Implementing Incremental View Maintenance in Nested Data Models

Akira Kawaguchi[1] **Daniel Lieuwen**[2]
Inderpal Mumick[3] **Kenneth Ross**[1]

[1] Columbia University, {akira,kar}@cs.columbia.edu
[2] Bell Laboratories/Lucent Technologies, lieuwen@research.bell-labs.com
[3] Savera Systems (work performed at AT&T Laboratories), mumick@savera.com

Abstract. Previous research on materialized views has primarily been in the context of flat relational databases—materialized views defined in terms of one or more flat relations. This paper discusses a broader class of view definitions—materialized views defined over a nested data model such as the nested relational model or an object-oriented data model. An attribute of a tuple deriving the view can be a reference (*i.e.*, a pointer) to a nested relation, with arbitrary levels of nesting possible. The extended capability of this nested data model, together with materialized views, simplifies data modeling and gives more flexibility.

Simple extensions of standard view maintenance techniques to the nested model would do too much work for maintenance: a change in a nested set would re-process the entire nested set, not just the changed parts. We show how existing incremental maintenance algorithms can be extended to maintain the views without performing this additional work.

We describe the implementation of these techniques in the SWORD interface to the Ode database system. The implementation is based on the representation of nested structures by classes and the use of an SQL-like language to define materialized views. We outline the data structures and algorithms used in the implementation and examine performance.

This is one of the first pieces of work to explore the applicability of materialized views over complex objects.

1 Introduction

Most past research on materialized views has focused on high level incremental algorithms for updating materialized views efficiently when the base relations are updated [9, 37, 8, 31, 10, 17, 16, 25, 40]. Those studies are in the context of first-normal-form relational databases—materialized views are defined in terms of one or more relations using a relational query language. In this paper, we allow a materialized view to be defined over a nested data model, where a tuple can have arbitrarily deeply nested structures. For the simplicity of our discussion

we assume in the rest of paper that all materialized views are *snapshot* views [2], *i.e.*, views that are maintained when an explicit maintenance request is made.

The extension of the relational model to non-first-normal-form goes back to [27], which introduced the concept of nesting. The work of, *e.g.*, [18, 19, 4, 1, 38, 39] further investigated nested properties including the algebra and calculus. Query languages and implementation issues are discussed in, *e.g.*, [5, 15, 30, 24, 33, 34]. The above research aims to extend the relational model to directly represent data that is usually organized hierarchically such as in CAD/CAM and multimedia applications. The idea of having relation names as arguments is presented in [32]. This extension allows relation names to be stored as attributes of other relations (essentially using the name as a pointer), and allows access to one relation or another based on the value of some other attribute. Thus, nested relations can be implemented without directly embedding relations in tuples. Object-oriented databases (OODBs) share this property. An object often has a complex sub-object that is referenced by an object identifier. Further, the nested structures are usually *unnamed* (*i.e.*, the structure has no external name by which it can be referenced; it can be referenced only by navigating through the containing relation/object). In this paper, we consider nesting by reference, as described for the relational and object-oriented models above. We leave the case of embedded nested values within a tuple or object for further work.

We consider the problem of maintaining materialized views over such nested structures. A simple extension of maintenance algorithms for views over normal form relations to views over nested structures yields inefficient algorithms. For example, suppose that p is a binary relation whose second argument is a nested relation with a single attribute. Let q be another binary relation. Let V be a view defined $V(X,Y) \stackrel{def}{=} p(X,Z), Z(W), q(W,Y)$. (in a HiLog style language [11, 32]). Thus, V is a join of p with q on the elements of p's nested attribute. Suppose that $p(a,s)$ are the tuples in the extension of p, and that $\{\langle 1\rangle, \langle 2\rangle, \ldots, \langle n\rangle\}$ are the values in the nested relation s. Let us modify s by adding $\langle n+1\rangle$ to the extension of s. A straightforward application of known view maintenance techniques could "undo" the effect of the whole set $\{\langle 1\rangle, \langle 2\rangle, \ldots, \langle n\rangle\}$ and then "redo" the effect of $\{\langle 1\rangle, \langle 2\rangle, \ldots, \langle n+1\rangle\}$. Much of this work is unnecessary—only work corresponding to inserting $\langle n+1\rangle$ is necessary. We develop auxiliary data structures to find such relevant changes efficiently.

The motivation of this study is the development of the SWORD interface [29] to the Ode OODB [3]. SWORD provides an SQL-like declarative language to define materialized views compositionally and hierarchically on collections of Ode objects. SWORD supports transparent incremental maintenance of those

views using the incremental view maintenance algorithm of [17]. The algorithm accumulates the changes made to each base relation of the materialized view in a *log*, a sequential file associated with a base relation. It uses the logs to maintain views. There is only one log for each base relation regardless of the number of the views it derives (*i.e.*, the log is shared by a set of views to be maintained). A log entry holds an *identifier* of the changed tuple (the change may be an insertion, deletion and modification). The data structures used to implement logs are described in [13].

Two problems arise when we enhance the materialized views in SWORD to work with a nested data model. First, how should the log structure be extended? Second, how can one capture the change made in a referenced complex relation? The implementation creates an entry in relation T's log in response to each update to a tuple of T. Suppose that a tuple t of T has a pointer to another relation R whose tuple r is updated. The change to r must be identified both for referring relation T and for referred relation R. How can we identify all such referring tuples t? Recording the change in the log of the referring relation T is not easy since the pointer value in the tuple t will not change unless the whole referred relation R is dropped or newly created. Furthermore, since nested sets are referenced through pointers, the insertion of a single tuple may affect many containing objects—thus, the space required for the log (and, consequently, the time for creating and scanning log records) may balloon. A more subtle problem arises if we consider the efficiency of the maintenance that involves implicit join operations between flat and nested attributes or between nested attributes.

Paper Outline: Section 2 reviews the nested data model [39, 33] used in this paper and then investigates maintaining materialized views over nested data models. The implementation architecture for the non-nested case is reviewed in Section 3; Section 4 presents the necessary extensions for handling nested structures. Section 5 discusses performance results. Related work is discussed in Section 6; Section 7 presents conclusions.

