WPI-CS-TR-99-33 Feb 1999

Aggregation Path Index for Incremental Web View Maintenance

by

Li Chen

Elke A. Rundensteiner

Computer Science

Technical Report

Series

WORCESTER POLYTECHNIC INSTITUTE

Computer Science Department

100 Institute Road, Worcester, Massachusetts 01609-2280

Aggregation Path Index for Incremental Web View Maintenance *

Li Chen, Elke A. Rundensteiner

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

Abstract

As web data becomes more essential in our work and play and it keeps growing in an explosive way, web view mechanisms are extensively employed to offer customized value-added services to customers and they are usually materialized to achieve fast query response time. However, the dynamicity problems of the underneath web information is not as easy to tackle as it is in the context of conventional database systems. Developing maintenance techniques for materialized web views over dynamic web data sources becomes more challenging because of the lack of a schema restricting the structure of all the web data sources and the shareability of web data sources enabling each update on a single data source to potentially affect many others in the web data graph. To compute web view "patches" for its incremental maintenance in response to an update, a large amount of accesses back to base data is usually inevitable, but it is obviously not desirable because of the likelihood of severe impact from the heavy network overhead and the intense contention for base data. In this paper, given a web view specification defined over a hierarchical web data graph, we analyze the query pattern, conduct the evaluation strategy along aggregation paths as to distill a subgraph of web data objects, for which we set up an index structure. By utilizing the precomputed value aggregation results stored in such an index, our algorithms show that both web view computation and its maintenance can be done more efficiently. Cost analysis and experiment studies on the gains of our incremental maintenance approach compared to the state-of-art solutions are also conducted.

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

1 Motivation

1.1 Introduction

As web data becomes more essential, a lot of work focusses on developing database and XML tools to aid the modeling of web data [PGMW95] and integration of diverse web data sources into one "unified"

*This work was supported in part by the NSF NYI grant #IRI 94-57609. We would also like to thank our industrial sponsors, in particular IBM for the IBM Partnership Award and our collaborators at IBM Toronto for their support.

resource. Techniques are being developed for querying web data sources as well as for building web views over them. Given that the volume of data available on the web is growing exponentially, web view mechanisms [LMSS95, SDJL96] are extensively employed to offer customized value-added services to customers. They can serve as filters over the huge network of inter-connected web sources and integrated bits and pieces of "raw" web data into a "personalized" view.

To achieve fast query response time, web views are often materialized. However, the dynamicity problems of information joining in as a new data source and leaving to be not available any more is not as easy to tackle as it is in conventional database systems. In the latter context materialized view mechanisms and their maintenance issues have for long been one well-studied topic [AMR$^+$98, GGMS97, GM95, GMS93, KLMR97, RKRC96, SLT91]. Developing maintenance techniques for materialized web views over dynamic web data sources becomes more challenging because of the lack of a schema restricting the structure of all the web data sources and the shareability of web data sources enabling each update on a single data source to potentially affect many others in the web data graph [CAW98]. To compute web view "patches" for its incremental maintenance in response to an update, a large number of accesses back to base data is usually inevitable. It is however obviously not desirable because of the likelihood of severe impact from the heavy network overhead and the intense contention for base data.

In this paper, we model the distributed web data sources as a hierarchical graph model, over which a web view can be specified. A strategy based on separating the web view evaluation into two phases is developed. We analyze the query pattern, conduct the first phase evaluation along aggregation paths so to distill a subgraph of web data objects and then set up index structures for them. By utilizing the value evaluation results that are computed along aggregation paths and storing them in such indexes in the second evaluation phase, we can conduct both the web view computation and its maintenance more efficiently. Especially in the process of maintaining a materialized web view, our approach can lead to big savings in terms of the costs of access time of the base data compared to alternate solutions in the literature by integrating the updated objects with their precomputed aggregation results.

1.2 Related Work

Incremental view maintenance techniques attempt to reduce the number of references back to remote distributed base data sites through a better utilization of local data resources. The naive recomputation method would heavily impact the query performance and worsen the load on base data sites. An incremental view maintenance approach considers referencing back to base data as the last resort and investigates strategies to minimize the examing scope of base data. If incremental maintenance can be done using the local cache site information only, we call this view self-maintainable with respect to these updates.

There is some work tackling the view maintenance problem in the context of semi-structured data available on the web. Suciu et al. [Suc96] assume semistructured data to be rooted graphs, composed of a subset (subtrees) by union, concatenation, juxtaposition and recursion operations. For both the relational and the nested relational models that are subsumed, the queries are join-free but the lengths of traversal paths are not restricted. Zhuge and Garcia-Molina [ZG98a] address general issues related to graph-structured views and their view maintenance. They simplify views by only considering select-project view specifications over tree-structured databases, and the resultant view is a flat collection of objects without any edges between objects.

Zhuge et al. [ZG98b] also study the characteristics of self-maintainability that can be utilized to avoid any access to base data for irrelevant updates. They also show how to perform those tests when different update information is available. But this strategy is not an all-purpose solution, especially it would turn back to conventional maintenance techniques for those relevant updates while no improvement can be achieved in these cases. The limitation of their work also lies in the strong assumption of a simple view specification.

Abiteboul et al. [AMR+98] generalize these previous studies to cover arbitrary graph-structured databases. Their approach can handle joins and the resultant view is a structured sub-graph of the base graph instead of just a flat collection of objects. Their incremental maintenance algorithm minimizes the searching scope by directly applying the updated object instance to the view specification, thus it avoids the accesses to all the other objects within the same target set of the corresponding variable. This approach needs an auxiliary structure for the relevant objects of the variables that appear in the web view specification.

1.3 Contributions

The contributions of our work are:

- Propose the Query Graph (QG) to represent explicitly the path query pattern over data graph required by a web view.
- Develop a view evaluation strategy to reuse the common aggregation path index structure among a set of web view specifications, which have the same path specification, but may differ from each other in value predication or view selection predicates.
- Establish the Aggregate Path Index (APIX) for objects that conform to the path specification and accommodate their value evaluation aggregation results.
- Describe algorithms for efficiently deriving a variety of view selections and maintaining the materialized web view upon updates by checking self-maintainability and cheaper computation of web view "patches" (in terms of minimizing accesses to base data) based on the APIX.
- Analyze the cost of our approach and demonstrate that it wins over alternative state-of-art solutions.

1.4 Outline

In Section 2, we give a detailed specification of the basic concepts of web views, the web data model, our assumptions and the problem description. In Section 3, we present our basic solution approach surrounding the QG and the APIX structures. In Section 4, our maintenance algorithms are described under different update scenarios. Cost analysis and experiment studies on the comparison of the costs between our algorithm and the state of art solutions is conducted in Section 5. We wrap up our discussion in Section 6.

