Self-Maintenance of Match Classes in Materialized Integrated Views

Vânia Maria Ponte Vidal, Valéria Magalhães Pequeno

Computer Department, Federal University of Ceará, Brazil
email: {vvidal, valeriap}@lia.ufc.br

Abstract

One approach that has been used for integrating data from multiple databases consists in creating materialized integrated views which are stored in a centralized repository. The queries on the view can be processed directly from the integrated view, with no need for accessing the remote sources. The main difficulty with this approach is to maintain the consistency of the materialized view with respect to the source databases updates. In a distributed environment, it is desirable that the materialized view be self-maintainable, i.e., that it can be maintained without having to access the source databases. Match class is a type of class that frequently occurs in integrated views and usually is not self-maintainable. In this paper, we present a technique for self-maintenance of full match classes. We also show that other types of match classes can easily become self-maintainable using full match classes as auxiliary classes.

1 Introduction

One of the leading issues in database research is to develop flexible mechanisms for providing integrated access to multiple, distributed, heterogeneous databases and other information sources. A wide range of techniques has been developed to address this problem, including approaches based on creation of virtual integrated views [1] and on creation of materialized integrated views [2]. In the latter approach, information from each source database is extracted in advance and, then, translated, filtered, merged and stored in a centralized repository. Thus, when a user’s query arrives, it can be evaluated directly at the repository, and no access to the source databases is required [2]. Therefore, the user’s queries can be answered quickly and efficiently since the information is directly available. The problem, however, is that the views must be updated to reflect changes made to source databases.

Basically, there are two strategies for view maintenance: re-materialization and incremental maintenance. The re-materialization strategy re-computes all the view’s data at pre-established times. The incremental maintenance strategy [3, 2], on the other hand, periodically modifies only part of the view’s data to reflect the updates occurred in the source databases. Although incremental maintenance is often significantly more efficient than re-materialization, it still can be expensive, since it may require querying remote source. There are situations where it is necessary to fetch additional data from the same or from different source databases in order to determine what view updates are necessary, to correctly maintain the view, in response to a source database update. In view
maintenance, a view is said to be self-maintainable [4], when no additional queries over source DBs are required to maintain that view. Most views, however, are not fully self-maintainable.

Self-maintainability is an important characteristic for integrated views, because it may be slow to access remote data sources or even impossible to do it, when those sources are not available. Self-maintainability is also important because of the so called update anomalies problem [5] which occurs since the local DBs and the views can be updated at the same time. These update anomalies might occur when the result of a global query, submitted to a source DB, corresponds to a state which differs from the state the source DB had at the moment in which the source rule notified the update for the global system. In this paper, the term global system is used to denote the integrated view environment.

In this paper, we focus on the problem of self-maintainability of a special type of classes, the match classes. These classes correspond to the relational model’s outerjoin view defined by a “natural outer-equijoin in the key attributes” [6]. Recently, this type of classes has gained importance because they are common in integrated views. However, in most situations, as we discuss in this paper, match classes are not self-maintainable. We present a strategy to perform the self-maintenance of a special type of match class that we call full match classes, which correspond to relational views defined by a “natural full outer-equijoin in the keys attributes”. We will also show, that other types of match classes can easily become self-maintainable using full match classes as auxiliary classes.

The remainder of this paper is organized as follows. In Section 2, we present the basic concepts of the object model used for modeling the view schema and the source DBs schema. In Section 3, we define match classes and how to specify them using Correspondence Assertions. In Section 4, we present our approach to perform the self-maintenance of full match classes. In Section 5, we show how Correspondence Assertions are used to generate the rules for full match classes maintenance. In Section 6, we discuss the related works. In Section 7, we present our concluding remarks.

2 Terminology

In this section, we present the basic concepts of the object model used to represent the integrated view schema and the source DBs schemas. In accordance with the object model described in ODMG-97 [7], we distinguish objects from literals as follows: objects represent real world entities and have an unique identifiers (OID), while literals are special types of objects that have no identifiers.