2 Maintaining Materialized Views over Nested Data

2.1 Notation and Definitions

We describe our techniques in the nested relational model context. It should, however, be understood that the techniques developed apply also to OODBs.[4]

[4] While we did not discuss the use of our algorithms for OODBs over data with cyclic references, our algorithms extend [17]'s counting algorithms which can incrementally

Following [39], a database scheme is a collection of rules of the form $R = [R_1, ..., R_k]$. Objects $R_1, \ldots, R_k$ are called *names*. An object is a *higher-order* name if it appears on some rule's left-hand side; else, it is a *zero-order* name. Nested schemes may contain any combination of zero or higher-order attributes on the right-hand side of the rules as long as the scheme remains non-recursive.

An instance of $R = [R_1, ..., R_k]$ is a collection of tuples such that each tuple contains arbitrary combinations of values, or *indirect references* (relation names or pointers to relations) based on the *types* of names $R_1, ..., R_k$. Some object, R_i say, may be a higher-order name, in which case the instance of R_i will be a recursive expansion of the rules in the right-hand side. Hence a tuple of R can be viewed as having arbitrarily deeply nested relation instances. A relation is a stored instance of the database scheme. We call a relation R, defined by schema $R = [R_1, ..., R_k]$, a *nesting relation* if at least one of its attributes $R_1, ..., R_k$, say R_i, is a higher order name. We also call the higher order name, R_i a *nested relation*. A relation that is not defined as a view is called a *base relation*.

Materialized Views: A *view* V is a relation that is defined using a query Q over some set of relations $\{R_1, ..., R_n\}$, denoted as $V \stackrel{def}{=} Q(R_1, ..., R_n)$. The query is said to be defined over relation R_i if R_i appears in the query definition, or if the query follows a reference to a relation of type R_i. $R_1, \ldots, R_n$ are called the *referenced* relations, and may be base relations or materialized view relations. The referenced relation can be a nesting relation or a nested relation. A *materialized view* is a view whose tuples are physically stored in the database. A *virtual view* is an unmaterialized view; it is computed when it is needed for a query using the R_i. (For brevity, we do not discuss virtual views in the rest of this paper until Section 5 on performance.) □

Log and Delta: For every referenced relation R_i, an additional structure, which we call a *log*, contains the changes made to R_i. Any update to a referenced relation causes a corresponding log entry to be created. The log is used to construct a set of tuples that will be used for the incremental maintenance of view V. We call this set a *delta*, denote as ΔR_i. If multiple views $V_1, \ldots, V_n$ are defined using relation R_i, the log must be capable of constructing $\Delta R_i(V_j)$ (*i.e.*, a delta of R_i with regard to view V_j), for $1 \leq j \leq n$. □

View-Dependency Graph: The dependency graph G of a view V is a graph with a node for each relation referenced in the view definition, a node labeled V, and a directed edge from the node for each referenced relation to the node for V.

---

maintain recursively defined views over circular data. Thus, we believe our techniques our applicable to arbitrary data in OODBs. Furthermore, our algorithms directly apply to non-recursive view definitions over circular data without extensions.

The dependency graph shows how the view is derived from base relations and/or other views. The view-dependency graph $\mathcal{G}$ of a database schema is the union of the dependency graphs for all the views in the schema. The view-dependency graph shows how all the views in the schema are derived from each other and from base relations. □

2.2 Incremental Change Computation

We consider the counting algorithm of [17] for view maintenance. The counting algorithm keeps a count of the number of derivations for each view tuple. For instance, given a join view $V \stackrel{def}{=} R_1 \bowtie R_2 \bowtie R_3$, the counting algorithm derives the following algebraic equation to compute the changes ΔV to view V:

$$\Delta V = \Delta R_1 \bowtie R_2^{new} \bowtie R_3^{new} \quad \uplus \quad R_1^{old} \bowtie \Delta R_2 \bowtie R_3^{new} \quad \uplus \quad R_1^{old} \bowtie R_2^{old} \bowtie \Delta R_3$$

where ΔR_i is the set of insertions and/or deletions to relation R_i, R_i^{old} is the old (or pre-update) state of the base relation R_i (before the updates of ΔR_i are applied to R_i), and R_i^{new} is the new (or post-update) state of the base relation R_i (after the updates of ΔR_i are applied to it). The $\uplus$ operator denotes bag union. In the set of changes, insertions (deletions) are represented with positive (negative) counts. Updates are represented by deletions of the before images, and insertions of the new tuples. The count value for each tuple is represented in the materialized view V, and the new materialized view is obtained by combining the changes ΔV with the stored value of view V as follows: Positive counts are added in; negative counts are subtracted. A tuple with a count of zero is deleted.

A view defined over the nested data model can be maintained by the counting algorithm if we can define and identify the nested delta sets correctly. We assume that the view definition language is based on a query language such as HiLog [11].

EXAMPLE 2.1 Consider the schema Emp = [Nm, Dependent], Dependent = [D-Nm, Age], Health-Ins = [D-Nm] with the following database extension: Emp = {⟨Fred, D1⟩, ⟨Mary, D2⟩}; Health-Ins = {⟨Dave⟩, ⟨Jane⟩}; D1 = {⟨Dave, 85⟩, ⟨Bob, 10⟩, ⟨Jane, 5⟩}; D2 = {⟨Dave, 85⟩, ⟨Alice, 3⟩}. ($D1$ and $D2$ are ids of otherwise unnamed relations of type Dependent). Consider the view V which contains the employees' dependents who have their own health insurance. V is defined by the following HiLog expression: $V(X, Z) \stackrel{def}{=}$ Emp$(X, Y), Y(Z, A)$, Health-Ins(Z). The materialization of V is {⟨Fred, Dave⟩, ⟨Fred, Jane⟩, ⟨Mary, Dave⟩}. To respond to a change in the relation Health-Ins, we apply the counting algorithm. For this kind of update, we can intuitively express ΔV as $\Delta V(X, Z)$ =

$\text{Emp}^{old}(X,Y) \bowtie Y^{old}(Z,A) \bowtie \Delta\text{Heath-Ins}(Z)$. Note, Y^{old} is not precise since Y is a variable. We discuss what Y^{old} means below. Deleting Jane from Health-Ins (written as $\Delta\text{Heath-Ins} = \{\langle\text{Jane}\rangle^{-1}\}$) yields $\Delta V = \{\langle\text{Fred, Jane}\rangle^{-1}\}$. □

Example 2.1 shows that handling updates to non-nested relations is straightforward. We now consider updates to the nested relations.

