VLDB '98 submission #653

Incremental Maintenance for Materialized Views over Semistructured Data*

Serge Abiteboul† Jason McHugh‡ Michael Rys‡ Vasilis Vassalos‡ Janet L. Wiener‡

† INRIA-Rocquencourt
F-78153 Le Chesnay, France
E-mail: Serge.Abiteboul@inria.fr

‡ Stanford University
Stanford, CA 94305, USA
E-mail: *Firstname.Lastname*@cs.stanford.edu

Abstract

Semistructured data is not strictly typed like relational or object-oriented data and may be irregular or incomplete. It often arises in practice, e.g., when heterogeneous data sources are integrated or data is taken from the World Wide Web. Views over semistructured data can be used to filter the data and to restructure (or provide structure to) it. To achieve fast query response time, these views are often materialized. This paper studies incremental maintenance techniques for materialized views over semistructured data. We use the graph-based data model OEM and the query language Lorel, developed at Stanford, as the framework for our work. We propose a new algorithm that produces a set of queries that compute the changes to the view based upon a change to the source. We develop an analytic cost model and compare the cost of executing our incremental maintenance algorithm to that of recomputing the view. We show that for nearly all types of database updates, it is more efficient to apply our incremental maintenance algorithm to the view than to recompute the view from the database, even when there are thousands of such updates.

1 Introduction

Database views increase the flexibility of a database system by adapting the data to user or application needs [17, 36, 46]. Views are frequently materialized to speed up querying when the underlying data is remote, e.g., distributed, or query response time is critical [28, 8]. Once a view is materialized, however, its contents must be maintained in order to preserve its consistency with the base data. Maintenance can be achieved either by recomputing the view contents from the database or by computing the incremental changes to the view based on changes to the database. In this paper, we study the maintenance of materialized views for a new category of data, semistructured data. We propose a simple view specification mechanism and an algorithm for incremental maintenance. We then demonstrate the algorithm's usefulness (and limitations) with an analysis of the maintenance cost.

Unlike relational or object-oriented data, semistructured data need not conform to a fixed schema. The data may be irregular or incomplete, and often arises in practice, e.g., when heterogeneous data sources are integrated or data is extracted from the World Wide Web [1, 33, 9].

*Research partially supported by NSF grant IRI–96–31952, ARO grant DAAH04–95–1–0192, Air Force contract F33615–93–1–1339, the Swiss National Science Foundation, and the Lilian Voudouri Foundation.

Views over semistructured data can be used to filter the data and to restructure (or provide structure to) it [33]. Filtering is crucial since semistructured data is often encountered by applications interested in a very small portion of the available data (e.g., some specific data from the Web). Furthermore, semistructured data is often heterogeneous and outside our control, so a view is the only way in which we can restructure it. By introducing structure to semistructured data, views can also facilitate query optimization or query formulation.

For performance reasons, views over semistructured data often need to be materialized. Queries over semistructured data (possibly traversing long paths) are expensive to evaluate, as Mike Carey argued recently [12]. A materialized view can be used to isolate the data of interest, allowing subsequent queries to run over a smaller, perhaps more structured, data set. Materialized views can also be used to rewrite queries applied over the base data to improve the query performance [35]. Furthermore, subsequent queries over the materialized view may be able to take advantage of standard query optimization techniques and access methods for structured data, even though the underlying base data of the view is semistructured.

View mechanisms and algorithms for materialized view maintenance have been studied extensively in the context of the relational model [8, 24, 23, 37, 22]. Incremental maintenance has been shown to dramatically improve performance for relational views [25]. Views are much richer in the object world [2] and, subsequently, algorithms for querying materialized views are significantly more intricate [2, 6, 44, 43, 39].

Previous results on incremental view maintenance for object databases [40, 42, 41] and nested data [26] are based on the extensive use of type information. Semistructured data provides no type information, so the same techniques do not apply. In particular, subobject sharing and the absence of a schema make it difficult to detect if a particular update affects a view. Subobject sharing also makes it impossible to apply the approach taken by Croque [20], where the (OQL) view definitions are limited to linear functions to avoid accessing the database when constructing the incremental view maintenance statements.

While Suciu [45] also considers incremental view maintenance for semistructured data, the view specification language is limited to select-project queries and only considers database insertions. Our approach allows joins in the view query and handles database insertions, deletions, and updates. Zhuge and Garcia-Molina [47] also investigate graph structured views and their incremental maintenance. However, their views consist of object collections only, while we include edges (structure) between objects. Also, their maintenance algorithms only work for select-project view specifications over tree-structured databases, while our approach handles joins and arbitrary graph-structured databases.

Our work is based on the Object Exchange Model (OEM) [34] for semistructured data and on the Lorel query language for OEM [4]. We propose a view specification extension to Lorel that introduces two sets of objects in the view: (1) the *select-from-where* part specifies the *primary* objects imported to the view and (2) the new *with* part specifies paths from the primary objects to *adjunct* objects. Both the paths and the adjunct objects appear in the view. We exploit this distinction between the different view objects to propose an algorithm to maintain the materialized

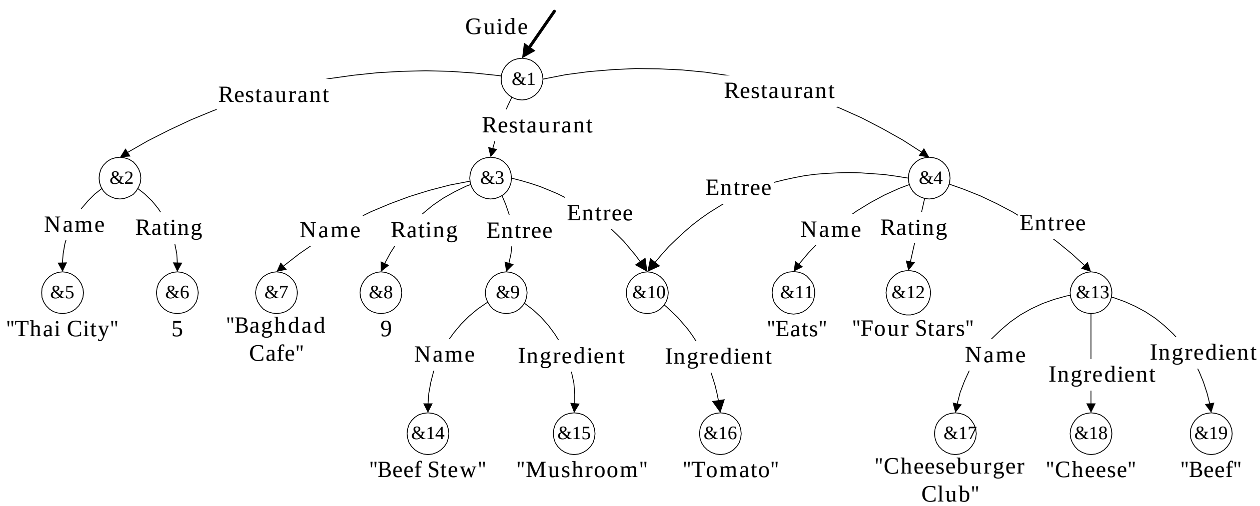

Figure 1: A Simple OEM Database

views incrementally. Given a view and a database update, the algorithm produces a set of maintenance statements, evaluates them on the database to yield a set of view updates, and installs the updates in the view.

We demonstrate the advantages of our algorithm with a cost model and a performance evaluation. We compare the cost of recomputation to the cost of incrementally computing the new view. Our results show that the incremental maintenance algorithm is several orders of magnitude better than recomputing the view for insertion and deletion of edges between objects. In addition, incremental maintenance is cheaper for small numbers of atomic value changes. However, in some cases, such as when a substantial portion of the database is updated, it may be cost effective to recompute the view. We implemented view specifications in Lore [29] following our early work [3], which then motivated the present work on incremental maintenance. Since the performance evaluation is (as we shall see) quite promising, we next intend to implement the algorithms described here.

The presented maintenance algorithm can be used both for immediate maintenance [8] and for deferred maintenance [38, 16] of the views. The general ideas presented here are also applicable to query languages for semistructured data [11], for the Web [27, 31], and (to some extent) to query languages for hypertext documents [14, 5].

2 View Specification

We use the Lore system [29] to investigate materialized view maintenance over semistructured data. Lore implements OEM. We now introduce OEM, the Lorel query language, the view specification language, and the update operations. [4] and [3] provide further details on Lorel and the view specification language, respectively.

