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
Serge.Abiteboul@inria.fr

‡ Stanford University
Stanford, CA 94305, USA
Firstname.Lastname@cs.stanford.edu
http://www-db.stanford.edu/lore

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 proposes an incremental maintenance algorithm 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. Our algorithm produces a set of queries that compute the updates to the view based upon an update of 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 updates.

1 Introduction

Database views increase the flexibility of a database system by adapting the data to user or application needs [37, 44]. Views are frequently materialized to speed up querying when the underlying data is remote or response time is critical [28, 9]. Once a view is materialized, however, its contents must be maintained in order to preserve its consistency with the base data. Maintenance can be performed either by recomputing the view contents from the database or by computing the incremental updates to the view based on the updates to the database. In this paper, we study the maintenance of materialized views for semistructured data. We propose a simple view specification mechanism and an algorithm for incremental maintenance. We then demonstrate the algorithm's strengths (and weaknesses) with a maintenance cost analysis.

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 [32, 1, 34, 10]. Views over semistructured data can be used to filter the data and to restructure (or provide structure to) it [34]. 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, a view is the only way in which we can restructure semistructured data that is outside of our control.

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 [13]. A materialized view can be used to isolate the data of interest, allowing subsequent queries to run over a smaller, often more structured, data set. Materialized views can also be used to rewrite queries over the base data and improve the query performance [36]. Furthermore, 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 [9, 24, 23, 38, 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, languages for specifying and querying materialized views are significantly more intricate [2, 7, 42, 41, 39].

Previous results on incremental view maintenance

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

Proceedings of the 24th VLDB Conference
New York, USA, 1998

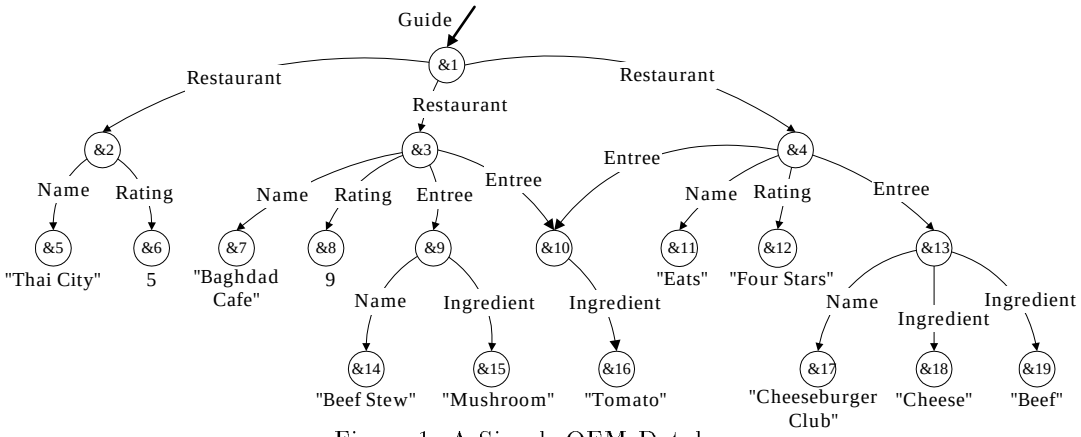

Figure 1: A Simple OEM Database

for object databases [39, 40] 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 along with the absence of a schema make it difficult to detect if a particular update affects a view. Gluche and colleagues [20] use a view maintenance scheme that is limited to linear OQL view definitions. Because of subobject sharing, most nontrivial semistructured view definitions are not linear, making their approach inapplicable in our context.

Suciu [43] 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 [45] 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 views over tree-structured databases, while our approach handles joins and arbitrary graph-structured databases.

Our work is based on the Object Exchange Model (OEM) [35] for semistructured data. In OEM, a database is a directed, labeled graph. OEM has strong similarities to XML [32], a proposed standard for a universal format for data on the Web. Our view specification language is based on the Lorel query language for OEM [5]. 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. The distinction between the two sets of objects is invisible to the user – it is only used to simplify the discussion of the incremental maintenance algorithm. 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 faster 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.

The presented maintenance algorithm can be used both for immediate maintenance [9] and for deferred maintenance [38, 17] of the views. The techniques presented here are also applicable to other query languages for semistructured data [12], for the Web [27, 31], and (to some extent) to query languages for hypertext documents [15, 6].

2 View Specification

We use the Lore system [29] to investigate materialized view maintenance over semistructured data. We now introduce OEM, the data model used by Lore; the Lorel query language; the view specification language; and the update operations. [5] and [3] provide further details on Lorel and the view specification language, respectively.

