Updating Relational Databases through Object-Based Views

Thierry Barsalou
IBM Thomas J. Watson Research Center
Arthur M. Keller
Advanced Decision Systems & Stanford University
Niki Siambela
Stanford University
Gio Wiederhold
Stanford University

Abstract

The view-object model provides a formal basis for representing and manipulating object-based views on relational databases. In this paper, we present a scheme for handling update operations on view objects. Because a typical view object encompasses multiple relations, a view-object update request must be translated into valid operations on the underlying relational database. Building on an existing approach to update relational views, we introduce algorithms to enumerate all valid translations of the various update operations on view objects. The process of choosing a translator for view-object update occurs at view-object generation time. Once chosen, the translator can handle any update request on the view object.

1 Introduction

Many application domains require database techniques for modeling and managing complex objects [6, 12, 16, 21, 24]. At the same time, a major incentive to exploit database management systems is the ability to support sharing of data among applications. In practice, however, these two objectives tend to conflict. Storing information in object format inhibits sharing, since the objects are configured according to one application view and hence cannot easily serve a variety of purposes. On the other hand, the relational model provides information sharing through, for example, the definition of views, but it lacks the expressive power to represent complex entities.

We have developed the *view-object model* as a first step toward reconciling the opposing objectives of object-oriented access to shared information [4, 22]. By combining the relational-database concept of view and the programming-language concept of object, the view-object model supports simultaneously abstract complex units of information and sharing of those units. Base information remains stored in a fully normalized relational database; this neutral representation facilitates sharing. View objects are then defined as uninstantiated, object-based views of arbitrary complexity onto such a database. Each object[1] is in effect a hierarchical subset of the underlying database schema that specifies a new class. Definition of multiple view objects with different configurations offers a view mechanism at a higher level of abstraction than that of relational views.

In our framework, a query on a view object is composed dynamically with the object's structure to obtain a relational query that can be executed against the database. View-object instances are assembled from the set of relational tuples satisfying the request. When update operations are performed on the instances, those updates have to be made persistent, and hence must be moved from the object representation into the base relations of the database. We need to translate the updates specified on the view objects into unique and semantically correct updates on the database—a problem akin to that of updating through relational views [10]. We present in this paper a formal method for handling all update operations reliably and predictably on view objects. Since each view object typically comprises many underlying relations, this method builds on an existing solution to the problem of updating through views involving multiple relations [15], and entails choosing a unique update translator at view-object definition time.

The paper is organized as follows. Section 2 details the semantic data model at the core of our formalism. Section 3 describes the view-object model for specifying object-based views in a relational framework. Section 4 presents an approach for updating through relational views, which we extend to handle updates through view objects in Section 5. Section 6 shows sample dialogs used in choosing a translator for a view object. We conclude in Section 7 with a discussion of our results.

[1] For brevity, we sometime use *object* as a short synonym for *view object*.

2 The Structural Model

The structural model of a relational database is a formal semantic data model constructed from relations that express entity classes and from relationships, or connections, among those classes [23]. The structural model defines a directed-graph representation of a database, where vertices correspond to relations and edges to connections.

Definition 2.1 *A* **connection** *is defined by the two relations R_1 and R_2 being connected, and by the two subsets of attributes X_1 of R_1 and X_2 of R_2 such that X_1 and X_2 have identical number of attributes and domains. R_1 and R_2 are then connected through the ordered pair $\langle X_1, X_2 \rangle$.*

For two connected relations R_1 and R_2, two tuples $t_1 \in R_1$ and $t_2 \in R_2$ are connected if and only if the values of the connecting attributes in t_1 and t_2 match.

The structural model defines three types of connections, according to the semantics of the relationship between the two relations. Most importantly for our purpose, the connection types carry precise integrity rules. Let K(R) and NK(R) be the key and nonkey attributes of relation R respectively.

Definition 2.2 *An* **ownership connection** *from R_1 to R_2 is specified by the following criteria:*

1. *Every tuple in R_2 must be connected to an owning tuple in R_1*
2. *Deletion of an owning tuple in R_1 requires deletion of all tuples connected to that tuple in R_2*
3. *Modification of X_1 in an owning tuple of R_1 requires either propagation of the modification to attributes X_2 of all owned tuples in R_2 or deletion of those tuples*

The ownership connection embodies the concept of dependency, where owned tuples are specifically related to a single owner tuple. As a result, we must have $X_1 = K(R_1)$, and $X_2 \subset K(R_2)$. The cardinality of the ownership connection is $1{:}n$. Its graphical symbol is $\longrightarrow\!\!*$.

Definition 2.3 *A* **reference connection** *from R_1 to R_2 is specified by the following criteria:*

1. *Every tuple in R_1 must either be connected to a referenced tuple in R_2 or have null values for X_1*
2. *Deletion of a tuple in R_2 requires either deletion of its referencing tuples in R_1 or assignment of valid or null values to attributes X_1 of all the referencing tuples in R_1*
3. *Modification of X_2 in a referenced tuple of R_2 requires any one of propagation of the modification to attributes X_1 of all referencing tuples in R_1, assignment of null values to attributes X_1 of all referencing tuples in R_1, or deletion of those tuples*

The reference connection relates one entity (the referencing relation) to another more abstract entity (the referenced relation). As a result, we must have $X_1 \subset K(R_1)$ or $X_1 \subset NK(R_1)$, and $X_2 = K(R_2)$. The cardinality of the reference connection is $n{:}1$. Its graphical symbol is $\longrightarrow$.