2.1 Data Model OEM

An OEM database is a labeled, directed graph such as the small example database given in Figure 1. The vertices in the graph are *objects*; each object has a unique *object identifier* (oid) such as &2. *Atomic objects* contain a value from one of the atomic types, e.g., `integer`, `real`, `string`, `gif`, `java`, `audio`. All other objects are *complex objects* and have a set of $\langle label, subobjects\rangle$ pairs as their value. In Figure 1, object &5 is atomic and has the value "Thai City". Object &4 is complex and has as its value $\{\langle Entree, \&10\rangle, \langle Name, \&11\rangle, \langle Rating, \&12\rangle, \langle Entree, \&13\rangle\}$. *Names* are special labels that each serve as an alias for a single object, and are used as entry points into the database. In Figure 1, *Guide* is a name that denotes object &1.

There is no notion of a schema in an OEM database. Semantic information is included in the labels, which are part of the data and can change dynamically. In this respect, an OEM database is *self-describing*. OEM has been designed to handle incompleteness of data, as well as the structural and type heterogeneity exhibited in Figure 1. For example, observe that the *Restaurant* object &2 has no *Entree* subobjects, while *Restaurants* &3 and &4 each have two.

2.2 Query Language

Lorel, for Lore Language, uses the familiar *select-from-where* syntax of SQL, and can be considered an extension to OQL [13] that provides powerful path expressions for traversing the data and extensive coercion rules for a more forgiving type system. Both additions are useful when operating in a semistructured environment. Consider the Lorel query in Example 1.

Example 1 (Lorel Query)

select *e*
from *Guide.Restaurant r, r.Entree e*
where *r.Name = "Baghdad Cafe"* **and** *e.Ingredient = "Mushroom";*

□

The query asks for all *Entree* subobjects of a *Restaurant* object where the restaurant's name is "Baghdad Cafe" and one of the ingredients of the entree has the value "Mushroom". The result of this query over the database in Figure 1 is the set {&9}.

The expression *Guide.Restaurant r, r.Entree e* is a *path expression* describing a traversal through the database. A path expression is composed of *one-step paths* of the form $x.L\ y$, where x is bound to a set of objects, L is the label for some outgoing edge and y designates the set of objects that are reached by starting from an object in the set x and traversing an edge labeled L. Each one-step path describes a single step traversal through the data and can be written $\langle x, L, y\rangle$.

While Lorel supports many ways for specifying paths (for example, by combining one-step path expressions, eliminating variables, or using wild cards), in this paper, we use one-step paths for clarity. Path expressions appearing in the *where* clause that are not quantified by the *from* clause are implicitly existentially quantified according to Lorel semantics.

2.3 View Specification

A view specification statement in Lorel [3] imports objects and edges from a source database into a view. In addition, new (virtual) objects and edges can be included in the view. Our view specification language can: (1) identify objects within a graph; (2) import arbitrary subgraphs; (3) add or remove objects appearing in the view. To specify views, we use Lorel's query and update operations and extend the *select-from-where* statement with a *with* clause.

The *with* clause is composed of path expressions where each path begins from an object chosen by the *select* clause. Each object in the path, along with its connecting edge, is included in the view. The *with* clause is a compromise between returning everything or nothing reachable from selected objects.

We call the objects included by the *select-from-where* part of the view specification the *primary objects* and the objects included in the view by the *with* clause the *adjunct objects*. An object can be both a primary and an adjunct object in a view. Although a view definition may consist of several view specification statements, we concentrate on views defined by a single statement in this paper.

The view specification in Example 2 materializes the result of the query in Example 1 (now written in an OQL-like syntax [4]) along with all *Name* and *Ingredient* subobjects of each *Entree*.

Example 2 (Canonical View Specification in OQL-like Syntax)

define view *FavoriteEntrees* **as**

Entrees =	**select**	*e*
	from	*Guide.Restaurant r, r.Entree e*
	where	**exists** *x* **in** *r.Name: x = "Baghdad Cafe"*
	and	**exists** *y* **in** *e.Ingredient: y = "Mushroom"*
	with	*e.Name n, e.Ingredient i;*

□

The objects bound to *e* are *primary* objects, while all the subobjects specified by the *with* clause are *adjunct* objects. Without the *with* clause, a view is a simple collection of objects that satisfy the query, without edges or subobjects.

2.4 Materialized Views

We now explain how views are materialized in Lore, using our top-down query evaluation strategy [29]. First, the *from* and *where* clauses are evaluated to obtain bindings for variables that appear in the *from* clause and satisfy the *where* clause. The *select* clause is evaluated for these bindings. Each primary object identified by the *select* clause is then augmented with the subobjects and edges in the *with* clause. In the view, each database object is represented by a new *delegate* object.

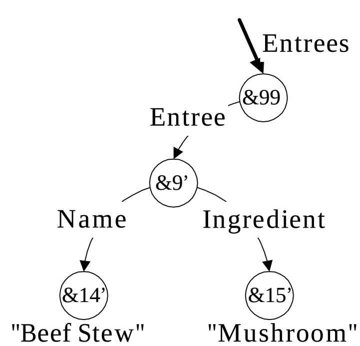

Figure 2: The materialized view defined by the view specification in Example 2

Figure 2 shows the materialized view for Example 2 applied to the database in Figure 1. The objects &9, &14, and &15 in Figure 1 provide bindings for *e*, *n* and *i*; the sole primary object &9' and the adjunct objects &14' and &15' are the corresponding delegate objects in the view.

2.5 Update Operations

The Lorel update statements [4] contain three elementary update operations that can affect a materialized view:

- Insertion and deletion of the edge with label L from the object with oid o_1 to the object with oid o_2, denoted $\langle Ins, o_1, L, o_2\rangle$ and $\langle Del, o_1, L, o_2\rangle$.
- Change of value of the atomic object with oid o_1 from *OldVal* to *NewVal*, denoted $\langle Chg, o_1, \mathit{OldVal}, \mathit{NewVal}\rangle$.

3 View Maintenance

When an update operation affects a materialized view, the view must be maintained to keep it consistent with the database. A view V is considered *consistent* with the database DB if the evaluation of the view specification S over the database yields the view instance ($V = S(DB)$). Therefore, when the database DB is updated to DB', we need to update the view V to $V' = S(DB')$ in order to preserve the consistency.

Our incremental maintenance algorithm computes the new state of the materialized view from the current state of the database, the view, and the database updates. Similar to relational view maintenance algorithms, the incremental maintenance algorithm uses the database updates to minimize the portion of the database examined when computing the view updates [23].

The algorithm applies to an important subset of Lorel [3]. More specifically, it handles every view specification statement without wild cards, subqueries, or negation (except on atomic objects, e.g., $x \neq 5$ is permitted). To simplify the presentation, we also assume in the examples that the *select* clause is of the form "*select y*" (although generalizing for any *select* clause is straightforward).

3.1 Overview of the Incremental Maintenance Algorithm

We treat the primary and adjunct objects (V_{prim} and V_{adj}) separately during maintenance. The algorithm's input is shown in Figure 3.

1. View specification statement S:

   select v_i

   from $v_0.L_1\ v_1,\ \ldots,\ v_j.L_k\ v_k,\ \ldots,\ v_{n-1}.L_n\ v_n$

   // where v_j could be any variable which already appeared in the sequence

   where *conditions*$(v_1, \ldots, v_n)$

   with $v_i.LW_{11}\ w_{11},\ w_{11}.LW_{12}\ w_{12},\ \ldots,\ w_{1(p_1-1)}.LW_{1p_1}\ w_{1p_1},$

   $u_j.LW_{j1}\ w_{j1},\ \ldots,\ w_{j(k-1)}.LW_{jk}\ w_{jk},\ \ldots,\ w_{j(p_j-1)}.LW_{jp_j}\ w_{jp_j},\ \ldots,$

   // where u_j is v_i or w_{kl} ($2 \leq j$, $1 \leq k \leq (j-1)$, $1 \leq l$)
2. Update U: $\langle Ins, o_1, L, o_2\rangle$, $\langle Del, o_1, L, o_2\rangle$, or $\langle Chg, o_1, OldVal, NewVal\rangle$
3. New database state DB'
4. View instance V

Figure 3: Input to the incremental maintenance algorithm

The view specification S, the database update U, and the database state DB' after the update

1. Check for relevance of update U to the view instance V defined by the view specification S. Generate a set of relevant variables R. If R is empty, stop.
2. Generate maintenance statements and create ADD_{prim} and DEL_{prim} using U, S, and R.
3. Generate maintenance statements and create ADD_{adj} and DEL_{adj} using U, S, R, and ADD_{prim} or DEL_{prim}.
4. Install ADD_{prim}, DEL_{prim}, ADD_{adj}, and DEL_{adj} in V.