An object is an instance of a class. Thus, classes serve as templates for their instances. We define the extent of a class C as the set of all instances of C at a given moment. There are two types of classes: the object classes, whose instances are objects; and the literal classes, whose instances are literals. A class definition is a tuple \(<\text{properties}, \text{operations}>\) that specifies the set of properties and the set of operations that are common to all instances of the class. The set of properties of a class C is denoted by \(\text{props}(C)\).

A property of a given class can be defined as a functional relationship between that class objects and the objects (or literals) of the property’s domain. Depending upon the type of its domain, the properties can be classified into attributes and relationships. The domain of an attribute is a literal class. For instance, the attribute \texttt{employeeName}, whose domain is the class \texttt{string}, is an attribute that relates an object of the class \texttt{employee} with an instance of the class \texttt{string}, which is a literal. On the other hand,
the domain of a relationship is an object class. For example, the property \texttt{dept} of the class \texttt{EMPLOYEE}, whose domain is the class \texttt{DEPARTMENT} (Dom (\texttt{dept}) = \texttt{DEPARTMENT}), relates the objects of the class \texttt{EMPLOYEE} with objects of the object class \texttt{DEPARTMENT}.

Properties also can be classified into singlevalued and multivalued. A class property is denoted singlevalued when each instance of the class can be related to at most one object of the property’s domain. Therefore, the value of this property is an atomic object which is an instance of the property’s domain. For example, the property \texttt{employeeName} of the class \texttt{EMPLOYEE} is a singlevalued property. A class property is denoted multivalued when each instance of the class can be associated to many instances of the property’s domain. Therefore, the value of a multivalued property is a collection of objects, which are instances of the property’s domain. For example, the property \texttt{telephone} of the class \texttt{EMPLOYEE} associates a collection of instances of the class \texttt{STRING} to one object of the class \texttt{EMPLOYEE}.

An object oriented schema is a set of classes definitions that serve as templates to generate the application domain objects. In the diagram of Figure 1, rectangles represent classes; single arrows represent singlevalued relationships and double arrows represent multivalued relationships.

![Diagram of an object oriented schema](image)

Figure 1: An object oriented schema

Objects also can be related through paths connecting two or more properties. Taking, for instance, the schema of Figure 1, one can observe that an employee is related to his/her department manager through the path \texttt{dept} \texttt{manager}. Now, we formally define path as:

**Definition 1 (Path)** Let \( C_1, C_2, \ldots, C_{n+1} \) be classes of a schema and \( p_1, \ldots, p_n \) be properties such that \( p_i \in \text{props}(C_i) \) and \( \text{Dom}(p_i) = C_{i+1}, 1 \leq i \leq n \). Therefore, \( p_1 \cdot p_2 \cdot \ldots \cdot p_n \) is a path of \( C_1 \). This means that the instances of \( C_1 \) are related with the instances of \( C_{n+1} \) through the path \( p_1 \cdot p_2 \cdot \ldots \cdot p_n \).

### 3 Integrated views and match classes

Several systems support integrated views (virtual or materialized) to provide integrated access to multiple, distributed, heterogeneous databases. In the object model, a view schema consists of a set of view classes definitions [8]. A kind of class that frequently occurs in integrated views is the so called \textit{match class}, which corresponds to a special type of outerjoin views, those defined by a “natural outer-equi-join in the key attributes” [6]. The objects of a match class are synthesized by merging the objects of different databases which correspond to the same object in the real world.

Figure 2 shows the source databases schemas \texttt{DB}_1 and \texttt{DB}_2, and the views schemas \texttt{V}_1 and \texttt{V}_2. The view class \texttt{EST\&EMP} is a match class whose objects are synthesized by merging objects of the class \texttt{STUDENT} and objects of the class \texttt{EMPLOYEE}, as follows:
• **Case 1:** if a student is also an employee, then there will be only one corresponding object in the view class EST&EMP for which the values of the properties `name`, `telephone`, `major` and `ic` (identification card) are defined from the corresponding object in STUDENT, and the values of the properties `wage` and `deptName` are defined from the corresponding object in EMPLOYEE.

