Views for Semistructured Data

Serge Abiteboul, Roy Goldman, Jason McHugh, Vasilis Vassalos, Yue Zhuge
Department of Computer Science
Stanford University
{abitebou,royg,mchughj,vassalos,zhuge}@db.stanford.edu

Abstract

Defining a view over a semistructured database introduces many new problems. In this paper we propose a view specification language and consider the problem of answering queries posed over views. The two main approaches, query rewriting and view materialization, are outlined with focus on the difficult problems caused by the semistructured nature of the data.

1 Introduction

Views are intended primarily to increase the flexibility of a database system by adapting the data to user or application needs. View mechanisms have been studied extensively in the context of the relational model [Ull89, KS91] and, more relevant to us, in the context of the object database model [Ber91, AB91, SAD94, LST91, Run92]. This paper describes some preliminary work towards implementing a view mechanism in the Lore system [MAG+97], a DBMS specifically designed to query semistructured data [Abi97].

A unique motivation for views when dealing with semistructured data is that views can be used to introduce some structure. A view can group together arbitrary portions of a database into a logical unit. Writing and processing a query over a view can be both simpler and more efficient than applying the query to the entire database. For example, consider a large data warehouse stored in Lore that integrates information about millions of people from many heterogeneous sources. In the warehouse a person could be represented by objects with many different structures, but a view would help to present objects in a more structured and regular manner.

Another important motivation for views over semistructured data is that a view mechanism provides a way of creating "stand-alone" databases from the original database. A view can be a subgraph of the original database (perhaps with new objects and new relationships added) whose objects can be either replicated (partially or fully) or pointers to objects in the source. A client/server architecture, where a portion of the database stored at the server is replicated at the client, could utilize this notion of views by treating the replicated information as a view defined over the source.

There are two major difficulties in introducing a view mechanism to a semistructured database. The first difficulty, also found in object database views, comes from the intermixing of queries and objects. A query in the relational model returns a relation that is by itself a "small" database that makes sense as an independent entity. In contrast, the result of a Lorel query [AQM+96] contains "bare" objects that do not have semantics independent of the original database. The second difficulty in introducing a view mechanism is the absence of a schema. For instance, if we observe the view specification for the ODMG data model [Cat94] used in O2Views [SAD94], the schema, and more precisely the class structure, plays a central role. Since there is no such precise notion of schema for semistructured data, our task is made more difficult.

The work presented here is on-going. Due to space limitations, we have to ignore many details and only highlight a few important techniques. We believe that a view specification could be richer than the one presented here. In particular, we do not consider issues such as using a Skolem-name mechanism in the style of Tsimmis [PAGM96], which is useful for data integration, or defining a mechanism for virtually classifying objects, which would provide specifications more similar to those of O2Views. In Section 2 we introduce the proposed view specification language. We consider query processing strategies in the presence of views in Section 3. Section 4 deals with materialized views and maintenance issues in the face of updates. Conclusions and future work are found in Section 5.

2 View Specification Language

Before introducing our proposed view specification language, we briefly introduce Lore's data model and query language. The Object Exchange Model (OEM) is a simple, flexible model for representing heterogeneous, semistructured data [PGMW95]. Intuitively, one can think of an OEM database as a labeled graph in which vertices correspond to objects and edges represent the object-subobject relationship with a label describing the precise nature of the relationship. A similar model is used in [BDHS96]. Lorel [AQM$^+$96] uses the familiar `select-from-where` syntax, and can be thought of as an extension of OQL [Cat94] in two major ways. First, Lorel encourages the use of path expressions in order to traverse semistructured data. For instance, the path expression `Staff.Employee.Salary` specifies the salary subobjects of employees where each employee is a subobject of the named object Staff. Second, in contrast to OQL, Lorel has a very forgiving type system and uses coercion extensively.

The main goal of the view specification language is to provide a mechanism for importing into a view arbitrary objects and edges between these objects from a source database. In addition, new objects and edges can be included in the view. A view definition must be able to specify entire subgraphs of the source database to be included in the view. To do so, we utilize `select-from-where-with` statements, where the `select-from-where` part is standard. The `with` clause allows the user to specify portions of the database (starting from selected objects) that are to be included in the view. More precisely, the `with` is made up of path expressions beginning from selected objects, where each object in the path, along with the edge, is also included in the view. Note that without the `with` clause it would not be possible to directly include subobjects of selected objects along with their edges. Let us consider the following simple example:

```
Define_View MyView
   TheManagers = select E
      from  Staff.Emp E, Staff.Emp E1, E1.Dept D
      where D.Manager = E
      with E.Name, E.Salary, E.Dept.Manager
```

This statement creates a view of all the manager objects along with name, salary, department and department manager subobjects if they exist[1].

