Knowledge and Information Systems (2002) 4: 283–304
Ownership and Copyright
Springer-Verlag London Ltd. © 2002

Knowledge and Information Systems

Schema Evolution in Data Warehouses

Zohra Bellahsene
LIRMM UMR 55060 CNRS, Montpellier, France

Abstract. In this paper, we address the issues related to the evolution and maintenance of data warehousing systems, when underlying data sources change their schema capabilities. These changes can invalidate views at the data warehousing system. We present an approach for dynamically adapting views according to schema changes arising on source relations. This type of maintenance concerns both the schema and the data of the data warehouse. The main issue is to avoid the view recomputation from scratch especially when views are defined from multiple sources. The data of the data warehouse is used primarily in organizational decision-making and may be strategic. Therefore, the schema of the data warehouse can evolve for modeling new requirements resulting from analysis or data-mining processing. Our approach provides means to support schema evolution of the data warehouse independently of the data sources.

Keywords: Containment; Data warehouse evolution; Structural view maintenance; View adaptation; View maintenance

1. Introduction

A data warehouse is a repository integrating information from heterogeneous and autonomous sources for the sake of efficiently implementing decision-support or OLAP queries. Data of interest to the clients is extracted from the sources and stored in the data warehouse to quickly answer user queries independently of the availability of the data sources (Hammer et al., 1995; Hull and Zhou, 1996). At the conceptual level, a data warehouse can be seen as a set of materialized views over multiple sources. An important feature in a data warehouse is taking changes arising on the data sources into account. In the literature, most of work dealt with the problem of view maintenance, which consists in maintaining the materialized view in response to data modifications of the base relations (Gupta and Mumick, 1995; Zhuge et al., 1996). An approach attempting to adapt views

Received 20 March 2000
Revised 5 January 2001
Accepted 20 April 2001

after redefinition has been proposed (Gupta et al., 1995). However, this approach does not take into account the invalidation of views due to schema changes arising on the data sources.

1.1. Structural View Maintenance

The view adaptation problem in data warehouses can be considered from two different points of view:

- From the user and/or the data warehouse designer point of view: the view definition can be changed and the system must be able to recompute the view. Such dynamic views are needed in interactive applications such as dynamic queries, data visualization applications, biological databases, etc. Some approaches for adapting materialized views after view redefinition are given in Gupta et al. (1995), Mohania and Dong (1996), and Mohania (1997). This topic is called view adaptation. The issue of the view adaptation is recomputing a materialized view in response to changes of the view definition (Gupta et al., 1995). The key idea of adaptation techniques is using the old materialization and the base relations instead of recomputing extent of views from scratch, especially when the views are defined over base relations from multiple sources.
- From the point of view of data sources: schema changes arising on data sources may invalidate the materialized views. The issue is preserving the structural consistency of the data warehouse. We call this form of view adaptation: *structural view maintenance*. This technique is needed for the maintenance of data warehouses in large-scalable environments including evolving information sources. To the best of our knowledge, the only related work dealing with structural view maintenance has been done in the EVE project (Nica et al., 1998a, 1998b). However, this approach emphasizes the replacement of deleted information.

A more detailed discussion about related work is given Section 7. In this paper, we investigate the structural view maintenance problem, which can be seen as a variant of the view adaptation problem. It consists in adapting materialized views in response to schema changes of base relations located at different sites. We also consider the data warehouse evolution independently of the data sources. Indeed, views can evolve also for integrating results of analysis or data mining.

1.2. Contributions

In earlier work, we investigated the view evolution problem (Bellahsene, 1998). However, the solution we proposed leads to an important cost of maintaining materialized view versions. This paper focuses on the adaptation of the data warehouse in response to schema changes arising on source relations that are located on multiple sites. The goal is minimizing both the communication and the computation costs. In our framework, propagating a schema change on a view relies in creating a new view, which reflects the semantics of the schema change. However, the extent of the new view is *adapted* rather than recomputed from scratch. The key idea of adaptation is computing the new view from the old one.

Schema evolution in data warehousing systems may result from schema changes to information sources or from direct schema changes to the views in the data warehouse. The present paper deals with both types of evolution. We present a comprehensive study of different types of schema changes that can be performed on the base relations and on the views. Then, we present rewriting algorithms to adapt the extent of views in response to these changes.

1.3. Paper Outline

Section 2 is devoted to the definition of the structural view maintenance problem. In Section 3, we present our view definition language. Section 4 describes our view adaptation techniques after schema changes altering base relations. In Section 5, we discuss schema evolution of the data warehouse independently of the data sources. Performance evaluation of our approach versus the recomputing approach is discussed in Section 6. The related work is presented in Section 7. We conclude in Section 8.

2. Preliminaries

View. Let Q be an SQL query over a set R of relations that are called source relations. Let A be a set of attributes introduced in the view for modeling data warehouse specific requirements. We call these attributes *additional attributes.* Then, the view V is denoted by the tuple (Q(R), A).

Materialized view. A view is said to be materialized when its data (i.e., its extent) is computed and persistently stored. Otherwise, it is said to be virtual.

2.1. Structural View Maintenance Problem

Let S and S′ be the schema of the views V and V′ respectively, where V is the original view and V′ is the view resulting from schema changes applied to V. Let I and I′ be the extent of the views V and V′ respectively. A view evolution is a couple <u: $S \rightarrow S'$, u′: $I \rightarrow I'$>, where u is a schema modification and u′ is an adaptation operation on the materialized view.

The problem of structural view maintenance conveys two sub-problems:

(i) propagating the schema change to the view schema (i.e., computing S′);
(ii) adapting the view extent (i.e., computing I′).

The main issue is formulating V′, which simulates the schema change u such that the cost of u′ is minimized. This means that the view extent of V′ (say I′) will be computed from the view extent of V (say I) if possible rather than from scratch.

2.2. Notation and Assumptions

We consider SPJ (Selection–Projection–Join) views and we adopt the same syntactic shorthand as in Gupta et al. (1995). Let $A_1, \ldots, A_n$ be attributes from relations $R_1, \ldots, R_m$. The syntax of view creation is as follows:

```
Create view V As
Select $A_1, \ldots, A_n$
From $R_1, \ldots, R_m$
Where $C_1$ and ... and $C_k$;
```

```
Part (p_partkey, p_name, p_brand, p_type, p_size, p_retailprice, p_comment)
Supplier (s_suppkey, s_name, s_address, s_nationkey, s_phone, s_acctbal)
PartSupp (ps_partkey, ps_suppkey, ps_availability, ps_supplycost)
Customer (c_custkey, c_name, c_address, c_nationkey, c_phone, c_acctbal)
Orders (o_orderkey, o_custkey, o_orderstatus, o_totalprice, o_orderdate)
Lineitem (l_orderkey, l_linenumber, l_partkey, ps_partkey, l_suppkey,
  ps_suppkey, l_quantity)
Nation (n_nationkey, n_name, n_name, n_regionkey, n_comment)
```

Fig. 1. Tables of TPC-D benchmark.

where $C_i (i \in [1, k])$ is a selection predicate or a join predicate. This query is called view definition.