Figure 4: Basic structure of the incremental maintenance algorithm

are used to compute the view maintenance statements in Lorel syntax.[1] These statements generate the sets ADD_{prim}, DEL_{prim}, ADD_{adj}, and DEL_{adj} of objects and edges to add to and remove from the view. In Figure 3, we abbreviate the *where* clause with "conditions($v_1, \ldots, v_n$)." Conditions are written in disjunctive normal form using boolean expressions, such as: **exists** y **in** *e.Ingredient:* y = *"Mushroom"*.

Figure 4 outlines the steps of the view maintenance algorithm. For simplicity, we describe the algorithm as it operates on a single update. First, it checks whether the update is *relevant* to the view, that is, if update U could cause a change to the view instance V. If so, the algorithm creates the Lorel statements that generate ADD_{prim} and DEL_{prim}. ADD_{prim} and DEL_{prim} identify the primary objects to add and remove by explicitly binding the objects in the update to the view specification. The algorithm then creates the sets of maintenance statements that generate ADD^x_{adj} and DEL^x_{adj}. ADD^x_{adj} and DEL^x_{adj} contain the adjunct objects and edges to add and remove for each *with* clause variable x. Adjunct objects may be affected in three ways: (1) by newly inserted or deleted primary objects; (2) by current adjunct objects that are the source of an inserted or deleted edge; and (3) by atomic value changes.

3.2 Relevance of an Update

To avoid generating (and evaluating!) unnecessary maintenance statements, we first perform some simple relevance checks. We use an auxiliary data structure, *RelevantOids*, to keep information that would be inferred from the schema in a structured database. *RelevantOids* contains the object identifier of every object touched during the evaluation of a particular view, paired with the variable to which it was bound, whether or not the object eventually appears in the view. It is used to check quickly whether a database update could possibly affect the view. For example, if object o_1 in a *Chg* update does not appear in *RelevantOids*, then it was not examined during view evaluation, its value is not relevant, and the update can be ignored. Note that we need to use our top-down evaluation strategy to compute *RelevantOids*.

We also use syntactic checks that indicate whether specific atomic value changes could affect the view. For each comparison in the view specification *where* clause that involves a constant value, we compare the constant to the update's *OldVal* and *NewVal*. If both or neither of *OldVal* and *NewVal* satisfy the comparison, then the change cannot affect the view.

Figure 5 presents the procedure *RelevantVars*, which determines the set of relevant variables

[1] We extend Lorel to allow the use of explicit object identifiers wherever names are allowed within a statement.

for the given update and view specification.

procedure *RelevantVars*(Update U, View specification S)
// If updated object is not in *RelevantOids*, then it's not relevant
if $\langle o_1(U), \cdot\rangle \notin$ *RelevantOids* **then return** $\emptyset$;
// Find out which variables are relevant
$vars \leftarrow \emptyset$; $relvars \leftarrow \emptyset$;
foreach $v \in$ variables(S) **do**
    // If updated object is not in *RelevantOids*, then it's not relevant
    if $\langle o_1(U), v\rangle \in$ *RelevantOids* **then** $vars \leftarrow vars \cup \{v\}$;
// If update is an atomic change, do syntactic check
if type(U) = Chg **then**
    foreach $v \in vars$ **do**
        foreach $c \in$ constants(S, v) **do**
            // See if there's a constant in the view spec whose value may have changed
            if (*OldVal*(U) $\neq$ c **and** *NewVal*(U) = c) **or** (*OldVal*(U) = c **and** *NewVal*(U) $\neq$ c) **then**
                $relvars \leftarrow relvars \cup \{v\}$;
else $relvars \leftarrow vars$;
return $relvars$;

Figure 5: *RelevantVars*(U, S, R) returns the set of view specification variables for which the update U is relevant.

For example, suppose that object &5's value in Figure 1 is changed from "Thai City" to "Hunan Wok". Then we can infer that this update does not affect the view in Example 2, because the view specification mentions neither "Thai City" nor "Hunan Wok". On the other hand, if the value of &5 is changed to "Baghdad Cafe", which is the constant used in the comparison *x.Name = "Baghdad Cafe"*, then the update may be relevant.

We do not attempt to quantify the savings achieved by using *RelevantOids* in this paper. However, we note that for views defined over a small portion of the database, most updates are irrelevant.

3.3 Generating View Maintenance Statements

We now describe how to generate the maintenance statements for each type of update: edge insertion, edge deletion, or atomic value change. For each one-step path in the view specification, we generate a maintenance statement that checks whether the updated edge binds to it. If so, the statement produces updates to the view. We use auxiliary data structures to represent the one-step paths. $OneStepPath_{from}$, $OneStepPath_{prim}$, and $OneStepPath_{adj}$ contain all the one-step paths that appear in the *from* clause; *from* and *where* clauses; and *with* clause; respectively. For example, $OneStepPath_{prim}$ for the view specification in Example 2 is {*Guide.Restaurant r, r.Entree e, r.Name x, e.Ingredient y*}. Note that each *OneStepPath* is small since it depends on the query and not on the database.

3.3.1 Edge Insertion

For edge insertion, let the update be $\langle Ins, o_1, L, o_2\rangle$. We generate a primary object maintenance statement for every possible pair of bindings of o_1 and o_2 using the procedure *GenAddPrim* in Figure 6.

procedure *GenAddPrim*(Update $U = \langle Ins, o_1, L, o_2\rangle$, View specification S, *RelevantVars* R)
// For each relevant variable
foreach $a \in R$ **do**
  // For each place where the update can be substituted in the view spec
  foreach $\langle a,\ L,\ b\rangle \in OneStepPath_{prim}$ **do**
    // Write a maintenance stmt based on the view specification:
    ADD_{prim}+= copy S except for the *with* clause and
      replace "$a.L_i$" with "$o_1.L_i$" $\forall L_i$ and "$b.L_j$" with "$o_2.L_j$" $\forall L_j$ in *from* clause,
      replace "a" with "o_1" and "b" with "o_2" in *where* clause,
      add "**and** $a = o_1$" to each disjunct in *where* clause,
      add "**and** $b = o_2$" to each disjunct in *where* clause if $\langle a,\ L,\ b\rangle \in OneStepPath_{from}$

Figure 6: *GenAddPrim* generates the ADD_{prim} maintenance statements.

Example 3 (Inserting an Edge: Generating ADD_{prim})
Suppose that update $\langle Ins, \&10, Ingredient, \&15\rangle$ is performed on the database in Figure 1. The Baghdad Cafe restaurant now has two entrees with the ingredient "Mushroom". Given the view specification, *RelevantVars* returns the set {e}. *GenAddPrim* then generates one statement.

ADD_{prim} +=
select *e*
from *Guide.Restaurant r, r.Entree e*
where **exists** *x* **in** *r.Name: x = "Baghdad Cafe"*
and **exists** &15 **in** &10.*Ingredient:* &15 = *"Mushroom"*
and *e* = &10;

□

We then generate the maintenance statements for the adjunct objects. There are two cases to consider: (1) adjunct objects attached to the new primary objects in ADD_{prim} and (2) adjunct objects that are newly connected to the view by the inserted edge from o_1 to o_2 (when o_1 is an adjunct object).

For the first case, we generate maintenance statements starting from the set ADD_{prim}. For the second case, we first test whether the inserted edge matches a relevant (adjunct object) variable and has a matching label. If so, then we generate a set of maintenance statements that add the inserted edge and all subsequent paths in $OneStepPath_{adj}$. Both cases are handled by procedure *GenAddAdj* in Figure 7.

Example 4 (Inserting an Edge: Generating ADD_{adj})
GenAddAdj generates the following maintenance statements for the update $\langle Ins, \&10, Ingredient, \&15\rangle$.

