Paper Number AMERICA197

Self-Maintainability of Graph Structured Views

Yue Zhuge and Hector Garcia-Molina
Computer Science Department
Stanford University
Stanford, CA 94305-2140, USA
{zhuge,hector}@cs.stanford.edu

Abstract

Materialized views need to be maintained in response to changes of base data. It is desirable to do this maintenance without accessing base data, to reduce overhead and contention for base data. In this paper we study the self-maintenance problem for materialized views defined over a graph-structured database (GSDB). We formally define the notion of self-maintainability for such views, and show how to check if a base update can be incorporated into a view without referencing base data. We also characterize views that are always self-maintainable, and compute the auxiliary information that needs to be stored in order to make a view self-maintainable.

1 Introduction

Materialized views copy selected or summarized data of interest to improve query performance and data availability. They are especially useful when the query site is separate from the base data, e.g., views cached for Web browsing, or stored in a data warehouse. In such an environment, materialized views can be stored close to the queries, and they may significantly reduce the query response time. Of course, it is important to maintain the materialized views up to date without too much overhead, or else the advantages may be lost.

As semi-structured data, and XML [7] in particular, become popular, it becomes important to understand how such data can be efficiently maintained in a materialized view. To illustrate, consider a Web server that manages a collection of documents and exports them in XML format. The client Web browser may cache the table of content (TOC) of selected server documents. A document TOC is an XML object containing the title, section headers, a list of figures, and links to the appropriate parts of the document on the server. Users may browse or query the TOC, and may retrieve document content by following links. (For a demo of a caching Web browser and server that operates in a similar fashion, please visit [8].) When the document at the server is modified, the server must notify the client so that changes can be incorporated into the existing client cache. Propagating the changes may need access to server data, for example, when a new section is created, links to the figures in this section should be added to the list of figures in the TOC. The cached TOC is a materialized view defined, and must be efficiently maintained.

Maintaining views without accessing source data may greatly reduce maintenance time and complexity, and may avoid data inconsistency problems. Intuitively, we say that a materialized view V is *self-maintainable* if V can be incrementally maintained without referencing the base data.

In this paper, we study self-maintainability issues for materialized views defined over a *graph-structured database* (GSDB). Informally, a GSDB is a collection of "objects" that may contain "pointers" (graph edges) to other objects. A GSDB can represent semi-structured information such as XML objects, web pages or integrated heterogeneous data. Instead of using XML or other semantically rich data model, we have chosen to use a simpler model that captures features necessary to understand self-maintainability (see Section 2). In the model we use, a materialized view is a collection of objects, which is also a graph-structured database. We also chose an OQL-style simple language to define our views. Because of this simple view definition, materialized views can be maintained much more easily, and self-maintainability is possible.

We model our view maintenance process in the following environment. The base data is stored at a *base site*, the materialized views are stored at a *view site*. As shown in Figure 1, a module called view *maintainer* receives the base updates and maintains the materialized views. The maintainer may use a *self-maintainability test* (SMT) to decide whether an update is self-maintainable with respect to a view. During the test, the maintainer may access auxiliary information such as indices stored for the purpose of view maintenance. The maintainer may also query the view. After deciding the self-maintainability of an update, the maintainer can maintain the view using a view maintenance algorithm, with or without querying the base data. In reality, the base site and the view site may not be physically separate.

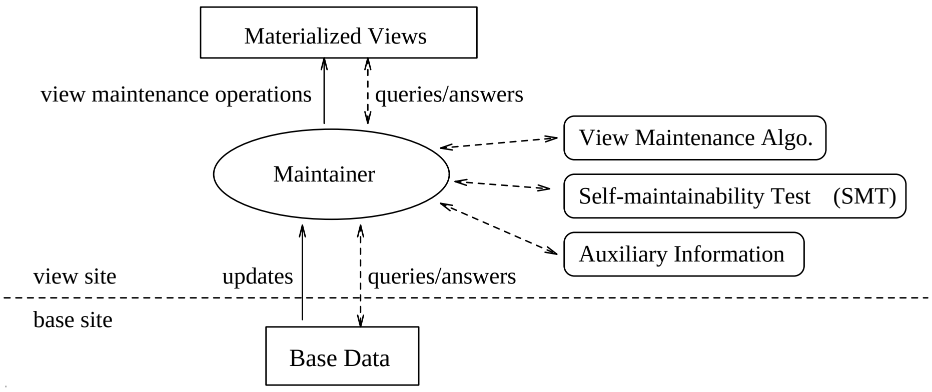


Figure 1: Conceptual view maintenance environment

Self-maintainability has been studied in the relational context but not for graph-structured data. Because graph-structured views are defined differently from relational views, view maintenance techniques vary substantially for these two data models, as do the approaches to decide whether a view is self-maintainable. For the relational model, the update information usually contains the updated tuple, but representing update information for the graph-structured data model is more complicated. For example, if an object O is involved in an update, will O's children objects be considered part of the update information? The same view may or may not be self-maintainable when different update information is available. Another complication for graph-structured views is the lacking of "scope restrictions" on base updates. In particular, the unit of relational updates is a tuple, while in a GSDB, a simple change like adding a link between two objects can potentially connect two graphs. If we modify a tuple in a relation, we know that only views defined on that relation will be affected. In a GSDB, however, adding a link between two objects can dramatically change the set of objects in a view.

Although the problems are analogous, the techniques for deciding the self-maintainability of graph-structured views are different from those for relational views, and these are the focus of this paper. The major contributions of this paper are

- We formally define self-maintainability for graph-structured views.
- We design a self-maintainability test that decides whether an update is relevant or self-maintainable to a view. We also show how to perform the SMT given different kinds of information on the base database, updates and views.
- We characterize the views that are always self-maintainable.
- We compute the auxiliary information that needs to be stored in order to make a view self-maintainable.

We review the data model, view definition and materialized view maintenance in Section 2, define self-maintainability and related terminologies in Section 3, and present self-maintainability test in Section 4. Section 5 addresses how the maintainer may evaluate the test conditions, and provide some "rules of thumb" that can be used to simplify the evaluation. Section 6 presents one type of view that is always self-maintainable, and, based on that, discusses how to keep auxiliary data to help maintain a view. The rest of this introduction is devoted to related work.

1.1 Related Work

Currently at least two teams are at work in defining and maintaining materialized views over graph-structured (or semi-structured) data. In both cases, the data models are labeled graphs, but views are defined using different query languages. The views of the first, Suciu et al. [13], are defined using the query language UnQL [5]. These views are defined using "join-free" queries, and updates are either concatenation or replacement of two disjointed graphs. In this study all views are always self-maintainable, and an algebraic approach can be used to maintain the views. That is, algebraic expressions can be found to compute delta views corresponding to the changes of base data.