• **Case 2:** If a student is not an employee, then there will be one corresponding object in the view class EST&EMP which has null value for the properties `wage` and `deptName`.

• **Case 3:** If an employee is not a student, then there will be one corresponding object in the view class EST&EMP which has null value for the properties `telephone` and `major`.

![Image](image-url)

**Figure 2:** Source databases schemas DB₁ and DB₂ and views schemas V₁ and V₂

In our approach, as we discuss in Section 5, to generate the rules for maintaining a match class, we use the class’ Correspondence Assertions. The Correspondence Assertions (CAs) of a view class formally specify the relationships between the view class and its base classes. Correspondence assertions are special types of integrity constraints that are used to assert the correspondence among schemas components. In this work, we use the view correspondence assertions to specify how the view objects are synthesized from the base class objects. In the case of match classes, since they are object-preserving classes, their relationships with the base classes can be specified by the following three types of Correspondence Assertions: Extension Correspondence Assertion, Property Correspondence Assertion and Path Correspondence Assertion.

The Extension Correspondence Assertions (ECA) are used to specify which objects of the base classes should have a corresponding *semantically equivalent* object in the view. Two objects e₁ and e₂ are semantically equivalent (e₁ ≡ e₂) if e₁ and e₂ represent the same object in the real world. We define the *root classes* of a view class V as all the classes that
are related to $V$ through some ECA, which means that in the root classes there are objects that are *semantically equivalent* to the objects of the view class. Consider, for example, the match class $\text{STUDENT}_v$ shown in Figure 2. The $\text{STUDENT}_v$ root classes are $\text{STUDENT}$ and $\text{EMPLOYEE}$, which are related to $\text{STUDENT}_v$ through the following ECAs: $\Psi_1$: $\text{STUDENT}_v \equiv \text{STUDENT}$ and $\Psi_2$: $\text{STUDENT}_v \cup \text{EMPLOYEE}$. $\Psi_1$ specifies that $\text{STUDENT}_v$ and $\text{STUDENT}$ are equivalent; i.e., for each $\text{STUDENT}$ object there is one semantically equivalent object in $\text{STUDENT}_v$, and vice-versa. $\Psi_2$ specifies that the classes $\text{STUDENT}_v$ and $\text{EMPLOYEE}$ can have objects in common. In accordance with the type of ECA relating a view class with its root classes, we distinguish six different types of view classes [9]: *equivalence*, *selection*, *union*, *difference*, *intersection* and *overlapping*.

The Properties Correspondence Assertions (PrCAs) and Path Correspondence Assertions (PaCAs) specify how the properties' values of view class objects are derived from the properties' values of its base class objects. For example, the property $\text{wage}_v$ of the view class $\text{STUDENT}_v$ (Figure 2) is defined by the PrCA: $\text{STUDENT}_v.\text{wage}_v \equiv \text{EMPLOYEE.} \text{wage}_1$, which specifies that given an instance $s$ of $\text{STUDENT}_v$, if there is an instance $e$ of \text{EMPLOYEE}, such that $s \equiv e$, then $s.\text{wage}_v = e.\text{wage}_1$. Otherwise, $s.\text{wage}_v = \text{nil}$. The property $\text{deptName}_v$ of the class $\text{STUDENT}_v$ is defined by the PaCA $\text{STUDENT}_v.\text{deptName}_v \equiv \text{EMPLOYEE.} \text{dept}_1 \cdot \text{name}_1$, which specifies that given an instance $s$ of $\text{STUDENT}_v$, if there is an instance $e$ of \text{EMPLOYEE}, such that $s \equiv e$, then $s.\text{deptName}_v = e.\text{dept}_1.\text{name}_1$. Otherwise, $s.\text{deptName}_v = \text{nil}$. It is important to notice that a view class property can be associated with more than one PrCA and/or PaCA. For example, the property $\text{name}_v$ of the view class $\text{STUDENT}_v$ has the following PrCAs: $\text{STUDENT}_v.\text{name}_v \equiv \text{EMPLOYEE.} \text{name}_1$ and $\text{STUDENT}_v.\text{name}_v \equiv \text{STUDENT.} \text{name}_2$. This means that the values of $\text{name}_v$ can be derived from $\text{name}_1$ and $\text{name}_2$.