The propagation of schema changes arising on base relations to a view consists in modifying some parameters in the SELECT, FROM and WHERE clauses. However, in structural view maintenance setting these modifications result from schema changes arising on data sources. While in view adaptation setting (Gupta et al., 1995; Mohania and Dong, 1996; Mohania, 1997) these modifications are invoked by the user on the view definition.

We support the following comprehensive changes arising on the base relations:

- addition, deletion or modification of attributes;
- deletion of relations.

This set of changes is comprehensive since it contains all possible changes that may invalidate the view. For instance, adding a relation at the data source level will not affect the existing views.

2.3. The Sample Schema

The sample tables we used are from the TPC{XE "TPC"}-D Benchmark (1999). Figure 1 gives the schema of these tables. The underlined attributes are the keys of the relations.

3. The Extended View Definition Language

3.1. Modeling Specific Requirements of the Data Warehouse

In relational database systems, a view is a derived relation, (i.e., table) which includes a subset of attributes belonging to the database schema. We propose to extend the relational view definition language in order to allow the specification of views with additional attributes that do not belong to the base relations. Additional attributes provide means for modeling new requirements resulting from analysis or data-mining processing in the data warehouse.

3.2. Hiding Attributes

Defining a view consists in specifying what information derived from the base relations (or from other views) should be visible. In relational database systems, this feature is performed by the *projection* operation, which includes all attributes that should be visible. This approach presents the drawback that the definition of view should evolve according to the schema changes performed on the base relation even if the view attributes are not affected by these changes. To avoid this drawback, we propose the *hide* operator, which enables one to specify the attributes to be hidden. This possibility will be exploited for view rewriting according to the old view. The extension we propose concerns the view definition language and not the query language. However, in general, the *hide* operator is suitable when few attributes are to be hidden from one relation, while the *projection* is more appropriate when the view query includes several relations, an *aggregate* function, or a *group-by* clause.

Example. This example illustrates a case where the classical *projection* is more appropriate than the *hide* operator since the use of group-by requires the select clause having the same attributes as the group-by. Besides, using the standard projection makes easier to check this requirement visually.

Let us consider the view providing the customers who have ever ordered after 25 December 1999. The view query lists the customer name, customer key, the order key, date, total price and quantity of the order.

```
Create View Customers_99 As
Select C.c_name, C.c_custkey, O.o_orderkey, O.o_orderdate,
                                                   sum(l_quantity)
From customer C, orders O, lineitem L
Where O.o_orderkey = L.l_orderkey and
      C.c_custkey = O.o_custkey and
      O.o_orderdate > ‘1999/25/12’
Group by C.c_name, C.c_custkey, O.o_orderkey, O.o_orderdate;
```

Example. Let us present an example where the *hide* operator is more appropriate than the standard projection. The following view provides information on the all suppliers of red wines that are produced in France. For each supplier, the query lists the supplier’s key, name, address, and account balance and phone number.

```
Create View RedWine_View As
Select *
From supplier S
Where S.s_suppkey in (Select PS.ps_suppkey
                      From Nation N, Part P, Partsupp PS
                      Where P.p_partkey = PS.ps_partkey
                            and P.p_type like ‘red wine’
                            and N.n_name = ‘France’
                            and S.s_nationkey = N.n_nationkey)
Hide S.s_nationkey;
```

In this example, the *hide* operator is more appropriate than the standard *projection* because the number of attributes to be displayed is greater than the number of attributes to be hidden. Furthermore, even if the schema of the relation supplier changes the view query will remain correct. On the contrary, if we use the classical projection to express this view, every change to the base relation will lead to the modification of the view. For example, let us consider adding an attribute to the base relation supplier, say email. The previous view query using the hide primitive remains correct after adding this attribute. Now, if the standard projection is used in place of hide, the view query should be changed in order to take the added attribute into account as follows.

```
Create View RedWine_View As
Select s_suppkey, s_name, s_address, s_phone, s_acctbal, s_email
From supplier S
Where S.s_suppkey in (Select PS.ps_suppkey
                      From Nation N, Part P, Partsupp PS
                      Where P.p_partkey = PS.ps_partkey
                            and P.p_type like 'red wine'
                            and N.n_name = 'France'
and S.s_nationkey = N.n_nationkey);
```

Finally, as we shall see in the next section, the *hide* operator provides means to achieve the simulation of deleting an attribute from a source relation. In general, a delete operation includes rarely more than one attribute. Therefore, we propose to use the hide operator in this context.

The general syntax of view definition command has the following structure:

```
[Create] view < view name > As
< query expression >; /* this is an SQL like query */
[Add Attribute < attribute name >: < query expression > / < type >]
[Hide Attribute < attribute name >];
```

The semantics of this statement is creating a view. The *add attribute* clause allows a view to own additional attributes (i.e., that did not belong to base relations). For example, an attribute may be added to the view defining historical information although source information does not. This possibility is also useful for simulating schema extension or modification operations. Finally, it is possible to hide some attributes in a view by using the *hide attribute* clause.

4. Impact of Changing the Schema of Data Sources

In this section, we examine the impact on the related views, of the schema changes arising on data sources.

4.1. Principle of our Method

The schema changes arising on information sources should be propagated to the affected views in order to ensure structural consistency. For each possible schema change that can arise on base relations we provide the new view resulting from this change. The old view is replaced by the new one. The issue is computing the data of the new view from the materialization of the old view as much as possible in order to avoid the recomputation of the view from scratch. For this purpose, we make use of containment checking. Only the part of the new view that is not contained in the old view will be computed. Therefore, the issue is finding containment rewriting of the new view by using the old view. Our idea is formulating the new view in terms of subqueries that can be subsumed by existing materialized views. Besides, some extra information, such as keys, are kept to help the later adaptation as it has been suggested in Gupta et al. (1995) and Mohania and Dong (1996).

