Incremental Maintenance of Materialized Object-Oriented Views in MultiView: Strategies and Performance Evaluation

Harumi A. Kuno, *Member*, *IEEE Computer Society*, and
Elke A. Rundensteiner, *Member*, *IEEE Computer Society*

Abstract—View materialization is a promising technique for achieving the data sharing and virtual restructuring capabilities needed by advanced applications such as data warehousing and workflow management systems. Much existing work addresses the problem of how to maintain the consistency of materialized relational views under update operations. However, little progress has been made thus far regarding the topic of view materialization in object-oriented databases (OODBs). In this paper, we demonstrate that there are several significant differences between the relational and object-oriented paradigms that can be exploited when addressing the object-oriented view materialization problem. First, we propose techniques that prune update propagation by exploiting knowledge of the subsumption relationships between classes to identify branches of classes to which we do not need to propagate updates and by using derivation ordering to eliminate self-canceling propagation. Second, we use encapsulated interfaces, combined with the fact that any unique database property is inherited from a single location, to provide a "registration service" by which virtual classes can register their interest in specific properties and be notified upon modification of those properties. Third, we introduce the notion of hierarchical registrations that further optimizes update propagation by organizing the registration structures according to the class generalization hierarchy, thereby pruning the set of classes that are notified of updates. We have successfully implemented all proposed techniques in the *MultiView* system on top of the GemStone OODBMS. To the best of our knowledge, *MultiView* is the first OODB view system to provide updatable materialized virtual classes and virtual schemata. In this paper, we also present a cost model for our update algorithms, and we report results from the experimental studies we have run on the *MultiView* system, measuring the impact of various optimization strategies incorporated into our materialization update algorithms.

Index Terms—View materialization, incremental view maintenance, object-oriented views, hierarchical registration, classification, view updatability, data warehousing.

1 Introduction

There is a growing need for strategies that provide the means to cache and use query results, for mechanisms that support customized interfaces to shared data, and for the integration of such mechanisms with the powerful constructs of the object-oriented programming model. For example, because of the dynamic nature of modern data acquisition, the schemata created for support of scientific data need to go through different levels of interpretation, i.e., stages of processing [14], spurring a need for the *virtual restructuring* of data. Similarly, as organizations begin to computerize more of the aspects of their daily operations, there is a growing need to provide flexible customized interfaces to shared data formed by the synthesis of multiple data sources (e.g., *workflow management*) [14], [42]. These needs are complicated by the fact that the scale of the data used by business, government, and scientific applications is steadily growing from the megabyte and gigabyte range into the realm of terabytes and petabytes, inciting an overwhelming need for scalability [14], [11]. An interest in supporting queries across multiple information sources in a transparent fashion (e.g., *data warehouses* and *digital libraries* [59], [3]) has developed in response to this need for improved access to diverse data sources. Finally, the powerful and flexible constructs of the object-oriented model are moving even relational vendors to explore the use of object-oriented technology in sophisticated data analysis tools that support multidimensional data analysis and online analytic processing (OLAP) [11].

Object-oriented view technology promises to address all of these needs. In database terminology, a *view* is a query that is stored and given a name by which it can be used in other queries. A view-defining query restructures the format of data, modifying the properties and/or items associated with a set of data. *View materialization* refers to the technique of storing and maintaining the results of view queries, as opposed to recomputing them on demand. The primary motivation for view materialization is to improve query performance by caching the results of frequently used queries. View materialization has been demonstrated to improve the performance of queries for relational databases by eliminating the need for query modification

- *H.A. Kuno is with the Research and Development Laboratory, Hewlett-Packard, Santa Clara, CA 95052. E-mail: harumi_kuno@hp.com.*
- *E.A. Rundensteiner is with the Department of Computer Science, Worcester Polytechnic Institute, Worcester, MA 01609. E-mail: rundenst@cs.wpi.edu.*

Manuscript received 15 Aug. 1996.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 104466.

[22], [28], [39]. Furthermore, because a materialized view effectively contains the cached results of the view-defining query, view materialization can be used during query optimization to improve performance. A number of researchers are examining the use of relational materialized views to support data warehouses—integrating selected data from multiple, probably heterogeneous, information sources into a single repository that supports queries [59], [3]. Other researchers propose the use of the more powerful object-oriented views to preserve existing views through schema changes [7], [8], [46], [51].

The problem of view materialization is that the contents of a materialized view must be kept consistent in the face of updates to the view's sources. This view materialization problem has been studied extensively in the relational context. However, although many researchers have proposed view systems for object-oriented databases [57], [1], [4], [15], [24], [25], [43], [44], [53], [30], [56] due to the fledgling state of OODB views (only a few OODB view systems have been implemented [43], [53], [54]), barely any work addresses the problem of maintaining materialized OODB views [24], [55], [32]. In relational database systems, a view-defining query results in a *virtual table* with *columns* and *rows* that are derived from its base source(s). Similarly, in our model of views for object-oriented database systems, we provide for the definition of *virtual classes* that have customized *types* and *extents*. Because the object-oriented model arranges classes in a generalization hierarchy reflecting the subsumption relationships between classes, we also support *view schemata*—virtual schemata containing arbitrarily selected sets of base and/or virtual classes. Because the restructuring is virtual, applications can use views to customize shared data objects without affecting other applications' perceptions of the data.

We can apply the extensive body of research regarding view materialization in relational (and deductive) databases [5], [22], [6], [10], [19], [17], [21], [2], [59], [3], [16], [22], [27], [28] to the OODB view materialization problem, but an outstanding need remains for an examination of how the OO paradigm affects the view materialization problem. A fundamental issue is the question of what semantics of an object-oriented view system should be and what data model properties are needed to support these semantics. Furthermore, the object-oriented model includes more information about the relationships between objects (such as subsumption relationships between classes and explicit aggregation relationships); we would like to develop view maintenance algorithms that exploit such knowledge.

At the University of Michigan, we have developed *MultiView*, to the best of our knowledge the first implemented view management system capable of supporting *updatable, incrementally maintained, materialized object-oriented views* [47], [48], [49], [35], [36], [33]. *MultiView* users can create virtual classes using difference, hide, intersect, join, select, or union queries. The system automatically integrates both base and virtual classes into a unified global hierarchy, where the virtual classes act as first-class database citizens. By this, we mean that virtual classes insofar as possible look and feel like "base" classes, i.e., they can define properties, and they are *not only queryable, but also updatable*. At any time, users can select both virtual and base classes for participation in view schemata. In addition, users can access *MultiView* either through a visually intuitive graphic user interface or through a Smalltalk command-level interface [37]. The implementation solutions described in our papers [35], [39], [33] are portable to other object-oriented systems.

The current prototype of the *MultiView* system [36] supports updatable incrementally materialized object-oriented views. The system incorporates the incremental update propagation strategies introduced in this paper. Our work utilizes known relational techniques for the optimization of update propagation, but also extends these solutions to exploit unique features of the object-oriented paradigm. For example, a number of papers have been written about filtering updates to test if the updated data fulfills view predicates [5], [21]. We adapt this solution to the OODB view materialization problem by exploiting the fact that the generalization hierarchy structure makes the subsumption relationships between classes readily apparent. In addition, other researchers have previously explored the use of active database capabilities to trigger maintenance operations to materialized relational views [10]. We use the encapsulation inherent in our object model to incorporate a triggering mechanism to propagate modification of properties used in predicates. We also exploit the fact that in our object model any given property has a single point of inheritance. This feature helps us provide a service by which virtual classes *register* their interest in the values of the properties used in their predicates. We then distribute these registration structures along the generalization hierarchy so as to identify branches of classes in the generalization hierarchy to which we do not need to propagate updates, which significantly prunes update propagation costs (as our experimental evaluation in Section 5 confirms).