4 Self-maintenance of match classes

In most cases the match classes are not self-maintainable. To exemplify one such case, we have chosen the match class $\text{STUDENT}_v$ of the view $S_v$ in Figure 2. Figure 3 shows the global rule to maintain $\text{STUDENT}_v$ with respect to insertions in $\text{STUDENT}$. According to this rule, the insertion of an object $s$ in $\text{STUDENT}$ will require the insertion of a corresponding object in the view class $\text{STUDENT}_v$. However, before doing that, we should first verify whether already exists an object $e$ in $\text{EMPLOYEE}$ such that $s \text{ match e (s\equiv e)}$. If it does exist, the object resulting from the fusion of $s$ and $e$ is inserted in $\text{STUDENT}_v$. Otherwise, the object created from $s$ is inserted in $\text{STUDENT}_v$.

---

*When* insert($s$) in $\text{STUDENT}$ /$s$ $s$ is the object inserted in $\text{STUDENT}$*

*Then* If there exists $e$ in $\text{EMPLOYEE}$ such that $s \equiv e$ then

\[
\text{Insert an object } s' \text{ in } \text{STUDENT}_v \text{ such that } s' \text{ is the fusion of } s \text{ and } e
\]

Else

\[
\text{Insert an object } s' \text{ in } \text{STUDENT}_v \text{ such that } s' \equiv s
\]

---

Figure 3: Rule for $\text{STUDENT}_v$ maintenance with respect to insertions in $\text{STUDENT}$

We present a strategy to perform the self-maintenance of a special type of match class, the *full match class*, which corresponds to relational views defined by a "natural
full outer-eqijoin in the keys attributes. In the following, we present our approach for performing the self-maintenance of full match classes, and we show that other types of match classes can become easily self-maintained by using full match classes as auxiliary classes.

4.1 Self-maintenance of full match classes

A match class $F$ is full when its extent is defined by the union of its root classes’ extents; a full match class has similar properties to those of a relational view defined by a “natural full outer-eqijoin in the key attributes” [6]. Formally, a full match class, $F$, is defined by the following ECA $F = C_1 \cup \ldots \cup C_m$, where $C_1, \ldots, C_m$ are the root classes of $F$. Note that, since a class cannot have duplicated objects, if $c_i$ is an instance of $C_i$ and $c_j$ is an instance of $C_j$ such that $c_i \equiv c_j$, for $1 \leq i, j \leq m$ and $i \neq j$, then there will exist only one object $f$ in the extent of $F$ such that $f \equiv c_i \equiv c_j$. The properties of $F$ are semantically equivalent to the properties or paths of its root classes; therefore, path and property assertions are used to formally specify how the properties’ values of $F$ are derived from the properties’ values of its root classes. In the view schemas of Figure 2, the class $EST\&EMP$ is a full match class, while the class $STUDENT_n$ is not.

In order to incrementally maintain full match classes, we use the architecture shown in Figure 4, which was based on the architecture adopted in the Squirrel project [2]. In our approach, the maintenance process consists of two steps:

1. When a relevant\(^1\) update $\tau$ occurs in a source DB $D$, a local rule is fired, sending to the queue of updates, $QU$, the sequence of updates that should be performed in the view to reflect the changes in $D$.

2. Periodically, the execution model processes the updates in $QU$, hence, updating the views classes.

![Figure 4: Architecture for self-maintenance of full match classes](image)

\(^1\)An update in a source DB $D$ is relevant to a view class if the new state of $D$ becomes inconsistent with respect to the state of the view class.
The difference of the Squirrel project's approach [2] from the one described previously is that its local rules just notify the global system about changes in the source DBs which are relevant to the view's class, and the global rules, which are defined in the global system, correctly update the materialized view to reflect the changes of the source DBs. As a consequence, in that approach, a view is self-maintainable with respect to an update \( \tau \) in a source DB \( D \) when the effect that \( \tau \) should have in the view class can be determined by using only the view information. The advantage of our approach as compared to Squirrel's, is that not only can it use the view information but also can use any information from the source DB where the update occurred. However, even these additional informations might not be enough for self-maintenance of full match classes.

