Graph Structured Views and Their Incremental Maintenance*

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

Abstract

We study the problem of maintaining materialized views of graph structured data. The base data consists of records containing identifiers of other records. The data could represent traditional objects (with methods, attributes, and a class hierarchy), but it could also represent a lower level data structure. We define simple views and materialized views for such graph structured data, analyzing options for representing record identity and references in the view. We develop incremental maintenance algorithms for these views.

1 Introduction

Relational views are useful for controlling data access, specifying contents of caches (or remote copies), and other data management tasks. In this paper we study how to extend this view concept and the associated maintenance algorithms to what we call a *graph structured database* (GSDB). Informally, a GSDB is a collection of "objects" that may contain "pointers" (graph edges) to other objects. A GSDB can represent Web pages, Lotus Notes documents, or other semi-structured information; it can also represent graph data structures such as a PERT chart, or a circuit design.

Example 1: Graph structured database
Figure 1 shows a graph structured database. Each

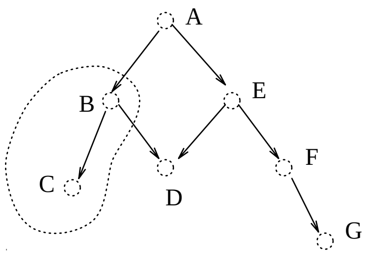


Figure 1: A graph structured database

node in the graph represents an object, each directed edge (pointer) represents a parent-child relationship between two objects. Users can traverse the graph by starting from an object and following the edges. We assume each object in the graph has an object identifier (OID) and some other fields that we will describe in Section 2. □

Views on a GSDB can be used for at least two tasks: defining cache contents and query filtering. For example, consider a set of interrelated Web pages. For now, assume that each page is an object, and the URLs in pages are the graph edges. Say that a user is interested in all Web pages containing the word "flower" and would like to copy them to his local disk for faster access. Using the constructs we define in this paper, a user will be able to define a *materialized view* to select the objects (and possibly edges) that should be copied. When the original objects change, the materialized view needs to be updated, and the incremental maintenance algorithm we discuss in Section 4 will keep the view up to date (at least for certain simple types of views).

A view can also be used to "filter" the objects that are accessed by a query. For example, a parent may wish to restrict access by his children to a particular subset of Web pages. For this he can define a *virtual view* (not materialized) that contains the allowed Web pages. Similarly, a user may have identified an interesting set of Web pages, and would like to use this set as a starting point for future queries. Again, a virtual view, defined using the constructs we will present, can be used for this purpose.

The data model we use in this paper is a "lightweight" one, without notions of object classes and inheritance. In particular, in Figure 1, objects A, B, ... are indistinguishable as far as the system is concerned. We believe that this makes our model more generic and widely applicable (e.g., to Web pages). Furthermore, even if an application requires classes or inheritance, the underlying data will most likely be stored by the database system as a GSDB. For example, the system may identify the class of an object by adding a special system field or tag to each object. (It could also link together all objects in a class.) An object A that inherits from another object B could have a special system pointer to B, for instance. Thus, the same view machinery we discuss in this paper could be used to cache or control access to the underlying data structures used to implement a more structured model.

*This work was supported by the Advanced Research and Development Committee of the Community Management Staff, as a project in the Massive Digital Data Systems Program.

Even though the notion of a GSDB view is similar to that of a relational view, there are a number of new challenges due to the richer nature of the GSDB model. For example, what exactly is a view on a GSDB? Is it just a collection of objects, or does it also include the edges between the objects? For instance, in Figure 1 say that our view includes the portion of the graph enclosed by the dotted line. Is the edge from B to C officially part of the view? In this case, perhaps a user that is given access to the view should not see the link from B to D. However, it is not clear how to achieve this, since the user could anyway retrieve the contents of B which somewhere contains the C, D pointers. Saying that the view simply contains objects B and C is simpler (no need to track edges), and if the view is materialized, then one could modify the contents of B so that it no longer contains the pointer to D. However, arbitrary "editing" of the materialized objects complicates view maintenance.

In a materialized view there is also the question of object identifiers. In our example, what should the OID of the copy of B be, and how can we identify the original object? Should the materialized copy of B contain a pointer to object C, or to the copy of C in the materialized view? Whatever the choice for materializing views, there is also the problem of incrementally maintaining them as the original data is modified. This is a much harder problem than for relational views. In particular, if we modify a tuple in a relation, we know that only views defined on that relation may be impacted. In a GSDB, a simple change like adding a link between two objects can dramatically change the set of objects in a view. (This will become clearer once we discuss how views are defined.) Thus, as we will see, incremental view maintenance is more complex and more expensive for GSDBs. As a matter of fact, it may only be feasible for relatively simple views.

In this paper we address some of these challenges. In particular, our contributions are:

- We formally define the notion of virtual and materialized views for GSDBs (Section 3). With these definitions, the result of a view definition on a GSDB is another GSDB, making it possible to define views on views and to query views in the same way GSDB are queried. (This same property holds for relational views and makes them especially useful.)

- We present a simple language for defining GSDB views, and mechanisms for restricting access to objects "within" a view. (Section 3). We also discuss implementation issues related to querying and materializing GSDB views.

- We present an algorithm for incrementally maintaining simple materialized views (Section 4). The algorithm takes as input a view definition and a sequence of updates to the base data, and propagates the changes to the copied data, querying the base data when necessary.

1.1 Related Work

Our work is based on previous work on materialized view maintenance [7], object technology [5], querying object oriented database [6, 9, 10], and semi-structured data models [16, 13]. In the rest of this subsection we specifically compare our GSDB views to relational views and object views defined using object classes.