Another group working on this subject is here at Stanford [1, 2, 15]. We define our views using an extension of OQL [6] and design incremental view maintenance algorithms using insertion, deletion, and update operations. These views are not join-free — updating some objects may affect the presence of other objects in a view. Thus, these views are generally not self-maintainable. Views defined in [1] may specify objects as well as edges to be kept (or discarded) in a view. As a result, maintaining such views is quite complicated, and maintenance expressions over base data are given in [2]. In this paper we use the notation and view definition in [15], where a view contains a set of objects only. Relationships among objects are implicitly kept and can be hidden only by defining multiple views.

Research on relational view self-maintainability can be found in [14, 4, 9, 12, 10]. As we have mentioned, different approaches must be taken for testing self-maintainability of graph-structured views.

2 Data Model, Views and View Maintenance

We use the same data model and view definition as in [15]. Here we give a brief review of related terminology.

2.1 Data Model

The data model we use is OEM [11], which was originally designed for information exchange among heterogeneous data sources, and is very similar to XML. A *database* is a collection of objects. Each *object* contains four fields: an OID, a label, a type, and a value. The OID of an object is a universally unique identifier. The label is a string that explains the meaning of the object and does not need to be unique. Each object either has an *atomic* type, such as integer or string, or has a *set* type, where its value is a set of OIDs of its children objects. The following is a sample database:

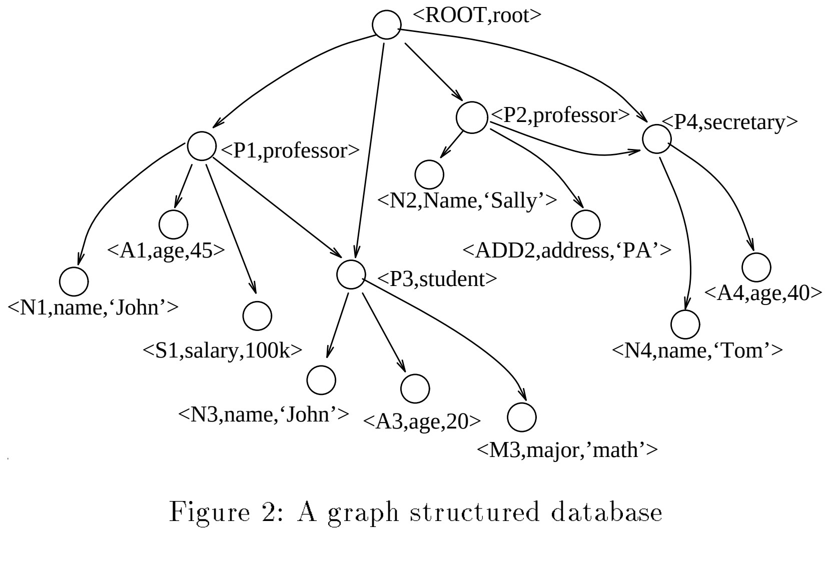


Figure 2: A graph structured database

In this database, object *ROOT* (we refer to an object by its OID) is an example of a *set* object. It has four children objects, representing two professors (i.e., with those labels), one student, and one secretary. Notice that the sub-objects of one professor are structured differently from those of the other professor. Object $A1$ is an example of an atomic object. We use $label(O)$ to refer to the label of object O and $value(O)$ for O's value, for example, $label(P2) = professor$ and $value(P2) = \{N2, ADD2\}$. This data model can be seen as a simple abstraction of XML. It also easily generalizes to other models such as objects with multiple fields or graph model with labels on edges (e.g., [5]).

A *path* is a sequence of zero or more object labels separated by dots. For example, $professor.student$ is a path. We use $N.p$ to denote the set of objects that can be reached following path p from the object with OID N. If $X \in Y.p$, then Y is an *ancestor* of X, and X is a *descendent* of Y. In this case, the first label in p is the label for one of Y's direct children, and the last label that of X. Let $path(Y, X) = \{p | X \in Y.p\}$; it contains the set of paths that can lead from Y to X. For example, in Figure 2, $path(ROOT, P3) = \{student, professor.student\}$. If Y is not an ancestor of X, then $path(Y, X) = \emptyset$. If Y and X are the same object, then $path(Y, X) = \{\lambda\}$. We define $path(Y, X).path(Z, W) = \{p_1.p_2 | p_1 \in path(Y, X), p_2 \in path(Z, W)\}$. For a path p, $\lambda.p = p$.

2.2 Materialized Views and View Maintenance

We study materialized views defined by the following statements (see [15]), in the style of OQL [6] or Lorel [3].

```
define mview MV as:                                              (2.1)
        SELECT    ROOT.sel_path Y
        WHERE     Y.cond_path X, sel_cond(X)
```

A materialized view is defined by a *view definition query*, in which the *sel_path* and *cond_path* are two paths. *ROOT.sel_path* contains a set of objects that can be reached from *ROOT* following *sel_path*. Those objects are bound to variable Y (each object is bound once). For each Y, *Y.cond_path* contains a set of objects reachable following *cond_path*. Those objects are bound to variable X. The Boolean function *sel_cond*() accepts an object and returns $true$ if the objects' value satisfies the condition. Often, the condition is defined on the value of an atomic object. When evaluating the query, *sel_cond*() is tested on all $X \in Y.cond\_path$. If at least one of the Xs satisfies the condition, Y is returned as an answer object of the view definition query.

A materialized view is a stored copy of those answer objects. Each answer object has a *delegate object* in the materialized view, which had the same label, type, and value as its original object.[1] (Thus, a delegate object may contain OIDs of base objects in its value.) Details on how to retain the relationship between a delegate object and its original and on whether to "swizzled" pointers in delegate objects can be found in [15].

We choose to use this simple view definition because it can be used to demonstrate fundamental issues of view self-maintainability. Features such as `FROM` clauses, multiple paths or path expressions in `SELECT` clauses, and multiple conditions in `WHERE` clauses can be added to the view definition language. Those additional features may complicate the testing of some conditions for deciding self-maintainability, as we will address briefly in Section 7.

[1] When context is clear, we sometimes say "object Y is inserted into the view" instead of "the delegate of object Y is inserted into the materialized view".

Corresponding to a base update, a view maintenance algorithm has two tasks: One, deciding whether there are any base objects whose delegates should be inserted into the view or removed from the view due to this update; Two, modifying the value of delegate objects if necessary. The second task, for example, removing a pointer from a delegate object, can always be performed given the corresponding base change. Thus, in the following discussion, we will address the first issue only and refer to it as the view maintenance problem.