As we will show in the next section, there are situations where it is necessary to know of which root classes a given view object is a member. In order to solve this problem, we adopted the following strategy: adding to the full match class a new property, \textit{classes}, whose value is the set of the root classes' names of which an object is a member. Thus, besides the ECA \( F \equiv C_1 \cup ... \cup C_m \), a full match class \( F \) has the following ECAs: \( C_k \equiv F["C_k" \in \text{classes}], 1 \leq k \leq m \), which specify that an instance \( f \) of \( F \) is member of the root class \( C_k \) if and only if "\( C_k \)" \( \in f.\text{classes} \). As we will show, full match classes with the property \textit{classes} are self-maintainable with respect to any update, \( \tau \), in the source DB, \( D \), because the effect that \( \tau \) should have in the view class can be determined by using only the information in \( D \) and the information in the view.

Match classes that are not full, are not self-maintainable, either. Consider, for example, the match class \textit{student}\(_v\) in Figure 2. Note that the rule of Figure 3 requires an access to the source DB\(_1\), in order to verify whether already exists an object in \textit{employee} which is semantically equivalent to the object inserted in \textit{student}. This information can be obtained neither locally, in DB\(_2\), nor in the view. Therefore, \textit{student}\(_v\) is not self-maintainable. However, it can become self-maintainable by using the full match class \textit{est\&amp;emp} as an auxiliary class. In this case, \textit{student}\(_v\) is derived directly from the class \textit{est\&amp;emp} and it can be implemented as a virtual class or as a materialized class. If it is implemented as a virtual class, its definition is a mere selection (\textit{select e from e in est\&amp;emp where "student" in e.classes}). If it is implemented as a materialized class, then it can be directly maintained from the auxiliary class \textit{est\&amp;emp}, and there is no need for generating local rules to maintain it. The idea is that the effect of updates in the source DBs are propagated (by the local rules) to the auxiliary class, \textit{est\&amp;emp}, and then, the effect of the updates in the auxiliary class, \textit{est\&amp;emp}, are propagated (by the global rules) to the \textit{student}\(_v\) class. In the next section, we present the local rules that should be generated to maintain full match classes.

5 Generating the rules for full match classes maintenance

In our approach, the rules for maintaining the full match class \( F \) are defined based on \( F \)'s correspondence assertion, which formally specify the relationships between the view class and its base classes. In our approach, CAs are treated as special types of integrity constraints that should be preserved by the update operations in the source DB. Thus, given an update in a source DB, it is necessary to define the sequence of updates that must be performed in the view to make it "synchronized" with the new source DB state. This problem can be solved by determining the sequences of view updates which are required for maintaining the view's correspondence assertions, which are relevant to that update.
A CA is relevant to an operation if it can be violated by the operation. In that way, the rules required for the incremental maintenance of a view class V are those required for the maintenance of V’s correspondence assertions. The process of defining the rules required for maintaining a given correspondence assertion Ψ consists of two steps:

1. First, one identifies the source DBs update operations that can violate Ψ, that is, the Ψ’s relevant operations;

2. Then, for each relevant operation identified in (1), a rule should be generated, whose event consists of the update operation identified and whose action consists in sending to the global system the sequence of updates which should be performed in the view so that Ψ is preserved.

The use of correspondence assertions allows us to formally justify that the rules generated by our approach correctly maintain the view. That means that the view updates requested by the local rules reflect exactly the changes that are needed in order to keep the view consistent with the sources DBs changes. To illustrate the process described above, consider the full match class EST&EMP shown in Figure 2. To maintain EST&EMP, we need to generate the rules to maintain the EST&EMP’s CAs defined in Figure 5. For the ECA Ψ₂, [EMPLOYEE ≡ EST&EMP [“EMPLOYEE” ∈ classes]], one should generate a rule for the following relevant operations: (i) deletions in EMPLOYEE and (ii) insertions in EMPLOYEE. For shortness of presentation, in the following, we write insertion/deletion in EMPLOYEE instead of insertion/deletion in the extent of EMPLOYEE.