Most of the incremental view maintenance work focuses on the relational model(see [7] for references). GSDB views are different from relation views in at least three major ways: (1) As mentioned earlier, in a GSDB view there is no schema to constrain changes to a particular "region;" with relational views, on the other hand, changes to a relation only impact views that refer to that relation. (2) In a GSDB view, there are relationships among objects that need to be preserved in the view. (3) View data may contain pointers and thus "lead access" to base data, those access need to be controlled. Instead of defining incremental view maintenance algorithms for GSDBs, one could instead represent the graph data as relations (e.g., with a relation storing all edges), and then simply use existing relational maintenance algorithms. However, as we discuss in Section 4.4, directly using the relational algorithms on graph data is not very effective.

Most previous research on object views (e.g., [1, 3, 14]) use object databases with classes. Views are defined by adding and hiding attributes to base classes. In contrast, since there is no class concept in a GSDB, views in a GSDB need to be defined by query expressions (as we do here). Views defined by adding and hiding attributes over a object classes are a subset of possible views defined using path expressions [9]. Also, most of the views considered by researchers so far are virtual views. In our work we consider both virtual and materialized views.

There is, nonetheless, some recent work on materialized object views by the *MultiView* group at University of Michigan [14]. The major difference between their approach and ours, besides that they define views as virtual classes, is that in their system each real value (attribute value or method of an object) has only one single physical copy. A materialized view stores only OIDs (pointers) of the base value. So their view maintenance algorithms do not need to worry about duplicating or propagating values of updated objects. However, the performance advantage of querying materialized views is reduced because each access to an object in a materialized view is decomposed into several accesses to the "real data". For the same reason, their approach is difficult to adapt to a distributed environment. On the contrary, the materialized views we design in this paper have the ability to duplicate both object pointers and values, and are able to independently maintain the relationship between an object and its duplicate.

Another work that is related to this paper is the view maintenance methods proposed in [15]. Their data model is edge-labeled trees without OIDs, which is similar to our model. They use a query language UnQL [4] to define their views, and use an algebraic approach to maintain the views. That is, they find expressions that can compute delta views corresponding to the changes of base data. However, their approach only works for views that are defined using "join free" queries, and updates that are either concatenation or replacement of two disjointed graphs. In their case, a view is always self-maintainable. In contrast, we define our views using an extension of OQL [5] and designing incremental view maintenance algorithms using common insertion, deletion and update operations. Views defined in this paper can not generally be handled by techniques discussed in [15].

2 Data model

In this section we introduce our object and database model. For objects, we use the OEM model [12]. Each *object* contains four fields: an OID, a label, a type and a value. The OID of an object is a universally unique identifier. The label is a string that explains the meaning of the object and does not need to be unique. Each object either has an *atomic* type, such as integer or string, or has a *set* type. The value of a set object is a set of OIDs of other objects. The following example shows a collection of objects.

Example 2: A collection of database objects We enclose each object within a pair of angle brackets and show the OID, label, type and value fields in order.[1] We use indentation as a visual aid to show the hierarchical relationship of the objects.

```
< ROOT, person, set, {P1,P2,P3,P4} >
   < P1, professor, set, {N1, A1, S1, P3} >
      < N1, name, string, 'John' >
      < A1, age, integer, 45 >
      < S1, salary, dollar, $100,000 >
      < P3, student, set, {N3, A3, M3} >
         < N3, name, string, 'John' >
         < A3, age, integer, 20 >
         < M3, major, string, 'education' >
   < P2, professor, set, {N2, S2} >
      < N2, name, string, 'Sally' >
      < ADD2, address, string, 'Palo Alto' >
   < P4, secretary, set, {N4, A4} >
      < N4, name, string, 'Tom' >
      < A4, age, integer, 40 >
```

In this database, object ROOT (we refer to an object by its OID) is an example of a *set* object. It has four children objects, representing two professors (i.e., with those labels), one student and one secretary. Notice that the subobjects of one professor are structured differently from those of the other professor. Object A1 is an example of an atomic object. We use $label(O)$ to refer to the label of object O and use $value(O)$ to refer to O's value. For example, $label(\texttt{P2}) = \texttt{professor}$ and $value(\texttt{P2}) = \{\texttt{N2}, \texttt{S2}\}$.

This set of objects can also be represented by the graph in Figure 2. For an atomic object, we omit the type since it can be inferred by its value. For a set object, we show the OID and label within brackets, and show its value by outgoing edges. □

[1] In our examples we try to use meaningful OIDs; in general, they can be arbitrary.

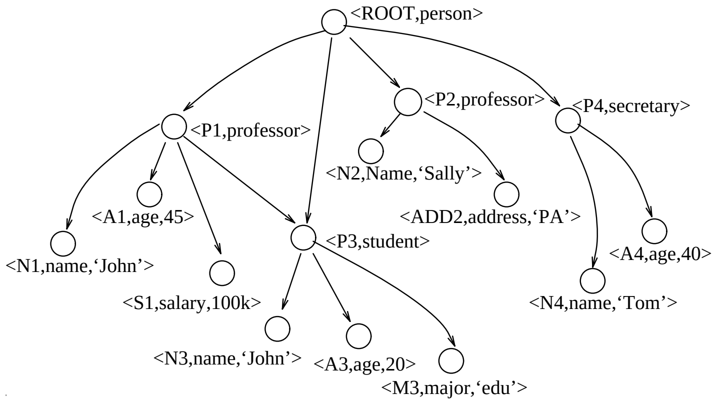


Figure 2: A graph structured database

Next we introduce the notion of a database in this data model. A *graph-structured database(GSDB)* is an object whose set value contains the OIDs of all objects in this database. Thus, a database is simply a way to group objects together. Objects can be grouped into a database for various reasons, for example, because they are semantically related, they are frequently accessed together, or they are physically located at the same site. If all the objects in Example 2 compose a database PERSON, then the object PERSON is:

```
< PERSON, database, set,
    { ROOT, P1, P2, P3, N1, A1, S1, N2,
      ADD2, N3, A3, M3, P4, N4, A4 }>
```

Notice that a database is not a special type of object; it is simply a conceptual aid and will be helpful when we discuss views.