2 Background on Web Views

2.1 Web Data Model

Numerous data models have been proposed in the literature for semi-structured data [Aro97, Mih96, AMM97, FFLS97, CRCK98]. Recently, XML is emerging as a standard of universal data exchange format on the Internet and it utilizes a hierarchical structure with rich, powerful links and naming mechanisms. We believe that these qualities of XML make it a perfect fit for modeling web data sources. Hence, we vision that web data sources can be structured as a hierarchical graph model by parsing each tagged XML element as an object and by capturing each hyper-link of XML as a direct edge attached with a label indicating the type of this parent-child relationship. Basically, the model suggested for XML objects is quite similiar to the Object Exchange Model (OEM)[PGMW95] but has some extensions (such as each object has knowledge of its parent object).

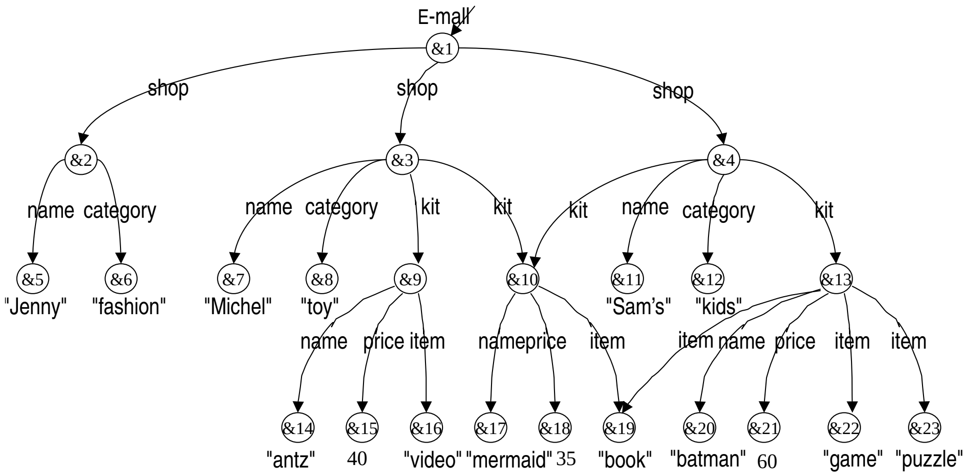


Figure 1: Motivating Web Database Example

Figure 1 shows the structure of our running example E-mall web database. An E-mall integrates information sources from a large amount of shops, each of which has its products advertised. For instance, most of the shops within this E-mall have their names, sale categories and product information published. For each product, information such as its name, price and component items are expected to be provided. However, this product information structure is not fixed and can be flexible. We characterize basic features of a database as below:

- A database is a single-rooted, labeled, acyclic directed graph. Root is the only entry point of it. Each node in the graph is an object with a unique oid (such as *&10*) and a unique label (such as *kit*). Each object can have multiple parent objects but with the same label linked to it. Each labeled edge represents a single-step path from a parent object to its child object.

- Each object in OEM is either atomic or complex. An atomic object has a value of one primary type (such as an integer, string, or image). The value for a complex object can be seen as a collection of subobjects taking the form of <label, oid> pairs. A complex object never has a primary type value of itself (as in XML, the value for one attribute of an XML element can be modeled as a child atomic object).
- An object with the Null value is a specific case. An object in such a state can either become a complex object by adding an outgoing edge to an atomic object, or turn out to be a real atomic object by changing the null value to a value of another primary type. On the other hand, a complex object can be changed back to an atomic object by removing all the links to its child objects.

2.2 Web View Specification

In this paper, we focus on exact path expressions that specify each single-step path. We give the general form of a WVS as:

Define web view *favorite_products* as
select *c, k, p* – a view variable list
from *E-mall.shop s, s.category c, s.kit k, k.price p, k.item i* – a set of path conditions
where *c = "toy" and p < $50 and z = "book"* – a set of value predicates
with *k.category c, k.price p* – a web view construction statement

Figure 2: Example Web View Specification

In our WVS definition, the *selection list* can specify more than one type of object to be returned by the *path and value conditions* that consider joins.

Given the example in Figure 1, the WVS is shown in Figure 2 get a collection of favorite_products, each of which is found in "toy" shops of this E-mall, costs less than $50, and at least contains one item of *book*. This web view is constructed by each *product* object with its *category* and *price* object. Variables (such as s, k, c, p and i) that are attached to both ends of each path are designated for a set of objects respectively (for example, s is binding to the object set of *&2, &3, &4*). Variables can be distinguished according to their location in the global graph.

2.3 Basic Types of Web Updates

Like previous work [AMR+98], we consider three types of basic updates on web data source: *Ins*, *Del* and *Chg*. <*Ins*, o_1, *l*, o_2> and <*Del*, o_1, *l*, o_2> represent the insertion and deletion of the edge with label l from object o_1 to object o_2. <*Chg*, *o*, *OldVal*, *NewVal*> denotes the change of the value of the atomic object o from *OldVal* to *NewVal*. For both insertion and deletion, o_1 must be a complex object while o_2 can be either an atomic object or a complex one.

Note that these basic update transactions would affect nothing if o_1 is an unreachable object, and this reachability of o_1 will not be changed by an *Ins* or *Del* operation since it is at the start of the edge l. On the other hand, o_2 together with all its descendant objects becomes reachable after an insertion operation while it may be unreachable after a deletion operation.

3 Evaluation Strategy for Web View

In the web data graph, there is no strict schema restriction. Note however that WVS asserts over such a data graph a query specification, that imposes a query pattern to filter out only objects that conform to their corresponding aggregation paths. In this section, we study this query pattern imposed by a WVS, characterize it by exploiting a structural graph of aggregation paths, and develop our evaluation strategy based on it. Then we introduce an APIX structure for those objects to store their value evaluation results that are computed along these aggregation paths.

3.1 Query Pattern of Web View Specification

When initially setting up a web view, we need to access the base data to identify the relevant information. The WVS asserts a query pattern over the base data graph with two kinds of condition restrictions – **Path Conditions (PC)** and **Value Conditions (VC)**.

3.1.1 Path Conditions and Value Conditions

Web view evaluation involves path evaluation and value evaluation. The **PC** reflects the evaluation criteria on an object conforming the path specifications of the WVS. It corresponds conceptually to an overall path pattern graph structure, within which each object falls into one type of evaluation pattern on its outgoing aggregation path set. This road-map-like query pattern serves as a filter to distill out of the base data graph a small conforming subgraph. The **VC** on the other hand, only includes value evaluation criteria on atomic objects. In addition, there exists an implicit value aggregation function for each complex object to compute its aggregation value result along its required path set. The final web view consists of objects that satisfy both the **PC** and the **VC**. Based on the separation of these two types of conditions in a WVS, we propose a two-phase-evaluation strategy by first applying an overall path pattern graph against the base data graph for path evaluation and second by propagating bottom-upwards the computation of the aggregation values for the value evaluation starting from the primitive conditions at the atomic leaf objects.