However, as it has been noted in Gupta et al. (1995), view adaptation differs from the problem of query rewriting (Levy et al., 1995) since in view adaptation setting the new view is not always equivalent to the old view. Furthermore, in structural view maintenance setting, the view changes are assumed to be local, i.e., the new view schema is close to the old one. Therefore, the transformation between the schema of the old view and the schema of the new view is straightforward.

Definition: Containment (Abiteboul et al., 1995). Let Q_1 and Q_2 be two queries over a schema S, Q_1 is contained in Q_2, denoted $Q_1(I) \subseteq Q_2(I)$[1], if for each instance I over S, the answer of Q_1 is a subset of the answer of Q_2.

4.2. Impact on the SELECT Clause

In this subsection, we examine the impact of schema changes to the SELECT clause of the view query.

4.2.1. Adding an Attribute to a Source Relation

When an attribute is added to a source relation, the issue is to reconsider the visibility of the added attribute to the corresponding views (if the affected relation is a derivation root). The decision to integrate the new added attribute into the existing views is application context dependent. It is the responsibility of the data warehouse designer to appreciate the pertinence of the added attribute and to determine which views are potentially concerned. Therefore, the addition will be notified by a view redefinition operation.

There is an exception when the view is defined as SELECT*. In this case, every addition to a relation involved in this view should be propagated to the view. An appropriate algorithm for that can be found in Mohania and Dong (1996). This type of change does not require the use of query containment since the problem is adding a new attribute. If the view query includes only a selection projection, the recomputation approach provides almost the same performance as our approach if the view selectivity is low, that is, if the extent of the view is small in relation to the base relation. On the other hand, if the view involves joins, the recomputation should perform expensive joins over the source relations, while our approach just has to do look-ups from the view materialization in order to add the new attribute values.

4.2.2. Impact of Deleting an Attribute from the SELECT Clause

The propagation of deleting an attribute A1 from the data source to the view V should take three different cases into account: (i) the attribute can be involved in the Select clause; (ii) the attribute can be involved in a condition of the Where clause; (iii) both (i) and (ii).

[1] When the containment is commutative it entails an equivalence of the related queries.

First, we consider the deleted attribute that is involved in a SELECT clause:

```
Create view V1 As
Select *
From V
Hide attribute A1;
```

Since the extent of the new view is contained in the extent of the old view, the new view can be adapted just by deleting the attribute from the old view materialization, while the recomputing approach needs to perform joins on the base relations. Our approach is almost as expensive as the rematerialization approach when the view includes one selection condition with low selectivity because in this case the view extent may be close to the extent of the source relation.

We note, in case of a view involving a join and a selection projection, the cost of our approach is less than that of the recomputation since it reads from the view materialization, which is smaller than the source relations. Furthermore, the gap between the two approaches increases according to the number of joins involved in the view query.

Example. Let us suppose, for analysis purpose, the schema designer wants to define a view providing a special treatment for computer items sold in France. For the sake of optimization and because this view is frequently invoked it will be materialized. The corresponding view is as follows:

```
Create View French_Computers As
Select P.p_partkey, P.p_name, P.p_retailprice, P.p_brand,
                                                    P.p_container
From Customers C, Lineitem L, Nation N, Order O, Part P
Where P. p_type = ‘computer’ and N.n_name = ‘France’ and
     N.n_nationkey = C.c_nationkey and P.p_partkey = L.l_partkey
     and L.l_orderkey = O.o_orderkey and O.o_custkey = C.c_custkey;
```

Let us assume the attribute container has been deleted from the base relation Part. We note this attribute is included in the view French_computers. Therefore, deleting it entails creation of the new view, which replaces the old with the same name. The view is rewritten in order to exhibit the part of it that should be changed.

```
Create View V′ As
Select *
From French_computers;
Hide attribute container;
Drop view French_Computers;
Rename V′ as French_computers;
```

The rewriting of the view shows that the old materialization should be updated by deleting the values of the attribute container from it. Finally, the created view is renamed so that the schema change becomes transparent to the user of the view.

4.3. Impact on the WHERE Clause

In this subsection, we discuss the cases where the deleted attribute is involved in a query condition of V. There are two possible cases: (i) the deleted attribute is involved in a selection condition; (ii) the deleted attribute is involved in a join condition. The possibility of adapting the related materialized views depends on the availability of the deleted attribute. In this paper, we suppose that the to-be-deleted attribute is still available at the adaptation time.

4.3.1. Impact of Deleting an Attribute Involved in a Selection Condition

Deleting a selection condition is equivalent to a substitution by a tautology true condition. This deletion may lead to inserting into the new view the tuples that have been discarded by the to-be-deleted condition. Thus, the definition of the new view resulting from this schema change can be rewritten as follows:

```
Create View V′ As
(Select *
From V)
     union
(Select *
From R1,R2,...,Rm
Where not C1 and C2 and ... and Ck);
```

The new view uses the initial view and also includes a subquery that needs to access the base relations (i.e., the second subquery). The reason is that tuples corresponding to this subquery are at the data sources. Thus, it is not possible to avoid it.

The first subquery is contained in the query defining V and therefore it will not be computed, while the second subquery will be computed and then materialized. From the practical point of view, the adaptation process consists in inserting into the old view the tuples resulting from the second subquery. We note both our approach and rematerialization approach (e.g., computing the view from scratch) need to access source relations. However, adaptation saves the time of writing out those tuples already present in the materialization of V. Furthermore, our approach is almost as expensive as the rematerialization approach when the second subquery yields a large number of tuples in relation to the total result of the view.

4.3.2. The Deleted Attribute is Involved both in the SELECT Clause and in the WHERE Clause

Now, we discuss the situation where the deleted attribute is involved both in the SELECT clause and in the WHERE clause. Let A1 be the deleted attribute and let C1 be the condition involving A1. Deleting A1 should lead to deleting C1. This change has two consequences: (i) removing the attribute from the view; (ii) deleting the condition from the WHERE clause. This second consequence may entail insertion of new tuples into the view, as previously discussed. So, the following statements perform the adaptation of the view:

```
Create View V′ As
(Select A2, A3, ..., An
From V)
 union
(Select A2, A3, ..., An
From R1, R2,..., Rm
Where not C1 and C2 and ... and Ck);
```

In fact, this case combines the changes discussed previously in Sections 4.2.2 and 4.3.1. The first subquery is contained in the query that defines the view V. Only the second subquery will be computed from the sources. This is the query adaptation. As for the previous case, both adaptation and recomputing need to access source relations in order to compute the tuples of the second subquery. However, our approach saves the time of writing out those tuples already present in the materialization of V. Moreover, our approach becomes almost as expensive as recomputing the whole view when the number of tuples resulting from the query adaptation is high in relation of the view extent.