We have selected this particular data model because it is simple, and yet will let us study the important view management issues. Furthermore, we believe that the model and the algorithms we will present easily generalize to other models. For example, objects with multiple fields can be represented in our model by several objects. For instance, a multi-field employee object `< name:'Joe', salary:50k >` can be represented as

```
< E1, employee, set, {N1, S1} >
    < N1, name, string, 'Joe' >
    < S1, salary, dollars, 50k >
```

As another example, fixed format records can be represented in our model by repeating the field names (as labels) in each object. For instance, say that a schema defines the first field of a record to be a name and the second field to be a salary. Then the record `<'Joe', 50k>` can be represented by the same object above. As a third example, some graph structured data models (e.g., [4],[13]) have values or labels on edges. These models can be mapped to ours [4].

Incidentally, note that our model is a conceptual one. The actual implementation could differ. For example, the OID field may not be stored in the record, and could be inferred from the record's location on disk. Repeated labels or values could be compressed in various ways. The database objects we defined may or may not be physical objects; for instance, objects in a database could simply be those contained in a particular region of a disk.

We now define some GSDB terminology that will be used in this paper. A *path* is a sequence of zero or more object labels separated by dots: $p = l_1.l_2 \ldots l_n$. For example, `professor.student` is a path. We use $N.p$ to denote the set of objects that can be reached following path p from object with OID N. If $N_2 \in N_1.p$, then N_1 is an *ancestor* of N_2, and N_2 is a *descendent* of N_1. In this case, the first label in p is the label of one of N_1's direct children, and the last label in p is the label of N_2. If $N_2 \in N_1.p_1$ and $N_3 \in N_2.p_2$, then $N_3 \in N_1.p_1.p_2$. In the Figure 2 example, node `A1` is a descendent of `ROOT` and can be reached from node `ROOT` following path `professor.age`, that is, `A1` $\in$ `ROOT.professor.age`.[2]

A *path expression* is a regular expression of paths. For example, $*$, $professor.*$ and $professor.?$ are path expressions. A path is also a (constant) path expression. We say that a path p is an *instance* of path expression e if the wild cards in e can be substituted by paths to obtain p. For any path expression e, we define $N.e$ to be the union of all objects in $N.p$ for all instances p of e.

We allow set operations on objects of type set. In particular, let $S1$ and $S2$ be two set objects. We define $union(S1, S2)$ to be an object whose value is {`value`($S1$) $\cup$ `value`($S2$)}, and define $int(S1, S2)$ to be an object whose value is {`value`($S1$) $\cap$ `value`($S2$)}. We assume that these resulting objects have an arbitrary unique OID and take on the label of $S1$[3]. These operations are mainly used to manipulate database objects and query answers, although they could apply to any set objects.

Many languages have been proposed for querying object-oriented databases [9, 4, 5, 13, 2]. Here we use a simple but representative language that lets us study view management issues. Its basic syntax is:

```
SELECT    OBJ.sel_path_exp X              (2.1)
WHERE     cond(X.cond_path_exp)
[WITHIN   DB1 ]
[ANS_INT  DB2 ]
```

A query answer is also an object, with the format `<ANS, answer, set, value(ANS)>`, where `value(ANS)` is a set of OIDs. To evaluate the above query without the last two optional clauses, the system considers all objects in `OBJ.sel_path_exp`. For each object `X` in this set, the system checks if it satisfies condition *cond*(`X.cond_path_exp`). Boolean function *cond*() accepts a set of atomic objects, and returns *true* if one of those object values satisfy the condition. When the condition is true, `X` is placed in `value(ANS)`. For example, the query `SELECT ROOT.professor X WHERE X.age > 40` will return `<ANS, answer, set,` {`P1`}`>` as answer. To write a query, the user must provide an entry point(OID) like `ROOT`. This entry point could be obtained from a previous query, or it could somehow be known to the user. A database name `DB` can also be used as the entry point. Using `DB.?` means that the search starts at all objects in `DB`.

Notice that the above query can span multiple databases. In Example 2, say that all objects are in database `D1` except for `A1` which is in database `D2`. In this case `P1` points to a "remote" object. The query `SELECT ROOT.professor X WHERE X.age > 40` will still return the same answer, since the query is insensitive to the "location" of objects. (Contrast this to relational queries where the relation a tuple belongs to is important.) The two optional clauses we now define attempt to control the "scope" of a query. As far as we know these clauses are not part of traditional object-oriented query languages, but as we will see, they will be useful for posing queries on views.

The `WITHIN DB1` clause limits the search to a single database `DB1`. The effect is that any OIDs that are not in `DB1` are completely ignored by the query. For our example where all nodes are in `D1` except `A1`, then our sample query with the clause `WITHIN D1` will have an empty result.

The clause `ANS_INT DB2` specifies that the answer object should be intersected with the `DB2` object to yield the final answer. This means that the answer objects are constrained to be in `DB2`, but the evaluation of the `WHERE` clause can follow remote pointers. For the example in Figure 2, if all nodes are in `D1` except `A1`, then the query with `ANS_INT D1` will return `<ANS, answer, set,` {`P1`}`>`. However, if all nodes except `P1` are in `D1`, the same query will return an empty set. The `ANS_INT` clause is mainly used to restrict a query answer to objects within a certain database. As we will see in the next section, we can use this clause to restrict user queries to return only objects within certain views.

Finally, we want to stress that we have selected a simple language, not because other features are not useful, but because these other features are not necessary for discussing simple view management. Features such as `FROM` clauses (e.g., in OQL [5] or Lorel [2]), multiple paths in `SELECT` clauses, or multiple conditions in `WHERE` clauses could easily be added to our language. However, some of these features would make incremental view management (Section 4) more complex.