3.1.2 Query Graph

Definition 1 (Condition Relevant Path – CRP) Each single-step path that is represented as a label in the *from* as well as in the *where* clauses in a WVS is relevant to the path evaluation, and thus is referred to as a **Condition Relevant Path (CRP)**. In general, we refer to *CRP* in two different contexts. A complete path that concatenates adjacent *one-step paths* starting from the root variable and ending with some atomic variable, we express it using a specific term of *a-CRP* in this global context. For a variable v, the term *v-CRP* denotes the aggregation single-step path set that is imposed in such a local situation.

Example 1 For the given example in Figure 1, the single-step paths (each with two variables attached at both ends) in the WVS (given in Figure 2) are (e is the root variable for E-mall).

$$e.\text{shop } s, \quad s.\text{category } c, \quad s.\text{kit } k, \quad k.\text{price } p, \quad k.\text{item } i$$

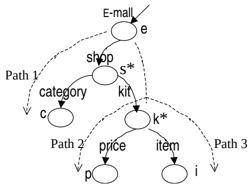
Three complete *a-CRPs* (attached variables are eliminated to avoid the interference):

E-mall.shop.category, E-mall.shop.kit.price, E-mall.shop.kit.item

The *v-CRP* for the root variable e is: *shop*. There is a procedure of computing *v-CRP* sets from a given set of complete *a-CRPs* of the WVS (given in Figure 2):

$a\text{-}CRP_1$: E-mall.shop.*category*
$a\text{-}CRP_2$: E-mall.shop.*kit.price*
$a\text{-}CRP_3$: E-mall.shop.*kit.item*

a-CRPs



Query Graph

We see for our given example, this would be done as follows. $a\text{-}CRP_2$ and $a\text{-}CRP_3$ overlap with $a\text{-}CRP_1$ for the segment of E-mall.shop but not after variable s. While they diverge from each other after their common segment of E-mall.shop.*kit*. Thus we obtain for variable s (represents E-mall.shop) its *v-CRP* set is category and kit and for k (represents E-mall.shop.*kit*) its *v-CRP* set is price and item.

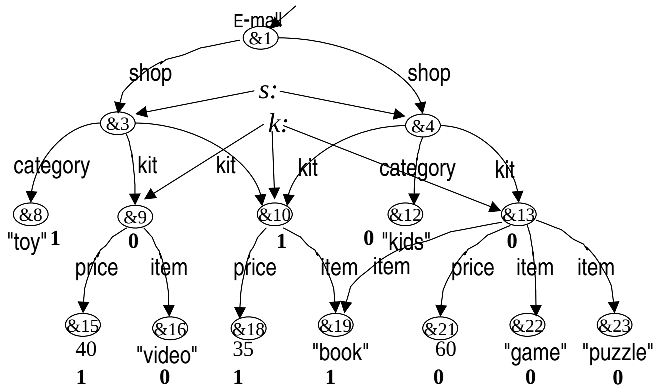
Definition 2 (Query Graph – QG) We construct an overall path pattern graph structure by overlapping the common segments of the *a-CRPs* and refer to this resultant graph as the **Query Graph (QG)** of a WVS. The **QG** can act as an overall query pattern against the base data graph for conducting the path evaluation.

Example 2 A Query Graph corresponds to the WVS (given in Figure 2). Variables marked by * are those that have its *v-CRP* set composed of more than one member outgoing path. Thus a type value aggregation function is prepared for each object of such variables. The *QG* summarizes all the path conditions and graphically illustrated the *v-CRP* set of each variable within it.

Given a set of WVSs, that have the same path specification part but may differ from each other in value predication or view selection parts, we can capture them by the same *QG*, i.e., the same path query pattern.

3.2 Our Two-Phase-Evaluation Strategy

Path evaluation is usually conducted in a Depth-First-Search (DFS) traversal process [AQM+97, Abi97]. The path evaluation may be conducted for the same object several times accessing it each time it is involved in the evaluation of one single path condition. Also, once we reach an atomic object via such a DFS path evaluation, we evaluate its VC (refered to as value evaluation) and then traverse upwards for other unprocessed path evaluations. This way, path evaluation and value evaluation are mixed, leading to evaluation efficiency. Hence, we propose our two-phase-evaluation strategy that separates path evaluation from value evaluation. In particular, for each object touched in a Breadth-First-Search (BFS) traversal, conduct a once-and-for-all path evaluation against its *v-CRP* path set.



(a) Evaluation Passage Graph of the Example Database

c

&8	&12
1	0

p

&15	&18	&21
1	1	0

i

&16	&19	&22	&23
0	1	0	0

(b) Seed truth values assigned according to the value evaluation of the atomic objects

Figure 3: Evaluation Passage Graph

With the *QG* serving as the guide for the required query paths and variables within it indicating their *v-CRP* sets, we first conduct the path evaluation. The result of the path evaluation pass is a subgraph of web data objects that are distilled from the base data graph. We refer to such a *virtual* subgraph structure as the *Evaluation Passage Graph (EPG)* (see Figure 3) and build an **Aggregate Path Index (APIX)** for objects captured by it.

3.2.1 Aggregate Path Index

Path evaluation proceeds as a BFS traversal process starting from the root object. For each object encountered in the traversal, we set up its *APIX* structure and initialize some auxiliary information needed for the second phase of computing value evaluation aggregation results.

The structure of *APIX* for each trapped object, as shown in Figure 4, is a cross-tabular: One tuple for each distinct object (for example, object &3); One column for each outgoing path member in its *v-CRP* set (represented by its corresponding variable, i.e., two columns for object &3 are “category” c and “kit” k). Each cross-bar hosts three measures of the child *oid* set of one outgoing path type (for example, &9, &10 is the child object set targeted by the “kit” path of object &3), the *oids*, the *Count* of children objects of each path type, and the cumulative truth value CT derived from each set of children objects, respectively. These three measures capture structural path information for the objects that conforms to the QG, hence we name it *Aggregate Path Index.*

APIX for objects of "shop" (s):

		c	k
&3	oids	{&8}	{&9,&10}
	Count	1	2
	CT	1	1
&4	oids	{&12}	{&10,&13}
	Count	1	2
	CT	0	1

APIX for objects of "kit" (k):

		c	k
&9	oids	{&15}	{&16}
	Count	1	1
	CT	1	0
&10	oids	{&18}	{&19}
	Count	1	1
	CT	1	1
&13	oids	{&21}	{&19,&22,&23}
	Count	1	3
	CT	0	1

Figure 4: Aggregation Path Index (APIX)

Besides these three explicit measures for each child object set of an object, as illustrated in Figure 4, there are more measures associated with an object. They are the aggregated truth value T and the signal *In_EPG* indicating whether the object itself belong to the *EPG* (if not, the space allocated for such index information can be released), and the only parent object set (since each object except the root has a unique type of incoming edges in Section 2).