<table>
<thead>
<tr>
<th>Extent CAs</th>
<th>Properties CAs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ψ₁:STUDENT ≡ EST&amp;EMP [“STUDENT” ∈ classes]</td>
<td>Ψ₄:EST&amp;EMP.name ≡ EMPLOYEE.name₁</td>
</tr>
<tr>
<td>Ψ₂:EMPLOYEE ≡ EST&amp;EMP [“EMPLOYEE” ∈ classes]</td>
<td>Ψ₅:EST&amp;EMP.name ≡ STUDENT.name₂</td>
</tr>
<tr>
<td>Ψ₃:EST&amp;EMP ≡ STUDENT U EMPLOYEE</td>
<td>Ψ₆:EST&amp;EMP.wage ≡ EMPLOYEE.wage₁</td>
</tr>
<tr>
<td>Path CAs</td>
<td>Ψ₇:EST&amp;EMP.major ≡ STUDENT.major₂</td>
</tr>
<tr>
<td>Ψ₁₁:EST&amp;EMP.deptName ≡ EMPLOYEE.dept₁ • name₁</td>
<td>Ψ₈:EST&amp;EMP.telephone ≡ STUDENT.telephone₂</td>
</tr>
<tr>
<td></td>
<td>Ψ₉:EST&amp;EMP.ic ≡ EMPLOYEE.ic₁</td>
</tr>
<tr>
<td></td>
<td>Ψ₁₀:EST&amp;EMP.ic ≡ STUDENT.ic₂</td>
</tr>
</tbody>
</table>

Figure 5: EST&EMP’s correspondence assertions

The rule generated for maintaining Ψ₂ with respect to insertions in EMPLOYEE is shown in Figure 6. Note that the rule’s event is parameterized with the object c that was inserted in EMPLOYEE, hence, c can be referenced in the rule’s action. The rule’s action consists of two steps:

1. First (line 2 in Figure 6), the object specification c’ is created, i.e., the method “CreateOS_of_EST&EMP” returns an object specification of EST&EMP, where the values for EST&EMP’s properties (name, wage, ic, telephone, major and nameDept) are assigned in accordance with the corresponding values of c’s properties. This correspondence is defined based on the Properties and Path assertions of EST&EMP that
specify the correspondence between the EST&EMP’s properties and the EMPLOYEE’s properties.

2. Then, the updates to be processed in the view class EST&EMP are sent to the global system GS (lines 3-7 in Figure 6), which can be one of the cases described below.

01. When insert(c) in EMPLOYEE  /* c is the inserted object in EMPLOYEE */
02. Then  c' = c.CreateOS.of(EST&EMP);
03. Request the following updates for GS:
04. < If there exists e in EST&EMP such that e ≡ c' then  /* Case 1 */
05.   e.MakeFusion(c');
06.  Else  /* Case 2 */
07.     EST&EMP.CreateObject(c');>

Figure 6: Rule for Ψ2’s maintenance with respect to insertions in EMPLOYEE

**Case 1:** There exists an object in EST&EMP that is semantically equivalent to c’

If there already exists an object e in EST&EMP, such that e ≡ c’, then the method “MakeFusion” is called, and c’ is passed as a parameter. The method “MakeFusion”, shown in Figure 7, makes the fusion of the object e with the object c’ in the following way:

1. It adds the root class’ name that originated c’ in the e.classes collection. c’ has a property sourceClass whose value is the class that originated c’. In our example, the sourceClass of c’ is the class EMPLOYEE. Thus, EMPLOYEE is inserted in the e.classes collection, which means that e has become a member of the class EMPLOYEE.