ADD^n_{adj} += **select** $\langle$*e, Name, n*$\rangle$ **from** ADD_{prim} *e, e.Name n;*
ADD^i_{adj} += **select** $\langle$*e, Ingredient, i*$\rangle$ **from** ADD_{prim} *e, e.Ingredient i;*

Since the inserted edge is not connected to an existing adjunct object (&10 is not currently an adjunct object in the view), no statements are generated by the second half of *GenAddAdj*. □

```
procedure GenAddAdj(Update U = ⟨Ins, o_1, L, o_2⟩, View specification S, RelevantVars R)
// (1) If primary objects were added, need to add adjunct objects from them
if ADD_prim ≠ ∅ then
  // For each adjunct edge and object
  foreach ⟨w_j(k-1), LW_jk, w_jk⟩ ∈ OneStepPath_adj do
    // Write a maintenance statement based on the view specification (no where or with clause):
    ADD_adj^{w_jk} += select ⟨w_j(k-1), LW_jk, w_jk⟩
                      from   ADD_prim v_i, v_i.LW_j1 w_j1, ..., w_j(k-1).LW_jk w_jk;
// (2) For each place that edge could be adjunct edge
foreach v ∈ R do
  foreach ⟨v, L, w_jk⟩ ∈ OneStepPath_adj do
    // Write a set of maintenance statements starting from added edge:
    // Add inserted edge to the view
    ADD_adj^{w_jk} += select ⟨o_1, L, o_2⟩;
    // From o_2, add any necessary edges
    ADD_adj^{w_j(k+1)} += select ⟨o_2, LW_j(k+1), w_j(k+1)⟩ from o_2.LW_j(k+1) w_j(k+1);
    // In a similar fashion, include all paths
    foreach ⟨w_j(k+n-1), LW_j(k+n), w_j(k+n)⟩ ∈ O do
      ADD_adj^{w_j(k+n)} += select ⟨w_j(k+n-1), LW_j(k+n), w_j(k+n)⟩
                            from   o_2.LW_j(k+1) w_j(k+1), ..., w_j(k+n-1).LW_j(k+n) w_j(k+n);
```

Figure 7: *GenAddAdj* generates the ADD_{adj} maintenance statements.

Because the addition of an edge in the absence of negation cannot cause a deletion, we do not have to generate DEL_{prim} or DEL_{adj}.

3.3.2 Edge Deletion

Let the update be $\langle Del, o_1, L, o_2\rangle$. A deleted edge may: (1) be irrelevant and not affect the view; (2) cause a primary object to be deleted; (3) appear directly in the (adjunct) view and need to be removed. Either (2) or (3) could cause additional adjunct edges to be removed from the view. In principle, a delete edge update generates maintenance statements similar to an insert edge update. However, the delete edge statements must simulate the now deleted edge in the view to determine whether it originally contributed to the appearance of objects or edges in the view. Also, the delete edge statements must check (using a subquery) whether a potentially deleted object or edge should remain in the view due to paths not involving the deleted edge. For example, if the *Entree* object &9 in Figure 1 had *two* "Mushroom" ingredients then applying the update $\langle Del, \&9, Ingredient, \&15\rangle$ should not remove the *Entree* object &9 from the view.

Figure 8 shows the procedure *GenDelPrim*, used to generate the maintenance statements for the primary objects.

Example 5 (Deleting an Edge: Generating DEL_{prim})
Suppose the update $U = \langle Del, \&3, Entree, \&9\rangle$ is applied to the database of Figure 1. The object &9 must be removed from the view. *GenDelPrim* generates one statement.

procedure *GenDelPrim*(Update $U = \langle Del, o_1, L, o_2\rangle$, View specification S, *RelevantVars* R)
// For each relevant variable
foreach $a \in R$ **do**
    // For each place where the update can be substituted in the view spec
    foreach $\langle a,\ L,\ b\rangle \in OneStepPath_{prim}$ **do**
        // Write a maintenance statement based on the view specification:
        DEL_{prim}+= copy S except for the *with* clause and
            // The first two replacements reconstruct the before state
            replace "$a.L\ b$" with "$(o_1.L \cup \{o_2\})\ b$" in *from* clause.
            replace "**exists** b **in** $a.L$" with "**exists** b **in** $(o_1.L \cup \{o_2\})$" in *where* clause.
            // The remaining replacements handle normal appearance of bound variables
            replace "$a.L_i$" with "$o_1.L_i$" $\forall L_i$ in *from* clause
            replace "$b.L_j$" with "$o_2.L_j$" $\forall L_j$ in *from* clause
            replace "a" with "o_1" in *where* clause
            replace "b" with "o_2" in *where* clause
            add "**and** $a = o_1$" to each disjunct in *where* clause
            add "**and** $b = o_2$" to each disjunct in *where* clause if $\langle a,\ L,\ b\rangle \in OneStepPath_{from}$
            // The duplicate test is a subquery that ensures that the object bound
            // to v_i is not in the view for another reason
            add to *where* clause "**and not exists** (S')" where S' is
            S without *with* clause and new variables v'_j for each v'_j
            and an additional *where* condition: "$v'_i = v_i$" (v_i is the selected variable in S)

Figure 8: Generating maintenance statements for DEL_{prim}

```
DEL_prim +=  select  e
             from    Guide.Restaurant r, (&3.Entree ∪{&9}) e
             where   exists x in &3.Name: x = "Baghdad Cafe"
             and     exists y in &9.Ingredient: y = "Mushroom"
             and     r = &3 and e = &9
             and     not exists ( select  e'
                                  from    Guide.Restaurant r', r'.Entree e'
                                  where   exists x' in r'.Name: x' = "Baghdad Cafe"
                                  and     exists y' in e'.Ingredient: y' = "Mushroom"
                                  and     e' = e);
```

This statement adds bindings for r and *r.Entree* to the view specification S and reconstructs the before state by binding e to &9. The transformations to the original query are summarized in the table shown in Figure 9. □

We then generate the maintenance statements for the adjunct objects and edges. However, like the objects in the primary zone, an adjunct object or edge can be included in the view due to multiple paths. Reachability via a deleted edge is not a sufficient condition for deleting an adjunct object or edge, as we explain further in Appendix A. Instead, a subquery of the *where* clause looks for other variable bindings for the edge to be removed. If another binding is found, then the edge is not deleted. Procedure *GenDelAdj* in Figure 10 generates the maintenance statements for the

Clause	Original	Incremental Statement	General Rule
From	Guide.Restaurant r	Guide.Restaurant r	$v_j.L_k\ v_k$ s.t. $(v_j$ and $v_k) \neq (a$ or $b) \rightarrow$ No Change
From	r.Entree e	(&3.Entree $\cup${&9})e	$a.L\ b \rightarrow (o_1.L \cup \{o_2\})\ b$
Where	$\exists x$ **in** r.Name: x = "Baghdad Cafe"	$\exists x$ **in** &3.Name: x = "Baghdad Cafe"	$a.L_j\ v_j$ s.t. $v_j \neq b \rightarrow o_1.L_j\ v_j$
Where	$\exists y$ **in** e.Ingredient: y = "Mushroom"	$\exists y$ **in** &9.Ingredient: y = "Mushroom"	$b.L_j\ v_j \rightarrow o_2.L_j\ v_j$

Figure 9: Transformations for incremental maintenance statements for Example 5

adjunct objects and edges.

Example 6 (Deleting an Edge: Generating DEL_{adj})

For the update $\langle Del, \&3, Entree, \&9\rangle$, procedure *GenDelAdj* creates one maintenance statement for the paths in $OneStepPath_{adj}$.

DEL^n_{adj} += **select** $\langle e,\ Name,\ n\rangle$ **from** DEL_{prim} *e, e.Name n;*
DEL^i_{adj} += **select** $\langle e,\ Ingredient,\ i\rangle$ **from** DEL_{prim} *e, e.Ingredient i;*

Neither statement in our simple example includes a *where* subclause. More complex cases, however, do require the full generality of *GenDelAdj*. □

3.3.3 Atomic Value Change

Let the update U be $\langle Chg, o_1, OldVal, NewVal\ \rangle$. This value change may cause both deletions *and* insertions in the view or there might be no change at all, e.g., if the change is irrelevant to the view. Given the value update information, we have no way of knowing the incoming labels to object o_1. Due to object sharing, an object may have many incoming labels and knowing those that were traversed in the original view evaluation does not suffice. Therefore, we consider all possible variable bindings for the value change. Note that the predicate relevance test in *RelevantVars* in Figure 5 may be omitted when testing the values is expensive. *RelevantVars* can also simplify matters by tracking whether the changed value could potentially cause the *addition* versus the *removal* of objects.