3.2.2 Path Evaluation

Now we illustrate the process of path evaluation by procedure *PE* (see Figure 3.2.2. Starting from an object o (usually it is the root object since BFS(*"root"*) is called), a BFS traversal of the base data graph conducts the evaluation for each encountered object against its aggregation path set. If the object has all types of the outgoing paths that are asserted by its corresponding *v-CRP* set, we mark its *In_EPG* as *True* and assign it an *APIX* tuple of its type. For example, since the *v-CRP* set for variable s is category and kit, object &2 fails the path evaluation because it lacks the "kit" type of child object, while object &3 meets the requirement and is thus distilled.

Along with the traversal, we fill in the *oids* and the *Count* information (see Figure 3.2.2). At the end of this path evaluation pass, an *EPG* of data objects has been distilled with their path index as well as some initialization information captured in the *APIX* structure.

3.3 Aggregation Function for Value Evaluation

Value predicates specified in a WVS can be directly dealt with by the evaluation on atomic objects. Then these value evaluation results need to combined with the path evaluation results of the distilled subgraph of objects, each of which has an aggregation function attached for propagating upwards the value evaluation results of its children objects. For example, in Figure 3, the truth value of the object with oid &9 is the

```
Procedure BFS (Labels)
// Labels is a queue that stores labels to be evaluated on;
// BFS gets a label l from the top of Labels and evaluates the object set of l;
    get a label l from the queue Labels;
    if the end variable v of l is a leaf variable
        if Labels is empty
            return;
    else    // begin setting up API structure for the variable v
        for each of its v-CRP label l_i
            // put it into the end of queue Labels;
            Labels = Labels + li;
            // initialize information for its child object set
            Objs[li] = null; Count[li] = 0; CT[li] = 0;In_EPG = True;
            for each of the child object o_ij
                o.Objs[li] = o.Objs[li] + o_ij;
                o.Count[li] ++;
                                        if o.Count[li] = 0
                o.In_EPG = False;
    BFS (Labels);
```

Figure 5: Path Evaluation

conjunction of the truth values of its children objects &15 and &16.

We start the value evaluation from the atomic objects of the *EPG*, and assign for each a truth value *true/false*, or 1/0 based on if complying with the predicates of the WVS. Figure 3 shows the truth values that are attached to the atomic objects of the *EPG* (as shown by the Figure 3). For example, 1 is the truth value for object &8 that is binding to variable c, since it satisfies the predicate asserted on c: "exists x in c: x = "toy"".

Theorem 1 (Up-Propagating Truth Values) Each value aggregation function is decided by the aggregation path pattern of a variable, thus for objects with the same *v-CRP* pattern the function is the same. However, the aggregation result for each object is decided by its actual measures. The CT value for each of its children object sets is derived by a cumulative computation of the T value of all children object members within the set, then its own T value is computed via a conjunction of all the derived CT values for its children object sets.

The reason for the conjunctive method to compute the truth value T from all the CT values of its path divisions is obvious: the aggregated paths at their meeting point naturally assert to all the participated paths a conjunctive relationship, that reflects both the path evaluation and the value evaluation. However, within the same path division, all the children objects are under the same value evaluation and thus are considered by their parent object as one single contributor to the value evaluation result along this path. The aggregation conducted in difference place is coordinated in a bottom-up way, like the reverse process of BFS. At the end of this pass of value evaluation, all the truth T values for objects of *EPG* are obtained via this up-propagation.

3.4 Deriving the Web View

For each data object distilled by the path evaluation, its *APIX* structure is constructed to capture its aggregation path pattern as well as to accommodate its materialized value evaluation result. Thus a variety

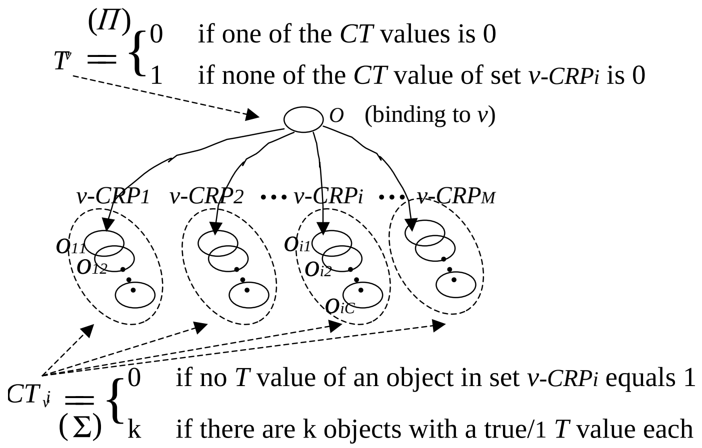
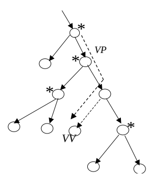


Figure 6: Aggregation Function for CT Value Computation

of web views could be easily derived by reusing this *APIX* structure.

Definition 3 (View Variables & View Paths) In a WVS, a list of variables specified in the *select* clause indicates the desired data, called **View Variables (VV)**. A **View Path (VP)** is a path that leads to a *VV*.

Example 3 Both the *a-CRPs* and *VPs* can be shown in one *QG* by augmenting the *QG* with dashed path segments of the *VPs*. The overlapped path segment of a *VP* with the *QG* is called **View Passing Path**. In the *QG* depicted in Figure View Path and Query Graph, there are six complete *a-CRPs* and the dashed path is the *VP* to a *VV*.



View Path and Query Graph

Web View Object Selection

select a view variable list
from a single-path set (attached with two variables, v_1 and v_2) of all CRPs
foreach single-path
apply each possible pair of bindings of satisfiable o_1 and o_2 to v_1 and v_2.
with view constructs

Theorem 2 (View Objects Selection) Each view object set can be selected via the objects that are along the *View Passing Path* and with satisfiable value evaluation results (T values are 1s).

The reason lies in the simple fact that the objects of the *EPG* with satisfiable *T* values indicate that they successfully passed both the path evaluation and the value evaluation and thus lead to the right view objects to be chosen. Thus no matter how many view variables are specified in the WVS, their view objects can easily be retrieved by utilizing the satisfiable objects along their *View Passing Paths*. Once we have retrieved all desired view objects, restructuring among them becomes a trivial job using local computation costs only.