Definition 2.4 *A* **subset connection** *from R_1 to R_2 is specified by the following criteria:*

1. *Every tuple in R_2 must be connected to one tuple in R_1*
2. *Deletion of a tuple in R_1 requires deletion of the connected tuple in R_2 (if the latter exists)*
3. *Modification of X_1 in a tuple of R_1 requires either propagation of the modification to attributes X_2 of its connected tuple in R_2 or deletion of the R_2 tuple*

Specialization of a general entity can be implemented by defining more specific entities connected to the main one through subset relationships. As a result, we must have $X_1 = K(R_1)$, and $X_2 = K(R_2)$. The cardinality of the subset connection is $1{:}[0,1]$. Its graphical symbol is $\longrightarrow\!\!\ni$.

Note that $m{:}n$ relationships are not modeled directly in the structural model but can be represented using combinations of connections. Finally, if there is a connection C from relation R_i to relation R_j, there is an inverse connection C^{-1} from relation R_j to relation R_i.

3 View Objects

Through the semantics provided by the structural model, the view-object model enables us to combine the abstraction concepts of *view* and *object* and to support complex units of information as well as sharing of this information. From the concept of view, we borrow the idea of virtuality. A view object is an uninstantiated window onto the underlying database; that is, only its definition is saved while base data remains stored in the relational database. Views, however, are still in first normal form and thus inadequate for many applications. From the concept of object, we hence borrow the notion of hierarchical structures and of set and record constructors. A view object is a hierarchical subset of the underlying database's structural model. During instantiation operations, the object provides the necessary composition knowledge to support proper rearrangement of the unstructured relational data into hierarchical instances that have atomic-valued, tuple-valued, and set-valued attributes.

We give the full construction in [4] and present only the salient results of our model here. Let $\mathcal{R}$ be the domain of all relations for a given relational database, let Π denote the domain of all projections π defined on $\mathcal{R}$, and let Set(Π) designate the domain of all finite sets of projections. In addition, we specify a function $d : \Pi \rightarrow \mathcal{R}$ such that $d(\pi)$ is the relation on which π is defined.

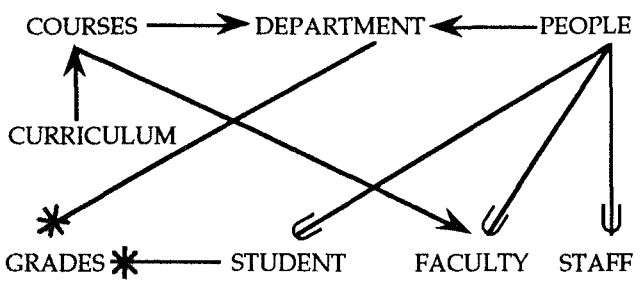

Figure 1: Structural schema of a university database.

Definition 3.1 *A* **view object** ω *is a nonempty element of* $Set(\Pi)$ *and is denoted* $\omega = \{\pi_1, \pi_2, ..., \pi_t\}$, *where the* πs *are projections defined on* $\mathcal{R}$. *The complexity of* ω *is the number of projections included in the object.*

Definition 3.2 *For each object* ω, *we further define a* **pivot relation** $R_1 \in \mathcal{R}$ *such that*

- $\exists!\ \pi_j \in \omega \mid [d(\pi_j) = R_1] \wedge [K(R_1) \subseteq \pi_j]$
- $K(\omega) = K(R_1)$
- $\forall\ k = 1 \ldots i, k \neq j, d(\pi_k) \neq R_1$

The notion of *pivot relation* is central to the formalism. Each object is "anchored" on one base relation, which will constitute its core component. (For convenience, we shall assume $j = 1$; that is, $d(\pi_1) = \mathrm{R}_1$.)

We extend here the notion of a relational key to an object key, such that the key of an object ω is isomorphic to the key of its pivot relation—hence the requirement that all the key attributes be included in the projection π_1. As its relational counterpart, the object key permits unique identification of any given instance of ω. Evidently, since the pivot relation uniquely defines each object instance, no projection other than π_1 in the object can be defined on the pivot relation. However, several objects can be anchored on the same pivot relation, and multiple copies of a non-pivot relation can be included in one object.

A view object is a set of projections on base relations, where one of those is the pivot relation for the object. As detailed in [4], we further refine this definition. In short, we apply an information-metric model for specifying which relations can be included in a particular object given that object's pivot relation, and we demonstrate that, using this model, each object is arranged into a unique tree of projections rooted at the pivot relation.