We have developed a formal cost model for our update algorithms. Because we have implemented the *MultiView* system,[1] we were able to evaluate actual implementations of each proposed update propagation algorithm (vs. simulations). As *MultiView* is the only implemented system to support updatable materialized views for OODBs, we are the first to evaluate the performance of maintenance techniques for object-oriented views in such a system. Specifically, we analyze the performance of nonmaterialized views and materialized views with various combinations of our update propagation optimizations. In constructing our cost models, we identify a number of factors that affect the cost of propagating updates to materialized views and retrieving the contents of nonmaterialized views. The results of our experiments validate our cost models.

In previous papers, we described our implementation of the *MultiView* model for object-oriented view materialization [35], [36], and presented initial results of our research regarding the optimization of update propagation [39], [50]. In this paper, we expand upon our previous work in several ways. We now provide detailed descriptions of algorithms that can be used to maintain materialized views in the face of updates, including strategies for maintaining

1. The current *MultiView* prototype has been built on top of GemStone Systems' GemStone OODB product.

materialized virtual classed created using join queries. We present a formal cost model for these update strategies, and lastly we describe experimental studies conducted on our system.

In Section 2, we review the *MultiView* object model and describe the representational model underlying our implementation. We exploit unique features of the object-oriented model in Section 3 to develop efficient update propagation techniques. In Section 4 we derive cost formulae and compare the performance of each of the proposed algorithms in terms of representative models. We present the results of some of our performance evaluations in Section 5. We discuss work related to ours in Section 6. Finally, in Section 7 we present conclusions and propose future work.

2 THE MULTIVIEW MODEL AND SYSTEM

In this section, we review the basic object model principles of the *MultiView* system and outline the architectural model underlying our implementation.

2.1 Basic MultiView Object Model

Let O be an infinite set of object instances. Each object $O_i \in O$ consists of state (the **instance variables** or attributes), behavior (the **methods** to which the object can respond), and a unique object identifier. Because our model is object-oriented and assumes full encapsulation, access to the state of an object is only through **accessing methods** (an accessing method modifies or retrieves the value of an instance variable). Together, the **methods** and **instance variables** of an object are referred to as its **properties.**

Objects that share a common structure and behavior are grouped into sets, denoted **classes**. We use the term **type** to indicate the set of applicable property functions shared by all object-instances of the class. Let C be the set of all classes in a database. A **class** $C_i \in C$ has a unique class name, a **type**, and a set membership denoted by **localExtent**(C_i) or short, **extent**(C_i).[2] The instance variables of a class's type can be **constrained** to a specific class.

For two classes C_i and $C_j \in C$, C_i is a **subtype** of C_j, denoted $C_i \preccurlyeq C_j$ if and only if (iff) (**properties**(C_i) $\supseteq$ **properties**(C_j)). C_j is a **supertype** of C_i iff C_i is a subtype of C_j. All properties defined for a supertype are **inherited** by its subtypes. Similarly, C_i is a **subset** of C_j, denoted $C_i \subseteq C_j$, iff $(\forall o \in O)\ ((o \in C_i) \Rightarrow (o \in C_j))$. Thus **globalExtent**($C_i$) = $\bigcup$ **localExtent**(C_j) $\forall\ C_j \subseteq C_i$. C_j is a **superset** of C_i, denoted $C_j \supseteq C_i$, iff $C_i \subseteq C_j$. C_i is a **subclass** of C_j, denoted C_i *is-a* C_j, iff $(C_i \subseteq C_j)$ and $(C_i \preccurlyeq C_j)$. C_i is a **direct subclass** of C_j if $\nexists\ C_k \in C$ s.t. $k \neq i \neq j$, C_i *is-a* of C_k, and C_k *is-a* of C_j.

An **object schema** is a rooted directed acyclic graph $G = (V, E)$, where V, the finite set of vertices, corresponds to classes $C_i \in C$ and E, the finite set of directed edges, corresponds to a binary relation on $V \times V$ that represents all direct *is-a* relationships. Each directed edge $e \in E$ from V_1 to V_2 represents the relationship C_1 *is-a* C_2. Two classes $C_i, C_j \in C$ share a common property iff they inherit it from the same superclass. The designated root node, **Object**, has a global extent equal to all database instances and an empty type description.

2.2 Object-Oriented Views

We now extend the basic definitions given above to include the concept of virtual classes and virtual schemas. In the relational model, a **view,** or **virtual table**, is defined to be a **named, stored query**. Similarly, our object-oriented model permits the application of a query operator to a **source class** (or classes) that restructures the source class's type and/or extent membership in order to form a **virtual class** with a type and extent derived from its source class(es).

The set of all classes, C, is partitioned into two sets—BC is the set of all base classes, and VC is the set of all virtual classes in the database. **Base** classes are defined during the initial schema definition. **Virtual** classes are defined by the application of a query operator to a **source class(es)** that restructures the source's type and/or extent membership. Virtual classes can be added dynamically to the schema throughout the lifetime of the database.

We constrain a **query** $Q_i \in Q$ used to define a virtual class to correspond to a single algebra operation. Table 1 displays the view-forming queries currently supported by our model. These queries determine the methods, instance variables, and extent of the virtual classes. The queries can be applied to both base and virtual classes. The join operator is *object-generating*, which means that new view objects are created to represent the "join" objects (which can contain aggregate information from more than one object); all other operators are *object-preserving*. Note that we do not support aggregation queries (e.g., a virtual instance variable that represents a sum of other instance variables). If a complex query is specified by nesting algebra operators, then each intermediate algebra operator generates a separate virtual class [47].

DEFINITION 1. *A **query** $Q_i \in Q$ includes the following components:*

<*QueryType, SourceClasses, TypeModifiers, MemberPredicates, ValuePredicates***>**,

where

QueryType $\in$ {*Select*, *Hide*, *Refine*, *Union*, *Intersect*, *Difference*};

SourceClasses $\in$ *C is an ordered list of the classes from which the class is derived (that is, the classes to which the query is applied); TypeModifiers defines the derivation relationship between the type of the virtual class and that of the source class from which it is derived; MemberPredicates is an ordered list of the class membership requirements that apply to instances of the virtual class, listed in the same order as the classes in SourceClasses to which they correspond (e.g., a difference query is associated with the constraint that qualifying objects be members of one source lass and not be members of the other source class); and ValuePredicates is the set of simple value predicates that constrain the extent of the virtual class. We refer to the query that defines a virtual class,* $VC_i \in VC$, *as query(*VC_i*).*

2. Although there is no general agreement on whether classes in OODBs should incorporate their own extents or require users to maintain their own collections of class-instances, several systems follow this philosophy, including [31], [26]. Furthermore, the proposed ODMG standard [41] recently formulated by several key OODB vendors also follows this approach.