Satisfiable objects are applied to their corresponding variables in the WVS to generate the web view as shown by the *Web View Object Selection* procedure.

4 Approach for Materialized Web View Maintenance

To keep a materialized web view up-to-date with dynamic web data sources, we now propose efficient maintenance algorithms based on the cached *APIX* structure.

4.1 Checking Self-Maintainability

Updates are said to be irrelevant to the materialized web view if they would not cause any effect to it. With the local materialized auxiliary information stored in the *API* structure, we have a set of self-maintenance tests that avoid any remote access to base data for such cases. We illustrate a complete list of our best cases in which irrelevant updates are discovered by our algorithm. The last two of them cannot be identified by other approaches without an *APIX* [ZG98b].

- For a <*Chg, o, OldVal, NewVal*> update, if the value evaluation result of its *OldVal* is the same with that of its *NewVal*, then this update is irrelevant. For example, given the base databases example shown by Figure 1, <*Chg, &6, "fashion", "bakery"*> is an obviously irrelevant update since neither "fashion" nor "bakery" belong to the category of "toy".
- For an *Ins* or *Del* operation, if either o_1 or o_2 do not belong to any of the *CRPs*, then the update is irrelevant. For example, operation <*Ins, &2, "location", &24*> (assume the value of the atomic object &24 is "boston") has nothing to do with the path evaluation thus no effect on the materialized web view.
- In a <*Del*, o_1, *l*, o_2 > case, if neither o_1 nor o_2 is an object with its information materialized in the *APIX*, then it is is irrelevant. An example for this case is <*Del, &2, "name", &5*>, before which &2 wasn't materialized in the *APIX* since it didn't pass the path evaluation. Thus deletion of such object would have no effect on the materialized view.
- For a <*Chg, o, OldVal, NewVal*> transaction, in which its value evaluation result is changed from true to false, we check whether any parent object of *o* is materialized in the *APIX*. If no such parent object exists, this update is irrelevant. For example, if the value for object &6 is changed to "toy", there is still no chance for re-evaluating its value since the path evaluation is stopped by its parent object &2, which doesn't have an outgoing path "kit".

The above checks indicate a sequence from simple to complex. The first test only requires to check on the *Chg* update type and the old and new values of the updated object. The second one needs to check on the path relevance of the affected object pair. These first two kinds of self-maintenance checks also appear in other work [ZG98b]. Our self-maintenance tests are more effective as they are able to also discover the last two cases of irrelevant updates based on the information materialized in the *APIX*.

4.2 Accessing Base Data

After these self-maintainability tests, only relevant updates remain. Hence we now would need to refer back to base data for maintaining the materialized web view and also the *APIX*. The maintenance task of the materialized *APIX* includes adding/deleting object tuples and keeping their measures up-to-date according to the structrual changes as well as the modified value evaluation results. The latter could in turn trigger maintenance procedures for maintaining the materialized web view. Later we show that the cost of this maintenance approach in terms of the number of accesses to base data is much reduced compared to the alternate solutions.

```
Procedure Ins (o1, l, o2)
  If o2 ∉ EPG
  BFS(l) from o2 // result in a o2-"EPG";
    If o2 ∉ "EPG" or o2.T = 0
      Judged to be an Irrelevant Update;
      exit;
    else
      if o1 ∉ EPG
        re-evaluate on o1;
      else
        cache o2 in EPG;
        o1.Count[l] ++; o1.CT[l] ++;
        if o1.CT[l] > 0 and o1.T = 0
          for each of other labels l^i
            o1.T = o1.T × o1.CT [l^i];
          if o1.T = 1
            propagate on its parents
```

Figure 7: Insertion Maintenance on APIX

```
Procedure Del (o1, l, o2)
  If o1 ∉ EPG or o2 ∉ EPG
    Judged to be an Irrelevant Update;
    exit;
     o1.Count[l]–;
  if o1.Count[l] = 0
    drop o1 from API;
    propagate the effect to parent objects.
  else
    o1.CT[l] – ;
  if o1.CT[l] = 0
    o1.T = 0;
    propagate the effect to parent objects.
```

Figure 8: Deletion Maintenance on APIX

4.2.1 Insertion Scenarios

Maintenance of the *APIX* upon an insertion case *<Ins, o_1, l, o_2>* is shown in procedure *Ins(o_1, l, o_2)* (see Figure 7). More tuples of objects are usually newly cached into the *APIX* due to the insertion updates.

The edge l has beed checked by the self-maintainability test and hence is sure to be one of the *CRPs*. We conduct the two-phase-evaluation on the data subgraph starting from o_2. If the aggregated value evaluation result of o_2 turns out to be 0, then this insertion is an irrelevant update. Otherwise the method *Inc_CT* is called for propagating the effect of the newly added edge to the satisfiable object o_1.

We have no materialized information about an object if it fails the path evaluation of its *v-CRP* set. Thus if o_1 wasn't materialized in the *API* at that time, then the newly introducted edge l from o_1 to o_2 causes us to re-evaluate the *v-CRP* set of o_1. Only if the path evaluation succeeds for o_1 and none of its parent objects also wasn't materialized in the *API*, then the next upper level path evaluation is carried on. Otherwise, if the path evaluation for o_1 fails, we can stop the process since the update is already judged to be irrelevant. If the path evaluation for o_1 succeeds and o_1's parent objects exist in the *API*, then the former broken path passages via these objects to o_2 now are conductive. Along with this upwards path evaluation, we carry on the value evaluation and accommodate their value evaluation results in the *APIX* (see Figure 9). We present two insertion cases to illustrate the effect on the materialized *API* of the maintenance process.

Scenario 1: <Ins, &2, "kit", &10> :

		c	k
&2	oids	*{&6}*	*{&10}*
	Count	*1*	*1*
	CT	*1*	*1*
&3	oids	{&8}	{&9,&10}
	Count	1	2
	CT	1	1
&4	oids	{&12}	{&10,&13}
	Count	1	2
	CT	0	1

Scenario 2: <Ins, &10, "item", &16> :

		c	k
&9	oids	{&15}	{&16}
	Count	1	1
	CT	1	0
&10	oids	{&18}	*{&16,&19}*
	Count	1	1
	CT	1	*2*
&13	oids	{&21}	{&19,&22,&23}
	Count	1	3
	CT	0	1

Figure 9: Updated Aggregation Path Index (APIX)

4.2.2 Deletion Scenarios

Upon a deletion case *<Del, o_1, l, o_2>*, the update first is screened for irrelevancy by the self-maintainability test if either *o1* or *o2* does not exist in the *APIX*. However, if o_1 has the only one outgoing path of type l to o_2, this deletion would dissatisfy the aggregation path restriction on o_1 and thus cause the deletion of its tuple from the *APIX*. Correspondingly, we propagate the effect of this deletion. The deletion procedure *Del(o_1, l, o_2)* is depicted in Figure 8.