An example will best serve to fix ideas. We model a university department by eight relations: DEPARTMENT, PEOPLE, STUDENT, FACULTY, STAFF, CURRICULUM, COURSES, and GRADES. The structural model for this schema, shown in Figure 1, indicates that courses and people relate to a department, that a person is either a student, a faculty, or a staff, that a curriculum describes the required courses for a given degree, and that grades are associated with courses and students.

To define a complex entity that will represent detailed course information, we first select COURSES as the pivot relation of the new object ω. An algorithm subsequently analyzes the topology of the structural model and extracts a subgraph G that isolates all the relations deemed to be relevant to the new object according to our information metric (Figure 2a). G is then converted into a tree T (Figure 2b). That translation demands that the circuits in G be broken. For that purpose, we expand all the paths in G emanating from the pivot relation until either we can go no further without creating a cycle or we reach a relation that is no longer relevant. The resulting T specifies all possible configurations for view objects anchored on COURSES; that is, once the pivot relation has been determined, we have the choice to either include in or exclude from ω every other relation in the tree.

Figure 2(c) shows the final hierarchical structure of our course-information object, which has a complexity of 5. As Figure 3 illustrates, an alternate perspective on the same underlying data repository is easily specified in the form of a new view object with a different configuration. The view-object model hence supports sharing of the database-resident information among diverse applications by providing multiple object configurations that map to the same underlying data repository.

We have a complete set of structural concepts to represent object-based views in a relational framework. To round out the model, we designed a query model that (1) defines the practical mapping between view objects and databases, (2) specifies a query language that supports adhoc, declarative queries on view objects, and (3) creates dynamically view-object instances from the base data stored in relational format. Figure 4 shows such a view-object instance for ω that has been dynamically generated following an application's request.

A first prototype of our view-object model has been implemented in the PENGUIN system [5].

Note that the query representation can also be used to formulate update requests. However, as we shall now discuss, the problem associated with handling update operations lies not in specifying these requests, but rather in translating them into semantically valid operations on the database.

4 Updating through Relational Views

The problem of updating through relational views has been addressed by many researchers [3, 9, 11, 20]. Keller's ap-