Example 1: Defining a materialized view
The following materialized view contains a set of objects that represent professors who have at least one student majoring in math.

```
define mview MATH-PROF as:                                         (2.2)
        SELECT    ROOT.professor Y
        WHERE     Y.student.major = 'math'
```

In the `WHERE` clause of the view definition, the condition `Y.student.major = 'math'` is a shorthand of `Y.student.major X, value(X) = 'math'`. We use this shorthand when the select condition is a simple comparison. Using the base database in Figure 2, the materialized view contains one object, the delegate object of $P1$. □

2.3 Base updates

A *base update* is an update on base objects. It can be an insertion, a deletion, or a value change of an atomic object. Let N,N_1 and N_2 be OIDs.

1. $insert(N_1, N_2)$ adds the OID of N_2 into $value(N_1)$. N_1 must have a set type; object N_2 becomes a child of N_1 after this operation.
2. $delete(N_1, N_2)$ removes OID N_2 from $value(N_1)$, assuming that N_2 was a child object of N_1. (If no objects point to N_2 any more, N_2 may be garbage collected. However, we do not discuss garbage collection here.)
3. $modify(N_2, oldv, newv)$ changes the value of atomic object N_2 from $oldv$ to $newv$.

By default, we assume that when the maintainer receives notification of a base update, it always receives the OIDs of the updated base objects (N_1 and N_2) and the type of the update (insert, delete, or modify). Additional update information such as labels of updated objects may also be sent to the maintainer. We will discuss these issues in Section 5.

3 Self-maintainability and Irrelevance

Self-maintainability means that without the base data, the maintainer has all necessary information to maintain a view. Like relational views, whether a graph-structured view can be self-maintained depends on what information is available to the maintainer. Table 1 shows the possible information that can be used for view maintenance, most of which are discussed separately in the specified sections.

	Relational	Graph Structured
basic information	view definitions	view definitions
	updated tuples	OIDs of updated objects
	type of base update	type of base update
additional information	—	additional update information (Section 5)
	base relation schema	base database information (Section 5)
	base key information [9, 12]	unique label information (Section 5)
	constraints [14, 12]	constraints
	auxiliary views [12]	auxiliary views (Section 6)
	view contents [10]	view contents (Section 6)

Table 1: Information that can be used to maintain a materialized view

The first three categories of information are almost always necessary for view maintenance, the rest are additional ones that may be used to help determine self-maintainability. In practice, whether some information can be used depends on whether they are available to the maintainer and whether it is cost effective to use them. For example, in Section 6 we discuss what auxiliary data can be stored such that a view together with the auxiliary data are always self-maintainable.

In general, with Φ as a subset of information specified in Table 1, we give the following definition:

Definition: An update U is *self-maintainable* with respect to a view V given information Φ if, corresponding to U, V can be maintained (i.e., the new V can be obtained) using information in Φ only. We also say that the view V is *self-maintainable* with respect to update U given Φ. □

If V is self-maintainable with respect to all updates, we simply say V is *self-maintainable.* We want to re-emphasize here that there are two aspects of view self-maintainability. First, there is the question of whether a view can always be self-maintained regardless of the base update. This view property may be decided when the view is defined. Second, with respect to a specific update, there is the issue of whether a view can be self-maintained. This decision must be made at run time when the update is available.

A notion related to self-maintainability is *relevance*, which indicates whether a view is affected by a base update. Intuitively, an update U is irrelevant to a view V if V stays the same when evaluated over the base database before and after update U.

Definition: An update U is *maintenance irrelevant* to a view V given information Φ if it is possible to decide from Φ that V stays the same when evaluated over the base database before and after update U. □

Detecting irrelevant updates may save a lot of unnecessary work to maintain views unaffected by base updates. Obviously, for the same update, view, and Φ, maintenance irrelevance implies self-maintainability, but not vice-versa. Just like deciding self-maintainability, deciding maintenance irrelevance depends on Φ. For example, if the label of an updated object is known, and the label is not used in any of the paths in the definition of view V, then one can determine that this update is irrelevant to V. On the other hand, if only OIDs of updated objects are available, one is not able to tell whether the same update affects the view.

Finally, we want to point out that most of the research on self-maintainability of relational views first gives the set of information used (e.g., updated tuples and keys), then defines self-maintainability as "maintainable." Implicitly, their maintainable means maintainable given this fixed set of information. Therefore, definitions in different previous papers categorize different set of updates as self-maintainable, e.g., maintainable given key information, or maintainable given view content. We believe that the information used to maintain the view (i.e., Φ) should explicitly be given by the definition.

4 Self-maintainability and Irrelevance Test

In this section we develop a self-maintainability test, SMT. This test is performed by the maintainer (recall Figure 1) to determine if an update is maintenance irrelevant, self-maintainable, or not self-maintainable. For SMT, the maintainer must evaluate certain conditions (given below). Some of the conditions refer to the state of the base data. When the maintainer is unable to evaluate those conditions, the update is not self-maintainable. In other cases, the maintainer may know enough (from Φ) to evaluate the condition and determine irrelevance or self-maintainability of the update.

4.1 Testing Maintenance Irrelevance

Figure 3 graphically illustrates what SMT is trying to discover. The view being updated is defined by expression (2.1), and we let $view\_path = sel\_path.cond\_path$. In the figure we assume that the update was an insert or a delete involving objects N_1 and N_2. The figure illustrates scenarios where the view may be affected. In particular, the view may be affected when the updated link between N_1 and N_2 (either added or removed) is on a path from $ROOT$ to one of the objects X that satisfy the select condition of the view definition. Removing the N_1, N_2 link may mean that one of the Y objects that were selected by the view should no longer be in the view. Similarly, adding link between N_1 and N_2 may mean that a new Y should be added. Note that the candidate Y object may follow N_1, N_2 on the path from $ROOT$ to X (Scenario 1 in the figure), or may precede N_1, N_2 (Scenario 2). The situation for modify is similar to Scenario 2, where Y can only be on the path from $ROOT$ to N_2.

Our first SMT condition checks if there is a path from $ROOT$ to the updated object(s) that is relevant to the view.

Condition (a) In the base database, right after the update completed,

- For an insert or a delete: There exists a path $p \in path(ROOT, N_1)$ such that $p.label(N_2)$ is a prefix of $view\_path$.
- For a modify: There exists $p \in path(ROOT, N_2)$ such that p is a prefix of $view\_path$. $\square$

In a graph-structured database, there may be more than one path from $ROOT$ to N_1. Any one of those paths may make Condition (a) true. If the maintainer can ascertain that Condition (a) was false at the base database, then it knows that the update is irrelevant to the view.

