Reduction of Update Propagation Overhead for Data Warehouses Incorporating OODBs

Zhiyong PENG
Advanced Soft. Tech. & Mech. Research Institute of Kyoto, Japan
peng@astem.or.jp

Yahiko KAMBA YASHI
Department of Information Science Kyoto University, Japan
yahiko@kuis.kyoto-u.ac.jp

Abstract

Since the requirement of data warehouse is increasing, it is necessary to develop efficient mechanisms for data warehouses incorporating various kinds of database systems. Essentially, materialized views are used for data warehouses and various methods for relational databases have been developed. In this paper, we will first discuss a method to realize materialized views for object-oriented databases (OODBs) using deputy objects. Since materialized views require to generate duplication of data, improvement of efficiency for update operations is important. We will handle this problem by the following two approaches: (1) reduction of duplicates by considering inheritance property, (2) speed-up update propagation by considering communication delay. Our approach allows objects to be updated at data warehouse sides. If a data object is updated, the effect of update propagation in the same data warehouse must be completed as soon as possible. If update information is transmitted to the original OODB and the induced update operations are transmitted from the OODB to the data warehouse, the delay required for update completion will be large due to the communication delay between the data warehouse and the OODB systems. We will try to avoid communication delays for update propagation in the same data warehouse.

Keywords data warehouses, materialized views, OODBs, object deputy model

1 Introduction

Data warehousing has drawn more and more attention from industrial and academic circles. A data warehouse is a data repository containing integrated information from distributed, autonomous and heterogeneous sources. When a user's query is submitted to the data warehouse, the required information is already there, with inconsistencies and differences already resolved.

The current efforts on data warehousing are mainly focused on relational databases. In the recent years, object-oriented databases have extensively been used to manage complex information, such as multimedia databases and digital libraries. Therefore, it becomes important to develop advanced data warehouse systems that can incorporate object-oriented databases. Suppose that various academic papers, technique reports and scientific books are classified, stored and managed by object-oriented databases distributed on the internet. As an application example, a document warehouse system can be realized for researchers to collect documents related to their research themes from these sources and find valuable information through various analysis tools. The document warehouse is maintained through update propagation. For example, when a new document is added into some source, it will be automatically dispatched to all the related document warehouses. In addition, there are some documents written by users that may be included in their warehouses. In order to allow users to update their own documents, updates operations at data warehouse sides should also be supported. However, most of data warehouse systems can only provide supports for updates made to the source databases.

Materialization technology of views of databases is required for realization of data warehouses. Materialized views of relational databases have greatly been studied, but very few papers are concerned with materialized views of object-oriented databases. In the paper[8], we proposed to realize object views by deputy objects. The major features of deputy objects are summarized as follows.

1. A deputy object has its own persistent identifier, and may have additional attributes and methods that are not derived from its source object(s).

2. The attributes and methods of an object can be inherited by its deputy objects. The inheritance is realized through switching operations that can change the names and types of the inherited attributes and methods.
3. There is a bilateral link between an object and one of its deputy objects, which allows not only inheritance but also update propagation between them.

Views realized by deputy mechanisms are semi-materialized since attributes and methods of objects are inherited by switching operations that are executed on demand. In this paper, we extend deputy mechanisms to allow the attribute values of objects to be replicated by their deputy objects. Thus, deputy mechanisms can be used to realize materialized views suitable for data warehousing.

At data warehouses, redundant data are introduced to achieve query optimization, but they may cause increment of update propagation overhead. We will handle this problem by the following approaches. One is to reduce duplication of data, and the other is to design algorithms that can speed up update propagation. The features of object-oriented databases and data warehouses are exploited to achieve such purposes.

In object-oriented databases, objects are classified by classes which are organized into a hierarchy structure that can reduce information redundancy. If both the superclass and the subclass are extracted into data warehouses without considering their inheritance relationship, instances of the subclass will be replicated twice since they also indirectly belong to the superclass. By considering inheritance property, we have developed an algorithm that can extract the class hierarchy selected from an OODB into data warehouses without the above unnecessary data duplication.

Suppose that an object has two copies at the same data warehouse. When a copy is updated, the update will be propagated to the underlying OODB and then transmitted back to the data warehouse for modifying another copy. In our way, since an object has a single copy at the data warehouse, it can avoid such communication delays between the data warehouse and the OODB systems.