TABLE 1
OBJECT ALGEBRA OPERATORS: SYNTAX, SEMANTICS, AND CLASS RELATIONSHIPS

hide	syntax	<*virtual*-class>) := (<*source*-class>) **createHideClass:** [<new-class-name>] **hideMethods:** [<prop-functions>])
	semantics	**type**(<*virtual*-class>) := {p ∈ P | p∈**properties**(<*source*-class>) ∧ p∉<prop-functions>} **extent**(<*virtual*-class>) := **extent**(<*source*-class>)
	class rels	<*source*-class> ⪯ <*virtual*-class> <*source*-class> ⊆ <*virtual*-class> <*source*-class> *is-a* <*virtual*-class>
refine	syntax	<*virtual*-class> := (<*source*-class>) **createRefineClass:** [<new-class-name>] **withProperties:** [<prop-function-defs>]
	semantics	**type**(<*virtual*-class>) := {p∈P | p∈**properties**(<*source*-class>) ∨ p∈<prop-function-def>} **extent**(<*virtual*-class>) := **extent**(<*source*-class>)
	class rels	<*virtual*-class> ⪯ <*source*-class> <*virtual*-class> ⊆ <*source*-class> <*virtual*-class> *is-a* <*source*-class>
select	syntax	<*virtual*-class> := (<*source*-class>) **createSelectClass:** (<*new-class-name*>) **query:** (<predicate>)
	semantics	**type**(<*virtual*-class>) := **type**(<*source*-class>) **extent**(<*virtual*-class>) := {o∈O | o∈<*source*-class> ∧ <predicate>(o)=true}
	class rels	<*virtual*-class> ⪯ <*source*-class> <*virtual*-class> ⊆ <*source*-class> <*virtual*-class> *is-a* <*source*-class>
union	syntax	<*virtual*-class> := (<*source*-class1>) **createUnionClassWith:** (<*source*-class2>) **named:** (<*new-class-name*>)
	semantics	**type**(<*virtual*-class>) := **type**(<*source*-class1>) ⊓ **type**(<*source*-class2>) **extent**(<*virtual*-class>) := {o∈O | o∈<*source*-class1> ∨ o∈<*source*-class2>}
	class rels	<*source*-class1> ⪯ <*virtual*-class> ∧ <*source*-class2> ⪯ <*virtual*-class> <*source*-class1> ⊆ <*virtual*-class> ∧ <*source*-class2> ⊆ <*virtual*-class> <*source*-class1> *is-a* <*virtual*-class> ∧ <*source*-class2> *is-a* <*virtual*-class>
intersect	syntax	<*virtual*-class> := (<*source*-class1>) **createIntersectClassWith:** (<*source*-class2>) **named:** (<*new-class-name*>)
	semantics	**type**(<*virtual*-class>) := **type**(<*source*-class1>) ⊔ **type**(<*source*-class2>) **extent**(<*virtual*-class>) := {o∈O | o∈<*source*-class1> ∧ o∈<*source*-class2>}
	class rels	<*virtual*-class> ⪯ <*source*-class1> ∧ <*virtual*-class> ⪯ <*source*-class2> <*virtual*-class> ⊆ <*source*-class1> ∧ <*virtual*-class> ⊆ <*source*-class2> <*virtual*-class> *is-a* <*source*-class1> ∧ <*virtual*-class> *is-a* <*source*-class2>
difference	syntax	<*virtual*-class> := (<*source*-class1>) **createDifferenceClassWith:** (<*source*-class2>) **named:** (<*new-class-name*>)
	semantics	**type**(<*virtual*-class>) := **type**(<*source*-class1>) **extent**(<*virtual*-class>) := {o∈O | o∈<*source*-class1> ∧ o∉<*source*-class2>}
	class rels	<*virtual*-class> ⪯ <*source*-class1> <*virtual*-class> ⊆ <*source*-class1> <*virtual*-class> *is-a* <*source*-class1>
implicitJoin	syntax	<*virtual*-class> := (<*source*-class1>) **createImplicitJoinClass:** (<*new-class-name*>) **on:** (<*instanceVariable*>)
	semantics	**type**(<*virtual*-class>) := newly-generated type **extent**(<*virtual*-class>) := {o∈O | o∈<*source*-class1>
	class rels	<*virtual*-class> *is-a* <*Root*>

We identify two types of predicates used in virtual-class defining queries. **Class membership predicates**, inherent in hide, union, intersect, refine, and difference queries, are predicate terms that depend upon the classes to which an instance belongs.[3] **Value predicates**, used by select queries, are conjunctive simple predicate terms constraining instances based on the values of their local instance variables.

The **membership-dependent** and **value-dependent** dependencies between classes follow from these predicates. For example, a virtual class derived by applying a *hide* query to a source class is **membership-dependent** upon that source class. Similarly, a virtual class derived by applying a *select* query that uses some instance variable's

3. Set operations are the most typical type of queries using class membership predicates, because they function by using the presence of objects in source classes rather than by checking value-based predicates.

value is **value-dependent** upon that property (and hence the class that defines that property). We later use these definitions of dependencies to identify the classes that are affected by an update.

Each virtual class $VC_i \in VC$ maintains the query that initially defined it. In addition, each class C_i maintains a set of all virtual classes $VC_j \in VC$ directly derived from it. We use the notation VC_j **derived-from** C_i to indicate that a class C_i is a source class of a virtual class VC_j. We use the term **Derived-from-Sub-Graph** of V_i to refer to the schema $DS(V_i) = (DV, DE)$ containing all the classes either directly or indirectly derived from V_i.

We define a **materialized virtual class** as a virtual class that caches its extent rather than computing it upon access. In this implementation, we do not replicate objects that belong to materialized virtual classes, but instead store references to them.[4] We refer to this feature as **membership materialization**. As it depends upon oid support, membership materialization is unique to the object-oriented model. This feature reduces the storage overhead of materialization as well as the time and effort required for view update propagation (demonstrated in Section 3 and Section 4). The technique of membership materialization lies between the "fully virtual" and the "fully materialized" approaches. This means that although we do not have to recalculate the extent of a virtual class (meaning that we can readily identify which objects qualify to belong to a virtual class), the state of these objects is not not necessarily stored with the virtual class. The cost of retrieving each object's state would depend on the implementation. Note that this is an implementation detail belonging to this particular prototype; our paradigm easily lends itself to implementations that completely replicate virtual objects—though of course updates would then need to be propagated to the replicated instaces, thereby increasing view maintenance costs.

We define a **global schema** as a database schema containing all database classes, both base and virtual, and a **view schema** as containing a user-selected subset of the classes from the global schema. These schemata each have an integrated generalization hierarchy. In our model, virtual classes function as first-class citizens in the global hierarchy, i.e., they inherit and provide properties for inheritance just like base classes. In order to maintain this global class hierarchy, our view management system supports a flexible classification mechanism that is able to make dynamic changes to the class hierarchy, e.g., inserting a new class between two existing classes. We have proposed elsewhere [49], [36] algorithms and techniques to automatically maintain the global class hierarchy. Classification offers unique opportunities for optimizing update propagation to materialized views, as will be discussed in Section 3.

2.3 Object-Slicing Representation

As noted in [34], introducing virtual classes and schemata requires the underlying data model to support the following features. If an object instance qualifies for membership in two virtual classes, then it should belong to both even if no subsumption relationship exists between the virtual classes, so the model must support *multiple classification*. *Dynamic reclassification* is necessary because an object instance should dynamically gain (or lose) the type of a *virtual class* (e.g., select or join view) if its data values change so that it fulfills (or ceases to fulfill) the class's selection predicate. Objects must be able to flexibly gain and lose types (e.g., select views) during their lifetimes, including both the data stored in their state as well as the set of methods to which they can respond (*dynamic restructuring*). Finally, *multiple inheritance* is necessary because certain virtual classes, such as those formed by *intersect* queries, must inherit properties from both their source class types.

Because to the best of our knowledge, no existing OODBMS supports all of these features, we chose to develop an object manager to provide these required properties. To this end, we utilized an object-slicing architecture, which we describe and evaluate elsewhere [45], [35]. Please note that although here we present a brief overview of the object-slicing mechanism used in the implementation of our prototype, the view materialization techniques described in this paper are not coupled to the object-slicing architecture.