Example. Now, imagine the database designer of the related data sources wants to delete the attribute o_orderdate from the table Order. We note this attribute is included both in the SELECT clause and in the where clause of the view Customers_99. Deleting this attribute entails the creation of a new view as follows:

```
Create View Customers_99_V1 as
(Select c_name, c_custkey, o_orderkey, sum(l_quantity)
From Customers_99)
UNION
(Select C.c_name, C.c_custkey, O.o_orderkey, sum(l_quantity)
From customer C, orders O, lineitem L
Where O.o_orderkey = L.l_orderkey and
      C.c_custkey = O.o_custkey and
      Not(O.o_orderdate > '1999/25/12'));
Drop view Customers_99;
Rename view Customers_99_V1 As Customers_99;
```

The first subquery performs a projection of the old view in order to delete the attribute orderdate from the view.

The second part of the view corresponds to the fact that deleting a condition may add tuples that did not match the condition (O.o_orderdate >'1999/25/12'). Thus, adapting the view consists in computing this subquery and inserting the resulting tuples into the view materialization.

4.4. Impact on the FROM Clause

In this section, we discuss the impact of deleting a source relation involved in a view. If the affected view is defined over a single relation then deleting this relation implies deleting the view. Otherwise, if the deleted relation is involved in the SELECT clause and/or in the WHERE clause, the new view should be adapted accordingly. We have to consider two possible cases: (i) if the join condition is alone in the WHERE clause, the new view is computed by the projection on the attributes of the remaining relation; (ii) the second case is detailed in the following subsection.

4.4.1. The Deleted Relation is Involved in a Join Condition

Deleting a relation from the view query has, as consequence inserting into the view new tuples, which did not match the corresponding join condition. We suppose that the to-be-deleted relation is still available at the adaptation time.

Example. Let us consider the view called `Join_View` defined by the following query:

```
Create View Join_View
As Select R.A, S.C
From R, S, T
Where R.A = S.B and S.C = T.C;
```

Let us suppose the relation T has been deleted from the data source. This entails deleting the join condition `S.C=T.C`. For the adaptation purpose, it is necessary to insert into the view the tuples that match the join condition `R.A=S.B` but do not match the join condition `S.C=T.C`.

The issue is to rewrite the new view in such a way that a subquery contained in the old view can be produced. The rewritten view in response to deleting relation T is as follows:

```
Create View V′ As
   (Select R.A, S.C
      From Join_View)
   union
   (Select R.A, S.C
      From R, S
      Where not (S.C = T.C) and (R.A = S.B));
```

The first subquery of this view query is contained in the view V; hence it will not have access to the sources. Only the second subquery will be computed from the base relations and the resulting tuples will be added to the view materialization. The algorithm proposed in Mohania and Dong (1996) can be used to compute this subquery. This algorithm is based on optimizing the communication cost since the involved relations may be located at different sites. For that, extra information is stored with each base relation and the view. More precisely, a join-count variable allows storing for each tuple the number of joins with tuples from another relation. Also, a derive-count gives the number of derivations of each view tuple. The same principle can be used if the deleted relation is involved in the SELECT clause and in the WHERE clause. In addition, one can project out the attributes of the deleted relation that were involved in the SELECT clause. Although both adaptation and recomputing need to have access to source relations, adaptation saves the time of writing out those tuples already present in the materialization of V. The gap between the two performances depends on the result of the second subquery. The recomputation approach should be preferable whenever the extent of the view is small in relation to the result of the second subquery.

5. Modifying the Schema of the Data Warehouse

The aim of this section is not to show that the adaptation approach is more appropriate than recomputing the view from scratch. The issue we consider in this section is how to provide means to support evolution of the data warehouse independently of the data sources. Indeed, the clients or the data warehouse administrator can invoke schema changes on views independently of the data sources. These schema changes should not be propagated to the data sources since these are autonomous. Furthermore, some of these schema changes (e.g., modifying an attribute domain and deletion of an attribute in a view schema) can lead to inconsistencies with regard to the data source schema. More precisely, handling a schema change to the view will entail creation of a new view version, which will be materialized in place of the old one. However, for the sake of optimization, the view extent should not be recomputed from scratch but adapted.

5.1. Extending the Data Warehouse Schema

The interest of including additional attributes in a data warehouse schema independently of the data sources lies in modeling new requirements resulting from analysis or data-mining processing in the data warehouse. The semantics of `add attribute` operation is augmenting the type of a view with the specified attribute. This operation does not entail a risk of inconsistency with regard to the data source schema or the existing views. The value of additional attributes is persistently stored. The syntactic form of this primitive is as follows:

```
Add Attribute [to < view name >] (< attribute name >: < query
expression > / < type >)
```

The value of an additional attribute may be inserted or computed by a query expression.

Example. Let us illustrate adding a new attribute named `category` to the view `Customers_99`. The attribute `category` indicates whether a customer is a 'big customer' or a 'small customer' according to the quantity of their orders.

```
Add attribute to Customers_99 (category : string);
```

This attribute is unknown from the data sources. We suppose it has been added for modeling a specific requirement of the data warehouse. Its value should be initialized and maintained by the data warehouse administrator.

5.2. Modifying an Attribute Domain in the Data Warehouse

Now, we discuss the modification of an attribute domain. This change is simulated by creating, first, a view which does not involve the specified attribute. Next, an attribute with the same name and having the specified new type is added to the created view. After that, the created view is renamed so that the schema change becomes transparent for the user. Note that the modification operation of an attribute domain has not been addressed in related work (Mohania, 1997; Nica et al., 1998a).

The following statement creates a new version of view V, which simulates the change that modifies the domain of the attribute A:

```
Create View V′ As
Select *
From V;
Hide attribute A;
Add attribute A  : T /* Now A has the new type T in V′ */
Drop view V;
Rename V′ as V;
```

At the implementation level, this view query will be decomposed into two subqueries such that each one corresponds to an atomic transaction in order to ensure the safe execution of the update operation. The first subquery is:

```
Create View V′ As
Select * From V
Hide attribute A;
```

It projects out the attribute to be modified. This subquery is contained in V. As previously discussed in Section 4.2.2, the adaptation process consists in removing from the view materialization the values of the to-be-deleted attribute.

The second subquery is:

```
Add attribute A  : T to V′;
```

This subquery consists in introducing a new attribute to the view, and containment checking falls. Consequently, the values of the new attribute will be inserted into the view and then stored persistently.

