1 + \frac{1}{2^4 - 1}(C'_m + C'_q + C'_p + C'_i)$$

6.3 Cost Comparison of Experimental Results

We show in Figure 22 the experiment tests on the maintenance costs under three different update scenarios to the example XML database (see Figure 2). The database contains one E-Mall, 1000 shops, 100 entries and 2 categories per shop, 10 items, 1 maker and 1 price per entry. We randomly generate update test cases to insert or delete objects of different levels, or update the values of leaf nodes. We observe from the experiment result that, in an *Ins* update situation, the cost of APIX maintenance is slightly less than that of the RelOid method, since both costs are much related to the subtree size of the affected object. The APIX approach improves more on deletion cases due to that it mainly involves the computations and propagation of the changed ER values conducted in the local APIX structure. The number of the APIX tuples that need to be maintained is in a linear relation with the distance of the affected object from its leaf nodes. An *Udp* update is most expensive especially for the naive and RelOid approaches since it affects a leaf node at the bottom of the data tree and thus requires the accesses almost to the whole base database. Even in our APIX approach, the maintenance involves a longest reverse evaluation from bottom upwards, hence is likely to be long. However, Figure 22 shows that our approach wins significantly over the other two methods in the *Udp* situation.

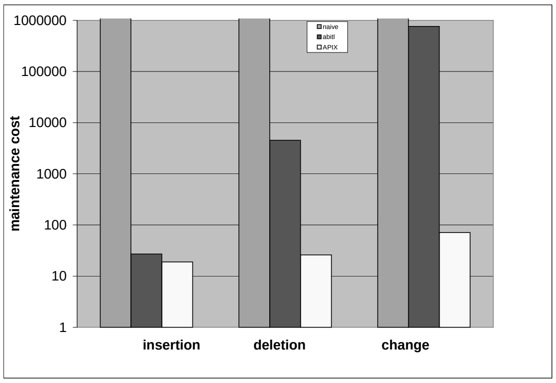


Figure 22: Maintenance costs for updates

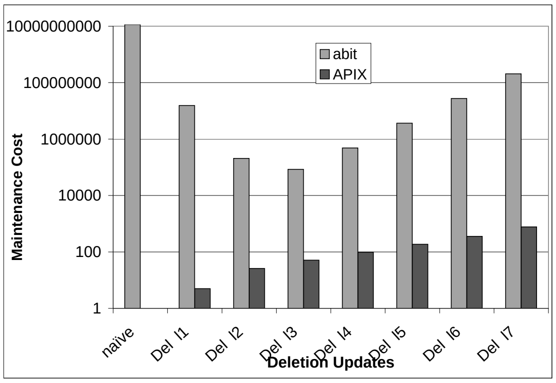


Figure 23: Varying the position of effected object

The second experiment is particularly designed for the *Del* situation. Suppose we have an XQL query containing a view path composed of eight single-step paths $l_1, l_2, l_3, ..., l_8$. Figure 23 shows that our algorithm achieves the more markable improvement in the terms of maintenance costs if the deletion operation occurs closer to the root object. This is because the higher the objects are (opposed to the lowest leaf objects), the shorter path they need to check upwards and propagate up their changed values in the APIX.

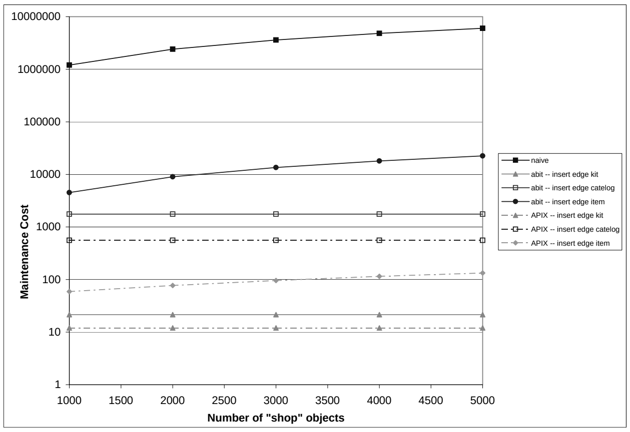


Figure 24: Varying the database size

For the experiment shown by Figure 24, we use the example XQL query shown in Figure 3. We increase the number of shops in the database from 1000 to 5000, and keep the same average ratio of entries per shop, items per entry, etc. Therefore, when the number of shops doubled, for example, the size of its subtree is doubled. We conduct three kind of insertion operations by inserting entries, categories and items respectively to the XML database and test the costs caused by our algorithm against the RelOid approach. We see that both sets of maintenance costs grow linearly with the size of relevant subtree. Our approach gains much compared to the alternative one when inserting a lower object such as item (we observe an opposite preference for deletion cases from the second experiment). The reason for this is that the access time to the base data by our insertion maintenance approach is related to the size of the subtree that stems from the inserted object.

All three experiments are designed to be similar to Abiteboul's work [AMR+98] to set up a uniform testbed based on which the experimental results can be compared against. These experimental studies help us to identify cases for when our algorithm is superior: (1) The richer the query graph structure is in terms of its depth and fan-out, and the more strict the value predicates are, the larger the reduction factor of the APIX approach to the RelOid approach would be. Hence the more effective of the APIX approach to screen out update irrelevant cases. (2) *Del* operations happening to the upper objects while the *Ins* operations occuring to lower ones. (3) Data objects corresponding to the variables of the query graph are much diverse and all possible variables are more evenly distributed to the children objects. (4) The costs for network communication are high or the times of connections to be established are long.