2. Assign a value to each property p_j of e whose value is null and the value of the corresponding property in c’ is not null. As shown in Figure 7, if the property p_j is an attribute, then the corresponding property’s value in c’ is a literal, which can be directly assigned as the value of p_j. For example, the values of EST&EMP’s attributes name, wage, telephone, ic and nameDept are directly defined from the corresponding properties in c’. If the property p_j is a singlevalued relationship, then the corresponding property’s value in c’ is an Object Identifier Specification (OIS) which is used for matching the corresponding object in the view. For example, the property major of EST&EMP is a singlevalued relationship whose domain is DEPARTMENT. In this case, the value of the corresponding property in c’ is an OIS which is used to identify the corresponding object o in the view class DEPARTMENT_v. The object o is, then, assigned as the value of the property major of e. When the properties are multivalued relationships, we have to repeat the procedure used for the singlevalued relationships for each object in the collection.

**Case 2:** There exists no object in EST&EMP that is semantically equivalent to c’
\begin{verbatim}
MakeFusion(Object_Specification c')
{
    this.classes.insert_element(c'.SourceClass);
    For each property p_j \in this.properties, 1 \leq j \leq n, do
        If c'.p_j \neq nil and this.p_j = nil then
            If p_j is an attribute then
                this.p_j = c'.p_j;
            Else /* p_j is a relationship */
                Select the object o of Dom(p_j) matching c'.p_j;
                this.p_j = o;
        Else
            For each id in c'.p_j do
                Select the object o of Dom(p_j) matching id;
                this.p_j.insert_element(o);
}
\end{verbatim}

Figure 7: MakeFusion Method

If there exists no object e in est&emp, such that e \equiv c’, the method “CreateObject” is called and c’ is passed as parameter. The method “CreateObject” is very similar to the method “MakeFusion” discussed previously. In [10], we developed algorithms which take as input the CAs of a full match class and automatically generate all the rules needed for maintaining the class and the methods used by those rules.

6 Related works

There has been a significant amount of research in the database community related to the problem of incrementally maintaining materialized view [2, 6], most of which is focused on relational views. In [6], the authors present an algorithm for incremental maintenance of outerjoin and match views. They also identify the conditions under which an outerjoin view is self-maintainable. One such condition is that all view attributes should be semantically equivalent to attributes of its “root relations”, which is violated, for example, by the attribute dept.Name of the view est&emp. Therefore, in their approach, the view est&emp is not self-maintainable with respect to insertions in employee, because the value of dept.Name cannot be obtained directly from the attributes’ values of the object inserted in employee. This is clearly a limitation of their approach. Another limitation of that approach is that the algorithm cannot be adapted for the object model because it is not possible to deal with relationship properties. The values of a relationship property are objects, and the source OIDs have no meaning in the global system.

As demonstrated in this paper, our approach overcame these limitations, since it is the sources’ rule that determines the effect that the source updates should have in the view class. Hence, the sources’ rules can obtain, locally, the value of path derived properties, such as dept.Name, and the value of relationships properties, by generating the object identifier specifications (OIS) which are necessary to identify the corresponding objects.
in the global system.

7 Conclusions

In this paper, we considered the problem of incrementally maintaining match classes in integrated views. We presented a strategy for self-maintaining full match class, and we showed, that other types of match classes can easily become self-maintainable using full match classes as auxiliary classes.

In our approach, full match classes are maintained by the sources’ rules, which means that the rules of a given source database send to the global system the sequence of updates to be performed in the view in order to reflect the changes in that source database. An innovation of our rules is that, when additional information from their source database is needed, it can be locally extracted from the source database and sent to the global system. Avoiding, in that way, later access to the source database. As shown in the examples, our approach overcomes other limitations of [6], since it can manage self-maintenance of full match classes that have path derived properties and relationship properties.

Another innovation of our approach is that the rules are defined based on the correspondence assertions, which formally specify the relationships between the view class and its base classes. An advantage of using correspondence assertions is that the rule generation process can be automated. In [10], we developed algorithms which take as input the correspondence assertions of a full match class and automatically generate all the rules needed for maintaining the class and the methods used by those rules. Another advantage of using correspondence assertions is that they allow us to formally justify that the rules generated by our approach correctly maintain the view. That means that the view updates requested by the rules of a given source database reflect precisely the changes that are needed in order to keep the view consistent with the changes in that source database.

References