With the exception of the IRIS functional DBMS [13], which uses a relational database as storage structure, storing data from single objects across many relations, most commercial OODB systems represent each database object as a chunk of contiguous storage determined at object creation time. In object-slicing, a real-world object corresponds to a hierarchy of **implementation objects** (one for each class whose type the object possesses) linked to a **conceptual object** (used to represent the object-itself) rather than associating one implementation with each conceptual object as is commonly assumed in OODBs. This technique of using implementation objects to represent an object's membership in multiple classes is extremely flexible, and extends an OODB system to support all requirements needed to implement our view model.

The object-slicing paradigm can be formalized as follows. Let $O_i \in O$ be a user-defined object. In the object-slicing model, O_i is represented using two kinds of objects: a single conceptual object,

$$O_{i_{conc}}$$

and one implementation object for each class $C_j \in C$ to which the object belongs,

$$O_{i_{impl_{C_j}}}$$

A **conceptual object** consists of a tuple, ***<implObjects, OID>***, where *OID* is the unique, system-generated object identifier of the conceptual object, and *implObjects* is the set of **implementation objects** that are linked to the conceptual object. An **implementation object** is a tuple ***<OID, oid, class, state>*** where *OID* is the object-identifier of the linked conceptual object, *oid* is the object-identifier of the implementation object itself,[5] *class* is the class of which the

4. In fact, as we will describe in the following subsection, we employ an object-slicing technique for object materialization, which distributes the state of an object across the classes that define its instance variables.

5. Each implementation object by default possesses its own object identifier. However, because the implementation object serves as an interface for a specific conceptual object, the object-identifier of the conceptual object supersedes that of the implementation object for most practical purposes, such as determining object-equality.

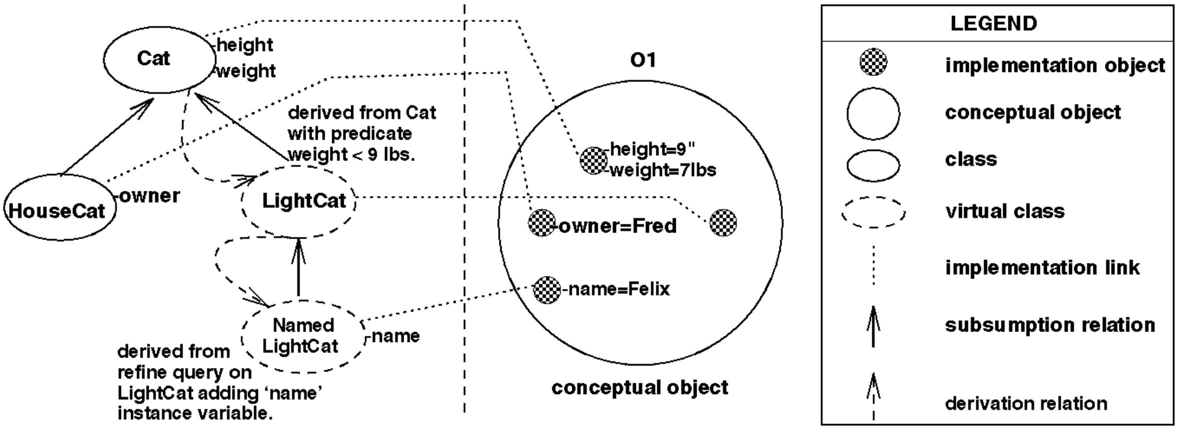

Fig. 1. Example of object-slicing.

implementation object is a direct instance, and *state* corresponds to the values of the local instance variables stored for the given object. Each implementation object

$$O_{i_{impl C_j}}$$

is an object instance of the database class $C_j \in C$ it represents. Conceptual objects are object instances of a special system class, *ConceptualObject*, rather than of a user-defined class.

For example, Fig. 1 depicts a schema composed of two base classes, *Cat* and *HouseCat*, and two virtual classes, *LightCat* (derived from a selection query upon the *Cat* class) and *NamedLightCat* (derived by refining *LightCat* to add a new instance variable, `name`). The right-hand side of the figure illustrates a single instance of the *HouseCat* class with instance variables `height = 9"`, `weight = 7 lbs`, and `owner = Fred`. Because the instance fulfills the membership requirements of *LightCat* (weight < 9 lbs), it belongs to both virtual classes. Note that the state of the object is maintained by the implementation objects of the classes defining each property.

Classification of virtual classes, as shown in Fig. 1, may lead to the problem of select upwards and downwards inheritance [1]. In contrast to previous work in this area, we have explored a unique solution for the preservation of uniform upwards inheritance—we **promote** methods and/or attributes from a subclass to a new superclass during the classification of a new virtual class into the global schema in order to ensure that the property will be located above all the classes that inherit that property.[6] This promotion, called **property migration**, ensures a single point of inheritance for both instance variables and methods. Instance variable values are thus shared by both base and virtual classes without duplication. The existence of a **single point of inheritance** combined with the object-slicing approach guarantees that no more than two classes (the original class where the properties currently reside, and the new class to which they are being moved) will ever be involved in such a migration [35].

The object-slicing technique, combined with the characteristic of **single point of inheritance** and **classification**, lends itself well to materialization, because

1) it elegantly avoids the need to duplicate data for materialized classes, and
2) any update to an object will take place at a unique location determined by the property involved regardless of the source of the update request.

For example, the *LightCat* virtual class in Fig. 1 directly shares the `height` attribute with the *Cat* class—without duplicating its value. Consequently, regardless ofwhether the update is specified via the *Cat* or the *LightCat* class, the same copy of the `height` attribute will be modified. Details on implementation issues in our model such as method resolution, inheritance mechanisms, etc., can be found elsewhere [34], [45].

Object-slicing overhead. As would be expected, extending an existing system with object-slicing techniques involves a certain amount of overhead, including additional data structures, maintenance costs, and processing time. In a separate paper [34], we describe our experimental results evaluating the relative costs and benefits of adopting the object-slicing technique, including an analytical assessment of the storage overhead of the object-slicing representation and its comparison against the conventional representational models. This work was done in the context of regular base classes only [34]. In the current paper we strive to isolate the impact of our propagation techniques for views from the effects of the object-slicing architecture. We thus tailor our experimental models to minimize the impact of the object-slicing implementation.

3 OPTIMIZATION STRATEGIES FOR OBJECT-ORIENTED VIEW MATERIALIZATION

3.1 Assumptions

In this section, we describe our update propagation techniques. Although object-slicing does have a large effect upon the performance of the *MultiView* system, the

6. The methods used to set or retrieve an instance variable's value are called **accessing methods**. Accessing methods are always located at the same class as the instance variables for which they are defined, and thus when an instance variable migrates from one class to another, that instance variable's accessing methods must make the same migration.

problems and solutions described in this section are independent from object-slicing. To place *MultiView* within the taxonomy presented by [18], we have full access to both base and virtual classes at the time of maintenance, including knowledge of the query used to define each virtual class and the subsumption relationships between classes. Our view maintenance algorithms handle the insertion and deletion of objects to base classes, as well as updates to base and virtual objects. We do not handle changes to a virtual class's definition or sets of modifications at this time. And finally, our virtual classes can be defined using select queries with conjunctive simple components of the form a_i θ *value*, where if a_i is an instance variable either defined or inherited by the target class; $\theta \in \{=, <, \neq, \leq, >, \geq\}$, and *value* is a atomic value; hide queries (equivalent to the relational "project" query); refine queries, which add state to base objects (we thus refer to them as "capacity augmenting views"); union, intersect, and difference queries, which are equivalent to their relational namesakes; and implicit join queries, which are object-generating.