3 Views and materialized views

A *view* is a set of imaginary objects defined in a precise way from real objects. A view can be used to hide or restructure objects from the underlying database. A view can also be used to specify what objects to cache for performance reasons.

As discussed in Section 1, defining views on GSDBs introduces some challenging problems regarding the structure of a view, how it can control access, and how it can be materialized. However, now that we have defined query answers to be the same as databases, we can think of views as other databases, available for querying. However, the situation is still different from a relational context because a query answer simply contains a set of OIDs that are not meaningful without the original data. In a relational view, on the other

[2] We use the terms node and object interchangeably.

[3] Typically set operations are only meaningful when objects $S1$ and $S2$ has the same label.

hand, the view contains all the relevant data. Another difference with relational views is that with the richer nested object model, view processing and view maintenance algorithms are substantially different.

3.1 Virtual Views

We start by defining *virtual views* to be the results of queries, as illustrated by the following example.

Example 3: A view on GSDB
Suppose that we are interested in all persons named 'John' from the database PERSON. We write:

```
define view VJ as:                        (3.2)
   SELECT    ROOT.* X
   WHERE     X.name = 'John'
   WITHIN    PERSON
```

In this case, objects P1 and P3 are selected, so value(VJ)= {P1, P3}. The view VJ is an object <VJ, view, set, value(VJ)>. Notice that we used a new label view for the view object, although this label will generally not be seen by queries. □

In general, a virtual view is defined by a *view definition query.* A view is an object <V, view, set, value(V)> where value(V) is the set of OIDs returned by the view definition query. We say that an object O is in a view V if O is in value(V). We also refer to the objects that are examined during query evaluation for a view as the *base objects* of the view; if those objects belong to some databases, we refer to them as *base databases.*

Virtual views can be used by queries in two ways. First, views can be used to constrain query results by using an ANS_INT clause. For example, query

```
   SELECT    ROOT.professor X
   ANS_INT   VJ                           (3.3)
```

will return {P1} as its answer. Object P2 is in ROOT.professor, but was excluded from the query answer because it is not in value(VJ).

In our example the ANS_INT clause was added by the query writer. We can also envision an authorization system where user queries are automatically expanded to include ANS_INT or WITHIN clauses for the union of views the user is authorized to access. This way users would only be able to access authorized data (or retrieve authorized data). Since views can be changed, it is easy to dynamically modify the privilege of a user.

A second way virtual views can be used is as "starting points" for queries. For instance, view VJ defined in Example 3 contains the set of person objects containing subobjects with value 'John'. If we are interested in the age of those persons named 'John', we do not need to write the full query. Instead we can write the query SELECT VJ.?.age, which gives us all subobjects of objects in view VJ with label age. Thus, views can be used as important intermediate results that can be further processed by follow-on queries on the views. This can clearly make the follow-on queries much simpler.

Our view concept is quite simple: views only contain sets of objects, and do not "restructure" the objects in any way. Yet, by defining views on views, one can essentially add new structure that may be useful to applications. To illustrate, consider the views defined on the objects in Figure 2:

```
define view PROF as:                      (3.4)
   SELECT    ROOT.*.professor X
define view STUDENT as:                   (3.5)
   SELECT    PROF.?.student X
```

In the original database, professor and student objects were reachable from ROOT, in any possible depth. Now we have created a database PROF that only contains professors, and one (STUDENT) that contains only their students. (A student who is not a subobject of some professor would not be included in STUDENT.) Queries can now access this new "professor–student" hierarchy by using the views as starting points (e.g., by starting at STUDENT we only reach information on students with professors), or by constraining the queries with the views (e.g., by using a ANS_INT clause we restrict our answer to students with professors).

3.2 Materialized views

A *materialized view* is a stored copy of the objects in a view. The materialized copy of a view can be stored either physically close to the base databases, or at a remote site. Views are typically materialized to improve query performance and data availability, e.g., data that is expected to be accessed frequently is included in the materialized view. However, materialized views can also be used for access control and as query starting points, just as virtual views.

For a view V, each base object in value(V) has a *delegate* object in the corresponding materialized view. A delegate object is a real object with the same label and type of its original object. For now, we assume that a delegate has the same value as the original object; we will discuss some variations later. A delegate object will have a new OID as described in the follow paragraph.

We need to relate the delegate of an object back to its original. Keeping this relationship is crucial for view maintenance. We use the OIDs of view objects to record this relationship. For a base object O, the OID of O's delegate object in a materialized view is obtained by concatenating the view OID with O. For example, in a materialized view MV, the delegate of base object $P1$ has OID $MV.P1$. This approach is similar to semantic OIDs as used in [11, 8, 9].

Since object identifiers in a materialized view have meaning, they cannot be changed arbitrarily. Some storage systems may not allow semantic OIDs (e.g., the system assigns its own OIDs). In such cases, the view management system can generate a table that maps the local OIDs used by the storage system to the semantic OIDs needed for view management.

A materialized view is again an ordinary GSDB, i.e., it is an ordinary object <MV, mview, set, value(MV)>, where MV is the OID of the view and value(MV) contains the OIDs of delegate objects. Our next example illustrates a materialized view.