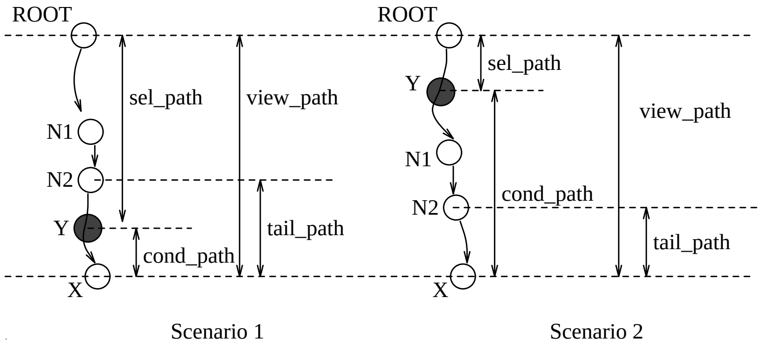


Figure 3: Updated base objects

If the maintainer knows that the condition was true, the maintainer can check for the rest of the path, from N_2 to object X (see Figure 3). To do so, for each $p \in path(ROOT, N_1)$ we define a *tail_path* to be the portion of the *view_path* that was not traversed to reach N_2. That is, for each $p \in path(ROOT, N_1)$ that makes Condition (a) true, *view_path* should equal $p.label(N_2).tail\_path$. If $view\_path = p.label(N_2)$, then let $tail\_path = \lambda$. (For a modify update, *view_path* should equal $p.tail\_path$). If there are more than one *p*s, there are more than one *tail_path*s. Let T be the set of all *tail_path*s. For example, if $view\_path = a.b.a$, and $label(N_2) = a$, then if *ROOT* can reach N_2 through both path a and path $a.b.a$, we have $tail\_path = b.a$ or $tail\_path = \lambda$. The second SMT condition checks whether the select condition is satisfied by some descendents of N_2.

Condition (b) In the base database, right after the update completed,

- For an insert or delete: there exists $tail\_path \in T$ and an object X such that $X \in N_2.tail\_path$ and $sel\_cond(X) = true$.
- For a modify, there exist a $tail\_path \in T$ such that $tail\_path = \lambda$ and $sel\_cond(oldv) \neq sel\_cond(newv)$. □

If the maintainer knows that Condition (b) is false, then the update is again irrelevant to the view. If both Conditions (a) and (b) are true, then the update potentially affects the view and the maintainer needs to evaluate the SMT Conditions (c) and (d).

Sometimes the maintainer is unable to evaluate Conditions (a) or (b) from its information in Φ. In this case, we say that the outcome of the condition evaluation is *unavailable*. For example, when the value of X is unknown, the maintainer cannot evaluate $sel\_cond(X)$. If the maintainer cannot evaluate Condition (b), then the view is not self-maintainable. If the maintainer cannot evaluate Condition (a), then it is impossible to obtain T. However, the maintainer can still evaluate Condition (b) for each potential *tail_path* that could be in T. For example, if $view\_path = professor.student.major$, then there are only three potential *tail_path*s: $major$, $student.major$, or $professor.student.major$. If Condition (b) is false for all potential *tail_path*s, then the update is maintenance irrelevant to the view. However, if Condition (b) is true or unavailable for any of the *tail_path*s, the update is not self-maintainable.

Table 2 summarizes the decisions after evaluating Conditions (a) and (b). Remember that maintenance irrelevance implies self-maintainability.

Condition (a)	Condition (b)	Decision
False	Any value	Maintenance irrelevant
True	False	Maintenance irrelevant
True	Unavailable	Not self-maintainable
True	True	Evaluate Conditions (c) and (d)
Unavailable	(all) False	Maintenance irrelevant
Unavailable	(any) Unavailable	Not self-maintainable
Unavailable	(any) True	Not self-maintainable

Table 2: Summary of decisions on Conditions (a) and (b)

The following example shows how the maintainer can use SMT Conditions (a) and (b) to determine whether an update is irrelevant to a view. It also shows that the maintainer may make different decisions when different information is available.

Example 2: Testing Maintenance Irrelevance
Consider a simple view `VSMITH` defined as `SELECT ROOT.professor Y, WHERE Y.name = 'smith'`. Consider a base update $delete(ROOT, Z)$. Assume the maintainer receives label information for updated objects and $label(Z) = professor$. For view `VSMITH`, $view\_path = professor.name$. Since $path(ROOT, ROOT) = \lambda$ and $label(Z)$ is a prefix of $view\_path$, Condition (a) is true and we have $T = \{name\}$. If the maintainer has no information about Z's sub-objects, then the maintainer cannot decide whether Condition (b) is true or false, and the update is not self-maintainable. On the other hand, if the maintainer knows that object Z has no sub-object with label $name$, then $Z.name = \emptyset$, Condition (b) is false, and the update is maintenance irrelevant to the view according to Table 2. □

4.2 Testing Self-maintainability

When both Conditions (a) and (b) are true, the update is relevant to the view, meaning that potentially there are Y objects (Figure 3) to be inserted into or deleted from the view. Condition (c) checks whether the maintainer can obtain all those Y objects from Φ.

Condition (c) We define **Y** as the set of objects that are potentially affected by the update:

- For an insert or a delete: Let $S = \{X | X \in N_2.tail\_path$, where $tail\_path \in T$ and $sel\_cond(X)\}$. S refers to the base database right after the update completed.

  For all $X \in S$,

  let $\mathbf{Y1} = \{Y\ | sel\_path \in path(ROOT, N_1).label(N_2).path(N_2, Y)$ and $cond\_path \in path(Y, X)\}$,

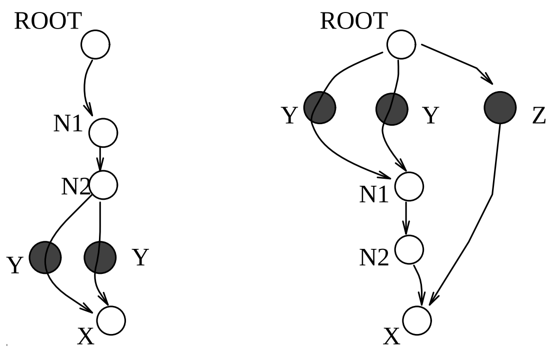
  let $\mathbf{Y2} = \{Y | sel\_path \in path(ROOT, Y)$ and $cond\_path \in path(Y, N_1).label(N_2).path(N_2, X)\}$.

  Let $\mathbf{Y} = \mathbf{Y1} \cup \mathbf{Y2}$.