Example. Let us suppose the data warehouse designer wants to modify the domain of the attribute `retailprice` in the view `French_Computers`. Because in this view the items of interest are sold in France, he/she would like to compute the `retailprice` according to the French VAT amount, which is 20.6%. Our view system rewrites the view query as follows:

```
Create View V′ As
Select * From French_Computers;
Hide retailprice;
Add attribute retailprice  : (Select retailprice*1.206
From French_Computers);
Drop view French_Computers;
Rename V′ as French_Computers;
```

Finally, the data warehouse administrator can delete attributes from the view by using just the *hide* primitive. We note that in the case where the added attribute is defined by an expression, it is better to keep the attribute virtual notably if the corresponding source attribute (e.g., `retailprice`) is frequently updated.

6. Performance Evaluation of View Adaptation versus Rematerialization

In this section, we present an analytic model to compare performance of our approach versus recomputing. This cost model has been inspired from the one proposed in Kuno and Rundensteiner (1998) and adapted to the structural view maintenance problem. Other related works investigated the adaptation after redefinition (Gupta et al., 1995; Mohania and Dong, 1996; Mohania, 1997). The issue we investigated concerns the impact, on the data warehouse, of schema changes arising on source relations. The goal of our performance evaluation is to quantify the impact of our adaptation method upon changes in propagation times. In our cost model, we count the number of fetched tuples since we cannot accurately determine whether two tuples will be on the same page. First, we begin by evaluating the cost of recomputing a view. We then evaluate the main adaptation operations and make a comparison with the recomputing method.

|Extent(R_i)| gives the size (i.e., the number of tuples) of the relation R_i,
V.query is the query defining the view V,
CostReadATuple(V.query, R_i) gives the cost to screen one tuple from R_i for the view query,
CostReadATuple(V) gives the cost to screen one tuple from V,
CostWriteATuple is the cost of writing one tuple,
K is the number of base relations involved in the view query,
Selectivity(query) is the selectivity of query,
|Extent(V)| gives the size (i.e., the number of tuples) of the view V,
Netcost is the communication cost to transfer tuples on the net.
Ranges of values for each variable in the following evaluations are:

· CostReadATuple:	[0.95, 1.05]
· |extent (R_i)|:	[4000, 5000]
· CostWriteATuple:	[0.95, 1.05]
· Netcost:	10% of |extent(R_i)|
· DropCost(A):	1.5
· selectivity(V.query)	[0.5, 0.9]

· |extent(V)| is 40% * $\sum_{i=1}^{K}$ |extent(R_i)|

Fig. 2. The parameters of the cost model.

The graphics presented in this section are obtained as a result of our analytic model. In Fig. 2 we give the range of values that have been used. Besides, in some cases, we will give the specific values for the evaluation of the cost formulae.

6.1. Cost of Recomputing a View

The cost of recomputing the view V from scratch includes the communication cost of the base relations transfer plus the cost of performing the view query:

$$\texttt{Cost(V, recomputing)} = \texttt{ExtentCost(V)} + \texttt{NetCost}. \tag{1}$$

where

- K is the total number of base relations involved in the view V.
- `ExtentCost(V)` is the estimated average number of fetched tuples from the base relations to compute the extent of the view V. We assume the view V is derived from K base relations:

$$\texttt{ExtentCost(V)} = \sum_{i=1}^{K} \texttt{CostReadATuple(V.query, } R_i\texttt{)} \times (\texttt{selectivity(V.query)} \times |\texttt{extent(}R_i\texttt{)}|$$

  where `selectivity(V.query)` gives the rate of tuples in each base relation, which participate to the view extent; it takes its values in `[0.5, 0.9]`.
- `CostReadATuple(V.query,`R_i`)` gives the cost to screen one tuple from R_i for the view query.

- `NetCost` corresponds to the cost of the base relations transfer to the data warehouse site. It is a function of the size of the involved base relations. If each base relation is in a different site, then

$$\mathtt{NetCost} = \sum_{i=1}^{K} |\mathtt{extent}(\mathtt{R_i})|$$

  This cost is high; therefore, to avoid a disproportion of the graphics, we assume some base relations are situated in the data warehouse site. More precisely, in our evaluation for the recomputation approach we consider

$$\text{The average net cost is } 10\% * \sum_{i=1}^{K} |\mathtt{extent}(\mathtt{R_i})|$$

Finally, the detailed formula for recomputing the view is

$$\mathtt{Cost(V,\ recomputing)} = \sum_{i=1}^{K} \mathtt{CostReadATuple(V.query, R_i)}$$

$$\times\ \mathtt{selectivity(V.query)} \times (|\mathtt{extent}(\mathtt{R_i})|$$

$$+ 10\% * \sum_{i=1}^{K} |\mathtt{extent}(\mathtt{R_i})|$$

6.2. Evaluating the Impact of Deleting an Attribute Involved in the Select Clause

Let us consider now the case where the deleted attribute is involved only in the SELECT clause. Thus the cost of adapting the view materialization includes the cost of fetching the view materialization and projecting out the unneeded attribute. The communication cost is null in this case since the adaptation does not have access to the base relation. On the contrary, in the recomputation approach it does.

$$\mathtt{DeletAttCost(V, adaptation)} = \mathtt{CostReadATuple(V)} \times (|\mathtt{extent(V)}| + \mathtt{Dropcost(A)} \qquad (2)$$

where

- `CostReadATuple(V)` gives the cost to screen one tuple from the view V.
- `Dropcost(A)` is the cost of projecting out the attribute A from the view V.

In equation (2), extent(V) varies according to K, the number of base relations involved in the view query, despite the fact that parameter K representing this number does not appear in this equation. The reason is that extent(V) is assumed to be already computed at the time of the adaptation since the view is materialized.

Figure 3 depicts the evaluation of the two approaches when deleting an attribute from a source relation while varying the number of relations involved in the view. We note our approach outperforms the recomputation approach because the latter should read from the source relation, while the adaptation reads from the view materialization, which is smaller than the source relation. Furthermore, the gap between the two approaches increases according to the

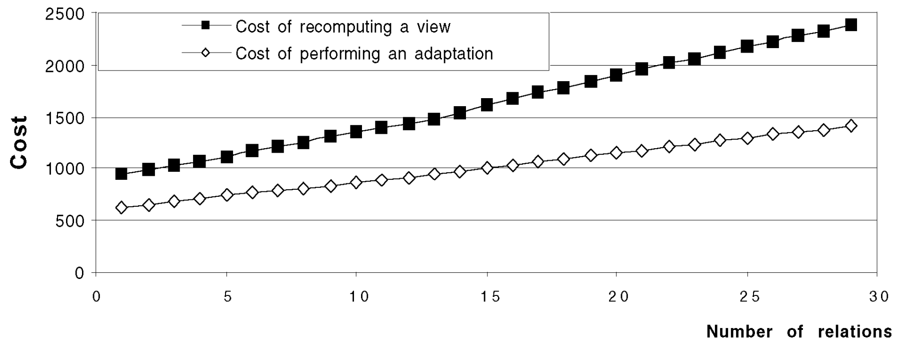

Fig. 3. Adapting an SPJ view after deleting an attribute.

number of joins[2] involved in the view. Indeed, If the view query involves joins the recomputation approach should perform these joins on the source relations, while our approach just has to access the old view materialization and to project out the deleted attribute.