Example 4: A materialized view
The following expression defines the materialized copy of the view in Example 3.

```
define mview MVJ as:                      (3.6)
   SELECT    ROOT.* X
   WHERE     X.name = 'John'
   WITHIN    PERSON
```

We use the keyword `mview` to specify that the view is materialized. The materialized view `MVJ` is shown in Figure 3. Each object in `obj(VJ)` has a delegate in the materialized view. The view object `MVJ` is also shown in the figure. □

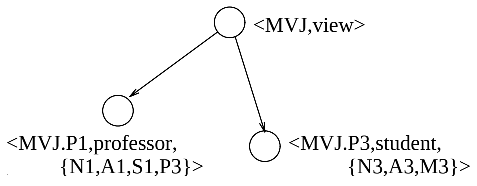


Figure 3: Materialized view database

Whether a view is materialized or not should not affect query results, as long as values in delegates are the same as in the original object. For example, a query posed to `MVJ` should return the same results as when the query is posed to `VJ`. Notice that making delegates contain the same value as their original means that delegates may contain OIDs of base objects. For example, `N1` in view `MVJ` is an OID of an object in database `PERSON`. A user query on `MVJ` can access `N1` if the query allows access to remote objects (e.g., no `WITHIN MVJ` clause).

Relationships (edges) between objects in a materialized view are implicitly kept. For example, `P3` is a child object of `P1` in database `PERSON`. In `MVJ`, *value*(`MVJ.P1`) contains `P3`. It is possible for the system that implements this view to detect the existence of `MVJ.P3` in the view, then change `P3` in *value*(`MVJ.P1`) to `MVJ.P3`. We call this operation *swizzling the edge* from `MVJ.P1` to `MVJ.P3`. It means changing a base OID to the OID of its delegate. Swizzling should not affect the results of queries.

The system or its manager may decide to swizzle edges to improve performance. In particular, there are two scenarios where swizzling is very useful. One is when the materialized view is stored at a site different from the base databases. In this case edge swizzling may enhance query performance by allowing local access to the referenced objects. A second scenario is when a query uses a materialized view `MV` as a starting point and also contains a `WITHIN MV` clause. In this case swizzling makes it easier to enforce the `WITHIN MV` clause. For example, a query `SELECT MVJ.professor.student WITHIN MVJ` should return `MVJ.P3` in the answer object. If edge swizzling is done, it is easy to check that the edges traversed are in `MVJ` (i.e., check that pointers start with the `MVJ` identifier). Without swizzling, when the system decides to follow the link in `MVJ.P1` to `P3`, it must then check if the delegate for `P3` is in `MVJ`. Since it is (i.e., `MVJ.P3` exists), then `MVJ.P3` is added to the answer.

Since a materialized view is stored independently from the base data, it is possible to "manually" change the object values without affecting base objects. However, this has to be done with care since queries on the modified view may give different results from the original materialized view (or from the equivalent virtual view). Furthermore, changing the view in an arbitrary way makes it impossible to maintain the view automatically.

Nevertheless, there may be cases where modification of materialized views could be useful. For example, say we chose to swizzle all edges in a materialized view `MV`, and then remove all remaining base OIDs that appear in delegate objects. As a result, any later user query using objects in `MV` will be restricted to access only `MV` objects. This "access control" is similar to attaching a `WITHIN MV` clause to all queries, but is not identical. (With the `WITHIN` clause, a query can still retrieve a view object that contains OIDs of base objects.) A second use of view modification could be to add timestamps or other auxiliary information to delegate objects. Queries can then refer to this auxiliary information, something they could not do on the equivalent virtual view.

Notice that if a remote site defines several views that share common objects, it may end up with multiple delegates for the same base object. Sharing delegate objects of interrelated views is an interesting idea that we plan to investigate further.

3.3 Querying views

Conceptually, a view is the same as an ordinary database, and objects in a view can be queried just like those in a GSDB. How the system processes a query on a view depends on whether the view is materialized or not. If the view is materialized, then the system accesses the stored objects and there is really no difference over normal query processing. (A materialized view must be kept up to date as the base data changes. This is the topic of the following section.)

When a view is virtual, one way to process a query over the view is to rewrite the query into an equivalent query that uses base objects only, just like query rewriting for relational views. However, since we lack an algebraic representation of queries, brute force rewriting may result in huge queries that are difficult to optimize.

Another method for answering queries on virtual views is to materialize the view when it is queried. Then the query is evaluated using the materialized view. The problem with this method is that the view could contain a large number of objects and the query could access a small number of them, thus resulting in a lot of wasted work.

In summary, we make view definition relatively simple. For virtual views, simplicity makes query rewriting feasible. For materialized views, as we will see, simplicity makes maintenance feasible. More complex views and relationships may be break into multiple simple stages (multiple simple views).

4 Incremental maintenance algorithm for simple views

An incremental view maintenance algorithm produces changes to a view given the changes to base objects. Incremental view maintenance algorithms have been developed for the relational model in both centralized and distributed cases [7]. In this section we develop an algorithm for GSDB views in a centralized system, where the base databases and the materialized view reside at the same site. In this environment, the view maintenance algorithm has direct access to the base data. We first briefly review basic updates in a GSDB, then describe the type of view that we will develop a maintenance algorithm for. We then design the incremental view maintenance algorithm, illustrate it by examples and discuss why this new algorithm is necessary.

4.1 Basic updates of GSDB

We consider three types of basic updates on a GSDB. Let N, N_1 and N_2 be OIDs.

1. $insert(N_1, N_2)$ adds the OID of N_2 into $value(N_1)$, N_1 must have a *set* type; object N_2 becomes a child of N_1 after this operation.

2. $delete(N_1, N_2)$ removes OID N_2 from $value(N_1)$, assuming that N_2 was a child object of N_1. (If no objects point to N_2 any more, N_2 may be garbage collected. However, we do not discuss garbage collection here.)

3. $modify(N, oldv, newv)$ changes the value of atomic object N from $oldv$ to $newv$.

The insertion and deletion operations add and delete edges in the base database graph. There are other possible update operations on a GSDB, for instance, the creation of a new object. Creating a new object that is not pointed at by any other object will have no impact on any queries, hence it has no effect on any views. Adding a new object O to a database DB can be modeled as $insert(DB, O)$. Most of other update operations on a GSDB can be represented as a series of base updates. For example, changing the value of a set object can be treated as insertions and deletions of its components. Changing the value of a label is also possible but not considered here.