Meta-Relation, Extraction Function: Consider a nesting relation $R = [\ldots, S, \ldots]$ where S is a higher-order name. For each such nested S in the database we define a *meta-relation* u_S as follows. $u_S(X)$ is true precisely when X is the id of a nested relation in the database extension, of the same type as S. For each such nested relation S, we define an *extraction function* $e_S(X, A_1, \ldots, A_k)$, where S is defined by the rule $S = [A_1, \ldots, A_k]$. (In other words, $A_1, \ldots, A_k$ are the attributes of relations of type S.)

$$e_S(X, A_1, \ldots, A_k) = u_S(X) \bowtie X(A_1, \ldots, A_k) \qquad \square$$

For the nested Dependent tables of Example 2.1, we would have $u_{\text{Dependent}} = \{D1, D2\}$ and $e_{\text{Dependent}} = \{\langle D1, \text{Dave}, 85\rangle, \langle D1, \text{Bob}, 10\rangle, \langle D1, \text{Jane}, 5\rangle, \langle D2, \text{Dave}, 85\rangle, \langle D2, \text{Alice}, 3\rangle\}$.

Consider again a nested relation $R = [\ldots, S, \ldots]$ where S is a higher-order name. Let a view V be defined using R and S, so that $V' = R(\ldots, X, \ldots) \bowtie X(\ldots)$ is a subexpression within V. (Here, X is a variable appearing in the position of the nested attribute of type S.) Rather than doing maintenance on this subexpression directly, we maintain the equivalent subexpression

$$R(\ldots, X, \ldots) \bowtie e_S(X, \ldots). \tag{1}$$

The benefit of this transformation is that we no longer have a HiLog variable as a relation name. Treating e_S as a relation, we can express $\Delta V'$ using the counting algorithm as $\Delta R(\ldots, X, \ldots) \bowtie e_S^{new}(X, \ldots) \uplus R^{old}(\ldots, X, \ldots) \bowtie \Delta e_S(X, \ldots)$ In the event that there are multiple levels of nesting, so that S itself contains a nested relation T as an attribute ($S = [\ldots, T, \ldots]$), we can recursively express Δe_S in terms of S and e_T as above.

The expression (1) also gives us a hint about where to keep the delta information for unnamed nested relations. The extraction function can be thought of as a materialized view. However, only the log of the view, not the view's extension, is stored. Conceptually, we should keep the delta information for nested relations of a given type S in a single place associated with S. For the efficient incremental change computation, $\Delta e_S(X, \ldots)$ must be quickly found in the database, so that the system can avoid scanning all tuples in R. In Section 4 we show how this is implemented using "nested descriptors."

EXAMPLE 2.2 The view V in Example 2.1 can be incrementally computed as follows:

$$\begin{aligned}\Delta V(X,Z) = {} & \Delta\text{Emp}(X,Y) \bowtie e^{new}_{\text{Dependent}}(Y,Z,A) \bowtie \text{Heath-Ins}^{new}(Z) \\ & \uplus \text{Emp}^{old}(X,Y) \bowtie \Delta e_{\text{Dependent}}(Y,Z,A) \bowtie \text{Heath-Ins}^{new}(Z) \\ & \uplus \text{Emp}^{old}(X,Y) \bowtie e^{old}_{\text{Dependent}}(Y,Z,A) \bowtie \Delta\text{Heath-Ins}(Z)\end{aligned}$$

Suppose Fred changed his name to Greg after V's materialization, and that he no longer has a dependent Dave. So $\Delta\text{Emp} = \{\langle \text{Fred}, D1\rangle^{-1}, \langle \text{Greg}, D1\rangle^{+1}\}$ and $\Delta e_{\text{Dependent}} = \{\langle D1, \text{Dave}, 85\rangle^{-1}\}$. Using the expression above, we compute ΔV as $\{\langle \text{Fred}, \text{Dave}\rangle^{-1}, \langle \text{Fred}, \text{Jane}\rangle^{-1}, \langle \text{Greg}, \text{Jane}\rangle^{+1}\}$. □

Observe that the incremental work needed to maintain a view over nested data is in principle proportional to the *changes* in the contents of the nested relations, and not proportional to the size of the nested relations themselves. In the next two sections we outline how we achieve this performance level *in practice.*

3 Implementation for First Normal Form Relations

This section briefly reviews the implementation of our view maintenance system presented in [13]. The extensions to handle nested data are discussed in Section 4.

A part of the effort in [13] addresses scalability and efficiency concerns since materialized views introduce additional system overheads (*e.g.,* space for storing log, log update time, view maintenance time). Some of the requirements are: (a) The overhead of making log entries must be independent of the number of views. Thus, we rule out a design based on a separate log for each relation-view pair. (b) Time required to compute the delta must be proportional to the size of the relevant log. This prevents us from scanning the entire log to determine the portion of the log's changes. (c) The total space used to store all the logs in the system should be proportional to the number of updates that need to be propagated into materialized views. Thus, log entries should not be replicated, and old log entries must be discarded. (d) Queries over a relation should not be slowed down when views are defined over the relation. (e) Given a view, we should be able to quickly check whether it needs to be refreshed.

A relation (base or materialized view) is implemented as a *collection* class in Ode. A collection has a *materialization* containing tuples and a *descriptor.* A tuple is implemented as an Ode object. The descriptor holds meta-information about the collection, such as the creation date, a pointer to the materialization,

and other pointers needed to support materialized views. The collection class provides a number of member functions (methods) such as `insert()`, `remove()`, and `replace()` that can be invoked from the O++ interface to Ode. The `insert()` function creates a new tuple with the given values, and inserts it into the materialization. The `remove()` function removes a tuple from the materialization by marking it as removed, and placing it in a pool of removed tuples. The tuple must stay in this pool until the effects of its removal are propagated to all views defined on the relation, after which point it can be garbage collected. The `replace()` function updates an existing tuple, and stores the pre-update value in a newly created tuple that is placed in the pool of removed tuples. A separate `Iterator` class is provided to iterate over the materialization of any relation.

Extracting Relevant Changes from Logs: The changes made since the last maintenance operation on a view V are the only log entries relevant for a maintenance operation on V. The `DeltaIterator` class is provided to iterate over the relevant changes of a relation R for V. When a `DeltaIterator` object is created for a given relation/view pair, we scan R's log starting from the last maintenance pointer for the view V stored in R's descriptor, and build an in-memory hash table by hashing the *oid* in the log entry. A bucket contains an in-memory copy of the log entry itself. The log is not modified. Hashing is used to compute the net effect of the changes in the log by eliminating and/or collapsing redundant log entries due to insert-remove pairs, replace-remove pairs, replace-replace pairs, and insert-replace pairs [35, 20]. Since the creation of the hash table requires time proportional to the size of the log relevant to the view, and using the hash table requires even less time, we clearly satisfy the efficiency requirement (b).