7 Conclusion

In general, previous techniques for incremental web view maintenance simply recompute it from scratch or just integrate the updated object directly with the variable it is binding to in the web views. We propose a novel index structure – APIX, which is a set of tabular structures, each of which is constructed for one variable node of the query graph of a web view definition. APIX is used to accommodate the update relevant objects as well as their value evaluation results. Also, the updated objects can be explored to derive the web view "patches" to be integrated into the materialized web view.

We conduct the cost analysis of the APIX index space costs and its maintenance compared to the alternative state-of-art solutions. It shows that our approach wins over its competitors most of the time and in some cases. The gains of our strategy are significant, as it is more effective in self-maintenance and causes fewer accesses to base data.

The implementation of our web view mechanism is based on the GMD XQL query engine [HM99]. We find that the storage space of APIX can be economized by compressing multi-step non-branching paths into one single-step path. The maintenance strategy could be adjusted correspondingly. Finally, we would like to consider exploiting XML schema, its linking mechanism and query language to optimize the web view maintenance.

References

[AMR$^+$98] S. Abiteboul, J. McHugh, M. Rys, V. Vassalos, and J. Wiener. Incremental Maintenance for Materialized Views over Semistructured Data. In *the Proceedings of the Twenty-Fourth International Conference on Very Large Databases, New York*, pages 38–49, August 1998.

[AQM$^+$97] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel Query Language for Semistructured Data. In *International Journal on Digital Libraries, 1(1)*, pages 68–88, April 1997.

[DFF$^+$99] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. XML-QL: A Query Language for XML. In *Proceedings of the Eighth International World Wide Web Conference (WWW-8)*, 1999.

[Eis99] Jonathan Eisenzopf. Perl-XML Module List. http://www.perlxml.com/modules/perl-xml-modules.html, 1999.

[GGMS97] D. Gluche, T. Grust, C. Mainberger, and M. H. Scholl. Incremental Updates for Materialized OQL Views. In *Proceedings of the 5th International Conference on Deductive and ObjectOriented Databases (DOOD), Montreux, Switzerland*, pages 52–66, December 1997.

[GM95] A. Gupta and I. S. Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. In *Bulletin of the Technical Committee on Data Engineering, 18(2)*, pages 3–18, June 1995.

[GMS93] A. Gupta, I. S. Mumick, and V. Subrahmanian. Maintaining Views Incrementally. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Washington, D.C.*, pages 157–166, May 1993.

[Gro98a] W3C Working Group. Extensible Markup Language (XML^{TM}). http://www.w3.org/XML/, 1998.

[Gro98b] W3C XSL Working Group. The Query Language Position Paper of the XSL Working Group. http://www.w3.org/TandS/QL/QL98/pp/xsl-wg-position.html, 1998.

[HM99] Gerald Huck and Ingo Macherius. GMD-IPSI XQL Engine. http://xml.darmstadt.gmd.de/xql/, 1999.

[KLMR97] A. Kawaguchi, D. F. Lieuwen, I. S. Mumick, and K. A. Ross. Implementing Incremental View Maintenance in Nested Data Models. In *In Proceedings of the Workshop on Database Programming Languages*, August 1997.

[LMSS95] Alon Y. Levy, Alberto O. Mendelzon, Yehoshua Sagiv, and Divesh Srivastava. Answering queries using views. In *Proceedings of the 14th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, San Jose, CA*, 1995.

[MAG$^+$97] 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, September 1997.

[PGMW95] 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.

[RKRC96] E. A. Rundensteiner, H. A. Kuno, Y. Ra, and V. CrestanaTaube. The MultiView Project: ObjectOriented View Technology and Applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, page 555, June 1996.

[RLS98] Jonathan Robie, Joe Lapp, and David Schach. XML Query Language (XQL). http://www.w3.org/Style/XSL/Group/1998/09/XQL-proposal.html, 1998.

[SC98] Jrme Simon and Sophie Cluet. Using YAT to Build a Web Server. In *International Workshop on the Web and Databases (WebDB)*, pages 118–135, 1998.

[SDJL96] Divesh Srivastava, Shaul Dar, H. V. Jagadish, and Alon Y. Levy. Answering sql queries using materialized views. In *Proceedings of VLDB*, 1996.

[SLT91] M. H. Scholl, C. Laasch, and M. Tresch. Updatable Views in ObjectOriented Databases. In *Proceedings of the 2nd International Conference on Deductive and ObjectOriented Databases (DOOD), Munich, Germany*, pages 189–207, December 1991.

[W3C98] W3C DTD Working Group. Data Type Definition (DTD). http://http://www.w3.org/XML/1998/06/xmlspec-report-19980910.htm, Sept. 1998.

[ZG98a] Y. Zhuge and H. GarciaMolina. Graph Structured Views and Their Incremental Maintenance. In *Proceedings of the 14th International Conference on Data Engineering, Orlando, Florida*, pages 116–125, February 1998.

[ZG98b] Y. Zhuge and H. GarciaMolina. Self-Maintainability of Graph Structured Views. In *Technical Report, Computer Science Department, Stanford University*, pages 116–125, October 1998.