[1]In a typical database instance a simpler view specification may return an equivalent set. We use this more intricate form to help demonstrate more general issues.

To illustrate the semantics of view specification statements, we explain how such a view could be materialized. The standard query evaluator is first used on the `from-where` part of the query to obtain bindings for the variables introduced in the `from` clause. From that, the select clause is evaluated, resulting in a graph where the leaf nodes are objects appearing in the source database and all other objects are newly created. Since a view is an independent unit, every source database object appearing in the view is replaced by a new *delegate* object. The rest of the view is constructed as specified by the `with` clause. Referring back to the previous example, if object o provides a binding for E, and there is a path *Dept.Manager* from o, then the edges and objects on that path[2] (or more precisely, delegates for these objects) are included in the view.

Note that the `with` clause does not do object filtering and is applied to all objects that satisfy the `from-where`. For instance, suppose we modify the view given above because we are only interested in managers with salaries above \$80,000. (We add the condition `E.Salary > 80000` to the `where` clause.) It is possible that some manager selected by the query works for a department where the department manager's salary is less that \$80,000. In such a case, the `with` clause will introduce the low paid manager to the view. Observe that this manager's name will not be in the view. To completely filter out low paid managers, we could create the view in two steps by first including too much information and then removing that which is not needed. Consider the following two statements that are used to create a view named `RichManagers`:

```
Statement 1:
Managers = select E
   from  Staff.Emp E, Staff.Emp E1,
         E1.Dept D
   where D.Manager = E and E.Salary > 80
   with E.Name, E.Salary, E.Dept.Manager
```

```
Statement 2:
 D.Manager -= B
 from Managers.Emp E, E.Dept D,
      D.Manager B
 where not exists ( B.Name)
```

The second statement illustrates the second facet of view specification. We use the Lorel update language [AQM+96] to specify how edges are (conceptually) added to or removed from the view. The operator "$-=$" indicates that *Manager* edges between bindings for D and B should not be in the view. As a result, objects for low paid managers are garbage collected. Note that there are clearly many alternative solutions. In particular, one could allow selections (nested `where`) in the `with` clause. This would greatly complicate view processing and will not be considered here.

Finally, observe that views naturally create the need for multiple *workspaces*. A view from the Lorel standpoint is a particular workspace, just like any other database. By default, path expressions are evaluated over the current workspace unless a *workspace selector* is used. The syntactic construct `WorkspaceName!PathExpression` allows access to data in workspaces other than the current one. We also provide the means to "switch" the workspace context of an object that might, because of views, live in more than one workspace. That is, you can go from a delegate object in a view to its original object, or vice versa.

[2] Each edge and object will only be included once in the view.

3 Virtual Views

A virtual view is a view that has not been materialized. There are at least three ways to process queries over a virtual view: (i) materialize the view on demand (when it is queried); (ii) rewrite the query posed to the view into a query posed to the source database, confining query processing to only discover objects defined by the view specification; (iii) given the query, rewrite the view so that, when materialized, only the portion of the view that is relevant to the query is constructed. We briefly discuss the latter two strategies through examples.

Query Rewrite: Continuing with `MyView`, introduced in Section 2, the following Lorel query asks for the employees who are in charge of a department to which they belong and have a salary of more than $100,000:

```
select E.Name
from   TheManagers.Emp E, E.Dept D
where  D.Manager = E and E.Salary > 100000
```

The query can be rewritten to eliminate references to the view as follows:

```
select E.Name
from   Staff.Emp E, E.Dept D
where  D.Manager = E and  E.Salary > 100000
  and  E in (select E
             from Staff.Emp E, Staff.Emp E1, E1.Dept D where D.Manager = E)
  and  D in (select E.Dept
             from Staff.Emp E, Staff.Emp E1, E1.Dept D where D.Manager = E)
  and  E.Salary in (select E.Salary
             from Staff.Emp E, Staff.Emp E1, E1.Dept D where D.Manager = E)
  and  D.Manager in (select E.Dept.Manager
             from Staff.Emp E, Staff.Emp E1, E1.Dept D where D.Manager = E)
```

The rewritten query can then be simplified to:

```
select E.Name
from   Staff.Emp E, E.Dept D
where  D.Manager = E and E.Salary > 100000
```

The example suggests a general technique for query rewrite: "constrain" each variable, via a subquery, to range over only those objects which would appear in the view based on the view definition. It also illustrates the main drawback of the query rewrite approach. Here, the resulting rewritten query is a complex expression, involving multiple subqueries, and the optimization of such a query would require a rather smart optimizer.