Our previous deputy mechanisms did not consider redundant data. Therefore, as complement of data update propagation, we develop method application propagation which is used to trigger update methods for rematerialization of the replicated attribute values of deputy objects when objects are updated. During design of procedures for update propagation, the following features of data warehouses are considered in order to speed up update propagation. That is, query at data warehouse does not interfere with local processing at sources since data are copied at data warehouses, where they are integrated in advance. Data can also be modified, annotated and summarized at the data warehouse. When a data object is updated, if all relevant data at the same data warehouse are also updated accordingly, the data warehouse can be quickly recovered without waiting the completion of update propagation at the source or the other data warehouses. Therefore, our algorithms are designed to avoid communication delays between data warehouses and OODBs by completing update propagation inside of the data warehouse as soon as possible.

Bilateral link structure of deputy objects also facilitates speed-up update propagation. Since each object holds identifiers of all of its deputy objects, when an object is updated, only its deputy objects are accessed without checking the other irrelevant objects or deputy objects. In addition, each replicated attribute of deputy objects has their own method to rematerialize their value. Once the original attribute value is updated, only the relevant method is applied. The other attribute values of the same object need not to be recomputed.

The paper is organized as follows. Section 2 introduces an object deputy approach for data warehousing. Reduction of update propagation overheads is discussed in Section 3. We will compare the related work in Section 4.

2 An Object Deputy Approach for Data Warehousing

An integrated access to multiple, distributed, heterogeneous information sources is usually based on view mechanisms that are realized by computation methods. That is, the user query is divided into several sub-queries that are executed by information sources. The results of the sub-queries are integrated and returned to the user. Data warehousing provides an alternative way that is based on materialized view mechanisms. At data warehouses, the data of interest are extracted from information source, and integrated in advance. When a user query is issued, the query is evaluated directly at the data warehouse, without accessing the underlying information sources.

The current data warehousing systems[11, 12] usually assume that sources and the warehouse use the relational data model. This is because a large amount of legacy data are stored in relational databases, of which materialized view techniques have been successfully developed. In this paper, we will try to realize data warehouses that also include object-oriented databases as their information sources. As shown in Fig. 1, the basic idea is (1) to select the objects of interest from object-oriented databases, (2) create their materialized deputy objects to avoid communication delays, and (3) form an integrated and application-specific view at data warehouses by properly selecting combination of computed and materialized deputy objects, considering the trade-off problems between storage cost and computation overhead.
The concept of deputy objects was at first introduced by authors for the unified realization of object views, roles and migration. In order to illustrate that it is also useful for data warehousing, we will review its definition.

In object-oriented databases, real-world entities are represented in term of objects. Objects are identified by system-defined identifiers which are independent of objects' states. An object has attributes which represent properties of a corresponding real-world entity. The state of an object is represented by its attribute values, which are read and written by basic methods. In addition, there are general methods that represent behavior of objects. Objects having the same attributes and methods are clustered into classes which make it possible to avoid specification and storage of redundant information. A formal definition of objects and classes is given as follows.

**Definition 1** Each object has an identifier, attributes and methods. Schema of objects with the same attributes and methods is defined by a class which consists of a name, an extent and a type. The extent of a class is a set of objects belonging to it, called its instances. The type of a class is definitions of its attributes and methods. A class named as C is represented as

\[ C = \{ o \}, \{ T'_a : a \}, \{ m : \{ T_p : p \} \} \]

1. \( \{ o \} \) is the extent of C, where o is one of instances of C.
2. \( \{ T'_a : a \} \) is the set of attribute definitions of C, where a and \( T'_a \) represent name and type of an attribute, respectively. The value of attribute a of object o is expressed by o.a. For each attribute \( T'_a : a \), there are two basic methods: read(o,a) for reading o.a and write(o,a,v) for writing o.a with the new value v, expressed as follows.

\[
\text{read}(o, a) \Rightarrow o.a, \\
\text{write}(o, a, v) \Rightarrow o.a := v
\]