4 Handling Nested Structures

This section describes key ideas to efficiently capture the incremental changes to nested components. Nested relations in SWORD are defined using an O++ class with an attribute that is an embedded collection class or a pointer to another collection class. We insisted both that view maintenance be transparent to the user and that it not do any (significant amount of) extra work if no views exist. Our algorithms trigger transparent view maintenance work only if the user instantiates a view using the view definition language.

Efficiency Requirements: A major challenge is efficiently detecting changes made to the elements that are in a nested attribute. We must establish associations between such elements and owner tuples. This must be done by the

time we construct the extraction function of Section 2.2 for maintenance. In an implementation, a change made to an element of the internally nested relation could be placed in the log of that nested relation. A naive approach is to propagate this change to the log of every *nesting* relation whose tuples reference the nesting relation being changed. This approach imposes a heavy burden on the update transaction since the transaction needs to find all owner tuples in order to insert the log entries. If a large number of tuples contain references to the updated relation, or if the nesting level is deep, the log operation can become very expensive. Log space may blow up. Thus, additional requirements imposed on the implementation are to find (1) an efficient way to collect and store the changes made to the nested elements, (2) an efficient way to establish nesting-nested associations, and (3) the smallest possible update transaction overhead. Section 3's requirements apply as well.

4.1 Capturing Changes in Nested Tables

To meet requirement (1), we create a system collection descriptor that owns a log for *all* nested relations of the same scheme (or class) definition. The descriptor is created when a view involving these nested relations is initially materialized. We call this descriptor a *nested descriptor*. Each SWORD view definition is inspected to find any nested relations (SWORD, like HiLog, requires specifying a variable that ranges over nested relations with the same scheme type). The tuples in the set of individual nested relations are of the same type. Thus, all log entries in the nested descriptor's log are of the same type. After creating a nested descriptor D, every update to a corresponding nested relation N is recorded in D's log (instead of in N's log). When the update transaction calls the `insert()`, `remove()` or `replace()` method of N, the method checks if a nested descriptor, D, of the same relation type exists. If so, it inserts a log entry into D's log. The insertion is done only once. This check is the only overhead to the update transaction, and this overhead is negligible. Thus, requirement (3) is met.

Consider a database scheme defined by the following collection of rules. $R = [T, \ldots], S = [U, \ldots], T = [U, \ldots], U = [\ldots]$. Figure 1 illustrates the structure of the relations (R, S, T_1, T_2, U_1, U_2) and the associated logs (disregard the indices for now). According to the figure, tuples of the relation R have pointers to T_1 and T_2, the tuple of S has a pointer to U_1, and so on. Boxes labeled R, T_1, T_2, ... in the figure represent *relation descriptors* since they contain control information about relations (*e.g.*, information about each relation's indices/views).

Suppose also that two views V_1 and V_2 are defined over R, T, and U (S is not used for the view definition). For example, V_1 is defined as a query of

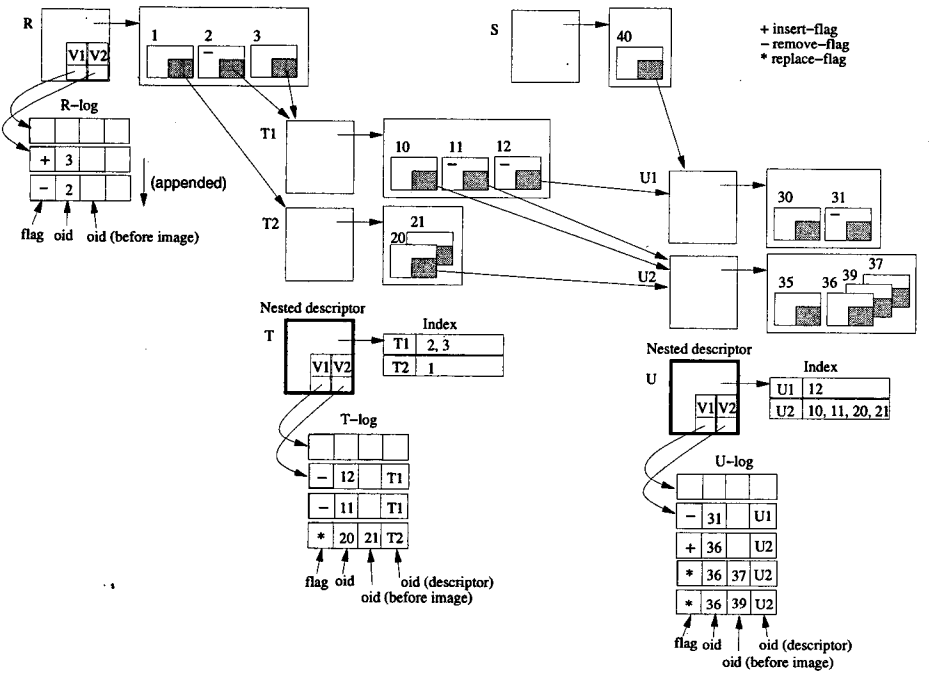


Fig. 1. Log Structure for Nested Tables

$V_1 \stackrel{def}{=} R(X,\ldots) \bowtie X(Y,\ldots) \bowtie Y(\ldots)$, where a variable X is in the position of the nested attribute T and a variable Y is in the position of the nested attribute of U. In Figure 1, the relation descriptor for R has pointers (labeled V_1 and V_2) into the log. The pointer for V_1 (V_2) points to the last log entry that has been applied to view V_1 (V_2) to have brought it up to date. R contains two current tuples with $oid = 1, 3$ which we will refer to as t_1 and t_3 ($oid = 2$ is a deleted tuple that must be kept around for view maintenance purposes for the time being). t_1 references the nested set T_2. T_2 contains a single active tuple with $oid = 20$, t_{20}. (Tuple t_{20} originally had the same contents as the tuple with $oid = 21$, t_{21}. When t_{20} was modified, a copy, t_{21} was made first for view maintenance. The last log entry for T's nested descriptor indicates that.) Notice that the log entry includes the *oid* of the relation descriptor. The principles of [13] are used to extract relevant changes from the log for views V_1 and V_2. For example, the maintenance of V_1 and V_2 requires computing the extraction function using t_1.