View Rewrite: View rewrite for partial materialization is a practical idea that may be easier to implement. Continuing with the example, the view definition can be rewritten (taking into account the query) as:

```
Define_View MyView
   TheManagersForQ = select E
      from  Staff.Emp E, Staff.Emp E1, E.Dept D, E1.Dept D1
      where D.Manager = E and D1.Manager = E and E.Salary > 100000
      with  E.Name, E.Salary, E.Dept.Manager
```

Intuitively, the idea is to find a relationship between the variables in the query and those in the view definition. If a mapping exists for all variables then the selections of the query can be "pushed" into the view definition. The rewritten (and hopefully much smaller) view can then be materialized to evaluate the query. Observe that in the example the query constructing the view could be optimized. Our intent here, however, is to illustrate a simple view rewriting technique. Note that the view rewrite approach could be less efficient than the query rewrite approach since view rewriting requires the partial materialization of the view and the execution of the original query over this materialization. On the other hand, the query rewrite approach requires the optimization of a (potentially huge) query which may be infeasible.

A drawback of the view rewrite approach is that we would like to be able to reuse these view fragments when answering subsequent queries over the original views. However, using partially materialized views is hard and even more challenging than in relational systems. For queries that are essentially select/project/join, techniques developed for relational systems [LY85, LMSS95, TSI94] can be used. For simple path queries, the problem is investigated in [AV97]. In the general case that combines both, the problem remains open.

4 Materialized views

In Section 2 we briefly described the construction of a materialized view. One advantage of view materialization is that the standard query evaluator can be used to answer queries posed over the view. A disadvantage is that it is difficult to keep the view consistent as the underlying source changes. While it is possible to only partially materialize a view, for simplicity, we only consider the problems associated with full materialization here. We focus on incremental view maintenance [GM95] instead of view recomputation. We also ignore the issue of updates through views.

View maintenance in a graph-based data model such as OEM is fundamentally more difficult than in the relational model. Because of data sharing, an update to a source object reached via some path may affect a view that involves the same object via a different path. For example, consider the `RichManagers` view given in Section 2. Suppose the database has another root of persistence, `Company`, and that an update statement reduces `Company.CEO.Salary` below $80,000. Assuming that the CEO is also a manager of some department, this would remove the CEO from the view, even though neither `Company` nor `CEO` is mentioned in the view specification. Indeed, the problem of detecting when a view is affected by a particular update is nontrivial and in general it is impossible to analyze the impact of an update based solely on the update statement and the view specification.

To detect whether updates to some objects may affect a view, we need to trace *inverse* path expressions from these objects. More precisely, let the updated object be o. An effective implementation of the update propagation to a view requires the ability to find all objects along any path p that reaches o, such that p is part of the view. We consider two approaches for efficiently evaluating inverse path expressions.

A first solution is to maintain "inverse pointers" for each edge in the database. Such data can be stored directly within each object or in an auxiliary index. After an update, these inverse pointers are used to find the objects and edges that are involved in path expressions relevant to the view. This done, we can incrementally evaluate the effect of the update to the view. Observe that this may result in wasting significant time following inverse paths that may not be relevant to the view.

Alternatively, for each path expression that is relevant to the view, we can keep a data structure that allows easy computation of inverse path expressions. For instance, if the path expression consists simply of a sequence $l_1.l_2...l_k$ of labels, it suffices to maintain one binary table for each label l_i in the path expression. Each tuple in the table has the oids of objects which are connected via the edge with label l_i. An update may modify one of the tables, but by using joins and appropriate indexes on the tables, it is not too expensive to maintain the tables and compute the impact on the entire path. When more complex paths are considered (with wild cards and regular expressions), we plan to evaluate more complex techniques such as those in [BKV91].

To conclude this section, it should be observed that many other aspects complicate the incremental maintenance of a materialized OEM view. We focused the discussion on the maintenance of inverse path expressions because this single point introduces a new dimension to the problem.

5 Conclusion and Future Work

This paper discussed several important issues arising in the specification and implementation of views in the Lore system. We first presented a syntax for specifying views. In OEM, object sharing and the lack of regular structure make query processing over virtual views and maintenance of materialized views challenging. We presented initial work towards addressing both problems. Implementing these techniques within the Lore DBMS is certainly high on our agenda. The paper also mentions a number of issues that we plan to investigate, e.g., the efficient evaluation of inverse path expressions. More traditional problems such as view updates or partial materialization take a different flavor in the context of semistructured data and will also be considered. We conclude by mentioning one other issue we wish to explore further.