Here, \( \Rightarrow, \Rightarrow \) and \( := \) stand for operation invoking, result returning and assignment, respectively.
3. \( \{ m : \{ T_p : p \} \} \) is the set of method definitions of C, where m and \( \{ T_p : p \} \) are method name and a set of parameters, p and \( T_p \) represent parameter name and type, respectively. Applying method m to object o with parameters \( \{ p \} \) is expressed as follows.

\[
\text{apply}(o, m, \{ p \})
\]

Deputy objects are defined as extension and customization of objects. An object can have many deputy objects that are used to customize objects for different applications or represent its many facet nature. The schemata of deputy objects are defined by deputy classes that are derived by creating deputy objects as their instances, generating
switching operations for inheritance of attributes and methods, and adding definitions for their additional attributes and methods. A formal definition of deputy objects and deputy classes is given as follows.

**Definition 2** A deputy object is generated from object(s) or other deputy object(s). The latter is called source object(s) of the former. A deputy object must inherit some attributes/methods from its source object. The schema of deputy objects with the same properties is defined by a deputy class, which includes a name, extent and type. Deputy classes are derived from classes of source objects, called source classes. In general, let $C^* = \{\{o^s\}, \{T_{o^s} : a^s\}, \{m^s \mid T_{p^s} : p^s\}\}$ be a source class. Its deputy class $C^d$ is defined as

$$C^d = \{\{o^d \mid (o^d \to o^s) \lor (o^d \to \ldots \times a^s \times \ldots) \lor (o^d \to \{o^s\})\lor sp(o^s) \lor cp(\ldots \times a^s \times \ldots) \lor gp(\{o^s\}) == \text{true}\}, \{T_{ad} : a^d\} \cup \{T_{ad} : a^d\}, \{m^d \mid T_{pd} : p^d\} \cup \{m^d \mid T_{pd} : p^d\}\}$$

1. $\{o^d\}(o^d \to o^*) \lor (o^d \to \ldots \times a^s \times \ldots) \lor (o^d \to \{o^s\})\lor sp(o^s) \lor cp(\ldots \times a^s \times \ldots) \lor gp(\{o^s\}) == \text{true}\}$ is the extent of $C^d$, where $(o^d \to o^*) \lor (o^d \to \ldots \times a^s \times \ldots) \lor (o^d \to \{o^s\})$ represents that $o^d$ is the deputy object of $a^s$, $\ldots \times a^s \times \ldots$, or $\{o^s\}$; $sp$, $cp$ and $gp$ represent selection, combination and grouping predicate, respectively.

2. $\{T_{ad} : a^d\} \cup \{T_{ad} : a^d\}$ is the set of attribute definitions of $C^d$.

   (a) $\{T_{ad} : a^d\}$ is the set of the attributes inherited from $\{T_{ad} : a^d\}$ of $C^*$, of which switching operations are defined as

   $\text{read}(o^d, o^s) \Rightarrow f_{T_{ad} \to T_{ad}}(\text{read}(o^*, o^s))$,

   $\text{write}(o^d, o^s, o^d) \Rightarrow \text{write}(o^d, o^s, f_{T_{ad} \to T_{ad}}(o^*))$,

   $\{T_{pd} : p^d\} \cup \{T_{pd} : p^d\}$ is the set of additional attributes of $C^d$, of which basic methods are defined as

   $\text{read}(o^d, a^d) \Rightarrow 1 o^d, a^d$,

   $\text{write}(o^d, a^d, o^d) \Rightarrow a^d : o^d$,

   (b) $\{T_{pd} : p^d\} \cup \{T_{pd} : p^d\}$ is the set of method definitions of $C^d$.

   (a) $\{T_{pd} : p^d\}$ is the set of the methods inherited from $\{T_{pd} : p^d\}$ of $C^*$, of which are applied through switching operations as

   $\text{apply}(o^d, m^s, \{o^s\}) \Rightarrow 1$,

   $\text{apply}(o^d, m^s, \{T_{pd} \to T_{pd}(p^*)\})$,

   (b) $\{T_{pd} : p^d\}$ is the set of the additional methods of $C^d$, which are applied as

   $\text{apply}(o^d, m^s, \{o^s\}) \Rightarrow 1$,

   $\text{apply}(o^d, m^s, \{T_{pd} \to T_{pd}(p^*)\})$.