3.2 Examples and Basic Definitions

Fig. 2 depicts a simple example that we will use throughout the remainder of this paper. Two virtual classes, *SmallCat* and *LightCat*, are derived from the *Cat* class using select queries. The *DenseCat* class is defined as containing those objects that are members of the *SmallCat* class but not members of the *LightCat* class. The *VeryLightCat* class contains the members of *LightCat* that weigh less than seven pounds.

The view materialization problem is that if an instance of the *Cat* class is updated, then we must determine the effect of this update on all of the virtual classes in the schema. *MultiView* supports the following update operations: creation, deletion, addition of a type, removal of a type, and modification of an instance variable. The first four operations alter the set-membership of a target class; the modification operation changes the value of a single object's instance variable. Once the update has taken place, the change must be propagated to the classes affected by it. For example, when the cat in Fig. 2 changes the value of its `weight` instance variable, then one direct effect of that update might be that the object gains the type of the *LightCat* class (i.e., a new implementation object of the select class is created and linked to the updated object).

We use the term **directly affected** to denote classes whose set-membership changes as a direct result of the update. We distinguish between two causes for such direct effects. One, a set-membership operation (e.g., the removal of a type) potentially directly affects classes that are **derived-from** the updated class. From our previous definition of **membership-dependency** in Section 2, we call such classes **membership-dependent** upon the updated class. Two, an operation that modifies the value of an object's instance variable potentially direct**ly** affects classes that are value-dependent upon a given instance variable. We consider such classes **value-dependent** upon the class that defines that updated property.

For example, given the schema shown in Fig. 2, suppose that we were to update an instance of the *Cat* class, changing its weight from 10 lbs to 12 lbs. Because the *LightCat* class is derived from the *Cat* class using a predicate involving `weight`, the *LightCat* class is potentially **directly affected** by the update. However, because the updated object's membership in the *LightCat* class will not change as a result of the update, the *LightCat* class is not **directly affected**. If the *LightCat* class were actually **directly affected** by the update, then the *NamedLightCat* class would be indirectly affected by the update because the latter is set-membership dependent upon the *LightCat* class.

We identify two invariants that define the consistency requirements of conceptual object instances under our data model. The invariants are conditions that hold true at the time of the update, and they must still be true after our update propagation procedures have completed. The first invariant states that an object possesses the type of a class $C_i \in C$ iff it also possesses the types of all superclasses of C_i. This assures that an object's class membership concurs with the class generalization relationships, which both guarantees the consistency of class extents and enables inheritance in our object-slicing model. The second invariant is that an object is a member of a virtual class $VC_i \in VC$ iff it

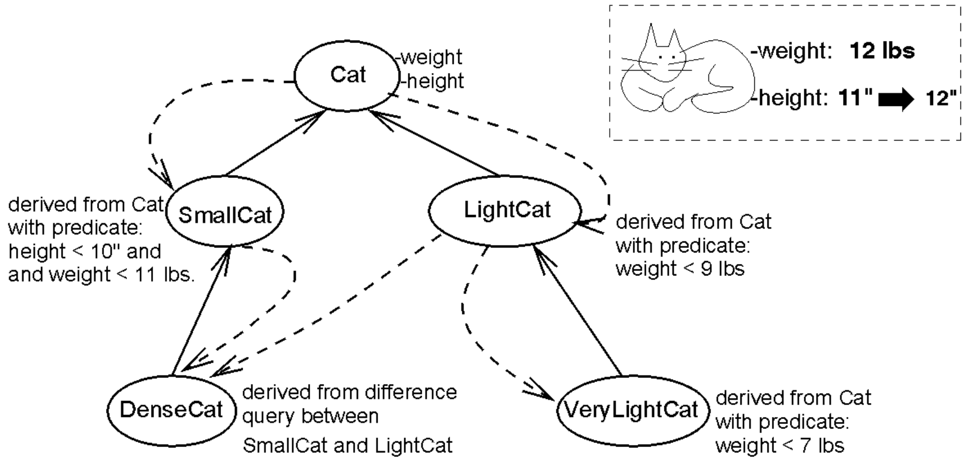

Fig. 2. Virtual classes must be maintained in the face of updates of the data.

fulfills the conditions of **query**(VC_i). Together, these invariants provide the basis for the specification of the semantics of update propagation.

One naive way to propagate an update correctly would be to completely rematerialize all virtual classes in order of derivation upon completion of the update. However, this would be needlessly expensive. Assuming random access or clustering of (implementation) objects by class, each time an object's membership in a given class is evaluated could lead to a random block access. Our goal is thus to minimize the number of times an updated object is evaluated to determine if the object should be a member of a virtual class. An initial propagation strategy is indicated by Proposition 1.

PROPOSITION 1. *An update that is propagated to all* ***directly affected*** *classes and to the classes in the* ***Derived-from-Sub-Graphs*** *of the* ***directly affected*** *classes will result in a correctly materialized database.*

A proof of this proposition is omitted due to space constraints and instead presented in [33]. We can thus correctly propagate an update to an object by propagating to all classes **directly affected** by the update, and all classes derived from the **directly affected** classes. A number of problems are associated with this strategy. First, we need an efficient method to reduce the set of potentially **directly affected** classes to those that are actually **directly affected**. Furthermore, not all of the classes in the **Derived-from-Sub-Graphs** of **directly affected** classes will be affected by the update, thus we want to calculate in an efficient manner the subset of classes that can be discarded. Finally, there may be some overlap among the classes in the **Derived-from-Sub-Graphs** of **directly affected** classes, so we must avoid update propagation operations whose effects must be "undone" by subsequent update propagation operations (**self-canceling update propagation**).

Our update optimization process addresses these problems in three steps. First, we identify the set of classes that are potentially **directly affected** by the update (Section 3.3.1 and Section 3.3.2). Next, utilizing information contained in the generalization hierarchy, relevance registration, etc., we use a screening process to eliminate from this set those classes that are not actually **directly affected** by the update. Finally, we propagate to the classes derived from the **directly affected** classes in an optimized manner, so as to avoid both propagating to classes that are not affected by the update and self-canceling update propagation.

3.3 Identifying Directly Affected Classes

In this subsection, we describe how we isolate the set of classes that are **directly affected** by the update from the set of classes that are potentially **directly affected** by the update. The type of the update operation (e.g., modify vs. add-type) determines the nature of the classes that are potentially **directly affected** by the update. Set-membership operators, e.g., create-object, potentially **directly affect** all classes derived from the updated class. Instance-variable-modification operators potentially **directly affect** only classes formed using queries that actually involve the modified property.

3.3.1 Classes Directly Affected by Set-Membership Updates

If the update is an operation upon the set-membership of a class (i.e., the addition or removal of an object from the extent of a class), then the set of potentially **directly affected** classes contains all classes directly derived from the target class, along with all superclasses of the target class if the operation is a type addition or all subclasses of the target class if the operation is a type removal. The set of classes directly derived from the target class is easy to obtain because, as stated in Section 2, each class maintains a set of all the virtual classes that are directly derived from it.

Our next task is to filter out from this set of potentially affected classes those classes that have set-memberships that are not affected by the update. The generalization hierarchy structure makes the subsumption relationships between classes readily apparent. As detailed in Procedure 1, we can use this information to identify and eliminate classes that are not **directly affected**, by processing classes in order of their level in the generalization hierarchy. If the update operation is a type addition, then potentially all of the superclasses of the affected class are **directly affected**; otherwise, if the update operation is a type removal, then potentially all of the subclasses of the affected class are **directly affected**. We must then identify which classes have set-memberships that are actually directly affected by the update. If the update operation is a type addition, we therefore traverse upwards through the superclasses of the target class, stopping our traversal to branches of classes above a class whose set-membership is not affected by the update. Otherwise, if the update operation is a type removal, then we traverse downwards through the subclasses of the target class, stopping our traversal for any branch rooted in a class whose set-membership is not affected by the update.