Note that t_1 is not found in the log of R since the changes were made to T_2 and U_2. Such changes are efficiently found in the logs of nested descriptors T and U. The details will be discussed in Section 4.3.

4.2 Nested-Nesting Associations

To meet requirement (2), we consider an index structure that is associated with each nested descriptor. The reason to have this index is to efficiently expand all changes (particularly those that have occurred in deeply nested elements) into the extraction function of Section 2.2 for maintenance. The index will be created at the same time as the corresponding descriptor. Subsequent view maintenance operations will maintain the index entries. The query transactions never use this index. This structure is similar to those used in nested indices and a path indices [7], but the motivation is quite different since those indices are used to answer queries.

Each key in the index is the *oid* of a relation descriptor (*e.g.*, T_1, T_2) that is directly referenced by some tuple in a nesting relation. This allows the *oid* of each tuple that is directly *nesting* another relation to be found (*e.g.*, it allows us to find that t_1 contains T_2). During the first view materialization operation, the index entries are initialized using mappings from the *oid* of an owned relation descriptor (*e.g.*, T_2) to the *oids* of each tuple that directly owns this relation descriptor (*e.g.*, t_1). Every tuple in the nesting relation must be inserted into this index, whether or not it contributes to the current instance of the materialized view.[5] After the index is created, new nesting tuples may be inserted into a relation relevant to a view. The *oids* of these new tuples will not appear in the nested descriptor's index until view maintenance occurs. Thus, during the subsequent view maintenance, mappings for recently inserted tuples must be inserted into the index (*e.g.*, the pair mapping T_1 to t_3 was added to the T index when the log entry for the insertion of t_3 was processed during the previous maintenance operation on view V_2). Garbage collection sweeps every removed *oid* in the index that is no longer used for any view maintenance. Since this index never adds overhead to update transactions, we satisfy requirement (3). Similarly, log entries may indicate that a nesting tuple has been deleted, in which case its index entry must be removed (*e.g.*, the mapping from T_1 to *oid* = 2 must be removed

[5] This is because we scan each relevant log only once to maintain views. If a tuple that previously did not participate in the view was modified so that it could contribute to the join and it was not already in the index, we would have to add it in an initial pass over the relevant logs and then use it in a second pass over the logs.

when the log record for the deletion of *oid* = 2 is processed). A modification is treated like a deletion followed by an insertion. (Note, the implementation actually optimizes the conceptual index described above by storing a pointer to a list of backpointers in the nested descriptor itself to reduce index lookup time—*e.g.*, T1 contains a pointer to a list containing 2 and 3.)

In Figure 1, two indices for T and U are shown. Currently both indices have sufficient information to locate which tuple directly owns the *oid* of the relation descriptor. Suppose that a new tuple is inserted in R. Then the view maintenance process finds it in the log of R and inserts into the index of T (it may also add its nested set's contents to the index for U). Notice that tuples of S do not participate to the index since S is not defined in any view.

4.3 Incremental Computation

We use a hashing method similar to [35] to quickly compute the net effect of the changes in the log. We create hash buckets by following the nesting levels. Recall from Section 4.1 that the view V_1 in Figure 1 is defined as $V_1 \stackrel{def}{=} R(X,\ldots) \bowtie X(Y,\ldots) \bowtie Y(\ldots)$, where the variables X and Y are respectively bound to T and U. Algorithms of extracting the changes of R and producing the extraction functions of e_T and e_U are highlighted in the following three steps:

1. (Non-nested sets:) For each set mentioned by name in the view definition (*i.e.*, for each non-nested set such as R, scan the corresponding log to create hash buckets in the standard way (see Section 3). Also,
   1–1. If the entry contains the insert flag and the *oid* o_{new}, add a mapping (if none exists) from each of o_{new}'s nested sets to o_{new} in the indices of the corresponding nested descriptors. If no mappings from the nested set previously appeared in the index, continue this recursively for nested sets of the nested set.
2. (Nested sets:) For each nested set mentioned at the next nesting level in the view definition (*e.g.*, T bound by X), scan the nested descriptor's log to create hash buckets similarly. Furthermore, the descriptor *oid* (*e.g.*, T_1 and T_2) stored in each scanned log entry L is used to probe the contents of the index in order to determine if L affects the tuple that owns this nested set:
   2–1. L's *oid* o is used to probe the nested descriptor index. The *oid*s of all tuples mapped to by o that do not yet appear in the hash table are added to the hash buckets as in step 1 (except for those that are marked as deleted or which are used as old versions of tuples). If o is contained in a nested set, recursively probe the nested descriptor of its containing set

and add the containing tuples to the hash table if they are not already there. Continue doing this until no such containing tuples exist or they are all in the hash relation already.

3. Recursively apply step 2 to each of the relations at the next level of nesting (*e.g.*, U bound by Y).

Consider maintenance of V_1. The incremental computation scans the logs of R, T and U to build the net effect of the changes made to them. In step 1, the log of the non-nested set R is hashed, and $oid = 2, 3$ are added to the hash table.

In step 2, the log of the nested descriptor T is scanned, and $oid = 11, 12, 20$ are added to the bucket. Also, step 2–1 adds $oid = 1$ to the hash table created for R in step 1 ($oid = 2, 3$ are already there. $oid = 1$ is identified by probing the index with T_2's descriptor *oid*—R's tuple t_1 is not updated itself but its nested set T_2 is updated). In step 3, the log of U is scanned to create hash buckets. (This is the recursive step of 2.) $oid = 31, 36$ are inserted into U's hash table. In step 2–1, $oid = 10$ is added to the hash table created for T because $oid = 10$ does not appear there yet. $oid = 20, 11$ are in the bucket already; $oid = 21$ is an old version and so is excluded.

In the end, the hash tables of R, T, and U contain the *oids* $\{1, 2, 3\}$, $\{10, 11, 12, 20\}$, and $\{31, 36\}$, respectively. The extraction functions then produce descriptor *oid*/tuple *oid* pairs by looking into these hash tables. For example, $\Delta e_T = \{(T_1, 10), (T_1, 11), (T_1, 12), (T_2, 20)\}$. The incremental join computation of ΔV_1 uses these descriptor *oids* to find matching pointer values bound to X (the joined values from the nested set can be obtained by the tuple *oids*). These hash tables are also used to compute pre-update states of the database during the incremental view maintenance [13].