Example 7 (Atomic Value Change)

Suppose the update U is $\langle Upd, \&7,$ *"Baghdad Cafe", "Wendy's"*$\rangle$. We identify x as the only relevant variable for Example 2. This atomic value change *cannot* result in adding new objects to the materialized view, because the new value "Wendy's" does not satisfy any condition on x. However, the old value "Baghdad Cafe" does. If x is bound to &7 then the condition's value changes from true to false and some objects may no longer be in the view. We therefore generate DEL_{prim} for the deletion of $\langle r,$ *Name*, &7$\rangle$ since *Name* is the label associated with x.

```
DEL_prim +=  select  e
             from    Guide.Restaurant r, r.Entree e
             where   exists &7 in r.Name : (OldVal(&7) = "Baghdad Cafe")
             and     exists y in e.Ingredient: y = "Mushroom"
             and     not exists ( select  e'
                                  from    Guide.Restaurant r', r'.Entree e'
                                  where   exists x' in r'.Name: x' = "Baghdad Cafe"
                                  and     exists y' in e'.Ingredient: y' = "Mushroom"
                                  and     e' = e);
```

```
procedure GenDelAdj(Update U = ⟨Del, o1, L, o2⟩, View Specification S, RelevantVars R)
if DEL_prim ≠ ∅ then
    // Deletion of primary objects could affect every one-step path in the adjunct.
    // Identify edges that need to be deleted because of primary object deletions:
    foreach ⟨w_j(k-1), LW_jk, w_jk⟩ ∈ OneStepPath_adj do
        // Write one maintenance statement for each one-step path in the with clause.
        // The path in the from clause has to match some path, starting at the selected
        // variable v_i, in the with clause of the view specification (see Figure 3).
        DEL_adj^{w_jk} +=  select  ⟨w_j(k-1), LW_jk, w_jk⟩
                           from    DEL_prim v_i, v_i.LW_j1 w_j1, ..., w_j(k-1).LW_jk w_jk
                           where   not exists (
                                     select   true
                                     // the where clause contains one subclause for each path in the with clause
                                     // of the view specification that leads to a variable that uses label LW_jk
                                     where    (V_prim v_i, v_i.LW_j1 w'_j1, ..., w'_j(k-1).LW_jk w'_jk,
                                              and w'_j(k-1) = w_j(k-1) and w'_jk = w_jk)
                                     or       ...);

// The adjunct zone could be affected if the label of the deleted edge appears in the with clause
foreach u ∈ R do
    foreach ⟨u, L, u_i⟩ ∈ OneStepPath_adj do
        // Ensure that u_i is relevant with respect to o2
        if ⟨o2, u_i⟩ ∈ RelevantOids then
            // Must remove the edge from the view since it is deleted from the database
            DEL_adj^{u_i} += select ⟨o1, L, o2⟩;
            // Write a set of maintenance statements "starting" from the deleted edge to delete
            // all the edges in the view along paths that start from variable u_i for u_i = o2
            foreach ⟨w_j(k-1), LW_jk, w_jk⟩ ∈ OneStepPath_adj do
                // The path in the from clause has to match some path starting at u_i in the with clause
                // of the view specification statement (see Figure 3)
                DEL_adj^{w_jk} +=  select  ⟨w_j(k-1), LW_jk, w_jk⟩
                                   from    o2.LW_j1 w_j1, ..., w_j(k-1).LW_jk w_jk
                                    // Same subquery as above
                                   where   not exists (
                                             select   true
                                             where    (V_prim v_i, v_i.LW_j1 w'_j1, ..., w'_j(k-1).LW_jk w'_jk,
                                                      and w'_j(k-1) = w_j(k-1) and w'_jk = w_jk)
                                             or       ...);
```

Figure 10: Generating maintenance statements for DEL_{adj}

Based on DEL_{prim}, DEL^n_{adj} and DEL^i_{adj} are calculated as shown in Example 6. □

3.4 Installing the Maintenance Changes

Finally, the changes represented by ADD_{prim}, ADD_{adj}, DEL_{prim}, and DEL_{adj} are installed in the materialized view. Since there is no duplication of objects in the view, deletions need to be installed in the view before insertions. If a view object ceases to be a primary object, it may still remain in the view as an adjunct object and vice versa. Then, if the update is an atomic value change of an object in the view, the new value is installed in the delegate object. Given ADD_{prim}, ADD_{adj}, DEL_{prim}, and DEL_{adj}, the installation process can use indices to identify the objects and edges that are already in the view.

4 Cost Model

In this section, we present an analytic model that evaluates the cost of both view recomputation and incremental maintenance for a given update. A more detailed cost model that follows a similar approach for an object-oriented system is presented in [7], and [30] presents a more complex cost model for Lore. The cost model can be used by the query optimizer to choose dynamically, for a given set of updates, whether to recompute or to incrementally maintain the view.

The cost assigned to a plan is the estimated number of object fetches during query processing. While more complex cost models have been proposed for object oriented systems, e.g., [19], they rely heavily on the object clustering guaranteed by the system. In the absence of clustering, we count the number of object fetches, since we cannot accurately determine whether two objects will be on the same page.

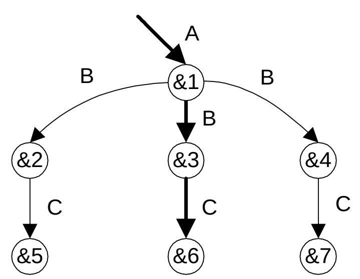

Figure 11: Path expression evaluation and statistics (path expression: *A.B b, b.C c*)

The cost formulas rely on our top-down query execution strategy, described in Section 2.4. This strategy results in a depth first traversal of the data starting from a named object [29]. Other query execution strategies for semistructured databases are investigated in [30, 15]. The following definitions are based on statistics kept by the system.

- *Fanout*(x, L): the estimated average number of children with the label L that are descendants of some object in the set x. The variable x must already be bound using a path expression. In Figure 11, *Fanout*(b, C) is 1 since (on average) each of the objects in the set b has a single C child.

- $|x|$: the estimated number of objects in the set corresponding to a variable x. In Figure 11, $|c| = 3$.
- $Cost(x, L, y)$: the estimated cost, i.e., the number of objects fetched in order to get all of the subobjects with edge labeled L originating from any object in x, where each resulting object is placed into set y. This cost is computed by $Cost(x, L, y) = |x| * Fanout(x, L)$.

For example, given the path expression $b.C\ c$ of Figure 11, the evaluation cost for $b.C\ c$ is $Cost(b, C, c) = |b| * Fanout(b, C) = |A| * Fanout(A, B) * Fanout(b, C) = 1 * 3 * 1 = 3$.

Without any bindings, the cost for evaluating a path expression P is

$$Cost(total) = \sum_{\langle x,L,y\rangle \in P} Cost(x, L, y)$$

The incremental maintenance statements bind variables to the objects contained in the update and use the bindings to prune the search space. The execution proceeds until a variable x bound by the update is encountered. If the object bound to x is not the updated object, then the evaluation short circuits and goes on to the next binding for x. A bound variable lowers the cost of the computation for the rest of the path expression, since it limits the remaining portion of a path to objects reachable from the bound variable. Insertions and deletions provide two object bindings, while an atomic value change provides only one. Both the cost model and the formulas ignore object sharing, which can reduce the actual cost.

We now apply our cost formula to the view specification of Example 2.

Example 8 (Cost of Full Recomputation)
The cost for the complete recomputation of the view is:

$$\begin{aligned}
&\sum_{\langle x,L,y\rangle \in OneStepPath_{prim} \cup OneStepPath_{adj}} Cost(x, L, y) \\
= \;& Cost(Guide, Restaurant, r) + Cost(r, Entree, e) + Cost(r, Name, x) + Cost(e, Ingredient, y) \\
& + Cost(e, Name, n) + Cost(e, Ingredient, i) \\
= \;& |Guide| * Fanout(Guide, Restaurant) + |r| * Fanout(r, Entree) + |r| * Fanout(r, Name) + \\
& |e| * Fanout(e, Ingredient) + |e| * Fanout(e, Name) + |e| * Fanout(e, Ingredient) \\
= \;& |Guide| * Fanout(Guide, Restaurant) + |Guide| * Fanout(Guide, Restaurant) * Fanout(r, Entree) + \\
& |Guide| * Fanout(Guide, Restaurant) * Fanout(r, Name) + \\
& |Guide| * Fanout(Guide, Restaurant) * Fanout(r, Entree) * Fanout(e, Name) + \\
& 2 * |Guide| * Fanout(Guide, Restaurant) * Fanout(r, Entree) * Fanout(e, Ingredient)
\end{aligned}$$

□

We now show how our cost formula applies to the maintenance statements in Example 3 for update $\langle Ins, \&10, Ingredient, \&15\rangle$.