6.3. Evaluating the Impact of Deleting a Condition from the Where Clause

Deleting a condition may arise after deleting an attribute or a relation at the source level. This case covers the changes described in Sections 4.3.1 and 4.4.1. We do not make a distinction between a selection condition and a join condition because the adaptation process is similar for both. Remember that this kind of change may lead to inserting new tuples into the view. Thus, the cost of adapting a view after this type of change includes the estimated cost of computing the part of the new view from the source relations plus the communication cost. Comparing the cost formulae of adaptation approach with that of recomputation, we note that the adaptation cost is dependent on the number of new tuples to be inserted into the view. Furthermore, the number of tuples is itself dependent on the selectivity of the to-be-deleted condition:

$$\mathtt{DeletcondCost(V, adaptation)} = \mathtt{QueryCost(query)} + \mathtt{NetCost} + \mathtt{Writecost} \tag{3}$$

where

- `QueryCost(query, V)` is the cost of computing part of the new view from the source relations and `query` corresponds to the query adaptation.

$$\mathtt{QueryCost(query, V)} = \sum_{\mathtt{i=1}}^{\mathtt{k}} \mathtt{CostReadATuple(query, R_i))} \\ \times(\mathtt{selectivity(query)} \times (|\mathtt{extent(R_i)}|$$

where `selectivity(query)` gives the rate of tuples in each base relation, which participate to the query adaptation result; it takes its values in `[0.5, 0.8]`.

[2] The number of joins depends on the number of relations involved in the view query.

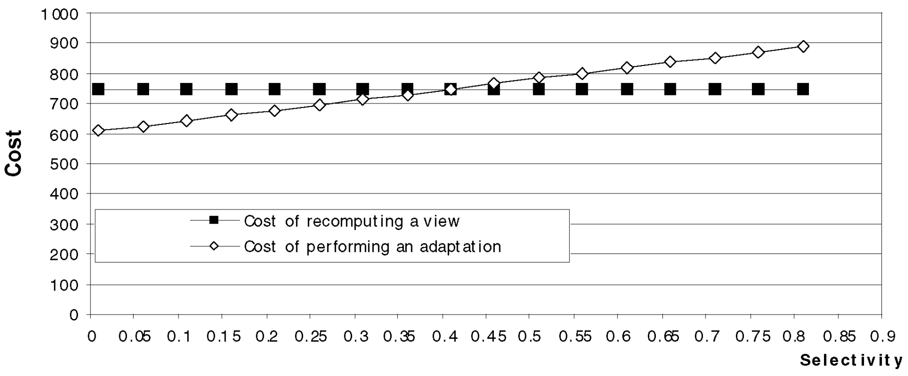

Fig. 4. Adapting a view after deleting a condition while varying its selectivity.

- NetCost is the cost of transferring the source relations in order to perform the subquery computing the tuples to be inserted into the view. If each base relation is in a different site then NetCost is

$$\sum_{i=1}^{K} |\mathtt{extent}(R_i)|$$

However, as for the recomputation and for the reason invoked previously, we consider

$$\text{The average net cost is } 10\% * \sum_{i=1}^{K} |\mathtt{extent}(R_i)|$$

- Writecost is the cost of writing in the view Vi the tuples resulting from the adaptation query.

Figure 4 depicts the performance evaluation of the two approaches when varying the selectivity of the query adaptation. In this figure, K, the number of base relations, is 5 and the selectivity of the recomputation query is fixed to 0.40. Our approach provides better performance than recomputation when the selectivity of the query adaptation is below 0.50. A low selectivity means the fraction of the number of tuples produced by the adaptation query is small. Above this value, our approach is more expensive than the recomputing approach. This result confirms our analysis that the recomputation is preferable when the adaptation query produces a high number of tuples in relation to the view extent.

6.4. Evaluating the Impact of deleting an Attribute Involved in both the Select Clause and in the Where Clause

We now discuss the situation where the deleted attribute is involved both in the SELECT clause and in the WHERE clause. This change leads to removing the attribute from the view and to inserting into the view the tuples that have been discarded by the deleted condition. Thus, the cost of this type of change includes the cost of deleting the attribute from the view (say, DeleteAttCost(A)) plus the cost of computing the new tuples from the sources (say ComputeTuplesCost(query)).

$$\mathtt{DeleteAttCond(V, adaptation)} = \mathtt{DeleteAttCost(A)} + \mathtt{ComputeTuplesCost(query)} + \mathtt{NetCost} \quad (4)$$

where

- $\mathtt{DeleteAttCost(A)} = \mathtt{CostReadATuple(V)} \times |\mathtt{extent(V)}| + \mathtt{Dropcost(A)}$
  where
  - CostReadATuple(V) gives the cost to screen one tuple from the view V.
  - Dropcost(A) is the cost of projecting out the unneeded attribute A from the view V.
- $\mathtt{ComputeTuplesCost(query)} = \sum_{\mathtt{i=1}}^{\mathtt{K}} (\mathtt{CostReadATuple(query, R_i)}) \times (\mathtt{selectivity(query)} \times |\mathtt{extent(R_i)}|$
  where
  - K is the total number of base relations involved in the view query.
  - query is the query adaptation
  - selectivity(query) gives the rate of tuples in each base relation, which participate to the query adaptation result.
- NetCost is the communication cost needed to transfer the involved source relations in order to compute the tuples to be inserted into the view. If we assume each base relation is situated in a different site then NetCost is

$$\sum_{\mathtt{i=1}}^{\mathtt{K}} |\mathtt{extent(R_i)}|$$

Comparing the cost formula (1) (of the recomputing approach) to the cost formula (4), we note that the recomputation would be preferable if the selectivity of the query adaptation was higher than the selectivity of the view query itself (see Fig. 5). This means the query adaptation result in terms of tuples is closed to the view extent. In this evaluation, selectivity(query) takes its values in [0.7, 1] and selectivity(V.query) in recomputation formulae takes its values in [0.5, 0.7].