4.2 Simple materialized view

We focus on a group of simple views, where the view definition query involves only tree paths instead of path expressions. We believe that studying this simple type of view illustrates the fundamental problems of view maintenance. A few examples of those problems are: When does a base update cause a materialized view to be changed? How can one determine which object(s) are to be inserted into or deleted from a view? Is incremental view maintenance more efficient than recomputing the entire view? In Section 5 we briefly discuss how to relax our assumptions and maintain more general views.

In this section, a materialized view `MV` is defined by the following expression.

```
define mview MV as:                    (4.7)
   SELECT    ROOT.sel_path X
   WHERE     cond(X.cond_path)
```

In the above expression, the selection path starts from object `ROOT`. The two paths `sel_path` and `cond_path` each contains a sequence of object labels without wild cards. Function `cond()` is as defined in Section 2; it returns *true* if any of the object values in `X.cond_path` makes the condition true.

As stated above, we assume that the base database has tree structure. This assumption is not crucial but simplifies the algorithm. The materialized view `MV` contains a set of delegate objects. From Section 3, we know that a new view object will be generated for each view defined. View object `MV` has type *set* and its components are all the delegate objects in this view.

Example 5: Basic update and corresponding view maintenance
Say we are interested in professors who are younger than 45. We define a view `YP` using the objects in Example 2.

```
define mview YP as:                    (4.8)
   SELECT    ROOT.professor X
   WHERE     X.age ≤ 45
```

The current materialized view is shown on the left hand side of Figure 4. The view `YP` includes a view object `YP` and an object `YP.P1` which is the delegate of base object `P1`.

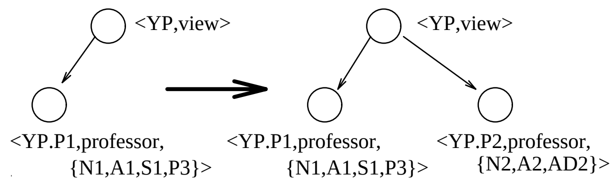


Figure 4: Change of materialized view

Assume that an update $insert$(`P2`, `A2`) occurs where `A2` is an atomic object `<A2, age, 40>`. Now `P2` becomes an object that satisfies the condition in the definition of view `YP`. Therefore, `YP.P2` should be inserted into `YP`. Figure 4 shows the correct view after processing this update. □

Notice that in a GSDB, labels of an object can be arbitrary and can be repeated. In this example, object `P2` does not have a child object with label `age` until `A2` is inserted. In other cases, one object may have two or more subobjects with the same label. Therefore there might be more than one "derivations" of a view object. The maintenance algorithm needs to consider all those scenarios.

4.3 Incremental maintenance algorithm

When base objects are changed, intuitively, either some new delegate objects need to be inserted into the view, or some existing delegates need to be deleted from the view. The task of an incremental view maintenance algorithm is to discover those delegates. With our model of materialized views, a view is just a collection of objects, so the maintenance algorithm need not worry about maintaining edges.

We first define a few constructs used by the algorithm. In a database with tree structure, it is clear that there is at most one path between two objects, so we use $path(N_1, N_2)$ to represent the path from N_1 to N_2. This path starts from the label of one of N_1's direct children and ends with the label of N_2. If N_1 is not

```
Algorithm 1:  Incremental maintenance of GSDB view MV
▷ When $insert(N_1, N_2)$ occurs:
  If $sel\_path.cond\_path = path(ROOT, N_1).label(N_2).p$ where $p$ is an arbitrary path
  then
      Let $S = eval(N_2, p, cond)$;
      For all $X \in S$ do $Vinsert(MV, MV.Y)$ where $Y = ancestor(X, cond\_path)$.
▷ When $delete(N_1, N_2)$ occurs:
  If $sel\_path.cond\_path = path(ROOT, N_1).label(N_2).p$ where $p$ is an arbitrary path
  then
      Let $S = eval(N_2, p, cond)$;
      For all $X \in S$, let $Y = ancestor(X, cond\_path)$;
      If $p = p_1.cond\_path$ where $p_1$ is an arbitrary path
          then do $Vdelete(MV, MV.Y)$;
          else if $eval(Y, cond\_path, cond) = \emptyset$, then do $Vdelete(MV, MV.Y)$.
▷ When $modify(N, oldv, newv)$ occurs:
  If $path(ROOT, N) = sel\_path.cond\_path$
  then
      Let $Y = ancestor(N, cond\_path))$;
      If $cond(newv)^4$, then do $Vinsert(MV, MV.Y)$;
          else if ($cond(oldv)$ and $eval(Y, cond\_path, cond) = \emptyset$), then do $Vdelete(MV, MV.Y)$.
End Algorithm 1
```

an ancestor of N_2, then $path(N_1, N_2) = \emptyset$. Let p_1, p_2 be two paths: $p_1 = l_1 l_2 \ldots l_m$ and $p_2 = k_1 k_2 \ldots k_n$. We say $p_1 = p_2$ if $m = n$ and $l_i = k_i$ for all $1 \leq i \leq n$.

Definition: Define $ancestor(N, p)$ to be the ancestor object X of N that satisfies $path(X, N) = p$. If there does not exist such an object, $ancestor(N, p) = \emptyset$. □

Definition: Define $eval(N, p, cond)$ to return the set of objects in $N.p$ that make $cond(N.p)$ true. The function returns $\emptyset$ when no such object satisfies the condition. □

In Example 5, $eval(\texttt{P1}, \texttt{age}, \texttt{cond}) = \{\texttt{A1}\}$ because `A1` is in `P1.age` and the value of `A1` satisfies the condition given in the view definition, i.e., $value(\texttt{A1}) \leq 45$.