According to the above definition, deputy objects at data warehouses have persistent identifiers but their attribute values inherited from source objects are still computed through switching operations that need communicate with the underlying information sources. However, data warehousing requires queries to be evaluated locally since information sources may be remote or unavailable for some time. For this reason, we extend deputy mechanisms to allow deputy objects to materialize their inherited attribute values. The definitions of basic methods for the inherited attribute of a materialized deputy object are changed as follows.

$$\text{read}(o^d, a^s) \Rightarrow \text{read}(o^*, a^s)$$

That is, the inherited attribute values are pre-computed and can be directly read from the deputy object. Thus, queries at warehouses need not interfere with objects at remote sources. The update of the inherited attribute value of a deputy object need to be reflected in its source object(s). Therefore, the writing method is first to update the precomputed value and then propagate the change into the original one through the switching operation. On the other hand, the update of the original attribute value requires to recompute the inherited attribute value of the deputy object. This operation is realized by introducing another basic method for each inherited attribute. The basic update method updates the inherited attribute value according to its function dependence relationship defined by the switching operation. It is triggered when the original attribute value is updated.

In object-oriented databases, there are general methods which are defined by classes. Since a method may invoke another method and its implementation depends on the definition of attributes, these methods can not be simply copied and executed at data warehouses. Only when the attributes and methods on which the method depends can be accessed and invoked with the same names and types at data warehouses, it can be copied and applied to deputy objects independently of objects at sources. For this reason, the initial deputy objects of objects selected from the underlying OODBs are materialized at data warehouses without changing their attribute and method definitions. Furthermore, deputy mechanisms allow deputy objects to have their own deputy objects. Thus, by defining deputy objects of deputy objects, we can customize data warehouses for different application objectives.

After the objects of interest are extracted from multiple heterogeneous object-oriented databases by creating their materialized deputy objects at
In order to reduce storage overhead, deputy objects derived for integration are usually not materialized. If the integration requires time-consumption type conversion, some of them will be materialized. For example, converting images from a format into another may take too much time. Therefore, the deputy objects of those image objects with formats different from the uniform one should be materialized, and the others need not. Thus, an integrated view at data warehouses is realized by the computed/materialized deputy objects or their mixtures (Even in the same deputy object, some of its inherited attributes can be defined to be materialized, and the others to be unmaterIALIZED).

3 Reduction of Update Propagation Overhead

Data warehousing realizes high performance queries by means of data duplication, of which side effects are increment of update propagation overhead. In addition, since data may be copied from the remote sources, communication delay is another major cause for slow recovery of data warehouses when updates occur. We will try to reduce update propagation overhead by (1) reducing unnecessary redundant data and (2) avoiding unnecessary communication delays between data warehouses and source OODBs.

(1) Reduction of Redundant Data

As discussed in Section 2, the extended deputy mechanisms can define the computed/materialized deputy objects and their mixtures, which can be used to realize selective materialized views so that the unnecessary redundant data at data warehouses can be reduced.

By exploiting features of OODBs, storage overhead can be reduced further. In object-oriented databases, objects are classified into classes, which are organized into a hierarchy. The upper classes are called superclasses of the lower ones that are conversely called subclasses of the upper ones. The superclass includes instances of its subclass, and the subclass inherits attributes and methods of its superclass. Suppose there are two classes, Rectangle and Right-Rectangle, where the former is a superclass of the latter. As shown in Fig. 2 (a), if both of them are extracted into data warehouses without considering their inheritance relationship, right rectangles will be replicated twice since right rectangles also indirectly belong to the class Rectangle.

In order to avoid twice replication of instances of the subclass, we extract the superclass by firstly creating its deputy class that only includes deputy objects of its direct instances. The deputy class is then merged with the deputy class of the subclass by the union operation to derive a deputy class that can be used at data warehouses as if it would
be the superclass. Since the deputy class derived by the union operation is not materialized, right rectangles will not be replicated twice as shown in Fig. 2 (b). In general, we can give the following algorithm that can derive deputy classes without unnecessary instance replication from the selected classes according to their inheritance relationships.

Let $S = \{C_1, \ldots, C_m\}$ be the classes selected from an object-oriented database. We assume that $T$ is a set of the classes that have been extracted, $D_i$ is a deputy class of $C_i$ and $D'_i$ is a deputy class of $C_i$ with the extent only including deputy objects of direct instances of $C_i$.