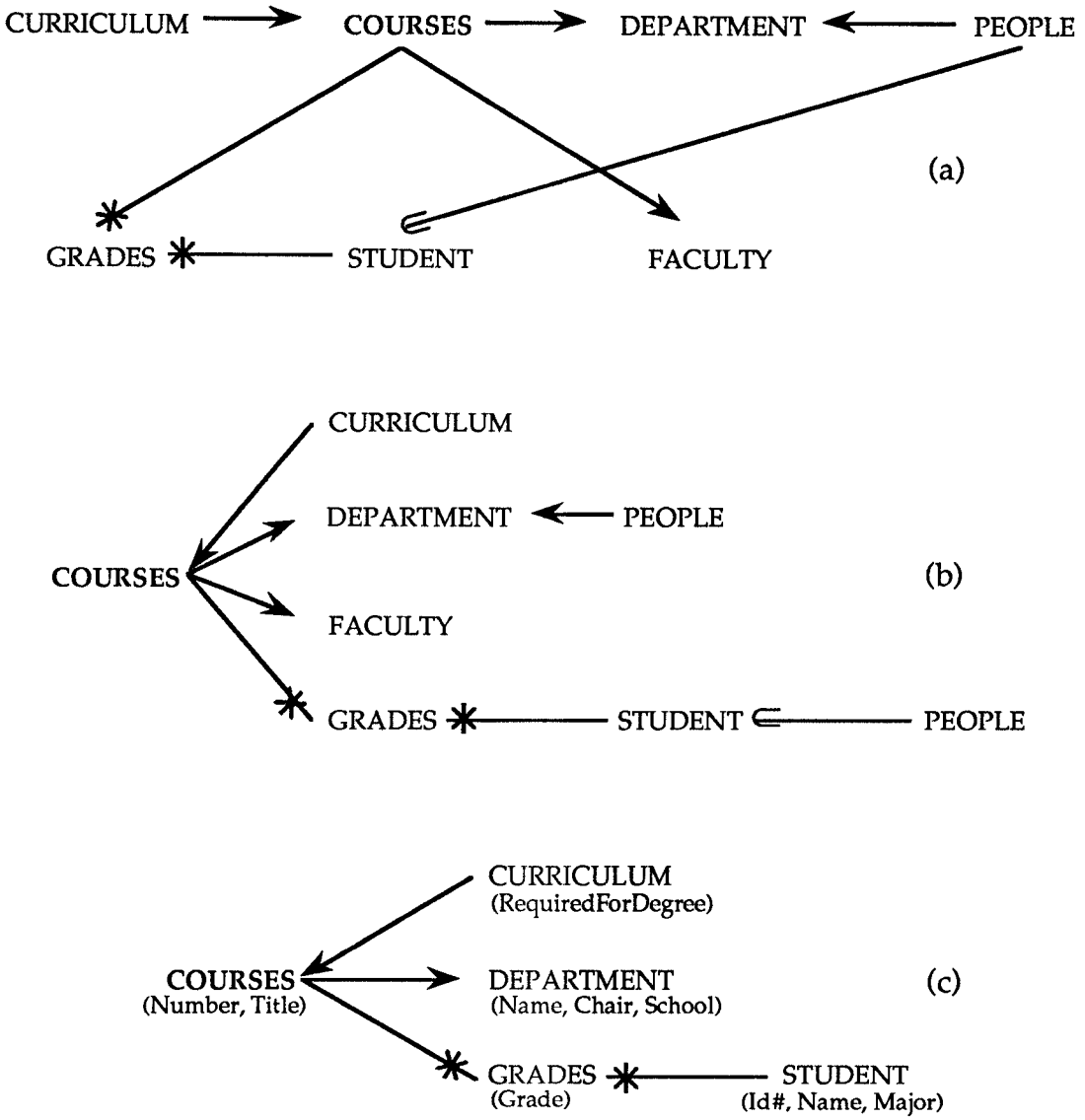

Figure 2: Definition of a view object.

This figure illustrates the creation of a view object ω anchored on COURSES. (a) A subgraph G, extracted from the schema of Figure 1, specifies all the relations that can contribute useful information in the context of the pivot relation, as measured by our metric. (b) A tree is then generated from the subgraph; note that, because of the presence of a circuit in G, there are now two copies of PEOPLE corresponding to the two paths from COURSES to PEOPLE. (c) The tree is pruned to define the final configuration of ω that includes DEPARTMENT, CURRICULUM, GRADES, and STUDENT in addition to COURSES. The attributes selected for each node of the tree are shown in parentheses.

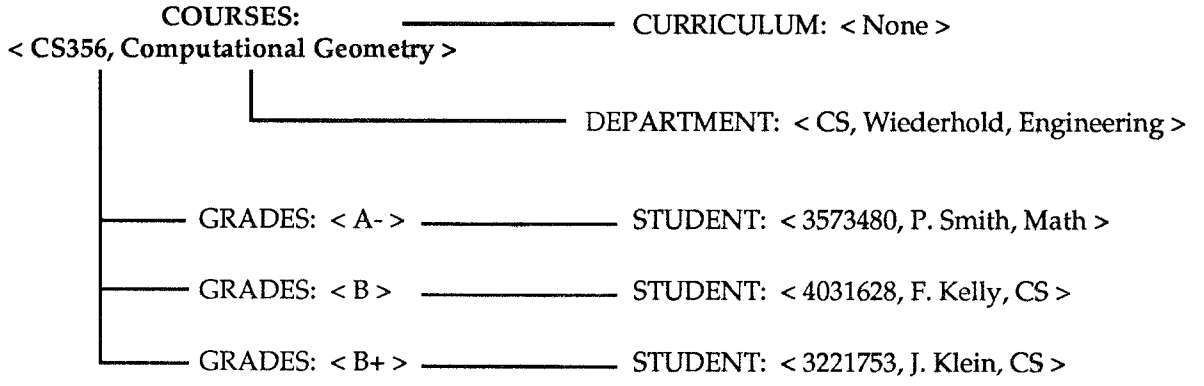

Figure 4: Instantiation of a view object.

An application's request to retrieve graduate courses with less than 5 students having enrolled produces one instance of ω. Such an instance is created by binding appropriately the set of relational tuples satisfying the query to the view object's structure.

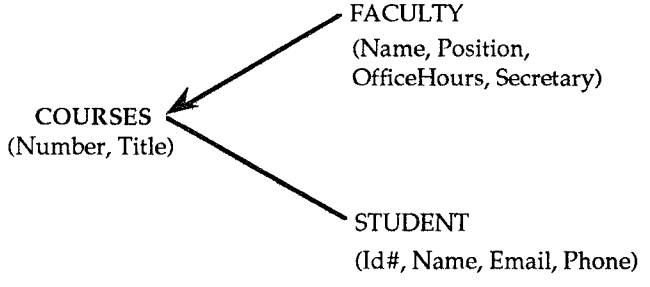

Figure 3: A different view of the database.

By defining a new view object ω', we can provide end users and applications with a different perspective on the data. ω' is still anchored on COURSES but includes additional information only on FACULTY and STUDENT. Note also that the edge from COURSES to STUDENT is no longer a structural connection but rather a path of two connections (COURSES ——*GRADES *—— STUDENT) since GRADES is not part of ω'.

proach to updating relational databases through views starts with a relational view definition. This relational view differs from a view object in that each tuple is in first normal form whereas a view-object instance is a fully unnormalized entity.

Conceptually, we specify an enumeration of all possible valid translations into sequences of database updates of each view update on the view. This enumeration is based on five validity criteria that must all be satisfied [13]. These criteria are syntactically based and they characterize the nature of the ambiguity in view-update translation. This ambiguity chiefly results from the existence of selections and projections in the view. We do not actually instantiate this enumeration, we merely use it to define the space of alternatives.