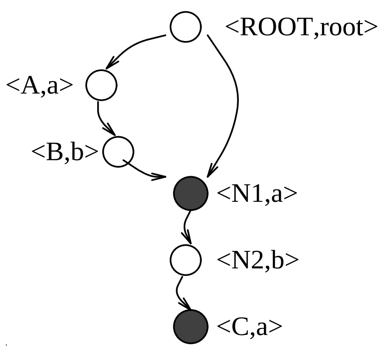
- For a modify: Let $\mathbf{Y} = \{ Y | sel_path \in path(ROOT, Y)$and $cond\_path \in path(Y, N_2)\}$.

Condition (c) is true if for an insert, OIDs, labels and values of all objects in $\mathbf{Y}$ can be obtained from Φ; for a delete, OIDs of all objects in $\mathbf{Y}$ can be obtained from Φ. (A modification is treated as an insert if not $sel\_cond(oldv)$ but $sel\_cond(newv)$; it is treated as a delete if $sel\_cond(oldv)$ but not $sel\_cond(newv)$.) □

In Condition (c), **Y1** contains Ys in Scenario 1 and **Y2** contains Ys in Scenario 2. Here we want to stress two issues related to the evaluation of Condition (c). First, there may be more than one Y object corresponding to one X, as shown in Figure 4(a), and the maintainer must find all of them. The maintainer has to be especially careful with objects like Z in Figure 4.2, which also satisfies $sel\_path \in path(ROOT, Z)$ and $cond\_path \in path(Z, X)$ but whose existence in the view is not affected by the current update.



(a) More than one Y object corresponding to one X



(b) Both Scenario 1 and 2 occur for the same view and update

Figure 4: Possible object layouts when evaluating Condition (c)

Second, when there are multiple $tail_path$s, the maintainer has to find all corresponding Ys to ascertain that (c) is true. Sometimes for the same view and the same update, there are Ys in both Scenario 1 and 2, as shown in Example 3.

Example 3: Multiple potential view objects corresponding to one source update
Let the view be defined as `SELECT ROOT.a.b.a Y, WHERE Y.b = 'value'`. In the base database shown in Figure 4(b), both object N_1 and C are potential Ys. In particular, since both $a \in path(ROOT, N_1)$ and $a.b.a \in path(ROOT, N_1)$, we have $T = \{a.b, \lambda\}$ and $\mathbf{Y} = \{C, N_1\}$. C is a potential Y object in Scenario 1 and N_1 is a potential Y object in Scenario 2. □

If the maintainer cannot obtain any Y object specified in Condition (c), then (c) is false and this view cannot be maintained. If (c) is true and the base update is an insertion, then the view can be maintained by inserting the delegate objects of Ys into the view. If the base update is a delete, however, we are not done yet. For a delete, the maintainer must know whether Y's delegate object should stay in the view after the deletion. Object Y may stay in the view because it can be reached from $ROOT$ via another path (for Scenario 1), or because it has more than one "derivation" due to other descendent objects (for Scenario 2). For instance, for the view `MATH-PROF` in Example 1, if a professor object has two students majoring in math and one of them graduates, this professor object should remain in the view. But if the professor has only one math student object who was just removed, the professor object should be removed from the view too. Remember that for modify, when $sel\_cond(oldv)$ but not $sel\_cond(newv)$, it is treated as a delete in Scenario 2.

Let **Y, Y1,Y2** be the collections of affected view objects defined in Condition (c). We first define a Condition (d') that indicates whether a single object $Y \in \mathbf{Y}$ should stay in the view after a delete.

Condition (d') In the base database, right after the delete, for $Y \in \mathbf{Y1}$, it is still true that $sel\_path \in path(ROOT, Y)$; for $Y \in \mathbf{Y2}$, there still exists $X \in Y.cond\_path$ and $sel\_cond(X) = true$. □

For a $Y \in \mathbf{Y}$, if the maintainer can decide that (d') was true, then Y stays in the view. If (d') was false, Y should be removed from the view. If the maintainer cannot make a decision on (d'), then it does not know whether Y should stay in the view, and hence the update is not self-maintainable. The last SMT Condition (d) should be evaluated for all deletes.

Condition (d) For all $Y \in \mathbf{Y}$, Condition (d') can be evaluated. □

If Condition (d) is true, i.e., Condition (d') is either true or false for all $Y \in \mathbf{Y}$, then the maintainer can decide whether a Y should stay in the view for all $Y \in \mathbf{Y}$. Therefore this update is self-maintainable. If Condition (d) is false, then there is at least one $Y' \in \mathbf{Y}$ such that Condition (d') is unavailable for Y', and the update is not self-maintainable. Table 3 summarizes the decisions after evaluating Conditions (c) and (d):

Condition (c)	Condition (d)	Decision
False	Any value	Not self-maintainable
True (insertions)	Any value	Self-maintainable
True (deletions)	True	Self-maintainable
	False	Not self-maintainable

Table 3: Summary of decisions on Conditions (c) and (d)

4.3 Discussion

The SMT we just developed is instance specific — it makes a decision for a given update and a view. It can also be used to prove the self-maintainability of a view (we give an example in [16]) or a type of view (e.g., complete views in Section 6), by proving that all updates will pass or fail specific tests.

The SMT is complete and sound. That is, given Φ, the maintainer can use SMT to detect all self-maintainable updates, and any self-maintainable update will be detected. Due to space limitation, we do not formally prove the completeness and soundness in this paper. Intuitively, the evaluation of the four conditions cover the process of maintaining a view given an update. If update

U is categorized as irrelevant by the SMT, then the maintainer needs to do nothing to maintain the view. If U is categorized as self-maintainable, Condition (c) must be true, so the set $\mathbf{Y}$ can be obtained and the view can be maintained. If U is categorized as not self-maintainable, there must be a decision or piece of data that is essential to the view maintenance and is missing from Φ.

5 Performing the Self-maintainability and Irrelevance Test

Relational self-maintainability tests are specified by standard relational operators (e.g., [14]) or query containment rules (e.g., [10]), thus there is no need to discuss how to perform the tests. In contrast, performing SMT may involve, for example, checking whether there is a path from one object to another in the base database. There are no standard methods to test such a condition.

In general, the more knowledge the maintainer has about the updated objects or the base data, the more updates can be self-maintained. In this section we show how additional information can be used to either evaluate or simplify the evaluation of the SMT conditions. Remember that by default the maintainer only has the OIDs of the updated objects. In the following we first consider options on additional update information provided to the maintainer as part of the update notification, then briefly address the use of additional knowledge on base databases.

5.1 Label and Value Information

It is often relatively easy to send the labels and values of the updated base objects to the maintainer. Example 4 shows how SMT Condition (a) can be evaluated given label information.