While ($S \neq \emptyset$) do
{
    For each $C_i \in S$ that has not any subclass in $S$,
    if $C_i$ has subclasses $\{C_{i1}, \ldots, C_{in}\}$ in $T$,
    then
    {
        if ($C_i$ has direct instances)
        then
        {
            1) To create a materialized deputy class $D_i$,
            2) To create a computed deputy class $D'_i$ as the union of $D_{i1}, \ldots, D_{in}$
        }
        else
        {
            1) To create a computed deputy class $D'_i$ as the union of $D_{i1}, \ldots, D_{in}$
            2) To delete $C_i$ from $S$ and input $C_i$ into $T$
        }
    }
    else
    {
        1) To create a computed deputy class $D'_i$ as the union of $D_{i1}, \ldots, D_{in}$
        2) To delete $C_i$ from $S$ and input $C_i$ into $T$
    }
}

In order to illustrate the above algorithm, we give an example as shown in Fig. 3. According to the algorithm, the class hierarchy of an OODB system is extracted into the data warehouse in the following way:

1. $S = \{C_1, C_2, C_3, C_4\}$ and $T = \emptyset$;
2. $C_2$ and $C_4$ are selected since they have not any subclass in $S$;
3. The materialized deputy classes $D_2$ and $D_4$ are created since $C_2$ and $C_4$ have not any subclass in $T$;
4. $S := S - \{C_2, C_4\}$ and $T := T + \{C_2, C_4\}$;
5. $S = \{C_1, C_3\}$ and $T = \{C_2, C_4\}$;
6. $C_3$ is selected since it has not any subclass in $S$;
7. $S := S - \{C_3\}$ and $T := T + \{C_3\}$;
7). A materialized deputy class $D_3$ is created since $C_3$ has a subclass $C_4$ in $T$ and has a direct instance;
8). A computed deputy class $D_3$ is derived as the union of $D_4$ and $D_5$;
9). $S := S \setminus \{C_3\}$ and $T := T + \{C_3\}$;
10). $S = \{C_1\}$ and $T = \{C_2, C_3, C_4\}$;
11). $C_1$ is selected since it has not any subclass in $S$;
12). A computed deputy class $D_1$ is created as the union of $D_2$ and $D_3$ since $C_1$ has subclasses $C_2, C_3$ in $T$ and has not any direct instance;
13). $S := S \setminus \{C_1\}$ and $T := T + \{C_1\}$;
14). $S = \emptyset$ and $T = \{C_1, C_2, C_3, C_4\}$;
15). Stop since $S = \emptyset$

Reduction of redundant data in this method can decrease communication delays for update propagation between data warehouses and OODBs. Consider the example shown in Fig. 2 again. If a right rectangle is replicated twice, when a copy is updated at the data warehouse, the update need be transmitted to the source which will transmit the update back to the data warehouse again for modifying another copy. Thus, there are twice communication delays. In this method, it is enough to communicate only once since each right rectangle has a single copy at the data warehouse.

(2) Speed-Up Update Propagation

Schemata of data warehouses are self-contained and queries can be executed at data warehouses independently of the sources. When a deputy object is updated, if all relevant deputy objects at the same data warehouse are also updated accordingly, the data warehouse can be recovered immediately without waiting completion of update propagation at sources. This feature can be used to speed up update propagation at data warehouses by designing algorithms that complete all of the relevant updates at data warehouses before transmit updates to the remote sources. Based on the bilateral link structures of deputy objects, we can realize update propagation procedures to satisfy such requirements.

As outlined in Section 2, there are the semantic constraints between objects and their deputy objects that are defined as predicates of deputy classes. The selection predicate determines existence of a deputy object according to the state of its single source object. Only when the source object satisfies some special condition, its deputy object can exist. The combination and grouping predicates define the existence conditions between a deputy object and several source objects. In order to enforce these semantic constraints, data update propagations between deputy objects and their source objects need to be supported.

There are three basic types of update operations, these being addition, deletion of an object and modification of attribute value of an object. When an object is added, the update should be propagated into all of deputy classes where possible, since the added object may satisfy existence conditions defined by some deputy classes. If so, new instances of these deputy classes will be created as deputy objects of the added object. Deleting an object may cause deletion of its deputy objects. Modification of an attribute value of an object may cause addition and deletion of its deputy objects since it may become satisfied or unsatisfied some existence conditions after modification.