We use semantics of the application to choose among the alternative translations of view updates. In the case of relational views, these semantics are obtained by a dialog during view definition time by asking a series of questions to the view definer, typically the database administrator [14, 15]. These questions are based on the view definition and on earlier questions asked in the dialog. The answers to these questions specify a view-update translator that is used to translate view updates into database updates. The effort of answering the series of questions once during view-definition time is amortized over all the times that updates against the view are subsequently requested.

5 Updating through View Objects

The classes of views that are supported by Keller's algorithms have commonalities with view objects. Beside the fully unnormalized nature of view objects, however, other important differences exist between Keller's views and view objects;[2] these differences introduce additional complexity and demand that significant extensions be made to his approach.

We can divide an operation of view-object update into four logical steps, as follows: (1) Local validation against the view-object definition; (2) propagation within the view object, (3) translation into a set of database update operations, and (4) global validation against the structural model.

Step 1 checks that the update request does not violate structural restrictions and user authorizations.[3] Step 2 enforces functional and key dependencies throughout the relations and the associated tuples that are included in the view-object instance. Step 3 performs the actual transformation of the request into database update operations according to the translator chosen for the view object. Finally, step 4 uses the structural model to maintain the global consistency of the data following these database updates. Steps 1, 2, and 3 hence correspond to the process of view-object decomposition; step 4 corresponds to the process of global integrity maintenance.

We define the following update operations. A *complete insertion* adds to the database a fully specified view-object instance. A *complete deletion* removes from the database a fully specified view-object instance. A *replacement* combines a complete deletion and a complete insertion; it needs a view-object instance and its fully specified replacing instance. A *complete update* is a complete insertion, a complete deletion, or a replacement. A complete update requires that the entire view-object instance be mapped back to the latter's underlying relations. The description of *partial* update operations for manipulating only a component of the view object (that is, a node in the object's tree of relations) can be found in [4].

Definition 5.1 *The* **dependency island** $\mathcal{D}_\omega$ *of a view object ω is the maximal subtree of the tree of projections such that (1) the root of the subtree is the pivot relation, and (2) all directed paths starting at the pivot relation must contain exclusively ownership and subset connections.*

Definition 5.2 *A* **referencing peninsula** *is a relation $R_j \in d(\omega)$ that is directly connected to any relation R_k of the dependency island by a reference connection; that is, $R_j \longrightarrow R_k$.*

[2] See [4] for a discussion of the differences.

[3] Because that step is straightforward, we shall assume that the local validation has been performed successfully prior to the processing of the update request.

The rationale behind the dependency island is that all the relations in $\mathcal{D}_\omega$ belong to the same entity—the entity that is centered on the pivot relation. As a result, any update operation on the view object should have consistent repercussions throughout the components of that object's dependency island. Referencing peninsulas, on the other hand, must be identified because of the constraints of referential integrity. For our view object ω shown in Figure 2(c), the dependency island is the subtree rooted at the pivot relation COURSES and including GRADES. The only referencing peninsula corresponds to relation CURRICULUM.

5.1 Translation of complete-deletion requests

Keller's deletion algorithm deletes the matching database tuple from the root relation in the query graph. This solution does not satisfy the semantic constraints of view objects, however, and needs to be extended.

Let us look at an example. Translating a deletion request on an instance of ω into a deletion of the matching tuple in its pivot relation COURSES, although appropriate, is not nearly enough. Clearly, the deletion in COURSES must be propagated to all the other elements of the dependency island (here, only GRADES). Furthermore, the tuples in the referencing peninsulas (here, only CURRICULUM) have to be modified to reflect the deletion of the tuples they were referencing. Accordingly, the algorithm for view-object complete deletion is as follows.

Algorithm VO-CD: The input is a request for deleting a view-object instance. The output is the set of database operations that implement that request.

- Isolate the dependency island
- For each projection in the island, delete all matching tuples from the underlying relation
- Identify the referencing peninsulas
- For each peninsula, perform a replacement on the foreign key of each matching tuple ♣

In a case where replacements are not allowed on any of the referencing peninsulas, the transaction cannot be completed and has to be rolled back.

This algorithm has precise effects on the database; the process of global integrity maintenance can therefore be simplified to require only two operations. First, for relations in the dependency island that have outgoing ownership or subset connections, the deletions must be propagated (repeatedly, if necessary) to those owned and subset relations. Second, in addition to the referencing peninsulas already handled, foreign-key replacements must be performed on any

relation referencing one of the relations involved in a deletion. Note that no further propagation is needed outside of the referencing peninsulas and the referencing relations.

5.2 Translation of complete-insertion requests

Keller's algorithm has three distinct cases for each projection involved in the view. The transaction is rejected only if an identical tuple already exists in the pivot relation. Here again, we have to take into account the entire dependency island, not just the pivot relation.

Inserting a view-object instance involves adding the tuples of each of the object's projections to the underlying base relations. We have not included in the algorithm the fact that each view-object tuple inserted in the database needs to be extended with some values for the attributes that have been projected out. How this operation is handled is dependent on the application.

Algorithm VO-CI: The input specifies a new view-object instance to be added to the database. The output is the set of database operations that implements the request.