PROCEDURE 1 (IdentifyDirectlyAffectedByTypeOperation).

1) Add all of the potentially **directly affected** (directly derived) classes to an ordered set, Directly Affected in breadth-first order of their level in the generalization hierarchy.[7]
2) For each class $C_i \in$ Directly Affected, do:
   a) Evaluate the updated object's membership status regarding C_i.
   b) If the update is a type addition and there is no change in the object's status regarding C_i, then remove all C_j s.t. C_i is-a C_j from Directly Affected.
   c) If the update is a type removal and there is no change in the object's status regarding C_i, then remove all C_j s.t. C_j is-a C_i from Directly Affected.

The classes remaining in *Directly Affected* are exactly the set of classes **directly affected** by the update. By placing the classes into the set of *Directly Affected* classes in order of their subsumption relationships in step 1, we can prune the set efficiently by avoiding the evaluation of the updated

7. The schema is assumed to be maintained in main-memory.

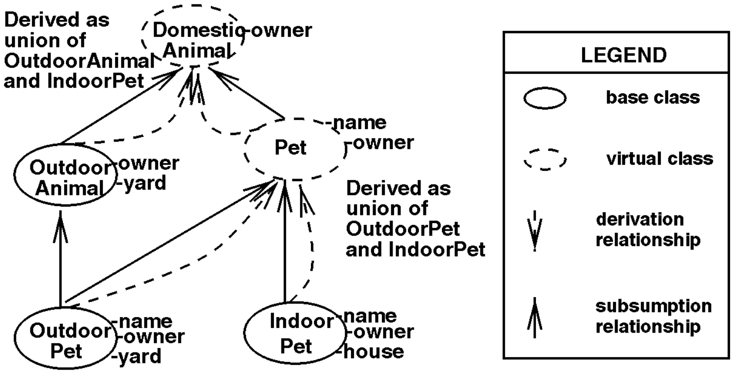

Fig. 3. Using subsumption to screen for directly affected classes.

object's membership regarding any class whose super or subclasses has already been evaluated without change.[8]

For example, Fig. 3 depicts a schema with three base classes, *OutdoorAnimal, OutdoorPet,* and *IndoorPet*. The *Pet* virtual class is formed from the union of the *OutdoorPet* and *IndoorPet* classes, and the *DomesticAnimal* class is formed by the union of *OutdoorAnimal* and *IndoorPet*. Now consider an object that is a member of the *OutdoorPet* class, but not the *IndoorPet* class. This means that this object qualifies to belong to all superclasses of the *OutdoorPet* class, so it also belongs to the *OutdoorAnimal, DomesticAnimal,* and the *Pet* classes. Now suppose that the object were to gain the type of the *IndoorPet* class (e.g., if it were to become an indoor/outdoor pet). The object must belong or be added to all superclasses of *IndoorPet*. Because the object's type memberships were consistent before the update, from step 2c of Procedure 1, we know that because the object already belongs to the *Pet* class it must already have the type of the *DomesticAnimal* class. That is, we do not have to evaluate its membership in any superclass of *Pet*, including other classes that are directly derived from the *IndoorPet* class.

3.3.2 Instance-Variable Value Updates and the Hierarchical Registration Strategy

Registration Strategy. In order to efficiently identify the set of classes **directly affected** by modification updates, we introduce a **registration** service by which virtual classes can register their interest in specific properties and be notified upon modification of those properties. This is facilitated by our data model's injunction that each property must reside at a single class that defines the property (as defined in Section 2). All classes whose membership may be affected by modifications to an attribute are thus registered with the attribute's **point of inheritance**. We associate triggers with the accessing methods to propagate updates to the registered classes if and only if the relevant property's value is modified.[9]

PROCEDURE 2 (Simple Registration of a Dependency). Each selection predicate term of the query of a newly created virtual "select" class C_n derived from a source class C_s is of the form *property* θ *value*, where comparison operator $\theta \in \{=, <, \neq, \leq, >, \geq\}$.

1) Determine the class *DefiningClass* from which the source class C_s inherits *property*.
2) Create an instantiation of a registration structure that records the property being tested, the name of the virtual class created by the query Q_n, and the comparison operator θ used in the test, e.g., <*property*, C_n, θ>.
3) Place this registration structure into the set of registration structures maintained by the *DefiningClass*, keyed by *property*.

For example, Fig. 4 represents a schema where an object of type *Cat* should possess the type of class *SmallCat* if the value of its `weight` instance variable is less than 11 pounds and the value of its `height` instance variable is less than 10 inches. Similarly, an object of type *Cat* should possess the type of class *LightCat* if the value of its `weight` instance variable is less than nine pounds. The *DenseCat* class is derived using a difference query between the *SmallCat* and *LightCat* classes. Because the *SmallCat* class's predicate involves the `weight` and `height` properties, the class is registered with the *Cat* class, which defines both of these properties. Similarly, the *LightCat* class is also registered with the *Cat* class for the `weight` property.

Suppose that an object of type *Cat* had the value of its `height` instance variable updated to eleven inches. An inefficient algorithm might evaluate the modified object's membership status in the *LightCat* class even though the value of the object's `weight` instance variable will not have changed. Fig. 4b illustrates how in the absence of registration an update to the `height` of an instance of *Cat* could be evaluated in terms of classes that are completely independent from that property. If, however, registration is used (see Fig. 4c), then we can readily identify exactly which simple selection classes are immediately affected by the update by looking at the registration structures associated with the modified property.

We can also use the comparison operators associated with each registration to analyze update operations and determine their potential effect upon the materialized virtual class. For example, if an object's value for an updated property fulfills a predicate involving a "greater-than" (>)

8. Note that, because both our upwards and our downwards traversals are rooted at the target class, in an actual implementation we would build up the set of *Directly Affected* classes dynamically.

9. Our registration service is similar in concept to the dependency mechanism provided by VisualWorks Smalltalk [40], but is tailored to our specific requirements, including optimization for subsumption relationships.

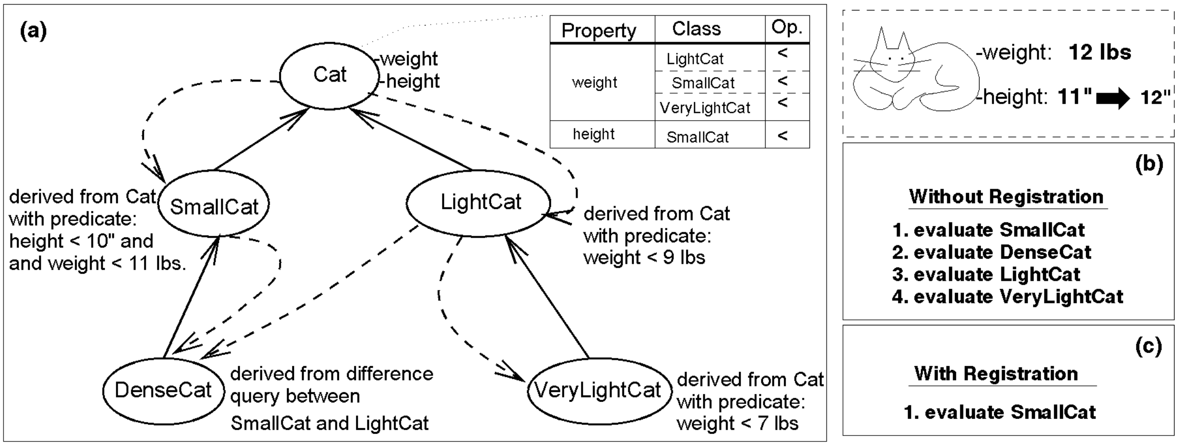

Fig. 4. Registrations at the point of inheritance.

comparison operator upon that property and the update increases the object's value for the property, then we do not need to re-evaluate the update in order to know that the updated object continues to fulfill the predicate.