4.2.3 Change Scenarios

A *<Chg, o, OldVal, NewVal>* update is a relevant if it bears different value evaluation results for o before and after its value change. If the value evaluation of *NewVal* is 1 (i.e., the value evaluation of *OldVal* is 0), then it equals to a set of *<Ins, o_1, l, o>* insertions, each of which with l standing for the only type of incoming edge of o and with o_1 representing one of the parent objects of o. Similiarly, if the value evaluation of *NewVal* is 0 (i.e., the value evaluation of *OldVal* is 1), then it is equivalent to a set of *<Del, o_1, l, o>* deletions.

4.3 Computation of Web View "Patches"

The maintenance of the materialized *APIX* involves the adding or deleting data object tuples and fixing value evaluation results of some data objects. In the *APIX*, the newly appeared *true/(1)* T values of data objects, either from the added data object tuples or due to the changed value evaluation results of data objects, would trigger the *ADD* maintenance statements for computing web view "patches". On the other hand, the disappeared *true/(1)* T values of data objects, either by the deletion of such data object tuples or due to the changed value evaluation results, would trigger the *DEL* maintenance statements.

Example 4 *We log the For generating view objects "patches" to be added, we can apply the ADD maintenance statements shown as below:*

ADD+ = **select** a view path list
from view paths
foreach view passing path
applying each possible pairs of bindings of objects with new *true/1* T values

5 Evaluation on Costs for Web View Maintenance

Like others' work, we assume that the main cost of the computation 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 object of base databases could be quite large in storage and its retrieval takes time. In fact, one could even assume that these objects (XML documents, for example) lie on different servers on the Internet. Hence, each base data object access may require a URL locating and an http transfer across networks.

5.1 Cost Factors

Next we consider key factors that account for the cost. The first two features depend on the query pattern, as shown by Figure 3.2.2, while the last two are more overall measures of a base database.

- $C(object\ occurrences)$: how many object instances bind to that variable.
- $M(outgoing\ label\ diversity)$: how rich in types of outgoing query paths a variable is.
- $H(height\ of\ base\ data\ graph)$: length of the longest path from the root to some atomic object.
- $N(total\ number\ of\ base\ data\ objects)$: the size of the base database.

From the first two parameters, we can estimate the population of the children objects of an object based on its binding variable characteristics. $M * C$ is the rough number of children for an object if C is fairly uniform. If the directed graph structure is balanced (the path lengths of atomic objects do not differ much from each other) and the deviation of M for each variable is quite small, then the data object explosion rate along one level down can be estimated as $M * C$. Thus after H levels down, the number of atomic data instances is about $(M * C)^H$. The total number of base data objects is a sum of the number of objects at each level.

On the reverse side, we use $C\prime$ to measure how many parent object instances an object has. The incoming label diversity $M\prime$ is 1 according to our data model specification. Usually, $C\prime$ is much smaller than C. Alternative maintenance techniques access base data from the root object may result in a large examination space. Our maintenance approach is carried on in a reverse direction. Starting from the touched object, our method to do maintenance examines upwards the parent objects, whose structual and value evaluation relevant information is already computed in the initialization phase and materialized in the *API* structure. Integrating this precomputed aggregation information with those of the updated object, we can quickly derive the new effects. The cost is a function of $C\prime$ instead of in the order of $M * C$.

The cost spend in the evaluation phase is decided by a *Reduction Factor* for each variable that describes the ratio of the number of objects being filtered out to the whole size of this object set. We formulate below the costs for the evaluation and maintenance phases.

5.2 Web View Computation Cost during the Evaluation Phase

Using the web view specification in Figure 2 to evaluate the base database shown as Figure 1, both the naive algorithm and Abiteboul's algorithm [AMR+98] conduct a DFS traversal during their evaluation phase, and the total number of objects they access is:

$Cost_{naive}^{eval} = C_e + C_s + C_c + C_k + C_p + C_i \quad (with\ C_e = 1)$

$Cost_{abit}^{eval} = C_e + C_s + C_c + C_k + C_p + C_i$

In both the naive algorithm and Abiteboul's algorithm, an object is evaluated no matter if it really has a complete set of outgoing paths that comform to its *v-CRP* or not. For example (see Figure 1), object &6 of variable *c* is evaluated in both the naive algorithm and Abiteboul's algorithm even if it does not have an outgoing path to objects of variable *k*. Such kind of accesses to base data objects is a waste of time. Our evaluation strategy conducts a once-and-for-all path evaluation in the process of BFS traversal and thus eliminates the unqualified objects from the evaluation space at a much earlier time. In this way, we considerably reduce the number of accesses to base data. We later refer to our approach as *APIX* as opposed to the other two approaches of *naive* and *abit*.

Theorem 3 (Reductive Factor for Objects to be Evaluated) During the evaluation phase, how much we cut down the costs is decided by the *Reductive Factor*. Assuming a uniform distribution of all combination probabilities of the outgoing paths of an object, the *Reductive Factor* for each variable is the ratio of the occurrences that at least encompass the required outgoing path set to the total occurrences. It is in the inverse proportion to an exponential function whose base is 2 and exponent is the number of the required outgoing paths by its *v-CRP* set. This *Reductive Factor* also indicates the storage space size needed by the *API.*

Illustrating it in more detail, suppose that for a variable v, we have M joint paths (outgoing labels) to evaluate and each of these paths leads to a variable v_i (i is from 1 to M). The evaluation of the subobjects of any variable v_i is worthwhile only if it also has subobjects of all the other $M-1$ variables. By a uniform distribution, the probability of having all the other $M-1$ variables is $\frac{1}{2^{M-1}}$. By applying this formula to the evaluation cost of objects of variable e and s in our example database, e has just one path leading to s, thus $\frac{1}{2^{1-1}} = 1$ times of the object occurrences of s need to be evaluated. While s has two joint paths to be evaluated, thus $\frac{1}{2^{2-1}} = \frac{1}{2}$ times of the object occurrences of variable c and k need to be evaluated. The access cost caused by using our approach is:

$Cost_{API}^{eval} = C_e + 1 * C_s + \frac{1}{2}C_c + \frac{1}{2}C_k + \frac{1}{2}C_p + \frac{1}{2}C_i \ \ (with\ C_e = 1)$

5.3 Referring Back Cost during Maintenance Phase

For below, suppose an operation <Ins, \$9, "item", \$24> happens with the value of the atomic object \$24 being "book". Thus \$9 is o_1 and \$24 is o_2, and o_1 is binding to variable k.

The maintenance by the naive approach involves the total recomputation of the view against the base databases. Hence the cost is the same as that of the initial phase.