Example 4: Using update label information to determine irrelevance
Say we have a view `SELECT ROOT.professor.student` and an update $U = insert(ROOT, Z)$. If the maintainer has the label of Z and $label(Z) = secretary$, then $label(Z)$ is not part of $view\_path$, hence, Condition (a) is false and U is irrelevant to the given view. □

We develop ad hoc rules called *SM rules* for the maintainer to perform SMT automatically. The following Rule (a1) applies to the case of Example 4.

Rule (a1) Using label information, if $label(N_1).label(N_2)$ is not a sub-path of $view\_path$, then Condition (a) is false. (Hence, the update is maintenance irrelevant to the view.) ◇

An SM rule is labeled as Rule (kn) if it is the nth rule and is related to the evaluation of Condition (k). We can see that an SM rule specifies some qualifications on Φ; if satisfied, the maintainer can directly obtain the result of an SMT condition. Providing SM rules is one way to test SMT conditions for a subset of updates or views.

5.2 Descendent Information

The maintainer may be provided with the OIDs, labels, and values of N_1 and N_2, plus the OIDs, labels and values of all N_2's descendent objects. In many scenarios, descendent information is essential to evaluate Conditions (b) to (d) and ensure that an update is self-maintainable.

Locating all descendent objects of N_2 may be difficult in a graph-structured base database. Fortunately, only a limited number of layers of N_2's descendents, not all, are necessary to evaluate Conditions (b) to (d). This number is equal to the size of the largest *tail_path* in T. Let $L = maximum(|t|)$ where $t \in T$. Because *tail_paths* are only known after evaluating Condition (a) and are update specific, the maintainer may set the required levels of descendent objects to $|view\_path|$, an upper bound of L. We use $descendent(N_2, L)$ to represent the first L levels of N_2's descendent objects (their OIDs, labels and values).

Here are a few example SM rules on how descendent information can be used. Since evaluating Condition (b) uses objects in $N_2.tail\_path$, we have

Rule (b2) Objects in $descendent(N_2, L)$ are sufficient to evaluate Condition (b). ◇

After evaluating Condition (a) the maintainer can decide whether each $Y \in \mathbf{Y}$ is on the path from N_2 to X, or on the path from *ROOT* to N_1, by comparing the corresponding *tail_path* with the *cond_path*. When Y is on the path from N_2 to X (Scenario 1 in Figure 3), if all descendents of N_2 are given to the maintainer, then Y can be obtained because Y itself is a descendent of N_2.

Rule (c3) If all $Y \in \mathbf{Y}$ are in Scenario 1, then if $descendent(N_2, L)$ is known, Condition (c) is true. ◇

Also, because all Y's descendent objects are descendent objects of N_2 as well, we have the following rule:

Rule (d4) If Y is in Scenario 1, then if $descendent(N_2, L)$ is known, the maintainer can decide whether Condition (d') is true or false and thus, Condition (d) is true. ◇

5.3 Root-path Information

In this case, the maintainer is provided with the OIDs and labels of N_1 and N_2, plus $path(ROOT, N_1)$. For a tree-structured base database, it is usually simple to send root-path information to the maintainer, since the base update transaction probably traversed the base database until it reached the updated objects. Therefore, it may record the object labels along the path and report them as part of the update. However, for graph-structured data there may be more than one path from *ROOT* to N_1 and all of them need to be reported to the maintainer. SM rule (a5) tells us that Condition (a) can be evaluated using root-path information:

Rule (a5) If $path(ROOT, N_1)$ is given as part of the update information, Condition (a) can be evaluated by path comparison. ◇

5.4 Restricted Base Updates

Knowledge of base update restrictions can also be used to simplify the evaluation of SMT conditions. The following are a few example scenarios:

- The base database only performs updates of a certain type. For example, many databases are append only, and for those databases, Condition (d) does not need to be evaluated.

- The base database only updates atomic objects. Remember that a set object with an empty value (i.e., no children) is also considered an atomic object. In a tree structured database, an insert or a delete of a subtree can be represented as a sequence of inserts or deletes of atomic (leaf) objects. One example of only deleting atomic objects is the UNIX delete file command `'rm -r dir'`, which removes each file in a directory first, then removes the empty directory.

  Rule (b6) When N_2 is atomic, if $\lambda \notin T$, Condition (b) is false; if $\lambda \in T$ and $value(N_2)$ is known to the maintainer, the maintainer can test Condition (b) by evaluating the select condition on the value of N_2. $\diamond$

  Since it is often easy to test the select condition on a given object, the maintainer may evaluate Condition (b) before (a), and only evaluate (a) when the result of (b) is true.

- The base database only updates objects with a certain label. For example, say a materialized view reflects stock data for a given portfolio. For this view, only changes of stock quotes are propagated incrementally into the view, and other changes (e.g., to the portfolios) can be reflected by a periodic refresh every evening. Then, the maintainer only needs to evaluate updates with label *quote* against SMT. In this scenario, for example, if this label is not part of the *view_path* of a view, then according to Rule (a1), the all updates are maintenance irrelevant.

5.5 Knowledge of Base Database

Schema information (e.g., DTDs of XML), unique labels and other constraints on the base database are all optional for a graph-structured database. When they are available, they can be used to simplify the evaluation of SMT conditions. For example, SMT conditions can be simplified when the underlying base database has tree structure.

Due to space limitation, we only give one example here on how knowledge of base database can be used to facilitate SMT evaluation. More detailed discussion can be found in [16]. In this example, we represent the additional knowledge on base data using *neighboring-label information (NLI)*. NLIs are expressions defined to represent the local structural information of a GSDB. They tell us what patterns of labels may appear together and which do not. The expressions may be defined using keywords like `ALWAYS`, `NEVER`, and `UNIQUE`.

Example 5: Using NLI to evaluate SMT conditions

Let the view definition be `SELECT ROOT.professor.student` and assume object *ROOT* has unique label *root*. Let the update be $U = insert(P, S)$. Assume the maintainer receives the labels of updated objects: $label(P) = professor$ and $label(S) = student$.

Let the base database information be given in an expression `ALWAYS(`*root*`).`*professor*, meaning that all professor objects are first-level children of *ROOT*. According to this information, the maintainer knows that Condition (a) is true because P is a child of *ROOT*, and $label(P).label(S)$ is a prefix of the *view_path*. The maintainer obtains the corresponding $tail\_path = \lambda$, and Condition

(b) is true since there exists an element of $S.tail\_path$, namely S itself, that satisfies the select condition (which is always true in this example). Condition (c) is also true because $Y = S$ and $\mathbf{Y}$= {S} is available to the maintainer. Thus, update U is self-maintainable. However, if the maintainer does not know `ALWAYS(`$root$`)`$.professor$ to hold, then P can appear without being a first-level child of $ROOT$, and the same update may or may not affect the view. □

We conclude this section by pointing out that update or base database information can be combined to perform more powerful test, as we did in Example 5. More sophisticated SM rules can also be developed for combined information.