2.1 The OEM Data Model

An OEM database is a labeled, directed graph such as the small example database given in Figure 1. Each vertex in the graph represents an *object*; 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 (in the Lore system) have a set of $\langle label, subobjectoid \rangle$ 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 {⟨*Entree*, &10⟩, ⟨*Name*, &11⟩, ⟨*Rating*, &12⟩, ⟨*Entree*, &13⟩}. *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 as 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 The Lorel Query Language

Lorel, for Lore Language, uses the familiar *select-from-where* syntax of SQL, and can be considered an extension to OQL [14] that provides powerful path expressions for traversing the data and extensive coercion rules for a more forgiving type system. Both features 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. In this paper, 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 in Lorel

A view specification statement in Lorel [3] imports objects and edges from a source database into a view. In addition, new objects and edges can be created 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 a variable appearing in the *select* clause. Each object and edge along a path in the *with* clause is included in the view. Intuitively, the *select-from-where* statement returns a flat set of objects. The *with* clause imports some of the structure of the database in the view. It is a compromise between returning everything or nothing reachable from selected objects.

We call the objects included in the view 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, in this paper, we concentrate on views defined by a single statement.

The view specification in Example 2 defines a view for the result of the query in Example 1 (now written in an OQL-like syntax [5]) along with all *Name* and *Ingredient* subobjects of each *Entree*.

Example 2 (Canonical View Specification)

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 discovered 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 present.

2.4 Materialized Views

We now explain how views are materialized in Lore, using a simple *top-down* query evaluation strategy [29]. First, the *from* then *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 imported database object is represented by a new *delegate* object.

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.

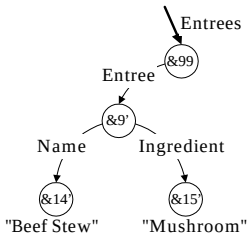

Figure 2: The materialized view for Example 2

2.5 Update Operations

The Lorel update statements [5] 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 $\langle Chg, o_1, OldVal, NewVal\rangle$.

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 its 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, in our examples the *select* clause is of the form "*select* y" (generalizing for any *select* clause is straightforward).

3.1 Overview of the 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.

The view specification S, the database update U, and the database state DB' after the update are used to compute the view maintenance statements in Lorel syntax.[1] These statements generate the sets

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

1. View specification statement S:
   select v_i
   from $v_0.L_1\ v_1,\ \ldots,\ v_j.L_k\ v_k,\ \ldots,\ v_{n-1}.L_n\ v_n$
   // v_j can be any variable that
   // already appeared in the sequence
   where $conditions(v_1, \ldots, v_n)$
   with $v_i.L_{11}\ w_{11},\ w_{11}.L_{12}\ w_{12},\ \ldots,\ w_{1(p-1)}.L_{1p}\ w_{1p},$
   $u_j.L_{j1}\ w_{j1},\ \ldots,\ w_{j(k-1)}.L_{jk}\ w_{jk},\ \ldots,$
   $w_{j(q-1)}.L_{jq}\ w_{jq}$
   // 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: Incremental maintenance algorithm input

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 $y =$ *"Mushroom"*, as in SQL.

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

Figure 4 outlines the steps of the view maintenance algorithm. 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}. The statements 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 sim-

```
function RelevantVars(Update U, View specification S)
// If updated object is not in RelevantOids, then it's not
// relevant. RelevantOids is {⟨oid, queryvariable⟩}.
if ⟨o1(U), ·⟩ ∉ RelevantOids then return ∅;
// Find out which variables are relevant to the update
vars ←∅; relvars ←∅;
foreach v ∈ variables(S) do
   // If updated object is not in RelevantOids, then it's
   // not relevant
   if ⟨o1(U), v⟩ ∈ RelevantOids then vars ←vars ∪ {v};
// If update is atomic change, do simple syntactic check
if type(U) = Chg then
   foreach v ∈ vars do
      // Let constants(S, v) be the constants appearing
      // in S compared to v, e.g., using = here
      foreach c ∈ constants(S, v) do
         // See if there's a predicate in the view spec
         // whose value may have changed
         if (OldVal(U) ≠ c and NewVal(U) = c) or
            (OldVal(U) = c and NewVal(U) ≠ c) then
            relvars←relvars∪{v};
elserelvars ←vars;
return relvars;
```

Figure 5: *RelevantVars* returns the view specification variables for which the update U is relevant.

ple relevance checks. We use an auxiliary data structure, *RelevantOids*, to keep information that would be available from the schema in a structured database. *RelevantOids* contains the object identifier of every object touched during the evaluation of a view specification, 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 and the update can be ignored.

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 function *RelevantVars*, which determines the set of variables appearing in the query that the update could be bound to given a view specification.

For example, suppose that the value of object &5 in Figure 1 is changed from "Thai City" to "Hunan Wok". 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 Maintenance Statements

We now describe how to generate the maintenance statements for each type of update: edge insertion, edge deletion, or atomic value change. Consider first the edge insertion and edge deletion cases. 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 appearing in the view specification. $OneStepPath_{from}$, $OneStepPath_{prim}$, 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* set 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.

Example 3 (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;
```

This maintenance statement can be evaluated more efficiently than the original view specification, as we show in Section 4. □

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

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.