5 Performance Study

This section describes an experimental performance study on top of the disk-based Ode<EOS> database system. The experiments compare the performance of maintaining snapshot views over data that is naturally represented using nesting/nested relationships. All experiments were run in single user mode on a 128 MB, 200 MHz, UltraSparc II station (running Solaris Sun OS5.5.1 operating system). The database was kept on the local disc attachment to eliminate NFS delays.

Experimental Setup: We build databases containing base tables and snapshot materialized views, run 1,000 transactions against each database, and gather various statistics for the set of 1,000 transactions.

Databases: Our experiments use materialized views of the following form:

$\texttt{Nest}(A, C) \stackrel{def}{=} \texttt{base1}(A, N), N(B, C), pred(B)$

$\texttt{Flat}(A, C) \stackrel{def}{=} \texttt{base2}(A, X), \texttt{base3}(X, B, C), pred(B)$

All the non-view tuples (*e.g.*, **base1**–**base3**, N) are 300 bytes long. A, B, C, and X are integer fields. There is the natural one-to-one mapping between tuples of **base1** and **base2** (the nested sets and **base3**) that one would expect—the X field of **base2**/**base3** contains the values 1–|**base2**| which correspond to the order in which the **base1**/**base2** tuples were generated. B+tree indices are built on the following attributes: A and C for **Nest** and **Flat**; A for **base1**; A and X for **base2**; and X for **base3**. These indices improve both query and incremental view maintenance performance.

Before each experiment, each base table is initialized with tuples of uniformly-distributed, randomly generated data, and each view is materialized. The fields B and C were randomly filled with values in the range [1, |**base3**|] (A from the range [1, |**base1**|]). In all experiments, **base3** contains 200,000 tuples.

Comparisons: We compared only incremental maintenance techniques. Full refresh of views with the associated indices takes about twenty minutes—far longer than incremental refresh. Since an approach to maintaining views over nested data without back pointers would require something similar to full refresh—each outer tuple examining its inner set for changes—we ignored this possibility.

Transactions: A program produces a stream of transactions, each of which either queries or updates the database. A *query transaction* contains only **display** operations on a randomly chosen view, while an *update transaction* contains either **insert**, **remove**, or **replace** operations on a randomly chosen base table. The **replace** operation updates the C field of a nested/**base3** tuple. The *update ratio* of the transaction stream is the number of update transactions divided by the total number of transactions in the stream. For instance, if the stream contains 750 updates and 250 queries, then the update ratio is 0.75.

Each transaction contains 1–8 (an average of 4.5) operations over the same table (in Section 5.3, it is 1–4). Thus, for example, a query transaction reads 1–8 tuples matching randomly chosen values from a single view table.

5.1 Comparison between Flat and Nested Representation

Purpose: Our first experiments compare the cost of incrementally maintaining snapshot views over nested and non-nested versions of data. Given the complexity of Section 4's algorithm, *NestMat*, we wanted to verify that it was competitive with maintenance over non-nested data. Since nesting offers superior

ease of data modeling, competitive performance is good enough to argue for nesting support. (Since *Nest* and *Flat* have identical contents, the read performance should be identical, so we will only consider update performance—the total time to update base tables and maintain views.)

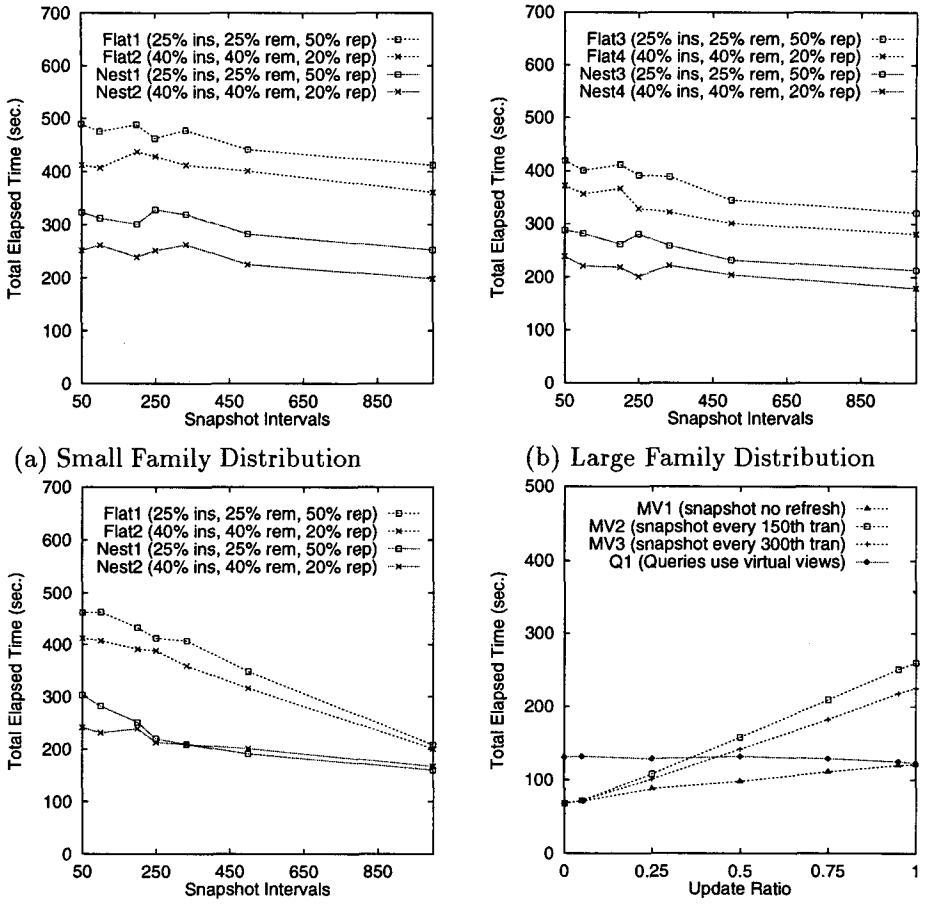
Method: In our first experiment, `base1` (`base2`) has 2,000 tuples; each `base1` tuple has its own non-shared nested set containing approximately 100 tuples. (There are 200,000 nested tuples randomly assigned to nesting tuples.) We call this the *Small Family* data distribution. *pred*(B) had a 50% selectivity, so the materialized view contained 100,000 tuples. Figure 2(a) contains the results for various snapshot frequencies. In our second experiment, `base1` (`base2`) has 200 tuples; each `base1` tuple has its own non-shared nested set containing (approximately) 1,000 tuples. We call this the *Large Family* data distribution. Other factors remain the same as in the previous experiment. Figure 2(b) contains the results.