The work presented in this paper centers around views by *extents*. That is, a view specifies a set of objects (usually via a query) and transformations to apply to these objects. A view by *intents* is just the transformations to be applied to objects in the database which match a description or have a certain form. During database traversals, the transformations will automatically be applied to those objects that match the description. In particular, there is no need to provide a query to specify the set of objects to transform and the rest of the database remains unchanged. Views by intents are supported in [SAD94] and allow an instance to modify the interface of all of the objects in a particular class. To do something similar in our context, we would like to "classify" objects, i.e., group in a class all objects that present the same pattern. Then we can specify how these particular patterns should be transformed. (Note that such transformations can be expressed in UnQL using *traverse* [BDHS96].) We believe that a classification mechanism, and its support by Lorel, would find uses beyond view specification, e.g., for querying by-example [GW97]. We are currently investigating this approach.

Acknowledgments

We wish to thank Janet Wiener for many fruitful discussions during the early stages of this work and Jeff Ullman for helpful suggestions and comments.

References

[AB91] S. Abiteboul and A. Bonner. Objects and views. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1991.

[Abi97] S. Abiteboul. Querying semistructured data. In *Proceedings of the International Conference on Database Theory*, Delphi, Greece, January 1997.

[AQM+96] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel query language for semistructured data. *Journal of Digital Libraries*, 1(1), November 1996.

[AV97] S. Abiteboul and V. Vianu. Regular path queries with constraints. In *Proceedings of ACM Conference on Principle of Database Systems*, Denver, 1997. to appear.

[BDHS96] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 505–516, Montreal, Canada, June 1996.

[Ber91] E. Bertino. A View Mechanism for Object-Oriented Databases. In *International Conference on Extending Data Base Technology*, pages 136–151, Vienna, March 1991.

[BKV91] A. Buchsbaum, P.C. Kanellakis, and J.S. Vitter. A data structure for arc insertion and regular path finding. *Annals of Math. and AI*, 3:187–210, 1991.

[Cat94] R.G.G. Cattell. *The Object Database Standard: ODMG-93.* Morgan Kaufmann, San Francisco, California, 1994.

[GM95] A. Gupta and I.S. 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.

[GW97] R. Goldman and J. Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. Technical report, Stanford University Database Group, February 1997.

[KS91] H. Korth and A. Silberschatz. *Database System Concepts.* McGraw-Hill, New York, New York, 1991.

[LMSS95] A. Levy, A. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *Proceedings PODS Conference*, pages 95–104, 1995.

[LST91] C. Laasch, M.H. Scholl, and M. Tresch. Updatable views in object–oriented databases. In *Proceedings of the 2nd Int. Conference on Deductive and Object–Oriented Databases*, number 566 in LNCS, Munich, Germany, December 1991. Springer Verlag.

[LY85] P.A. Larson and H.Z. Yang. Computing queries from derived relations. In *Proceedings VLDB Conference*, pages 259–69, 1985.

[MAG+97] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A database management system for semistructured data. Technical report, Stanford University Database Group, February 1997.

[PAGM96] Y. Papakonstantinou, S. Abiteboul, and H. Garcia-Molina. Object fusion in mediator systems. In *Proceedings of the Twenty-Second International Conference on Very Large Data Bases*, Bombay, India, 1996.

[PGMW95] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *Proceedings of the Eleventh International Conference on Data Engineering*, pages 251–260, Taipei, Taiwan, March 1995.

[Run92] E.A. Rundensteiner. Multi View: A Methodology for supporting Multiple View Schemata in Object–Oriented Databases. In *Proceedings of the 18th International Conference on Very Large Databases*, pages 187–198, Vancouver, Canada, August 1992. Morgan Kaufmann.

[SAD94] C. Souza, S. Abiteboul, and C. Delobel. Virtual schemas and bases. In *Proceedings EDBT, Cambridge*, 1994.

[TSI94] O. G. Tsatalos, M. H. Solomon, and Y. E. Ioannidis. The GMAP: A versatile tool for physical data independance. In *Proceedings VLDB Conference*, pages 367–378, 1994.

[Ull89] J.D. Ullman. *Principles of Database and Knowledge-Base Systems, Volumes I and II.* Computer Science Press, Rockville, Maryland, 1989.