6.5. Cost of Performing an Adaptation after Adding Attributes to a Source Relation

Remember, when the view is defined as SELECT*, the structural view maintenance states that every addition to a source relation involved in this view should be propagated to the view. The cost of this adaptation includes the cost of performing a look-up of each tuple of the view extent in order to add the new attributes to the view extent plus the cost of transferring the attribute values from the sources to the data warehouse site.

$$\mathtt{AddAttCost(V,\ adaptation)} = \mathtt{CostWriteATuple(V)} \times |\mathtt{extent(V)}| + \mathtt{NetCost} \quad (5)$$

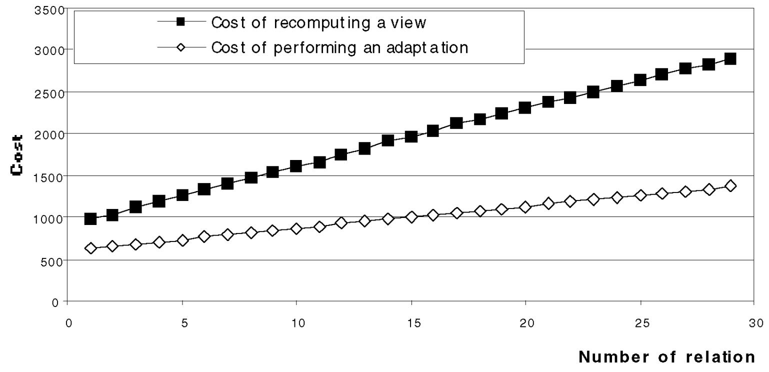

Fig. 5. Adapting a view after deleting an attribute and a condition.

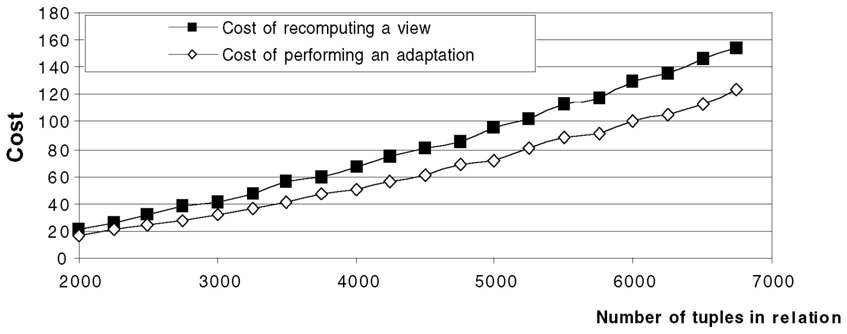

Fig. 6. Adapting SP views after adding attributes.

where

- `CostWriteATuple(V)` $\times$ |`extent(V)`| is the cost of writing the attribute values in the view materialization.
- `NetCost` corresponds to the communication cost of transferring the attribute values from the base relation to the data warehouse site. We note the projection of the involved relation R_i onto the deleted attribute A, Π_A (R_i). Then, `NetCost` is $|\Pi_A(R_i)|$.

In equation (5), the view extent is computed according to the number of base relations involved in the view query even if the parameter K representing this number is not explicitly stated. Indeed, several relations can be referenced in the view query. Thus, they are invoked to compute the view extent.

We distinguish selection–projection–Join (SPJ) views and selection–projection (SP) views (without join). First, we start by evaluating the cost of an SP view; we fix K =1 and vary the source relation size. In addition, we assume the `view extent` is 40% of the base relation extent and `Netcost` is 10% of the base relation extent.

The result of this evaluation is depicted in Fig. 6 and shows that adaptation wins within a small range when the relation size is small. The gap between adaptation and recomputation rises according to the size of the involved source relations.

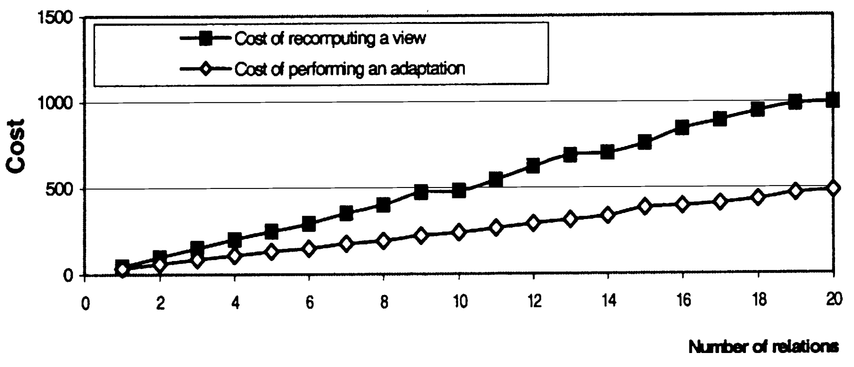

Fig. 7. Adapting SPJ views after adding attributes.

Conversely, the evaluation of the cost of SPJ views consists in varying K, the number of relations involved in the view query. As we can see in Fig. 7, our approach outperforms the recomputation approach. The reason is that recomputation should access the source relations in order to perform the joins involved in the view query, while our approach just has to do look-ups from the view materialization to add the new attribute values. In this case, our adaptation algorithm does not access the sources.

7. Related Work

Research problems in data warehousing systems are clearly identified in Widom (1995). Most work concerning view adaptation dealt with efficiently managing changes at the data level in order to keep materialized views consistent with the data at the sources. This problem is known as the view maintenance problem. The view has to be maintained incrementally by calculating the effects of the changes at the sources (Gupta and Mumick, 1995; Wiener et al., 1996; Zhuge et al., 1996).

The view adaptation after redefinition has been investigated first, in Gupta et al. (1995). In this approach, the adaptation process is expressed as an additional query or update upon the old view and the base relations that will be performed in order to adapt the view. For each possible view redefinition operation, a view adaptation algorithm is proposed. In some cases, extra information is stored with the views in order to make the adaptation process more efficient. Their experimental evaluation of adaptation techniques indicates that adaptation is more efficient than recomputation in most cases.