Analysis: We note first that replaces are more expensive than inserts/deletes because they must copy the old value of a tuple into a newly created tuple used by the log. We note also that *NestMat* is superior to the algorithm over flat data. In large part, this is because insertions/deletions into `base3` must modify the X index, while corresponding insertions/deletions of nested tuples do not modify an index—the representation eliminates the need for this index. Another important factor appears to be that it is cheaper to follow a pointer to a list of containing objects than to traverse a B+-tree on `base2`'s X field. We note in Figure 2 that (b) has considerably better performance in the flat case than (a), while the nested performance is about the same. This is because the nested case must do about the same amount of work traversing a list of backpointers in both cases. However, in the flat case, `base2`'s X index in (a) is ten times bigger than in (b), so more CPU and I/O costs are occurred using the index.

5.2 Skewed Access

Purpose: Our previous work [14] showed that incrementally maintaining a view over flat relations is considerably cheaper if the distribution of updates is highly skewed. Log trimming converts several updates to the same tuple into a single update which must be considered by the maintenance algorithm. We wanted to see if the same effect held in the nested case.

Method: In picking the tuples to modify, we picked a parent tuple initially, and then modified its children. 80% of our picks went to 20% of the `base1`/`base2` tuples. Figure 2(c) compares the effect of skew in flat/nested relations. The graph



(a) Small Family Distribution

(b) Large Family Distribution

(c) Skew's Effects (Small Family)

(d) Comparison with virtual view

Fig. 2. Materialized View Comparisons

to compare it to is (a), since they use the same initial base data, just different update patterns. Both graphs contain update and maintenance time.

Analysis: The nested performance did not change very much in the presence of skew unlike the flat case. The flat case improves significantly as fewer snapshots are produced since a longer refresh cycle means more log trimming, and hence fewer modifications to the materialized view. However, increased trimming had only minor impact in the nested case.

5.3 Comparison with Virtual Views

Purpose: Under heavy update loads, materialized views become too expensive to maintain, and it is better to use virtual views. We wished to determine at roughly what point this occurs in our system.
Method: We sent 1–4 updates/reads in each transaction, and varied the update ratio. We used the Small Family data distribution and the 40% insert / 40% delete / 20% replace update mix. Given an A value in the view, our queries find the corresponding C values. We compare the total costs of all transactions and view maintenance for snapshots with different snapshot periods and for virtual views (with no view maintenance). Note that with virtual views, the data is fresh. It is somewhat stale with snapshots. However, we are comparing costs when some staleness can be permitted. See Figure 2(d) for the results.
Analysis: MV1 (snapshot with no refresh) shows that it is more expensive to update a base tuple than to read a snapshot tuple. That is why, even with no refresh, the cost increases when the update ratio increases. MV2 and MV3 show the additional overhead of view maintenance on top of the raw base table update costs of MV1. Q1, which uses virtual views, has roughly constant performance across the update range. This is because the computation of a tuple of the virtual view required many I/Os to find `base3` tuples that match a given `base2` tuple—to check the B/C values. Consequently, computing a virtual view tuple and modifying a `base3` tuple (and the associated indices) had comparable cost. In our experiments, using virtual views proved superior to using materialized views once the update ratio reached 30%—40%.

6 Related Work

A preliminary version of this work was presented at a workshop [22]. That version describes earlier versions of the algorithms contained in this paper and contains more details on topics like garbage collection and log trimming. This paper extends that work with performance results.

This paper describes a nested data model based on prior work [39, 33, 32]. Various query languages and implementation frameworks for the nested relation model have been studied (*e.g.*, [5, 15, 30, 24, 33, 34]). These papers do not explicitly mention view definition/maintenance.

Our view definition language in SWORD is based on Noodle [28], and is similar to HiLog [11, 32], where relation names or references may appear as arguments of other relations. We described how [17]'s view maintenance algorithms

for flat data models can be extended to handle a nested data model. We then described an implementation based on this extension.

This paper assumes that all materialized views are snapshot views (*i.e.*, views that are maintained when an explicit maintenance request is made). Snapshots were first proposed in [2]. Snapshot view implementation techniques are described in [23, 20, 35]. These papers consider only SP (select-project) views. [23] focuses on detecting relevant changes to snapshots using update tags on base relations. [20, 35] present techniques for maintaining logs and computing the net update to a view. Our log structures are based on the ideas in [35]. However, since [35]'s techniques are limited to SP views, they are not concerned with providing efficient access to past states of relations. Oracle supports snapshot views. However, Oracle only incrementally maintains SP views—using full recomputation on join and aggregate views. In [14], a model that allows multiple views to be maintained with different policies (immediate, deferred and snapshot) is studied, and an experimental performance comparison is made.

Concurrency control algorithms and a serializability model to guarantee serializability in the presence of deferred views are discussed in [21]. The focus of that paper is on doing concurrency control when multiple transactions reading and updating relations are executing concurrently in the system.

Our nested descriptor indices are similar to structures used to maintain join indices (*e.g.*, [26, 7]) and for field replication [36]. All these techniques are based on creating index structures that invert access paths specified by users to allow efficient maintenance of the desired access path (which can be considered as a materialized view of sorts). The nested descriptor indices presented in this paper can be implemented with structures similar to the modified B-tree structures used to model nested indices and path indices [7, 6].

7 Conclusion

This paper describes implementation techniques for maintaining materialized views over a nested data model. We showed that such views can be maintained by simple extensions to the counting algorithm [17]. For efficient computation, we keep track of changes within nested relations by transparently creating a structure that flattens nested log records. We then outlined the data structures/algorithms for the implementation. The implementation was guided by specific goals to minimize view maintenance overhead. The techniques described allow these goals to be achieved. We also measured the performance of our techniques, demonstrating that our algorithm's view maintenance performance over nested data is superior to that of [17]'s counting algorithm over a normalized

representation of the data. This is one of the first pieces of work to explore the applicability of materialized views over complex objects.