6 Complete Views and Auxiliary Data

In this section we first describe one type of view, a *complete view*, that is always self-maintainable. Then using complete views, we show how to keep auxiliary data in order to maintain an arbitrary view without accessing base data.

6.1 Complete Views

We first give the definition of complete views, then discuss some of their properties.

Definition: For a given base database, given object O and path p, define a *complete view* $CV(O, p)$ as a view that contains all objects that can be reached from O using a prefix of p. We abbreviate $CV(ROOT, p)$ as $CV(p)$. □

Example 6: A complete view
The complete view $CM = CV(professor.student.major)$ contains all objects that are reachable from $ROOT$ through path $professor.student.major$. Figure 5 shows the base database given in Section 2 and the view CM over this database. The view CV contains (the delegate of the) five objects shown in the shaded area. Because λ is a prefix of $view\_path$, $ROOT$ itself is in the complete view. □

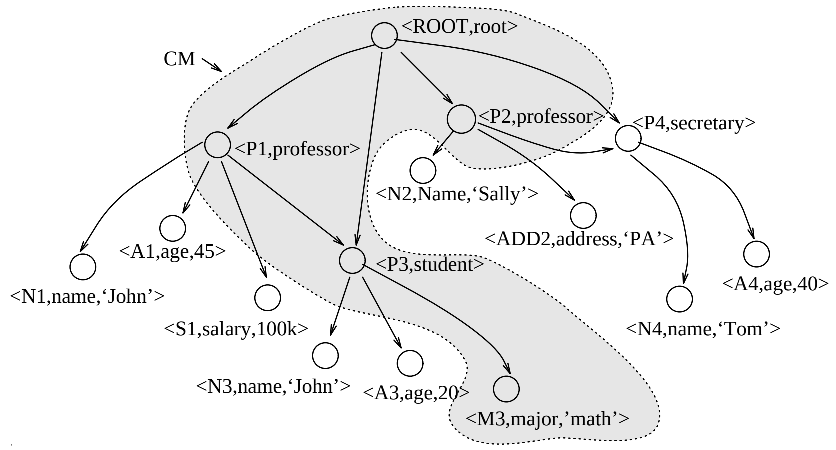


Figure 5: Complete view `CM`

Notice that a complete view cannot be defined by expression (2.1). However, a complete view can be seen as the union of views defined by expression (2.1). For example, view CM in Example 6 can be defined by four `SELECT` statements.

We use D_{-U} to denote the database D before an update U occurs, and D_{+U} to denote the same database right after U occurs. We use $V[D_{-U}]$ and $V[D_{+U}]$ to denote the extensions of view V over database D at the corresponding state. For a given update, let T be the set of *tail_path*s defined in Section 4.1. The following theorem states the maintenance properties of complete views.

Theorem 6.1 *For a complete view $C = CV(view\_path)$ and an update U,*

- *If U is a modify, then U is maintenance irrelevant to C.*
- *If U is a delete, then U is self-maintainable to C given $C[D_{-U}]$, the view content before the update.*
- *If U is $insert(N_1, N_2)$, then U is self-maintainable with respect to C given $C[D_{-U}]$ and $CV(N_2, tail\_path)[D_{+U}]$ for all $tail\_path \in T$.*

The correctness proof of Theorem 6.1 is given in [16]. Intuitively, a complete view C contains all objects that are reachable from $ROOT$ via certain paths. A modify does not change an object's reachability from $ROOT$, and thus does not change C. After a base deletion, N_2 and some of N_2's descendent objects may need to be removed from C. However, which objects should be removed can be decided from $C[D_{-U}]$. For insert, $C[D_{-U}] \cup CV(N_2, tail\_path)[D_{+U}]$ (for all $tail\_path \in T$) is $C[D_{+U}]$, which is the new view content after the update [16]. For all three cases, an update is maintainable for view C.

The purpose of studying complete views is that they can be stored as auxiliary data to maintain other views. The following theorem states the relationship between a complete view and the self-maintainability of a view with the same *view_path*, assuming the maintainer has the basic information in Table 1.

Theorem 6.2 *For a view V defined by expression (2.1) and an update U,*

- *If U is a modify or a delete, $CV(view\_path)[D_{-U}]$ contains all objects sufficient and necessary to maintain view V with respect to U.*
- *If U is an insert, $CV(view\_path)[D_{+U}]$ contains all objects sufficient and necessary to maintain view V with respect to U.*

The proof of correctness of Theorem 6.2 is given in [16]. Theorem 6.2 can be used by the maintainer to decide what auxiliary information to store in order to make a view self-maintainable. We discuss this issue next.

6.2 Keeping Auxiliary Data to Make Views Self-maintainable

The bottom line is that to maintain a view V defined by (2.1) with $view\_path = sel\_path.cond\_path$, the maintainer may need all the objects in $C[D_{-U}]$ and $C[D_{+U}]$ where $C = CV(view\_path)$. These objects can be found in the existing view, sent along with update information, or stored as auxiliary views. (Of course, an object in both $C[D_{-U}]$ and $C[D_{+U}]$ should preferably be stored or sent only once.) In the following we discuss two options on what to store and what to send. Notice that the auxiliary structures themselves (together with the view and update) must be self-maintainable.

1. The maintainer stores all objects that may potentially appear in the view or on a path that will be examined during the evaluation of SMT conditions. (Objects that are already in the current view need not be stored again.) All objects in $C[D_{-U}]$ are such objects. Actually, any object with a label that appears in $view\_path$ may potentially be examined. So for this method, if the base database is B, then the auxiliary view AV is the union of all such view defined by "`SELECT B.l Y`", where l is a label appearing in $view\_path$. (A database, like a view, is also an object that can be used in a query [15].) With respect to the three types of base updates given in Section 2.3, AV is itself self-maintainable. (Actually, AV is maintenance irrelevant.)

   For this method, the basic information in Table 1 is enough to maintain view V. But this method may result in keeping more information than necessary. For example, for a view `NAMES` defined as `SELECT ROOT.professor.name`, the maintainer needs to store all professor and name objects (even if they are not currently reachable from $ROOT$.) There is no need for the maintainer to keep a secretary object, for example, but it needs to keep the names of the secretaries. Although it does not make sense semantically, one could potentially insert a link from a professor object to the name sub-object of a secretary.