Another approach for view adaptation after redefinition in data warehousing systems has been proposed (Mohania and Dong, 1996; Mohania, 1997). The objective of this approach is to minimize the communication cost by avoiding sending data that is irrelevant to the view. In Mohania and Dong (1996), the authors consider SPJ views including at most two relations. The adaptation method proposed in Mohania (1997) deals with more general views that may involve selection, projection and join as well as union or difference. This adaptation method uses expression trees for determining what part of the view will be recomputed. Three cases have been considered according to the position of the node in the query tree. It has been shown that if changes are made at the root, it entails only the right-hand child to be evaluated. Secondly, when changes are performed at the first depth of the tree the view adaptation problem becomes an incremental problem (Mohania, 1997). Otherwise, it is necessary to maintain the intermediate results of the nodes in order to adapt the view.

More recently, the problem of *view synchronization* caused by external environmental changes has been addressed (Nica et al., 1998a, 1998b). The proposed approach focuses on the replacement of invalidated views. For this purpose an extension of SQL has been provided for expressing user preferences for view evolution. For example, a user can indicate what attributes are indispensable or what attributes are replaceable by similar information. This approach exploits meta-knowledge involving interrelationships existing between different data sources. Finally, our solution complements the work developed in Gupta et al. (1995) and in Mohania and Dong (1996) and Mohania (1997) since we consider the view adaptation after changes caused by the evolution of the data sources.

8. Conclusion

In this paper, we have shown how structural view maintenance can be supported in data warehouses including evolving data sources. This technique consists in maintaining the data warehouse in the face of schema changes to the data sources. In our approach, the view materialization is adapted dynamically by using the old materialized view as much as possible to avoid the entire recomputation of the view. Our approach is based on the query containment test. The key idea is that only the part of the new view that is not contained in the old view will be recomputed. The evaluation results show that our approach outperforms the recomputation approach of the view in most cases even when the adaptation needs to access to the sources.

Furthermore, our approach supports schema changes of the data warehouse independently of the data sources. These changes should not be propagated to the data sources since these are autonomous. Finally, a prototype validating our approach has been implemented on top of the database system Oracle V.7.3.

One of the most important decisions in designing a data warehouse is the selection of materialized views to be maintained at the warehouse. The goal is to select an appropriate set of views that minimizes total query response time and/or the cost of maintaining the selected views, given a limited amount of resources such as storage space or total view maintenance time (Gupta, 1997). We are currently investigating this topic. We believe that the view selection strategy has a great influence on the view adaptation techniques.

Acknowledgements. I would like to thank the anonymous referees for their helpful suggestions and comments on an earlier draft of this paper. I would like also to thank my students for their implementation of the prototype: in alphabetic order Tigist Alemu, Eric Boulat, Yvan Calas, Nadia Cheaib and Claude Lopez.

References

Abiteboul S, Bonner A (1991) Objects and views. In Proceedings of ACM SIGMOD conference on management of data, Denver, CO, pp 238–247

Abiteboul S, Hull R, Vianu V (1995) Foundations of databases. Addison-Wesley, Reading, MA

Bellahsene Z (1998) View adaptation in data warehousing systems. In Proceedings of international database and expert applications conference, DEXA'98. Lectures Notes in Computer Science, Springer, Vienna, pp 300–309

Gupta A, Mumick IS (1995) Maintenance of materialized views: problems, techniques, and applications. Data Engineering Bulletin 18(2): 3–18

Gupta A, Mumick IS, Ross KA (1995) Adapting materialized views after redefinitions. In Proceedings of ACM SIGMOD international conference on management of data, San Jose, CA, pp 211–222

Gupta H (1997) Selection of views to materialize in a data warehouse. In Proceedings of the international conference on database theory, Delphi, Greece, pp 98–112

Hammer J, Garcia-Molina H, Widom J et al. (1995) The Stanford data warehouse project. In IEEE Data Engineering Bulletin 18(2): 41–48

Hull R, Zhou G (1996) A framework for supporting data integration using the materialized and virtual approaches. In Proceedings of SIGMOD'96 conference, Montreal, Canada, pp 481–492

Kuno HA, Rundensteiner E (1998) Incremental maintenance of materialized object-oriented views in MultiView: strategies and performance evaluation. Transaction on Knowledge and Data Engineering, IEEE Press, 10(5): 768–792

Levy AY, Mendelzon AO, Sagiv Y et al. (1995) Answering queries using views. In Proceedings of the 14th ACM SIGACT-SIGMOD-SIGART symposium on principles of database systems, San Jose, CA, pp 95–104

Mohania M, Dong G (1996) Algorithms for adapting materialized views in data warehouses. In Proceedings of international symposium on cooperative database systems for advanced applications, Kyoto, Japan, pp 62–69

Mohania M (1997) Avoiding re-computation views adaptation in data warehouses. In Proceedings of 8th international database workshop, Springer, Hong Kong, pp 151–165

Nica A, Lee AJ, Rundensteiner EA (1998) The CVS algorithm for view synchronisation in evolvable large-scale information systems. In Proceedings of international conference on extending database technology, EDBT'98. Lectures Notes in Computer Science, Springer, Valencia, Spain, pp 359–373

Nica A, Lee AJ, Rundensteiner EA (1998) Using containment information for view evolution in dynamic distributed. In Proceedings of the workshop of environments data warehouse design and OLAP technology (DWDOT'98), Vienna, Austria, pp 212–217

TPC-D Benchmark (1999) Standard Specification 2.01, January 1999. http://www.tpc.org Widom J (1995) Research problems in data warehouse. In Proceedings of 4th international conference on information and knowledge management, CIKM. ACM Press, Baltimore, MD, pp 25–30

Wiener JL, Gupta H, Labio WJ, et al. (1996) A system prototype for warehouse view maintenance. In Proceedings of the workshop on materialised views: techniques and applications, Montreal, pp 26–33

Zhuge Y, Garcia-Molina H, Wiener JL (1996) The strobe algorithms for multi-source warehouse consistency. Proceedings of the Fourth International Conference on Parallel & Distributed Information Systems, December 1996: 146–157

Author Biography

Zohra Bellahsene has been an Assistant Professor in Computer Science at the University of Montpellier II, France, since 1987. She received her PhD degree in Computer Science from the University of Paris VI in 1982, and her *Habilitation à Diriger des Recherches* from the University of Montpellier II in 2000. She has devoted her recent research and publications on various aspects of view mechanisms, organizing database summer schools, and serving the committees of French and international conferences. Zohra Bellahsene has published in the following topics: query optimization, object-oriented database views, meta-modeling, human genome databases, schema evolution, distributed database systems, view adaptation in data warehousing systems, data warehouse design, and XML view management.

Correspondence and offprint requests to: Zohra Bellahsene, LIRMM UMR 55060
CNRS–Montpellier II, 161 rue ADA 34392 Montpellier Cedex 5, France.
Email: bella@lirmm.fr