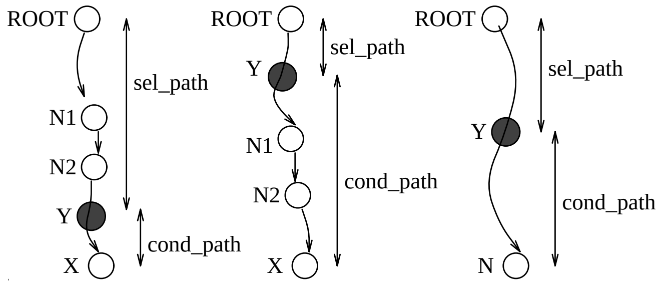
When a delegate for object X is created and added to a view V as part of view maintenance, we must copy the contents of X and make the OID of the new object $V.X$. For objects with type set, we assume that OIDs in $value(V.X)$ are not swizzled, i.e., the values in $value(V.X)$ are base OIDs.

Definition: Define $Vinsert(VN_1, VN_2)$ to be the operation that creates delegate object VN_2 and inserts it into $value(VN_1)$. We assume if an object with OID VN_2 was already a child of VN_1, then the insertion will be ignored and VN_1 stays the same. □

Definition: Define $Vdelete(VN_1, VN_2)$ to be the operation that removes object VN_2 from $value(VN_1)$. It is the same as the *delete* operation but it operates on view objects. If VN_2 is not a child of VN_1, then nothing happens to VN_1 after the operation. (Again, VN_2 may be garbage collected if necessary.) □

Algorithm 1 shows how to identify objects whose delegate is to be inserted into or deleted from `MV`. There is more than one way to identify those objects. Since we intend to develop algorithms that will eventually apply to a distributed warehousing architecture, the algorithm we provide here isolate the computations that need access to the base databases from those that can be done without base data. Specifically, the operations that may need to examine base data are encapsulated into functions $path(ROOT, N)$, $ancestor(N, p)$ and $eval(N, p, cond)$.

The algorithm is triggered once by each update on the base objects. In a centralized environment, view maintenance can be performed by the same transaction as the triggering update, so the algorithm uses the the base databases right after the triggering update and before any further updates.



To locate objects whose delegates are potentially affected (inserted into or deleted from the view) by a base update, Algorithm 1 examines *sel_path*, *cond_path* and the updated objects. In the algorithm, Y represents an object whose delegate is potentially affected, and X is Y's atomic descendent on which the condition is tested. After the algorithm locates Y, it tests whether the original condition that makes $MV.Y$

[4]The argument of $cond()$ here is a value; the function returns *true* if that value satisfies the condition specified.

appear or not appear in the view has been changed because of this recent update. If so, $MV.Y$ is inserted into or deleted from the view as appropriate. Notice that when a deletion occurs and Y is an ancestor of N_1, we can not simply delete $MV.Y$ from the view, because other descendents of Y may also make the condition true. This is a result of our non-unique label assumption. In this case, the condition on Y is reexamined. The following figures shows the relationship between the paths and objects used by Algorithm 1. The left two figures correspond to two possible objects layout scenarios when the base update is an insertion or a deletion, the rightmost figure corresponds to the base update *modify*().

Example 6: Incremental view maintenance
The following example illustrates the execution of Algorithm 1 on the scenario of Example 5.

Step 1: Update $insert(\texttt{P2}, \texttt{A2})$ occurs.

Step 2: Since $path(\texttt{ROOT}, \texttt{A2}) = \texttt{professor.age}$, $sel\_path = \texttt{professor}$ and $cond\_path = \texttt{age}$, it is true that $sel\_path.cond\_path = path(\texttt{ROOT}, \texttt{P2}).label(\texttt{A2}).p$ where $p = \emptyset$.

Step 3: Let $S = eval(\texttt{A2}, \emptyset, \texttt{cond}) = \{\texttt{A2}\}$, object `A2` is in S because $value(\texttt{A2}) = 40 < 45$. So the insertion of `A2` will enable one of `A2`'s ancestor to be inserted into the view.

Step 4: Let $Y = ancestor(\texttt{A2}, \texttt{age}) = \texttt{N2}$, do $Vinsert(\texttt{YP}, \texttt{YP.N2})$. After the insertion, node `YP.N2` appears in the view as a child of `YP`.

□

To prove that Algorithm 1 is correct, we need to show that starting from an initially correct materialized view, the view will be consistent with the base data after processing each update. That is, the delegates of all view objects are in `MV`, and there are no extra objects in `MV`. The details of the correctness proof are omitted here.

4.4 Discussion

An important issue is the cost of incremental maintenance, especially as compared to recomputation of the entire view. In Algorithm 1, the major cost lies in evaluating functions such as $ancestor(N, p)$ that may involve access to the base databases. However, recomputing the view also involves access to the base databases. Thus, the cost of each approach actually depends on the specifics of each scenario, such as the size of the databases, the type of view, the cost of query processing and the index structure of base databases. For example, if the base database has an "inverse index" such that from each node we can find out its parent, then evaluating $ancestor(N, p)$ is straightforward. If there does not exist such an index, evaluating the same function may require a traversal from *ROOT* to N. In general, incremental maintenance will be superior to recomputing the entire view if the view contains many delegate objects (in which case recomputation will be very expensive), and updates only impact a few, easily identifiable objects.

Another question is whether using our incremental view maintenance algorithm is better than using a relational model to represent both base data and views and then applying existing relational view maintenance algorithms. It is possible to represent objects of a GSDB in a relational fashion by "flattening" the object tree, as shown in the following example.