Procedure 3 summarizes our algorithm as presented thus far for identifying classes **directly affected** by an update to the value of an object's property. Given an update that changes an object o_i's value for property *property*$_j$ from *value*$_o$ to *value*$_n$ (a *mutator* operation), we would first use the registration structures to identify the set of virtual classes that have registered an interest in the modified property. We would then compare the old and new values for the modified property in the context of the comparison operator registered for each virtual class and eliminate virtual classes that are identifiably not affected by the update from this set. We thus must evaluate the complete predicate of only those classes that have both registered an interest in the modified property and passed our update-specific/comparison-operator filter. After the procedure has completed, the set *DirectlyAffected* will contain exactly those virtual classes that are directly affected by the update.

PROCEDURE 3 (IdentifyDirectlyAffectedByMutator). Given an update that changes an object o_i's value for property *property*$_j$ from *value*$_o$ to *value*$_n$:

1) Initialize the variable *DirectlyAffected* to be a set that will contain the set of classes directly affected by the update.
2) Identify the class C_p that serves as the point of inheritance for *property*$_j$.
3) For each class VC_r that has registered with C_p for *property*$_j$:
   a) If the comparison operator associated with VC_r is <, object o_i belongs to VC_r before the update, and *value*$_n$ < *value*$_o$, or if the comparison operator associated with VC_r is >, object o_i belongs to VC_r before the update, and *value*$_o$ < *value*$_n$, then we do not need to evaluate the predicate because object o_i must still fulfill the predicate.
   b) If the comparison operator associated with VC_r is =, and *value*$_n$ ≠ *value*$_o$, then add VC_r to the set *DirectlyAffected*.
   c) Otherwise, evaluate whether the modification affects object o_i's membership in VC_i, and if there is a change then add VC_r to the set *DirectlyAffected*.

Hierarchical Registration Strategy. When a modification to an instance variable occurs, not all of the classes that have registered an interest in the modified property as described above will be **directly affected** by the update. For example, using the schema in Fig. 5, suppose that the `weight` of an instance of *Cat* were updated from 10 to 8 lbs. In that case, although we know that an object that doesn't fulfill the predicate of the *LightCat* class couldn't possibly fulfill the predicate of the *VeryLightCat* class, Procedure 3 (described above) would process both the *LightCat* and the *VeryLightCat* classes.

Because all classes in our model are arranged in a generalization hierarchy, we therefore optimize our registration process to take advantage of subsumption relationships. We now propose the optimization of the registration process using the notion of *hierarchical registration*. When a class C_i receives a registration for a class/property/operator tuple <p_i, VC_j, *operator*>, it first searches its existing class/property registrations. If a previous registration for the given property <p_i, VC_k, *operator*> exists such that the registered class VC_k either subsumes or is subsumed by the new class VC_j, the registrations are rearranged so that the class that defines the property p_i possesses the registration for the superclass (say VC_k) and that class possesses the registration for the subclass (say, VC_j).

PROCEDURE 4 (Hierarchical Registration of a Dependency). When a new "selection" virtual class VC_n is created, each predicate term (of the form *property* θ *value*, where comparison operator $\theta \in \{=, <, \neq, \leq, >, \geq\}$) represents VC_n's dependence upon the state of *property*. The **Hierarchical Registration** process parses a predicate term to register this dependency relationship in a hierarchical fashion:

1) Determine the class *DefiningClass* where *property* is defined. Set a variable *CurrentClass* to *DefiningClass*.

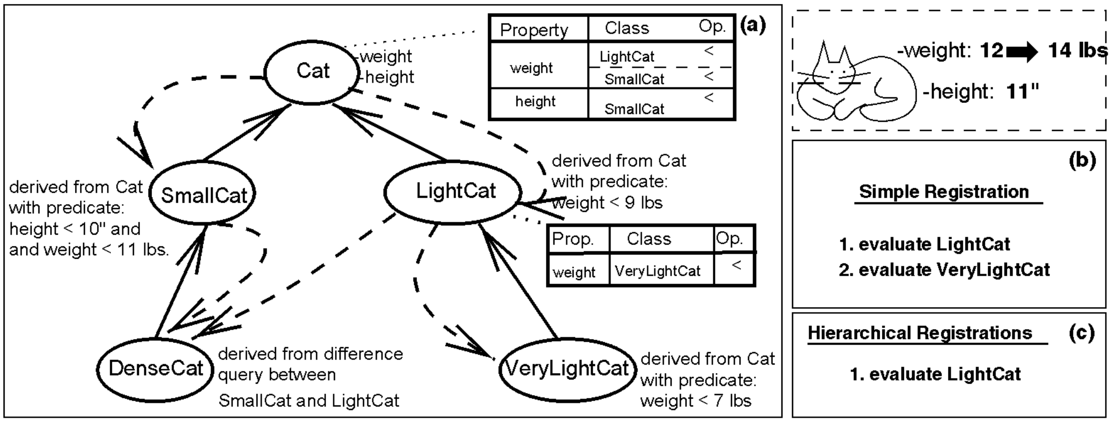

Fig. 5. Registrations use both point of inheritance and subsumption.

2) Retrieve the registration structures associated with *property* (e.g.,

$$<property, VC_i, \theta>, <property, VC_j, \theta>, \ldots)$$

from the *CurrentClass*'s set of registration structures. Collect the virtual classes from these structures, keyed on *property* into a set called *RegisteredClasses*.
3) If the new virtual class VC_n is subsumed by any class $VC_i \in$ *RegisteredClasses*, then set the *CurrentClass* variable to VC_i and discard *RegisteredClasses*'s previous contents. Go to Step 2.
4) For all $VC_j \in$ *RegisteredClasses* s.t. the new virtual class VC_n subsumes VC_j, remove the registration structure <*property*, VC_j, *operator*> from the *CurrentClass*'s registration structures and place it into the set of registration structures maintained by the new virtual class (keyed by *property*).
5) Create an instantiation of a registration structure that records the name of the new virtual class created by the query, the property being tested, and the test operator, i.e., <*property*, VC_n, **operator**>.
6) Place this registration structure <*property*, VC_n, *operator*> into the set of registration structures maintained by the *CurrentClass*, keyed by *property*.

This procedure builds a chain of registration structures into a hierarchy so that branches of this hierarchy can be eliminated, or "pruned," from the registration process. For example, the *registration* structures shown in Fig. 4 would thus be distributed along the generalization hierarchy, as illustrated in Fig. 5. Fig. 5a depicts the registration of the *SmallCat*, *LightCat*, and the *VeryLightCat* classes, all defined by selection queries. The *LightCat* class is registered with the *Cat* class for the `weight` property. However, because the *LightCat* class subsumes the *VeryLightCat* class and the former is registered with the *Cat* class, the latter's registration for the `weight` property is located with the *LightCat* class instead of the *Cat* class. Once the hierarchical registration structures have been constructed, then the following procedure can be embedded into the "set" (as opposed to "get") accessing method of each property used in a selection predicate.