Example 9 (Maintenance Cost of Inserting an Edge)
$P = \{\langle Guide,\ Restaurant,\ r\rangle, \langle r,\ Entree,\ e\rangle, \langle r,\ Name,\ x\rangle, \langle e,\ Ingredient,\ y\rangle\}$ is the set of one-step path expressions in the maintenance statement of Example 3 and $P' = \{\langle ADD_{prim},\ Name,\ n\rangle,$

$\langle ADD_{prim}$, *Ingredient*, $i\rangle\}$ is the set of one-step path expressions in the maintenance statement of Example 4. The bindings e = &10 and y = &15 is provided.

$$\begin{aligned}
&\sum_{(x,L,y)\in P} Cost(x,L,y) + \sum_{(x,L,y)\in P'} Cost(x,L,y) \\
&= Cost(Guide, Restaurant, r) + Cost(r, Entree, e) + Cost(r, Name, x) + Cost(e, Ingredient, y) \\
&\quad + Cost(ADD_{prim}, Name, n) + Cost(ADD_{prim}, Ingredient, i) \\
&= |Guide| * Fanout(Guide, Restaurant) + |r| * Fanout(r, Entree) + \\
&\quad |r| * Fanout(r, Name) + |e| * Fanout(e, Ingredient) + \\
&\quad |ADD_{prim}| * Fanout(ADD_{prim}, Name) + |ADD_{prim}| * Fanout(ADD_{prim}, Ingredient) \\
&= |Guide| * Fanout(Guide, Restaurant) + 1 + \\
&\quad |Guide| * Fanout(Guide, Restaurant) * Fanout(r, Name) + 1 * Fanout(e, Ingredient) + \\
&\quad |ADD_{prim}| * Fanout(ADD_{prim}, Name) + |ADD_{prim}| * Fanout(ADD_{prim}, Ingredient)
\end{aligned}$$

$|ADD_{prim}|$ depends upon the number of possible bindings for e and the selectivity of the *where* clause, as follows:

$$|ADD_{prim}| = |e| * Selectivity(where) = 1 * Selectivity(where) \leq 1. \qquad \square$$

If the selectivity of the *where* clause of a query is $a\%$, then only $a\%$ of all the objects that satisfy the view specification before applying the *where* clause are actually in the view. In order for an atomic value change from *OldVal* to *NewVal* to be relevant, the truth value of the *where* clause needs to change when *OldVal* is substituted by *NewVal*. As the 2×2 matrix in Figure 12 shows, an atomic change causes insertions to the view $a(1-a)\%$ of the time and deletions to the view $a(1-a)\%$ of the time. When computing the average cost of incremental maintenance after an atomic value change, we multiply the costs of updating the view by $a(1-a)$ to take the relevance of the update into account.

		OldVal true	*OldVal* false
NewVal	true	$a \cdot a$	$a(1-a)$
	false	$a(1-a)$	$(1-a)(1-a)$

Figure 12: Truth value of the *where* clause for *OldVal* and *NewVal*

5 Evaluation

Our evaluator program accepts a single view specification statement, a database, and a single change, and computes the cost for both recomputation and incremental maintenance using our cost model. In this section, we present the costs for a variety of view specifications, databases, and updates. We did not use the auxiliary structure *RelevantOids* in the cost model, so the actual costs for incremental maintenance will be lower in many cases. In all our graphs, the cost is shown on the y axis in a *logarithmic* scale.

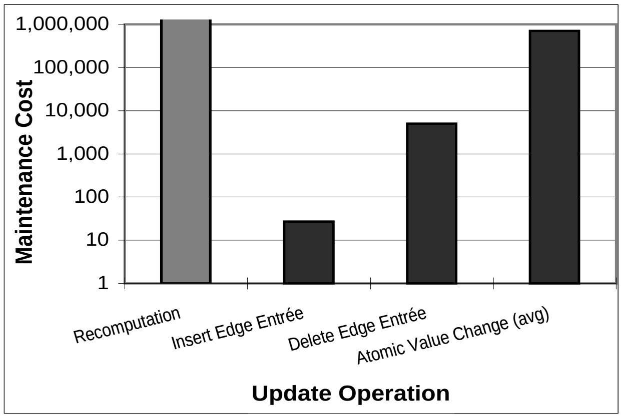

Figure 13: Base costs for update operations

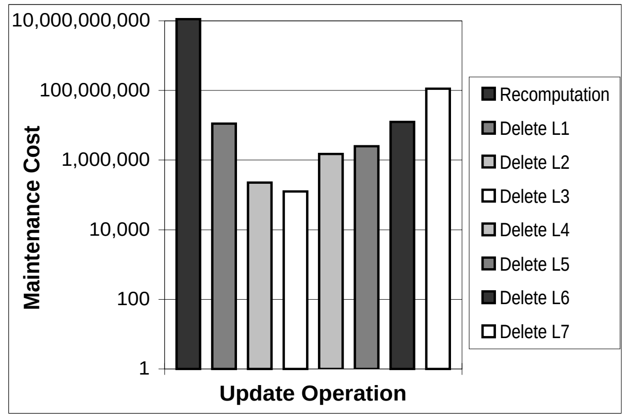

Figure 14: Varying the position of the bound variable in the *from* clause

5.1 Base Costs for Update Operations

In the first experiment, shown in Figure 13, we looked at the costs of different update operations for the view specification of Example 2. The database contained one Guide, 1000 restaurants, 100 entrees and 1 name per restaurant, and 10 ingredients and 2 names per entree, and possibly other portions of the database that were not traversed when computing or maintaining the view. We assumed a fixed selectivity for the *where* clause of 50%. Each bar shows the cost of maintaining the view after a single update for a different update operation.

Recomputation is over 100 times more expensive than incremental maintenance for insert or delete edge operations. These savings are due to binding the variables associated with the inserted or deleted edge. A much smaller portion of the database is traversed during execution of the incremental view maintenance statements compared to the view specification statement. Maintaining a view for edge insertions was significantly cheaper than for edge deletions due to the subquery in delete edge maintenance statements.

The maintenance cost for an atomic value change varies widely. Without conditional relevance tests, the incremental algorithm will generate a maintenance statement for each condition in the *where* clause. Although each statement will incorporate a variable binding for the changed object, there is only one such binding. Depending on where the binding occurs, the maintenance statement cost may vary from much to only slightly cheaper than the cost of recomputation. Given several *where* conditions, recomputation may be more cost effective. For example, for the view in Example 2, testing a single atomic change against both conditions in the *where* clause cost is almost as much as recomputation, as shown in Figure 13. However, relevance tests using *RelevantOids* can often determine that a single or none of the conditions in the *where* clause are relevant. For the same example, evaluating the maintenance statement for only one condition is always cheaper than recomputation.

5.2 Varying the Position of the Bound Variable in the *from* Clause

The position of the bound variable affects the cost of incremental maintenance. For this experiment, we used a view specification containing a chain of eight one-step paths in the *from* clause:

define view *VaryingFrom* **as**

$VF =$ **select** z_2 **from** $A.L_1\ z_1,\ z_1.L_2\ z_2,\ \ldots,\ z_7.L_8\ z_8$;

The database contained a single named object A, 1000 L_1 subobjects of A, 100 L_2 subobjects per z_1, and ten L_i subobjects per z_{i-1} for $3 \leq i \leq 8$. We deleted the edge $\langle o_{i-1}, L_i, o_i \rangle$, for all values of $3 \leq i \leq 8$ in turn. Figure 14 shows that recomputation is 10–500 times more expensive than incremental maintenance. When the bound variable is in the middle of a path expression, it effectively divides the path into two shorter paths whose costs are added rather than multiplied. Therefore, the variable binding provided by the newly inserted or deleted edge has the most beneficial effect when it occurs in the middle of the path expression.

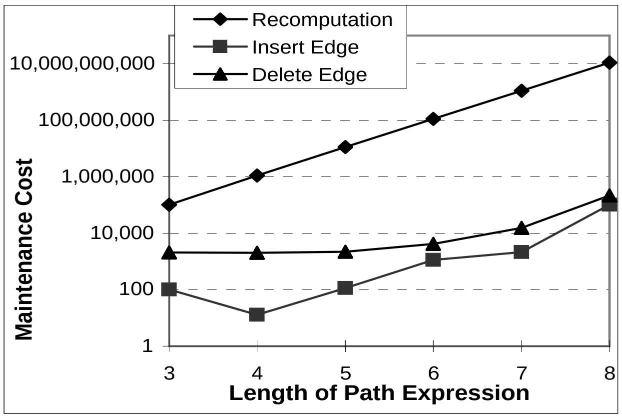

Figure 15: Varying the length of the *from* clause

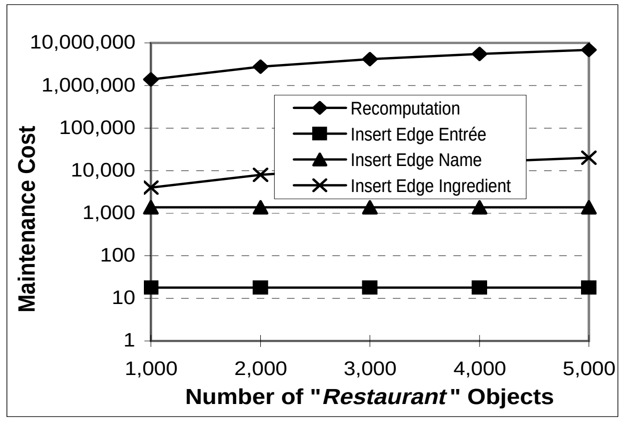

Figure 16: Varying the database size

5.3 Varying the Length of the *from* Clause

The number of variables in the *from* clause also affects the cost of incremental maintenance. For this experiment, we used view specifications of the following pattern and varied the length of the path expression in the *from* clause from three to eight one-step paths.