We have designed three procedures for data update propagation (for detail, to see [8]), which are caused by addition of an object, deletion of an
object and modification of attribute value of an object, respectively. They are realized based on bilateral links between objects/classes and their deputy objects/classes as well as various semantic constraints defined by deputy classes. When the above basic update operations occur on instances of some classes, their deputy classes will be examined so that the updates on instances of deputy classes are required in order to maintain semantic constraints defined by deputy classes. Since these update procedures are designed to propagate updates to the deputy objects ahead of to the source objects, they can complete update propagation at data warehouses before transmitting updates to the remote sources.

By the extended deputy mechanisms, some inherited attribute values of deputy objects are materialized. If the corresponding original attribute values are updated, the inherited ones of the deputy objects should be recomputed by the update methods. In order to trigger the application of the update methods for rematerialization of the inherited attributes, we will introduce method application propagation as complement of the above update propagation.

We classify the attributes of objects and deputy objects into the original ones and the inherited ones. The former includes the attributes of objects and the additional attributes of deputy objects, and the latter is the attributes of deputy objects except their additional ones. For the original attributes, their values are updated directly by the write methods. The modification of an inherited attribute of a deputy object will propagate the update to the source object. When an attribute value is updated, all of the materialized attributes that are inherited from it directly or indirectly need be recomputed. This is done by applying the update methods. To trigger the update method applications when an attribute is modified, the following procedure is designed for each class or deputy class.

In general, a deputy class is supposed to have the inherited attributes \( \{T_{a^*} : a^*\} \) and the original attributes \( \{T_{a^*} : a^*\} \). A class can be taken as a special case of a deputy class only with the original attributes. Suppose that \( T_{a^*} : a^* \) is inherited from \( T_{a^*} : a^* \). The procedure for triggering application
of update methods is realized by the following algorithm, where \( f^d(o) \) represents all deputy objects of \( o \), of which source object is represented as \( o' \).

\[
\text{trigger}(o, a) \\
\{ \\
\text{If } (a == a^o) \\
\text{Then} \\
\text{For All } o^d \in f^d(o) \text{ Do trigger}(o^d, a) \\
\ldots \\
\text{If } (a == a^t) \\
\text{Then trigger}(o, a^d) \\
\ldots \\
\text{If } (a == a^d) \\
\text{Then} \\
\{ \\
\text{If } (o.a^d \text{ is materialized}) \\
\text{Then} \\
\text{If } (o.a^d \neq \text{precomputed value}) \\
\text{Then} \\
\text{update}(o, a^d), \\
\text{For All } o^d \in f^d(o) \text{ Do trigger}(o^d, a) \\
\} \\
\text{Else For All } o^d \in f^d(o) \text{ Do trigger}(o^d, a) \\
\ldots \\
\text{If } (a \notin \{a^o \cup \{a^d\} \cup \{a^t\}) \text{ Then exit} \\
\}
\]

This procedure is invoked by the write operation of the original attribute or the inherited attribute with the precomputed value. The write operation first updates the corresponding attribute value, then invokes the above procedure and finally switches the update to the source object. In this way, the method application propagation can also complete rematerialization at data warehouses before transmitting updates to the remote sources. This can be illustrated by an example as shown in Fig. 4.

It is supposed that a materialized deputy object \( o \) has an attribute \( a \) with the value "2", which is copied from \( o' \) at an OODB. There are two deputy objects of \( o \) (named as \( o_{11}, o_{12} \)) which inherit the attribute \( a \) of \( o \) with the name \( a_{11} \) and \( a_{12} \) by converting its type from string into integer and float, respectively. In addition, \( o_{11} \) has its own deputy objects, \( o_{21} \) and \( o_{22} \), which inherit \( a_{11} \) with the name \( a_{21} \) and \( a_{22} \) through plusing and multiplying \( a_{11} \) by 3 respectively. We assume that \( o_{11}, a_{11}, o_{12}, a_{12} \) and \( o_{22}, a_{22} \) are materialized and \( o_{21}, a_{21} \) is not. \( o_{11}, a_{11} \) is assumed to be modified by \( \text{write}(o_{11}, a_{11}, 3) \). According to the above algorithm, the method application propagation caused by \( \text{write}(a_{11}, a_{11}, 3) \) will be done as annotated in Fig. 4. That is, after all materialized attribute values, \( o.a, o_{11}.a_{11}, o_{12}.a_{12} \) and \( o_{22}.a_{22} \) are updated through the method application propagation, \( \text{write}(o', a^o, 3) \) will be sent to the source. The data warehouse can be immediately recovered without waiting the completion of \( \text{write}(o', a^o, 3) \). Thus, we can avoid communication delays for update propagation between data warehouses and OODBs.