Currently, we only consider nested objects where an attribute of a tuple can be a reference (*i.e.*, a pointer) to a nested relation. We plan to extend out model to allow an attribute to be a nested relation, without the need to have pointers. Our implementation supports relational style SP and SPJ views over nested data. We plan to support aggregate views over nested data, based on the ideas in this paper. We also plan to improve the maintenance algorithms. For instance, the time for view maintenance can be further improved by having a separate asynchronous process that computes the incremental changes to the view and holds them in view differential files [12]. These view differential files would be updated periodically and be used to update the view relation when it is maintained. We are also investigating more efficient creation/maintenance of the index for nested-nesting associations.

References

1. S. Abiteboul and N. Bidoit. Non first normal form relations to represent hierarchically organized data. In *Proc. PODS*, 1984.
2. M. E. Adiba and Bruce Lindsay. Database snapshots. In *Proc. VLDB*, 1980.
3. R. Agrawal and N. Gehani. Ode (object database and environment): the language and the data model. In *Proc. SIGMOD*, 1989.
4. H. Arisawa, K. Moriya, and T. Miura. Properties on non-first-normal-form relational databases. In *Proc. VLDB*, 1983.
5. F. Bancilhon, P. Richard, and M. Scholl. On line processing of compacted relations. In *Proc. VLDB*, 1982.
6. E. Bertino. *Query Processing for Advanced Database Systems*, chapter A survey of indexing techniques for object-oriented databases. Morgan Kaufmann, 1994.
7. E. Bertino and W. Kim. Indexing techniques for queries on nested objects. *IEEE TKDE*, pages 196–214, June 1989.
8. J. A. Blakeley, P. Larson, and F. W. Tompa. Efficiently Updating Materialized Views. In *Proc. SIGMOD*, May 1986.
9. Peter O. Buneman and Eric K. Clemons. Efficiently monitoring relational databases. *ACM TODS*, 4(3):368–382, September 1979.
10. S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proc. VLDB*, pages 108–119, 1991.
11. W. Chen, M. Kifer, and D. Warren. HiLog: A first order semantics for higher-order logic programming constructs. In *Proc. N. American Logic Prog. Conf.*, June 1989.
12. L. Colby, T. Griffin, L. Libkin, I. Mumick, and H. Trickey. Algorithms for deferred view maintenance. In *Proc. SIGMOD*, 1996.
13. L. Colby, A. Kawaguchi, D. Lieuwen, I. Mumick, and K. Ross. Implementing materialized views, 1996. Unpublished manuscript.
14. L. Colby, A. Kawaguchi, D. Lieuwen, I. Mumick, and K. Ross. Supporting multiple view maintenance policies. In *Proc. SIGMOD*, May 1997.
15. P. Dadam, K. Kuespert, F. Andersen, H. Blanken, R. Erbe, J. Guenauer, V. Lum, P. Pistor, and G. Walch. A DBMS prototype to support extended NF^2 relations: An integrated view on flat tables and hierarchies. In *Proc. VLDB*, 1986.

16. T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *Proc. SIGMOD*, 1995.
17. A. Gupta, I. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proc. SIGMOD*, 1993.
18. R. Haskin and R. Lorie. On extending the functions of a relational database system. In *Proc. SIGMOD*, 1982.
19. G. Jaeshke and H. Sheck. Remarks on the algebra of non-first-normal-form relational database. In *Proc. PODS*, 1982.
20. B. Kähler and O. Risnes. Extended logging for database snapshots. In *Proc. VLDB*, pages 389–398, 1987.
21. A. Kawaguchi, D. Lieuwen, I. Mumick, D. Quass, and K. Ross. Concurrency control theory for deferred materialized views. In *Proc. ICDT*, January 1997.
22. A. Kawaguchi, D. Lieuwen, I. Mumick, and K. Ross. View maintenance in nested data models. In *Proc. Worshop on Materialized Views: Techniques and Applications (associated with SIGMOD96)*, June 1996.
23. B. Lindsay, L. Haas, C. Mohan, H. Pirahesh, and P. Wilms. A snapshot differential refresh algorithm. In *Proc. SIGMOD*, 1986.
24. V. Linnemann. Non first normal relations and recursive queries: An SQL-based approach. In *Proc. Data. Eng.*, 1987.
25. J. Lu, G. Moerkotte, J. Schu, and V. S. Subrahmanian. Efficient maintenance of materialized mediated views. In *Proc. SIGMOD*, 1995.
26. D. Maier and J. Stein. Indexing in an object-oriented DBMS. In *Workshop on OODB Sys.*, 1986.
27. A. Makinouch. A consideration of normal form of non-necessarily normalized relations in the relational data model. In *Proc. VLDB*, 1977.
28. I. Mumick and K. Ross. **Noodle**: A language for declarative querying in an object-oriented database. In *Proc. DOOD*, 1993.
29. I. Mumick, K. Ross, and S. Sudarshan. Design and implementation of the **SWORD** declarative object-oriented database system, 1993. Unpublished Manuscript.
30. P. Pistor and F. Andersen. Designing a generalized NF2 model with an SQL-type language interface. In *Proc. VLDB*, 1986.
31. Xiaolei Qian and Gio Wiederhold. Incremental recomputation of active relational expressions. *IEEE TKDE*, 3(3):337–341, 1991.
32. K. Ross. Relations with relation names as arguments: Algebra and calculus. In *Proc. PODS*, 1992.
33. M. Roth, H. Korth, and D Batory. SQL/NF: A query language for ¬1NF relational databases. *Information Systems*, 12(1):99–114, 1987.
34. M. Scholl, H. Paul, and H. Schek. Supporting flat relations by a nested relational kernel. In *Proc. VLDB*, 1987.
35. A. Segev and J. Park. Updating distributed materialized views. *IEEE TKDE*, 1(2):173–184, June 1989.
36. E. Shekita and M. Carey. Performance enhancement through replication in an object-oriented DBMS. In *Proc. SIGMOD*, 1989.
37. Oded Shmueli and A. Itai. Maintenance of Views. In *Proc. SIGMOD*, 1984.
38. L. Sterling and E. Shapiro. *The Art of Prolog. Advanced Programming Techniques.* MIT Press, Cambridge, MA, 1986.
39. S. Thomas and P. Fischer. Nested relational structures. In P. Kanellakis, editor, *The Theory of Databases.* JAI Press, 1986.
40. Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *Proc. SIGMOD*, 1995.