Example 7: GSDB Relational representation
We can represent a GSDB using three tables. The first table contains the OIDs and labels of all objects, the second table contains the OIDs of all objects of type set and their children OIDs, and the third one contains the OIDs of all atomic objects and their values. For simplicity, we assume that the VALUE attribute of the third relation can hold different data types (it is a union type). The three-table representation of database `PERSON` in Example 2 is the following.

OID	LABEL
ROOT	person
P1	professor
P2	professor
P3	student
...	...

PARENT	CHILD
ROOT	P1
ROOT	P2
ROOT	P4
P1	N1
...	...

OID	TYPE	VALUE
N1	string	'John'
A1	integer	45
...	...	...

Using this relational representation, one could in principle incrementally maintain the view, but there are disadvantages. First of all, with the relational representation, a single object update can involve multiple tables. For example, an insertion of an atomic object needs to modify all three tables. This would require several invocations of the relational incremental maintenance algorithm, and could lead to inconsistencies when only some of the updates are reflected on the materialized view. For example, it would be incorrect to have a tuple (A, B) in the PARENT-CHILD table without having both A and B in the OID-LABEL table.

Furthermore, a view defined using paths as in Section 3 needs to be defined by a Select-Project-Join expression with (many) self-joins. Incrementally maintaining such a view is not trivial, and we believe it could be more expensive to evaluate because the "path semantics" are hidden in the relations. Similarly, caching paths as suggested earlier will be harder to implement, again because the notion of a path is not explicit in the representation. □

4.5 View maintenance in a warehouse

Incremental maintenance for materialized views is very useful when the view is stored in a site separate from data sources, such as in a data warehouse. We discuss in [17] how to handle related issues. For example, how to realize function such as $eval(N, p, cond)$ in Algorithm 1 by communicating with a remote source, and how to cache relevant information to reduce query costs of view maintenance.

5 Conclusion

In this paper we studied the definition and realization of GSDB views. We provided a procedural algorithm for maintaining one type of view. We believe that our method for view definition, materialized view storage and management serve as a first step in solving the practical problem of GSDB views.

We only developed an incremental view maintenance algorithm for a group of simplified views to illustrate the fundamental problems. Due to space limitation, we cannot describe in detail how more general materialized views can be maintained. However, note that relaxing some of the restrictions we imposed on the view definition in Section 4 is easy. For example, handling views with more than one select path or more than one condition is straightforward. On the other hand, relaxing the following two assumptions (which we believe is important to do for some applications) is not simple:

- Allow the `sel_path` and `cond_path` to be general path expressions with wild cards. To maintain this type of view, the maintenance algorithm needs to be able to test path containment for general path expressions. For example, any path p is contained in path expression $*$. If a view is defined by expression `SELECT ROOT.*`, then any insertion of a `ROOT`'s descendent node will cause delegate objects to be inserted into the view.
- Allow more complex structure of base databases, for example, allow base databases to be arbitrary graphs. The maintenance algorithm will be similar to Algorithm 1, except that now there may be more than one path between two objects. Therefore, the actual implementation of the algorithm, e.g., computing $ancestor(X, p)$, is more difficult.

In addition to the maintenance algorithms for more general views, there are other open challenging issues for GSDB views that we are starting to address. In closing, we illustrate some of these issues.

- How does one define and maintain views whose edges (relationships) can be explicitly shown or hidden?
- How does one define and handle views in which the value of one delegate object is obtained from more than one base objects, for example, aggregate views?
- How does one define and maintain partially materialized views, for example, views that materialize a few levels of objects and leave the rest as pointers back to base data? This type of views may be useful for caching some but not all data of interest.
- How does one maintain materialized views when not only the updated base objects, but also the update query that generated them is known? For example, we may know what the salary of each person named 'Mark' was increased by $1000. Then a view containing the salary of persons named 'John' should be unaffected.

References

[1] S. Abiteboul and A. Bonner. Objects and views. In *SIGMOD*, pages 238–247, 1991.

[2] 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.

[3] F. Bancilhon, C. Delobel, and P. Kanellakis, editors. *Building an Object-Oriented Database System: The Story of O2.* Morgan Kaufmann, 1992.

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

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

[6] S. Cluet and C. Delobel. A general framework for the optimization of object-oriented queries. In *SIGMOD*, pages 383–392, June 1992.

[7] A. Gupta and I. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Engineering Bulletin, Special Issue on Materialized Views and Data Warehousing*, 18(2):3–18, June 1995.

[8] R. Hull and M. Yoshikawa. Ilog: Declarative creation and manipulation of object identifiers. In *VLDB*, pages 455–468, Aug. 1990.

[9] M. Kifer, W. Kim, and Y. Sagiv. Querying object oriented databases. In *SIGMOD*, pages 393–402, June 1992.

[10] D. Konopnicki and O. Shmueli. W3qs: A query system for the world wide web. In *VLDB*, pages 54–65, Sept. 1995.

[11] Y. Papakonstantinou, S. Abiteboul, and H. Garcia-Molina. Object fusion in mediator systems. In *VLDB*, Sept. 1996.

[12] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *ICDE*, pages 251–260, Mar. 1995.

[13] D. Quass, A. Rajaraman, J. Ullman, and J. Widom. Querying semistructured heterogeneous information. In *DOOD*, pages 319–344, Dec. 1995.

[14] E. Rundensteiner. Multiview: A methodology for supporting multiple views in object-oriented databases. In *VLDB*, pages 187–198, Aug. 1992.

[15] D. Suciu. Query decomposition and view maintenance for query language for unstructured data. In *VLDB*, pages 227–238, Sept. 1996.

[16] D. Suciu, editor. *Proceedings of the Workshop on Management of Semistructured Data*, May 1997.

[17] Y. Zhuge and H. Garcia-Molina. Graph structured views and their incremental maintenance. Technical report, Stanford University, Oct. 1997. `www-db.stanford.edu/pub/papers/gsvfull.ps`.