2. The maintainer stores $C[D_{-U}]$, that is, the content of the view C before the update. This information is sufficient to maintain view V with respect to delete and modify. For inserts, the maintainer also needs $CV(N_2, tail\_path)[D_{+U}]$ for all $tail\_path \in T$ to maintain V. In practice, the base site may simply send $descendent(N_2, L)$ (L is the maximum length of all $tail_path$s) or $descendent(N_2, \infty)$ (all descendents of N_2), supersets of necessary information.

   For example, to maintain the view `NAMES` above, name objects that are only reachable from secretary objects do not need to be stored. If they were to appear in the view, they can be obtained from update information.

   From Theorem 6.1, the auxiliary view $C[D_{-U}]$ is also self-maintainable given the update information. The current view content, $V[D_{-U}]$, contains a subset of objects in $C[D_{-U}]$. If the maintainer is allowed to access the current view content, it may not need to store those objects again as auxiliary data.

Of course, the maintainer may choose to store no auxiliary data and let the base site report all necessary information with updates. The cost of storing and maintaining the auxiliary information versus the cost and feasibility of querying base data determines which approach to use. Heuristics can be used to determine this relative cost. For example, if the expected size of $CV(view\_path)$ is substantially smaller than the size of the base database, it may be beneficial for the maintainer to store $CV(view\_path)$.

7 Conclusion

As semistructured and XML data become commonplace in database systems, it becomes important to understand how graph-structured materialized views can be efficiently maintained. In this paper we have studied this maintenance problem, and in particular, self-maintainability of graph-structured views. To our knowledge, this is the first research effort on this topic. We presented a self-maintainability test that decides the relevance of updates, and we showed how the test can be performed. We proved that complete views are self-maintainable, and we discussed how to keep auxiliary data to help maintain a view.

Techniques described in this chapter can be used to facilitate view maintenance, as shown in Figure 6. When a view is defined, the maintainer can decide whether it is self-maintainable. If not, the maintainer may decide whether to store auxiliary data or acquire more information about the base database to make the view self-maintainable. At run time, self-maintainable views can be maintained without accessing base data. For other views, the maintainer can decide whether a given update is irrelevant or maintainable. SMT (or a set of SM rules) is used in all three decision stages. If an update is not self-maintainable, then an algorithm accessing base data (e.g., one described in [15]) must be used.

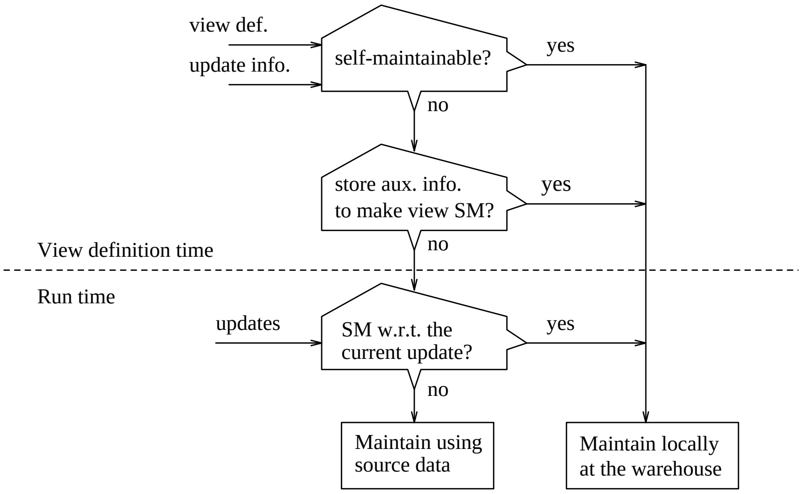


Figure 6: Using SMT to maintain materialized views

We assumed that in the view definition expression (2.1), both the sel_path and the $cond\_path$ are constant paths. In fact, when they are path expressions containing wild cards (e.g., $*$, $professor.*$

and $professor.?$), the four SMT conditions described in Section 4 can still be used to determine the self-maintainability of the view. However, when evaluating the test conditions, all possible instances of a path expression must be considered. In particular, computing $tail_path$s becomes more complicated. For example, for a view defined as `SELECT ROOT.* Y WHERE Y.major = 'math'` and a base update that modified an object with label $major$, we have $T = \{*, *.major\}$. In general, computing T requires path containment techniques, which we do not discuss further in this paper. Our definition and theorems for complete views also apply directly to views defined using path expressions.

References

[1] S. Abiteboul, R. Goldman, J. McHugh, V. Vassalos, and Y. Zhuge. Views for semistructured data. In *Workshop on Management of Semistructured Data*, pages 83–90, May 1997.

[2] S. Abiteboul, J. McHugh, , M. Rys, V. Vassalos, and J. Wiener. Incremental maintenance for materialized views over semistructured data. In *Proceedings of the 24th VLDB*, pages 38–49, August 1998.

[3] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, April 1997.

[4] J.A. Blakeley, N. Coburn, and P.-A. Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. *ACM Transactions on Database Systems*, 14(3):369–400, September 1989.

[5] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *SIGMOD*, pages 505–516, June 1996.

[6] R.G.G. Cattell, editor. *The Object Database Standard: ODMG-93*. Morgan Kaufmann, 1994.

[7] World Wide Web Consortium. Extensible Markup Language (XML). http://www.w3c.org/xml, 1998.

[8] Microsoft Corporation. XML demos. Available from http://www.microsoft.com/xml, 1998.

[9] A. Gupta, H. Jagadish, and I. Mumick. Data integration using self-maintainable views. In *EDBT*, pages 140–144, March 1996.

[10] N. Huyn. Efficient view self-maintenance. In *Workshop on Materialized Views*, pages 17–25, June 1996.

[11] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *Proceedings of the 11th ICDE*, pages 251–260, March 1995.

[12] D. Quass, A. Gupta, I. Mumick, and J. Widom. Making views self-maintainable for data warehousing. In *Proceedings of the 4th PDIS*, pages 158–169, December 1996.

[13] D. Suciu. Query decomposition and view maintenance for query language for unstructured data. In *Proceedings of the 22nd VLDB*, pages 227–238, September 1996.

[14] F.W. Tompa and J.A. Blakeley. Maintaining materialized views without accessing base data. *Information Systems*, 13(4):393–406, November 1988.

[15] Y. Zhuge and H. Garcia-Molina. Graph structured views and their incremental maintenance. In *Proceedings of the 14th ICDE*, pages 116–125, February 1998.

[16] Y. Zhuge and H. Garcia-Molina. Self-maintainability of graph-structured views. Technical report, Stanford University, March 1999. http://www-db.stanford.edu/pub/papers/zgselft.ps.