$Cost_{naive}^{maint} = C_e + C_s + C_c + C_k + C_p + C_i$

Abiteboul's algorithm still needs to go back to the root object and re-evaluate the base data. However, it can apply &9 directly to variable k while saving the accesses to other objects of k. For example, for <Ins, \$9, "item", \$24>, Abiteboul's algorithm needs to access all the objects that are attached to e, s and c and one object \$9 of variable k while ignoring all the other $(C_k - 1)$ objects. Also, the objects to-be-evaluated of variables p and i (descendant variables of variable k) are restricted to only those descendant of \$9.

Let the object occurences of variable p and i stemming from the object &9 are C_p' and C_i' respectively, then the number of total objects accessed using Abiteboul's algorithm is:

$Cost_{Abit}^{maint} = C_e + C_s + C_c + 1 + C_p' + C_i'$

As proposed in Section 4, our algorithms can avoid a large number of the accesses to base databases by detecting irrelevant updates. Even in the worst case when access to base databases is inevitable, we access only the base data objects from the affected one (i.e., &9). Thus the accesses to the objects of s and c are saved. The maximum number of total base objects evaluated by our algorithm under the same situation as Abiteboul's is:

$Cost_{API}^{maint} = 1 + \frac{1}{2}C_p' + \frac{1}{2}C_i'$

5.4 Cost Comparison of Experimental Results in Three Scenarios

Experiment tests on the maintenance costs under three different update scenarios to the base database (see Figure 1) are shown in Figure 10. The database contains one E-Mall, 1000 shops, 100 products and 2 categories per shop, and 10 items and 1 price per kit, and possibly other portions of database that are irrelevant to the WVS. We observe from the experiment result that, in an *Ins* update situation, the cost of maintenance is mainly associated with the size of the subgraph starting from the affected object. A deletion involves the propagation of the changed value evaluation result or the dropping of object tuples at the local *API* site. The number of the affected object tuples is in linear relation with the height of the affected object. An *Udp* update is most expensive since it affects an atomic data object, which is at the bottom of the data graph. The maintenance involves a longest reverse evaluation from bottom upwards.

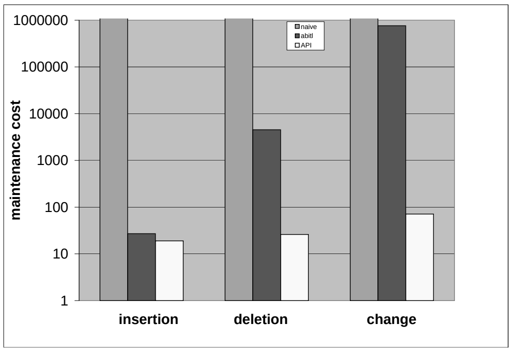


Figure 10: Maintenance costs for updates

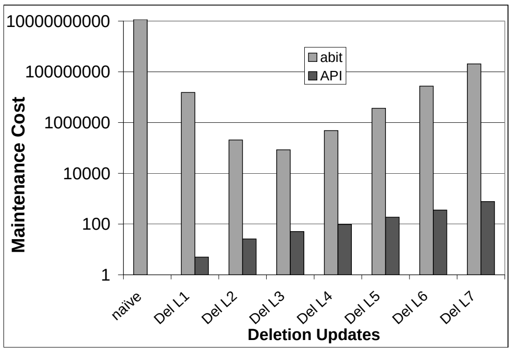


Figure 11: Varying the position of effected object

Figure 10 also shows that our approach wins significantly over the other two methods especially in the *Del* and *Udp* situations. In the second experiment, we use a view specification containing a chain of eight one-step paths in the *from* clause:

$$\text{select } z_i \text{ from } A.L1\ z_1,\ z_1.L\ z_2,\ \ldots,\ z_7.L\ z_8;$$

Figure 11 shows that our algorithm achieves the more significant improvement in the terms of maintenance costs for deletion cases if the deletion update occurs closer to the root object. This is because the higher the objects are (opposed to the lowest atomic objects), the shorter pathes they may go through to propagate up their dropped/changed value evaluation information to maintain the *APIX*.

For the experiment shown by Figure 12, we use the example WVS shown in Figure 2. We increase the number of *shops* in the database from 1000 to 5000, and keep the same average ratio of *kits* per shop, items per kit, etc. Therefore, when the number of *shops* doubled, for example, the size of its subgraph is doubled. We conduct three kind of insertion operations by adding the edges *kit* , *category* and *item* respectively to the database and test the costs caused by our algorithm against Abiteboul's. We see that both sets of maintenance costs grow linearly with the size of relevant subgraph. Our approach gains much compared

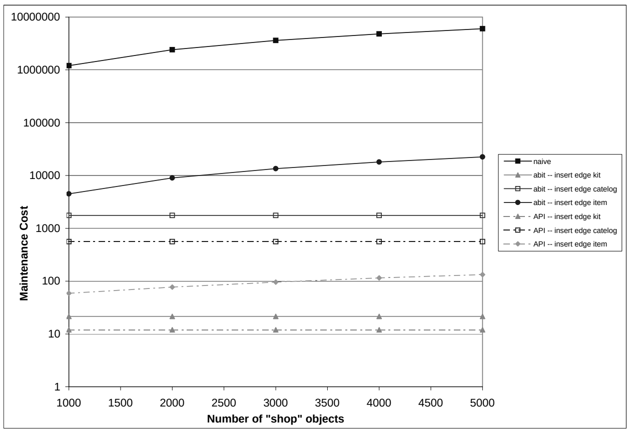


Figure 12: Varying the database size

to the alternative one when inserting a lower object such as *item* (we thus observe an opposite prefer from deletion cases). The reason for this is that the access time to the base data by our insertion maintenance approach is related to the size of subgraph that stems from the inserted data.

All three experiments are designed to be similiar to Abiteboul's work [AMR+98] to set up a reality uniform testbed based on which the experimental results can be compared against. These experimental studies help us to identify suitable cases for our algorithm: (1). The richer the *WVS* is in terms of path conditions and strict value predictates. Then, the base database is evaluated against a more complex *QG* and the evaluation is more effective to screen out undesired objects. (2). A big database bears a larger ratio of the average of object occurrences to the label diversity, this indicating a good reduction factor. (3). *Del* operations happening to the upper objects while the *Ins* operations occuring the lower ones. (4). Expense on storage is less important compared to the network communication or the times of connections to be established.

6 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 an index-like mechanism *APIX*, which construct itself according to the aggregation path restriction by the WVS and accommodates the conforming objects togther with their value evaluation results. This way, a set of web views specifications can reuse their common part of path evaluation criteria and compute the final view objects to restructure the web views from. Also, the updated objects can be explored on to derive the web view "patches" to be integrated into the materialized web view.