- Isolate the dependency island
- For each tuple in each projection of the view object, there are three possible cases:

  CASE 1: An identical tuple exists in the database. If the current relation belongs to the dependency island, reject the update; otherwise, do nothing.

  CASE 2: The new tuple does not match the key of any tuple in the underlying database relation. Perform an insertion.

  CASE 3: The new tuple matches the key of an existing tuple, but some values for nonkey attributes differ. If the current relation belongs to the dependency island, reject the update; otherwise, perform a replacement of the existing tuple with the new view-object tuple. ♣

Following the insertion of a new view-object instance, we need to run a number of checks to preserve the global consistency of the database. For all the relations where tuples have been inserted by algorithm VO-CI, outside relations along inverse ownership, inverse subset, and reference connections must be verified for proper dependencies. If no tuple satisfying the suitable dependency is found in any of those relations, one such tuple must be inserted, and the process must be applied recursively to that new insertion. Finally, for all the relations where tuples have been replaced, if some referencing attributes are involved in the replacement, the referenced relations must be checked for referential integrity.

5.3 Translation of replacement requests

As with replacements in relational views, replacements on view-object instances are more difficult to handle than are complete insertions and deletions. The main source of difficulty is the handling of replacements on keys. In such a case, all three steps of propagation within the view object, translation into database operations, and validation against the structural model have to be executed sequentially.

Before discussing the issue of propagating changes within the view object and translating the replacement request, we give the rationale for our approach to handling modifications of keys.

Handling of replacements on keys. The following rules apply:

- Any replacement operation on any element of the dependency island should be translated literally as a database replacement operation.
- If permitted, a replacement on the key of a relation referenced by any relation of the dependency island should lead to an insertion operation, rather than to a replacement operation.
- Replacements on keys of referencing peninsulas are not desirable, because they are inherently ambiguous. They are hence prohibited. For all other types of relations included in the view object, changes to the key are also precluded.

Propagation within the view object. From the previous discussion, we see that the replacement of a key in a view-object instance will translate into a database replacement only if the underlying relation belongs to the dependency island. Thus, we do not need to be concerned about propagating modification of a key outside the dependency island.

Let R_1 be the pivot relation of ω, and $\mathcal{D}_\omega$ its dependency island. We have

- $A_1 = K(R_1)$
- $A_j = K(R_j) - K(R_i)$, where R_i, $R_j \in D_\omega$, such that R_i is the parent of R_j in the dependency island.

On an ownership or subset path in the dependency island, $K(R_i)$ is the set of key attributes "inherited" from the parent relation R_i to the child relation R_j, and A_j is the complement of attributes that are necessary to make up a unique key for R_j.

By definition of a view object, in the dependency island, the complement A_j is the only part of R_j's key that is accessible at the level of R_j. As a result, we cannot modify in R_j the key of any ancestor of R_j—a desirable property. On the other hand, a change to A_j has to be propagated down to R_j's children in the dependency island.

Algorithm for translation of replacement requests. We assume here that all the necessary local propagation operations have already been performed.

Algorithm VO-R: Perform a depth-first search on the view-object's tree of relations. We are initially in State R at the pivot relation.

STATE R (replacing): Compare the old and new view-object tuples in this projection.

CASE R-1: The projections match exactly. Get the next view-object tuple in this projection, staying in state R. If there are no more tuples, move to the next relation down, then go to state I if we are outside the dependency island, and go to state R otherwise.

CASE R-2: The projections differ but the keys match. Perform a replacement in this projection. Get the next view-object tuple in this projection, staying in State R. If there are no more tuples, move to the next relation down, then go to state I if we are outside the dependency island, and go to state R otherwise.

CASE R-3: The projections differ and the keys differ. This case can happen for only those projections that are part of the dependency island. Perform a replacement in this projection. Get the next view-object tuple in this projection, staying in State R. If there are no more tuples, move to the next relation down, then go to state I if we are outside the dependency island, and go to State R otherwise.

STATE I (inserting): Compare the old and new view-object tuples in this projection. There are four cases:

CASE I-1: The keys match. Go to state R, staying with this tuple.

CASE I-2: The keys differ; the new tuple does not exist in the database relation. Insert the new tuple in the database. Get the next view-object tuple in this projection, staying in state I. If there are no more tuples, move to the next relation down, and go to state I.

CASE I-3: The keys differ; the new tuple exists in the database. Get the next view-object tuple in this projection, staying in state I. If there are no more tuples, move to the next relation down, and go to state I.

CASE I-4: The keys differ; the new tuple is in the database but some attributes have conflicting values. Perform a replacement in this projection. Get the next view-object tuple in this projection, staying in state I. If there are no more tuples, move to the next relation down, and go to state I. ♣

Note that the treatment of case R-3 may vary. The old view-object tuple is always deleted, but there are two alternatives for the new view-object tuple—insertion and replacement—depending on whether a tuple with the same key already exists in the database. If we have a deletion followed by an insertion, we perform a replacement instead, as it is a simpler operation.