define view *VaryingFrom2* **as**

$VF_2 =$ **select** z_2 **from** $A.L_1\ z_1,\ z_1.L_2\ z_2,\ \ldots,\ z_{n-1}.L_n\ z_n$;

The database was the same as in Section 5.2. For each view specification, we inserted the edge $\langle o_1, L_{\lfloor n/2 \rfloor+1}, o_2 \rangle$, which bound the middle variable in the path. Figure 15 shows that as the number of variables increased, the recomputation cost also increased. Each additional edge in the *from* clause caused the relevant portion of the database to expand by ten. The incremental maintenance costs are much lower and increase much more slowly due to the bound variables. The insert edge cost decreases when $n = 4$ because the bound variable appears in a more advantageous position in the path expression.

5.4 Varying the Database Size

For the fourth experiment, we used the view specification in Section 5.1, but varied the size of the database. We increased the number of restaurants in the database from 1000 to 5000, and kept the same average number of entrees per restaurant, ingredients per entree, etc. Therefore, when the number of restaurants doubled, for example, the size of the relevant portion of the database doubled. The maintenance costs after various edge insertions are shown in Figure 16. The cost of recomputation is consistently 100–100,000 times higher than the cost of incrementally maintaining the view.

The size of the database had negligible effect on inserting an *Entree* and *Name* edge, since the inserted edge provided a binding to a specific restaurant. When inserting an *Ingredient* edge the placement of the bound variable was not as fortunate, and the size of the database affected the execution cost of the maintenance statements. The recomputation cost grew linearly with the size of the relevant portion of the database, since it traversed the entire database.

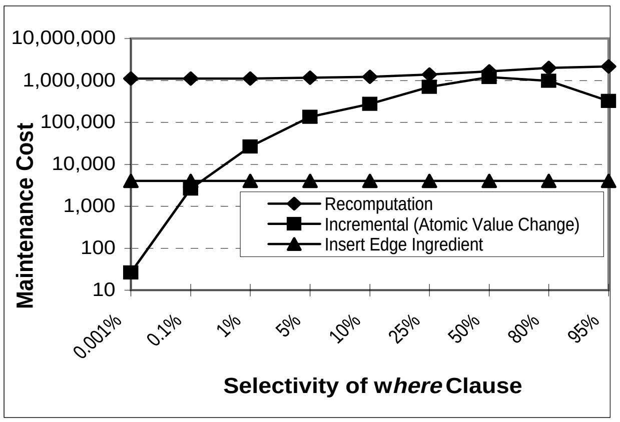

Figure 17: Varying the selectivity of the *where* clause

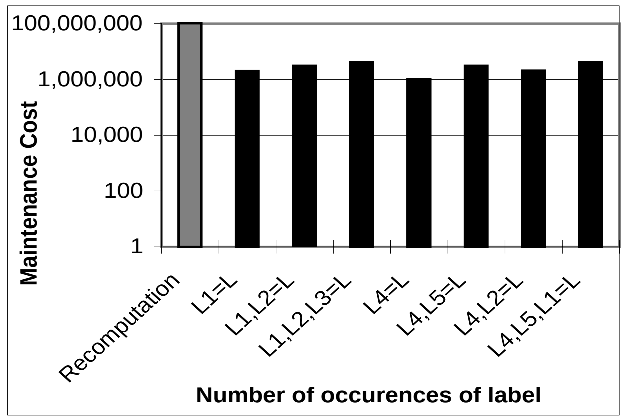

Figure 18: Varying the number of occurrences of a label in the view specification

5.5 Varying the Selectivity of the *where* Clause

Figure 17 shows the results of the fifth experiment. We kept the same view definition and database structure as in Section 5.1, but varied the selectivity of the *where* clause by changing the atomic values in the database. As the selectivity increases, more objects are included in the view. Therefore, the recomputation cost went up to find the increased number of adjunct objects. The incremental maintenance cost for atomic value changes is influenced significantly by the selectivity of the *where* clause. When the selectivity is low, most atomic value changes can be screened out by tge syntactic relevance test before running any queries. When the selectivity is high, most objects are already included in the view, so very few new objects need to be added to the view because of the change. Since syntactic relevance tests only apply to atomic value changes (and affect their cost!), the cost for an insert edge update does not change based on the atomic values and the selectivity.

5.6 Varying the Number of Occurrences of a Label in the View Specification

For the final experiment, we varied the number of times the label of the inserted or deleted edge matched a label in the view specification. We used view specification statements of the following form:

define view *VarLabel* **as**

$VL =$	**select**	x
	from	$A.L_1\ x,\ x.L_2\ y,\ y.L_3\ z$
	where	$y.L_4 < 10$ **and** $z.L_5 > 7$
	with	$x.L_6;$

We inserted or deleted the edge $\langle o_1, L, o_2 \rangle$. For each test, we changed some of the labels in the view specification (as well as the corresponding labels in the source database) to "L", as indicated by the legend for the results, shown in Figure 18. The database contained 100 subobjects of an object for each distinct label. The recomputation cost was unaffected by the specific labels, since the structure of the database remained the same. The incremental maintenance costs varied, however, since each appearance of the label L required an additional maintenance statement. However, even when the label L appeared three times in the view specification, incremental maintenance was still 20 times cheaper than recomputation.

6 Conclusion

Most approaches for incremental view maintenance rely on the database schema to generate maintenance statements. We described an incremental maintenance algorithm for views over semi-structured, or schemaless, data. Our algorithm identifies the needed view changes based on the information available from the view specification, the update operation, the database state after the update, and some auxiliary data structures that are generated when populating the view.

Our evaluation results show that our incremental maintenance algorithm outperforms recomputation, even for large numbers of insert and delete edge updates. However, in some situations, incremental maintenance can be as expensive as full recomputation of the view for a single atomic value change, due to the simple query execution strategy in our cost model. These numbers reflect only the portion of the database traversed during view evaluation, not the total size of the database. Furthermore, our algorithm scales well with increasing database size.

We have implemented view materialization within Lore [29]. We plan to implement the incremental maintenance algorithm as well. Several optimizations to our incremental maintenance algorithm are possible. First, we plan to extend the algorithm to handle sets of updates together. Second, if the data has a tree structure, then the maintenance statements can be simplified, e.g., by eliminating the subqueries when deleting objects or edges. Third, we would like to incorporate query rewriting and query optimization techniques [30] for semistructured data and provide more query execution choices to the query optimizer. Finally, we would like to consider using inferred schematic information such as DataGuides [32, 21] or graph schemas [10, 18] to optimize view maintenance.

7 Acknowlegements

We thank Yue Zhuge for organizing the group whose discussions eventually led to this paper. We would also like to thank Jennifer Widom and the rest of the Lore team at Stanford for many interesting discussions and comments.

References

[1] S. Abiteboul. Querying Semistructured Data. In *Proceedings of the ICDT*, 1997.

[2] S. Abiteboul and A. Bonner. Objects and Views. In *ACM SIGMOD Proceedings*, pages 238–247, Denver, Colorado, May 1991.

[3] 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, Tuscon, Arizona, May 1997.

[4] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel Query Language for Semistructured Data. *Journal of Digital Libraries*, 1(1), Nov. 1996.

[5] C. Beeri and Y. Kornatski. A Logical Query Language for Hypertext Systems. In *Proceedings of the 16th International Conference on Very Large Databases (VLDB)*, Brisbane, Australia, Aug. 1990.

[6] E. Bertino. A View Mechanism for Object-Oriented Databases. In *EDBT Proceedings*, pages 136–151, Vienna, Mar. 1991.

[7] E. Bertino and P. Foscoli. On Modeling Cost Functions of Object-Oriented Databases. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):500–508, May 1997.