Based on the bilateral link structures, the above update propagation doesn't touch irrelevant objects or deputy objects. Furthermore, each materialized attribute has its own update method. The whole object need not be updated while only the relevant attribute value is concerned with. Such a much finer-grained control can also reduce update propagation overhead. In addition, by comparing the precomputed value with what is computed from the original one, some unnecessary update propagation can be detected and cancelled.

4 Related Work

The main problem of object-oriented databases is lack of flexibility. Although a lot of view mechanisms [1, 2, 4, 5, 9] were published, to our knowledge, almost no commercial object-oriented databases provide true view supports. We know that flexibility of relational databases is due to their data independence that enables data to be divided and combined very easily. Similarly, a flexible object-oriented database should also allow objects to be restructured. Without this feature, view mechanisms are difficult to be incorporated into commercial object-oriented databases.

We introduced deputy mechanisms for object-oriented databases, where an object can have many deputy objects and a single deputy object can share multiple objects. Thus, although the encapsulation feature of object-oriented database limits capability of object restructuring, dividing and combining objects can still be done indirectly through their deputy objects. That is, deputy mechanisms enable object views to be easily realized.

Most of view mechanisms of object-oriented databases are realized by computation methods. There are also some papers [3, 7] discussing the materialized view techniques of object-oriented databases for concurrent engineering. They duplicated objects for virtual classes but did not provide enough update propagation supports for view rematerialization. The paper [3] gave a simple solution based on change file, where the change history is recorded. That is, if an application edits an object, the edit is represented as a change record that contains the identifier of the modified object, as well as the operation applied to the object. This change record can then be used to update other materialized views that contain the same object. As concluded in the paper, this method is only suitable for propagating changes between applications using the same materialized views. The paper [7]
proposed a methodology for constructing materialized OO views but update propagation was still under development.

In the paper[6], membership materialization was introduced for object-oriented databases where a virtual class does not replicate objects but instead stores references to them, of which attribute values are still computed on demand. It is actually a kind of semi-materialized views similar to those realized by the previous deputy mechanisms. In addition, the inheritance and update propagation among all the base and virtual classes require to maintain a ISA relationship. This way is not suitable for data warehousing since it is very difficult to maintain a single ISA hierarchy over multiple object-oriented databases at data warehouses.

Views realized by deputy mechanisms are also semi-materialized. In this paper, deputy mechanisms are extended to allow object views to be selectively materialized according to needs of data warehousing. In addition, we have developed method application propagation as complement of data update propagation so that materialized object views can be easily maintained. Furthermore, data warehouses have their own hierarchies that can be derived by object deputy algebra for various integration purposes.

5 Conclusion

To our best knowledge, there is not any paper discussing how to incorporate object-oriented databases into data warehouses. In this paper, we proposed an object deputy approach for realization of data warehouses incorporating object-oriented databases, which can reduce update propagation overhead by the following two approaches: (1) reduction of data duplication by considering inheritance property (2) speed-up update propagation by considering communication delays.

Since objects are allowed to be updated at data warehouse sides, there is a concurrency control problem. The concurrency control mechanisms are usually realized by the source OODBs. However, the well-organized data warehouses can also be used as sources from which we can construct data warehouses of data warehouses. Therefore, data warehouses also need provide concurrency control supports. By the methods discussed in this paper, the update operations can be locally processed at first, and then are transmitted to the source and the other data warehouses. Thus, we can realize distributed concurrency control mechanisms that can greatly reduce communication delay so that data warehouses can be recovered as soon as possible. We will address these problems in the future work.

References