Validation against the structural model. A replacement request may affect many components of the view object, and, therefore, its translation can affect many relations in the database. We have shown, however, that replacement of a key in a view-object instance translates into replacement of a key in the database for only those relations that are part of the dependency island. For referencing peninsulas, we must replace the foreign key of all tuples that were refering to any of the modified tuples in the dependency island. Similarly, if a relation outside of the object is attached to the dependency island by an ownership or subset connection, the replacement has to be propagated to it. In all other cases, propagation of the change outside of the island will produce only checking and insertion operations to satisfy the constraints of the structural model.

6 Choosing a Translator for View-Object Updates

We obtain the semantics for handling the translation of view-object updates during the view-object definition process. At that time, the database administrator (DBA) can address issues of global integrity maintenance over the entire database. In contrast, end users are typically restricted to specialized view objects and thus would not have the necessary knowledge about global database constraints to properly translate update operations on their objects.

The classes of translators from which we can choose are based on the translation algorithms. The DBA enters in a dialog with the object-definition facility; the sequence of answers to the system's questions defines the desired translator for the object at hand. Once the DBA has chosen the translator, users can specify updates through the view object, which are then translated into database update operations. The algorithms that drive the dialogs for choosing a translator follow closely the actual translation algorithms of Section 5 and can be found in [4].

We now show an actual dialog to choose a translator for the view object ω given in Figure 2(c) that will safely translate all update operations on ω.[4]

Is replacement of tuples in an object instance allowed? <YES>

[4]For brevity, we only present the portion of the dialog dealing with replacement. The DBA's inputs are displayed in **bold-faced type**. The system's questions appear in **typewriter style**.

The key of a tuple of relation COURSES could be modified during replacements.
Do you allow this? <YES>
Can we replace the key of the corresponding database tuple? <YES>
The system might need to delete the old database tuple, and replace it with an existing tuple with matching key. Do you allow this? <NO>
Can the relation CURRICULUM be modified during insertions (or replacements)? <YES>
Can a new tuple be inserted? <YES>
Can an existing tuple be modified? <YES>
Can the relation DEPARTMENT be modified during insertions (or replacements)? <YES>
Can a new tuple be inserted? <YES>
Can an existing tuple be modified? <YES>
The key of a tuple of relation GRADES could be modified during replacements.
Do you allow this? <YES>
Can we replace the key of the corresponding database tuple? <YES>
The system might need to delete the old database tuple, and replace it with an existing tuple with matching key. Do you allow this? <NO>
Can the relation STUDENT be modified during insertions (or replacements)? <YES>
Can a new tuple be inserted? <YES>
Can an existing tuple be modified? <YES>

Most answers for the replacement-related questions are affirmative; this translator will hence permit a wide range of procedures that the system might need to perform during replacement operations on ω.

For example, assuming that a department named "Engineering Economic Systems" does not exist in the database, the request to replace ω's instance (COURSE: CS345 (CURRICULUM: ...) (DEPARTMENT: Computer Science) (GRADES: ...) (STUDENT: ...)) with (COURSE: EES345 (CURRICULUM: ...) (DEPARTMENT: Engineering Economic Systems) (GRADES: ...) (STUDENT: ...)) will lead, among other things, to the insertion of a tuple ⟨ Engineering Economic Systems ⟩ in the DEPARTMENT relation.

On the other hand, if we do not wish to grant such privilege (of adding new departments to the university database) to the applications that use ω, the DBA can specify a different, more restrictive translator. More specifically, she can answer <NO> to the question

Can the relation DEPARTMENT be modified during insertions (or replacements)?

As a result of defining this new translator,[5] the same replacement request will be rejected, since the application is not allowed to insert tuples in DEPARTMENT.

[5]Note that, in this case, the two subsequent questions in the dialog dealing with DEPARTMENT are irrelevant and thus will not be asked to the DBA.

7 Discussion

In this paper, we address the question of updating relational databases through object-based views of arbitrary configuration, and we describe a formal method and its implementation for handling all three types of update operations reliably and predictably on a large class of view objects.

Because of the similarities between the view-object model and Keller's work on updating through relational views, we have built on his approach to handle update operations on view objects. More specifically, we have shown that the process of view-object update can be divided into four logical steps—local validation, propagation within the view object, translation into database updates, and global validation. We have presented our translation algorithms for complete update operations. We have described the process of choosing a translator for view-object update at object-generation time, and we have shown a dialog to select one translator. Such early conflict resolution obviates the need for tiresome and repetitive dialogs at execution time. As a result, the consistency of the underlying database is preserved in the face of update operations on view objects; these operations will be translated correctly and transparently into relational update operations.

There are a number of avenues, more or less closely related to the view-object model, for representing complex objects. IFO uses a graph-based framework to model structured objects as well as *is-a* and functional relationships [2]. Our step of global validation against the structural model is clearly related to IFO's study of update propagation stemming from the use of semantic relationships. In both models, for example, deletion of an entity propagates downwards in a specialization hierarchy. Although more expressive than the structural model, IFO, however, makes no distinction between the reference connection (modeled as a function) and the ownership connection. The structural model makes this distinction in order to propagate updates (e.g., deletions are propagated across an ownership connection) or prohibit an update (e.g., an entity may not be deleted if it is referenced). It is indeed the reference and ownership connections that we make most use of in determining how to translate object updates into underlying database updates. In addition, IFO does not support multiple virtual object configurations; as a result, its update operations do not require translation. Similarly, the nested-normal-form model imposes a single configuration on its nested entities [1, 8, 17, 18], so update constraints are simpler. Finally, two other implementations of the view-object concept have appeared in the literature [7, 19]; yet, neither handles the update problem satisfactorily.