[8] J. A. Blakeley, P. Larson, and F. W. Tompa. Efficiently Updating Materialized Views. In C. Zaniolo, editor, *ACM SIGMOD Proceedings*, pages 61–71, Washington, D.C., May 1986.

[9] P. Buneman. Semistructured Data, May 1997. Tutorial SIGMOD Conference.

[10] P. Buneman, S. Davidson, M. Fernandez, and D. Suciu. Adding Structure to Unstructured Data. In *Proceedings of the ICDT'97*, Delphi, Greece, Jan. 1997.

[11] P. Buneman, S. Davidson, and D. Suciu. Programming Constructs for Unstructured Data. In *Proceedings DBPL*, 1995.

[12] M. J. Carey. Panel on Semistructured Data, May 1997. Workshop on Management of Semistructured Data.

[13] R. Catell et al. *The Object Database Standard: ODMG-93.* Morgan Kaufmann, 1.1 edition, 1994.

[14] V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From Structured Documents to Novel Query Facilities. In *ACM SIGMOD Proceedings*, pages 313–324, Minneapolis, June 1994.

[15] V. Christophides, S. Cluet, and G. Moerkotte. Evaluating Queries with Generalized Path Expresssions. In *ACM SIGMOD Proceedings*, pages 413–422, Montreal, Canada, June 1996.

[16] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey. Algorithms for Deferred View Maintenance. In *ACM SIGMOD Proceedings*, pages 469–480, Montreal, Canada, June 1996.

[17] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems.* Benjamin/Cummings Publishing Company, Inc., Redwood City, California, second edition, 1994.

[18] M. Fernandez and D. Suciu. Optimizing Regular Path Expressions Using Graph Schemas. In *Proceedings of the 14th International Conference on Data Engineering (DE)*, Orlando, Florida, Feb. 1998.

[19] G. Gardarin, J. Gruser, and Z. Tang. A Cost Model for Clustered Object-Oriented Databases. In *Proceedings of the 21st International Conference on Very Large Databases (VLDB)*, pages 323–334, Zürich, Switzerland, Sept. 1995.

[20] 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 Object-Oriented Databases (DOOD)*, Montreux, Switzerland, Dec. 1997.

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

[22] T. Griffin and L. Libkin. Incremental Maintenance of Views with Duplicates. In *ACM SIGMOD Proceedings*, pages 328–339, San Jose, California, May 1995.

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

[24] A. Gupta, I. S. Mumick, and V. Subrahmanian. Maintaining Views Incrementally. In *ACM SIGMOD Proceedings*, pages 157–166, Washington, D.C., May 1993. Extended Abstract.

[25] E. N. Hanson. A Performance Analysis of View Materialisation Strategies. In *ACM SIGMOD Proceedings*, pages 440–453. ACM, 1987.

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

[27] D. Konopnicki and O. Shmueli. W3QS: A Query System for the World Wide Web. In *Proceedings of the 21st International Conference on Very Large Databases (VLDB)*, pages 54–65, Zürich, Switzerland, Sept. 1995.

[28] B. Lindsay, L. Haas, C. Mohan, H. Pirahesh, and P. Wilms. A Snapshot Differential Refresh Algorithm. In C. Zaniolo, editor, *ACM SIGMOD Proceedings*, pages 53–60, Washington, D.C., May 1986.

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

[30] J. McHugh and J. Widom. Query Optimization for Semistructured Data. Technical report, Stanford University Database Group, 1997. `http://www-db.stanford.edu/pup/papers/qo.ps`.

[31] A. O. Mendelzohn, G. A. Mihaila, and T. Milo. Querying the World Wide Web. In *Proceedings of the 4th International Conference on Parallel and Distributed Information Systems (PDIS)*, Miami Beach, Florida, Dec. 1996.

[32] S. Nestorov, J. Ullman, J. Wiener, and S. Chawathe. Representative Objects: Concise Representations of Semi-structured Hierarchical Data. In *Proceedings of the 13th International Conference on Data Engineering (DE)*, 1997.

[33] Y. Papakonstantinou, H. Garcia-Molina, and J. Ullman. MedMaker: A Mediation System Based on Declarative Specifications. In *Proceedings of the 12th International Conference on Data Engineering (DE)*, pages 132–141, New Orleans, Louisiana, Feb. 1996.

[34] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object Exchange across Heterogeneous Information Sources. In *Proceedings of the 11th International Conference on Data Engineering*, pages 251–260, Taipei, Taiwan, Mar. 1995.

[35] Y. Papakonstantinou and V. Vassalos. Query Rewriting using Semistructured Views. Stanford University Technical Report, 1998.

[36] R. Ramakrishnan. *Database Management Systems*. McGraw-Hill, 1997.

[37] N. Roussopoulos, C. M. Chen, S. Kelley, A. Delis, and Y. Papakonstantinou. The Maryland ADMS Project: Views R Us. *Bulletin of the Technical Committee on Data Engineering*, 18(2):19–28, June 1995.

[38] N. Roussopoulos and H. Kang. Principles and Techniques in the Design of ADMS±. *IEEE Computer*, pages 19–25, Dec. 1986.

[39] E. A. Rundensteiner. MultiView: A Methodology for Supporting Multiple Views in Object-Oriented Databases. In *Proceedings of the 18th International Conference on Very Large Databases (VLDB)*, pages 187–198, Vancouver, Canada, Aug. 1992.

[40] E. A. Rundensteiner, H. A. Kuno, Y. Ra, V. Crestana-Taube, M. C. Jones, and P. Marron. The MultiView Project: Object-Oriented View Technology and Applications. In *ACM SIGMOD Proceedings*, 1996.

[41] M. Rys. *Materialisation and Parallelism in the Mapping of an Object Model to a Relational Multi-Processor System.* Diss. ETH Nr. 12282, ETH Zurich, 1997.

[42] M. Rys, M. C. Norrie, and H. Schek. Intra-Transaction Parallelism in the Mapping of an Object Model to a Relational Multi-Processor System. In *Proceedings of the 22nd International Conference on Very Large Databases (VLDB)*, Mumbai (Bombay), India, Sept. 1996.

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

[44] C. Souza, S. Abiteboul, and C. Delobel. Virtual Schemas and Bases. In *EDBT Proceedings*, Cambridge, U.K., 1994.

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

[46] J. D. Ullman and J. Widom. *A First Course in Database Systems.* Prentice Hall, 1997.

[47] Y. Zhuge and H. Garcia-Molina. Graph Structured Views and Their Incremental Maintenance. In *Proceedings of the 14th International Conference on Data Engineering (DE)*, 1998.

A Maintenance Statement Subquery

Consider Figure 19(a) where object o is reached via path expressions p and q. Suppose that o has a single child with label L which appears within the view only because the path $q.L$ appears in the *with* clause. Figure 19(b) shows the state of the database after some update has invalidated the path q. If we destroy objects and removed edges within the adjunct based soley on reachability then the L subchild of o would remain, as shown in (b). However, since the path expression in the *with* does not contain $p.L$ then recomputation of the view, shown in (c), illustrates that (b) is inconsistent.

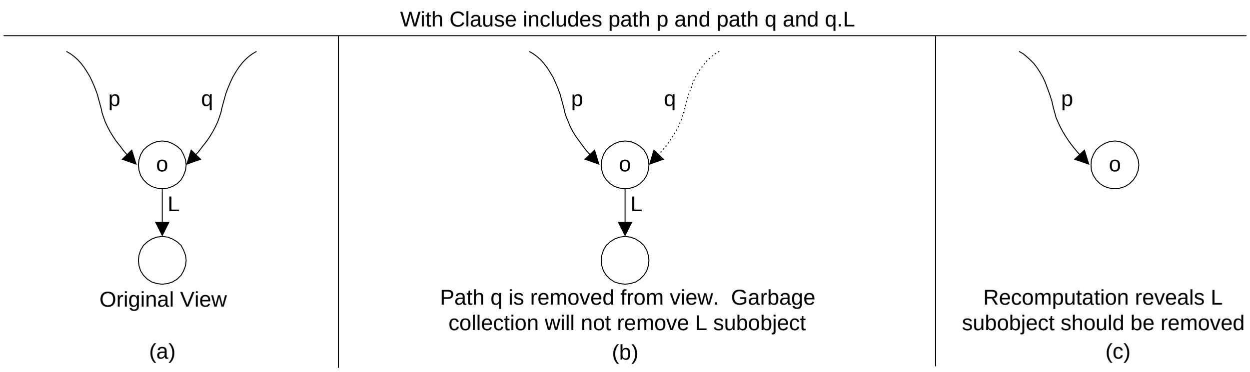

Figure 19: Motivation for subquery in maintenance statement