We conduct the cost analysis and the experimental studies on the maintenance performance comparisons with the alternative solutions at the state-of-art. Both the theoretical analysis and the experimental results show that our approach win over its competitors most of the time and in some cases the gains of our strategy are significantly with more probablity of self-maintenability or fewer accesses to base data. We develop a set of efficient strategies as for the initialization phase of web view evaluation as well as its incremental maintenance.

We use XML files to simulate web data sources and have implemented the web view mechanism based on that. We plan to extend our web view specification for accommodating also regular path expressions, and develop more general *APIX* structure to allow for such an extension. We find that the storage space *APIX* can be economized by compressing a multi-step non-branching paths into one single-step path. The corresponding maintenance strategy is possible. Finally, we would like to consider exploiting XML schema, its linking mechanism and query language to optimize the web view maintenance.

References

[Abi97] S. Abiteboul. Querying Semistructured Data. In *Proceedings of the Int. Conference on Database Theory*, pages 1–18, 1997.

[AMM97] P. Atzeni, G. Mecca, and P. Merialdo. To Weave the Web. In *Int. Conference on Very Large Data Bases*, pages 206–215, 1997.

[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.

[Aro97] G. Arocena. WebOQL: Exploiting Document Structure in Web Queries. In *Int. WWW Conference*, 1997.

[CAW98] S. Chawathe, S. Abiteboul, and J. Widom. Representing and Querying Changes in Semistructured Data. In *Proceedings of the Fourteenth International Conference on Data Engineering, Orlando, Florida*, pages 4–13, February 1998.

[CRCK98] K. T. Claypool, E. A. Rundensteiner, L. Chen, and B. Kothari. Re-usable odmg-based templates for web view generation and restructuring. In *CIKM'98 Workshop on Web Information and Data Management (WIDM'98), Washington, D.C., Nov.6*, pages 314–321, 1998.

[FFLS97] M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A Query Language System Demonstration - Strudel: A Web-site Management System. In *ACM SIGMOD Conference on Management of Data*, 1997.

[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.

[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.

[Mih96] G. Mihaila. WebSQL - an SQL-like query language for the WWW. Master's thesis, University of Toronto, 1996.

[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.

[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.

[Suc96] D. Suciu. Query Decomposition and View Maintenance for Query Languages for Unstructured Data. In *Proceedings of the 22nd International Conference on Very Large Databases, Mumbai (Bombay), India*, pages 227–238, September 1996.

[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.

A Appendix

A.1 Pseudo Code for Path Evaluation Algorithm

```
Procedure Path_Evaluation (o)
{
     if  BFS_CRPs ({"root"}, {o}) = True
         generate a "EPG" from o by including
         objects that their In_EPG = T
}
Boolean BFS (Labels, Objs)
{    int has_obj = 0;
     get a l from queue Labels;
     if  the ending variable of l, v is a leaf variable
         if Labels is empty
              return True;
     else  {
         for each li in v-CRP, put li in Labels;
         for each object o of Objs[l] {
            o.set_labels;
            if  o.In_EPG = T  {
                has_obj ++;
                for each label li in o.Label[ ]
                   Objs[li] = Objs[li] + o.Objs[li];
            }
         }
         if  has_obj = 0
            return False;
     }
     return BFS_CRPs (Labels, Objs);
}
```

(a) Path Condition Evaluation Conducted in a BFS Traversal

```
Object  ::
Member
{
  Boolean In_EPG = F;   int   T = 1;
  Set   Labels = Ø;         Set  Objs[ ] = Ø;
  int   Count[ ] = 0;         int   CT[ ] = 0;
}
Method  set_Labels
{ binding o with variable v;
  if v is a variable for leave nodes
    In_EPG = True;
  else {
    for each li in v-CRP {
       Labels = Labels + {li};
       Objs[li] = null;  Count[li] = 0;
       CT[li] = 0;
       for label li, for each of its paired subObj oij {
           Objs[li] = Objs[li] + {oij};
           Count[li] ++;
       }
    }
    for each label li in Label[ ] {
       if  Count [li] = 0
           In_EPG = False;
    }
  }
}
```

(b) Joint Variable Objects Index Structure Initialization

Figure 13: Path Evaluation Algorithm

A.2 Pseudo Code for Aggregation Function

```
Procedure Compute_Truth (EPG)
//Aggregation Function for computing truth value;
{
   According to the QG within this EPG,
   sort the variables bottom-up in a partial order;
   excluding leaf variables,
   for each variable v {
        for each o of variable v;
            o.comp_CT;
   }
}
```

```
Object :: Method  comp_CT
{   T = 1;
    for each label li in Label[ ] {
        for label li, for its paired subObj oij {
             CT [li] = CT [li] + oij.T;
             if  (CT [li] == 0)  {
                    T = 0;
                 exit;
             }
        }
    }
}
```

Figure 14: Aggregation Function for CT Value Computation

A.3 Pseudo Code for Insertion and Deletion Algorithm

```
Procedure Ins (o1, l, o2)
{    If  o2 ∉ EPG   {
         PE(o2);    // result in a sub-"EPG" from o2;
         If  o2 ∉ "EPG"   or   o2.T = 0 {
              Judged to be an Irrelevant Update;
              exit;
         }
         else  {
              if  o1 ∉ EPG
                  re-evaluate on o1;
              else  {
                  cache o2 in EPG;
                  o1.Count[l] ++; o1.CT[l] ++;
                  if  o1.CT[l] >0 and o1.T = 0  {
                       for each of other labels li
                          o1.T = o1.T × o1.CT [li];
                          if   o1.T = 1
                              propagate on its parents
                  }
              }
         }
    }
}
Procedure  Del (o1, l, o2)
{    If  (o1 ∉ EPG) or (o2 ∉ EPG)  {
        Judged to be an Irrelevant Update;
        exit;
    }
    o1.Drop_Obj (l);
}
```

```
Object  ::

Method  Inc_CT(l)
{   CT[l] ++;
    if  CT[l] = 1  {
         for each label li in Label[ ] other than l
             T = T × CT [li];
         if   CT = 1
             for each of its parent o´ with label ll
                 o´.Inc_CT (ll);
    }
}
Method  Dec_CT (l)
{   CT[1] -- ;
    if  CT[1] = 0  {
        T = 0;
        for each of its parent o´ with label ll
            o´.Dec_CT (ll);
    }
}
Method  Drop_Obj (l)
{   Count[l]--;
    if  Count[l] = 0 {
        In_EPG = F;
        for each parent o´ with label ll to it
            o´.Drop_Obj (ll);
    }
    else
        Dec_CT(l);
}
```

Figure 15: Maintenance Algorithms under Insertion and Deletion Scenarios