Acknowledgments. The author's address is IBM Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598. We thank Walter Sujansky for many useful discussions. This work was supported in part by the National Library of Medicine under Grant R01 LM04836, DARPA under contract N39-84-C-211, and the Stanford CIFE consortium.

References

[1] S. Abiteboul and N. Bidoit. Non first normal form relations: An algebra allowing data restructuring. *Journal of Computer and System Sciences*, 33:361–393, 1986.

[2] S. Abiteboul and R. Hull. IFO: A formal semantic database model. *ACM Trans. on Database Systems*, 12(4):525–565, 1987.

[3] F. Bancilhon and N. Spyratos. Update semantics and relational views. *ACM Trans. on Database Systems*, 6(4):557–575, 1981.

[4] T. Barsalou. *View objects for relational databases.* PhD thesis. Technical Report No. STAN-CS-90-1310, Computer Science Department, Stanford University, 1990.

[5] T. Barsalou and G. Wiederhold. Complex objects for relational databases. *Computer Aided Design*, 22(8):458–468, 1990. Special issue on object-oriented techniques for CAD.

[6] D.S. Batory and W. Kim. Modeling concepts for VSLI CAD objects. *ACM Trans. on Database Systems*, 10(5):322–346, 1985.

[7] B.C. Cohen. Views and objects in OB1: A Prolog-based view-object-oriented database. Technical Report PRRL-88-TR-005, David Sarnoff Research Center, Princeton, NJ, 1988.

[8] P. Dadam *et al.* A DBMS prototype to support extended NF2 relations: An integrated view on flat tables and hierarchies. In *Proceedings of the International Conference on Management of Data*, p. 356–367, Washington, D.C., 1986. ACM-SIGMOD.

[9] U. Dayal and P.A. Bernstein. On the correct translation of update operations on relational views. *ACM Trans. on Database Systems*, 7(3), 1982.

[10] A.L. Furtado and M.A. Casanova. Updating relational views. In *Query processing in database systems.* Springer-Verlag, New York, NY, 1985.

[11] A.L. Furtado *et al.* Permitting updates through views of databases. *Information Systems*, 4(4), 1979.

[12] S.J. Gibbs. *An object-oriented office data model.* PhD thesis, Computer Science Department, University of Toronto, 1983.

[13] A.M. Keller. Algorithms for translating view updates to database updates for views involving selections, projections, and joins. In *Proceedings of the Fourth Symposium on Principles of Database Systems*, Portland, OR, 1985. ACM SIGACT-SIGMOD.

[14] A.M. Keller. Choosing a view update translator by dialog at view definition time. In *Proceedings of the Twelfth International Conference on Very Large Data Bases*, p. 467–474, Kyoto, Japan, 1986.

[15] A.M. Keller. The role of semantics in translating view updates. *IEEE Computer*, 19(1):63–73, 1986.

[16] R.A. Lorie and W. Plouffe. Complex objects and their use in design transactions. In *Proceedings of Engineering Design Applications*, pages 115–121, San Jose, CA, 1983. ACM-SIGMOD/IEEE.

[17] P. Pistor and F. Andersen. Designing a general NFNF data model with an SQL-type language interface. In *Proceedings of the Twelfth International Conference on Very Large Data Bases*, p. 278–288, Kyoto, Japan, 1986.

[18] M.A. Roth *et al.* Extended algebra and calculus for nested relational databases. *ACM Trans. on Database Systems*, 13(4):389–417, 1988.

[19] N. Roussopoulos and H.S. Kim. ROOST: A relational object oriented system. In *Proceedings of the Third International Conference on Foundations of Data Organization and Algorithms*, p. 404–420, Paris, France, 1989.

[20] L. Rowe and K.A. Schoens. Data abstractions, views, and updates in RIGEL. In *Proceedings of the International Conference on Management of Data*, p. 214–225, Boston, MA, 1979. ACM-SIGMOD.

[21] M. Stonebraker *et al.* Application of abstract data types and abstract indices to CAD databases. In *Proceedings of Engineering Design Applications*, pages 107–114, San Jose, CA, 1983. ACM-SIGMOD/IEEE.

[22] G. Wiederhold. Views, objects and databases. *IEEE Computer*, 19(12):37–44, 1986.

[23] G. Wiederhold and R. ElMasri. The structural model for database design. In *Entity Relationship Approach to System Analysis and Design*, p. 237–257. North-Holland, 1980.

[24] C.C. Woo and F.H. Lochovsky. An object-based approach to modelling office work. *IEEE Database Engineering Bulletin*, 8(4), 1985.