Because our model includes complete encapsulation, when the value of an object o_u's instance variable is changed by a *mutator* operation, that change must take place using an accessing method defined by the class C_p that serves as the point of inheritance for that instance variable. Note that in the absence of encapsulation, generic operators could be used to achieve this end; however, encapsulation allows us to make our update mechanism transparent to the user. The following steps uses the registration structures as constructed using Procedure 5 to identify the set of classes directly affected by the update. After the procedure has completed, the set *DirectlyAffected* will contain exactly those virtual classes that are directly affected by the update.

PROCEDURE 5 (IdentifyDirectlyAffectedByMutator). Given an update that changes an object o_i's value for property *property*$_j$ from *value*$_o$ to *value*$_n$:

1) Initialize the variable *DirectlyAffected* to be a set that will contain the set of classes directly affected by the update.
2) Initialize the variable *PossiblyAffected* to be a set that will contain the set of classes that we must check to determine whether or not they are directly affected by the update.
3) Identify the class C_p that serves as the point of inheritance for *property*$_j$.
4) For each class VC_r that has registered with C_p for *property*$_j$, add VC_r to the set *PossiblyAffected*.
5) For each class $VC_p \in$ *PossiblyAffected*, follow the third step from Procedure 3. In addition, if o_i belonged to VC_p before the update, then for each class VC_q that has registered with VC_p for *property*$_j$, add VC_q to the set *PossiblyAffected*.

Fig. 5b illustrates how with just simple registration, an update to a *Cat*'s `weight` from 12 to 14 lbs would be evaluated in terms of both the *LightCat* and *VeryLightCat* classes. If, however, hierarchical registration is used (as illustrated in Fig. 5c), then we can readily identify exactly which simple selection classes are immediately affected by the update. Based on the registrations associated with the `weight` attribute, we would evaluate the updated object in terms of the *LightCat* class. Because the object does not gain the type

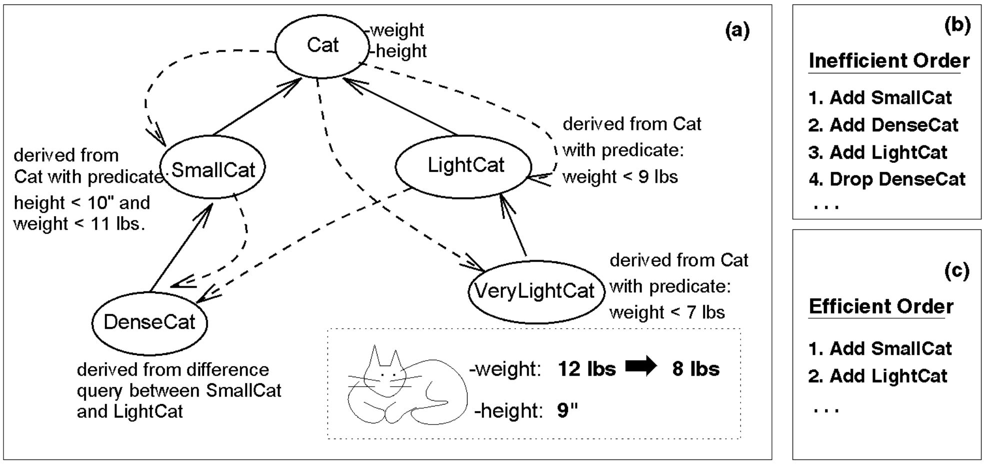

Fig. 6. An example schema where propagation order can lead to **self-canceling propagation**.

of *LightCat*, the propagation would terminate and the *VeryLightCat* class would not be processed. The procedure for performing a hierarchical registration of a newly created virtual class formed by a join query is similar to the procedures outlined above.

PROPOSITION 2. *The distribution of registration over the generalization hierarchy as defined in Section 3.3.2 ensures that we will identify exactly the set of simple selection classes that are* ***directly affected*** *by the modification of an instance variable.*[10]

3.4 Optimizing Propagation to Classes Derived from Directly Affected Classes

Once we have identified all classes that are **directly affected** by an update, it follows from Proposition 1 that we can complete the update propagation by propagating the update to the classes derived from the **directly affected** classes. In order to optimize propagation of the update to the **directly affected** classes and the classes derived from them, we have two goals:

1) **Identify branch termination conditions.** Under certain conditions, propagation to branches of the derivation or generalization hierarchy can be terminated; e.g., propagation to a branch of the derivation hierarchy can be terminated if the object does not qualify to belong to any of the entrance points into the branch either before or after the update. For example, consider the schema shown in Fig. 6, in which the *VeryLightCat* class is subsumed by the *LightCat* class. If an instance of the *Cat* class shown in Fig. 6 were to change the value of its *weight* instance variable from 12 to 10 lbs, then the instance would not qualify for membership in the *LightCat* either before or after the update. Our update algorithm should determine that since the instance does not belong to the *LightCat* class either before or after the update, there is no need to evaluate the instance's status in the *VeryLightCat* class, because the *VeryLightCat* class's membership predicate is subsumed by *LightCat*.

2) **Eliminate self-canceling update propagation**. The propagation process could involve updates that cancel out each other's effects. We call this the *self-canceling update propagation* problem because it could lead to repetitive evaluations of the object in the context of a single class, sometimes with self-canceling results. For example, Fig. 6a presents a schema vulnerable to the self-canceling problem. Fig. 6b illustrates how a decrease of the *weight* of an instance of the *Cat* class could cause it to first gain, then lose, the type of the *DenseCat* class. From this example it follows that a self-canceling propagation problem could extend to a potentially huge set of classes to the detriment of efficiency.

We can achieve both of these objectives by processing classes in the order of their derivation. We thus propagate changes to affected classes using a breadth-first traversal by order in the derivation hierarchy. Once the traversal reaches a class to which the updated object does not belong either before or after the update, it *terminates update propagation* to that class's branch. Because a derived class will only be processed after all of its source classes have been processed, we are guaranteed that no class will be processed more than once. (Note that it is possible that one of a class's source classes may not be processed at all.) This solution addresses the first problem (how to identify branch termination conditions) by not examining branches of classes that are immune to the effects of the update, thereby potentially eliminating a large set of dependent materialized classes. Derivation-ordered processing also avoids the second problem of self-canceling update propagations by ensuring that if DerNum(C_k) < DerNum(C_l), then a modified object's membership in C_k can be evaluated only before the object's membership in C_l has been evaluated.

For example, Fig. 6c shows how this strategy avoids the pitfall illustrated in Fig. 6b. Because we process the potentially relevant classes in order of derivation number, the status of an instance of *Cat* regarding class *DenseCat* will not be evaluated until after the object's membership status regarding both class *SmallCat* and class *LightCat* has been determined and stabilized.

10. A proof of this proposition is presented in [33].

TABLE 2
MODEL PARAMETERS

parameter	definition
k	# update transactions.
q	# query transactions.
P	Probability that a given operation is an update ($P = k/k + q$).
$extent(C_i)$	Set of objects in the extent of specified class C_i.
$class(obj)$	Type of object obj.
$f(Query)$	Selectivity of $Query$.
$ExtentCost(C_i)$	Cost in ms to retrieve the extent of a base or materialized class C_i; or calculate the extent of a non-materialized virtual class C_i.
$PerformCost(Update, obj)$	Cost in ms to perform the specified update upon the object.
$size(extent(C_i))$	Number of objects in class C_i's extent.
$AddCost(VC_i, obj)$	Cost in ms to add the type of virtual class VC_i to the object.
$DropCost(VC_i, obj)$	Cost in ms to drop the type of virtual class VC_i from the object.
$ScreenCost(Query, C_i)$	Cost in ms to screen one object from class C_i for $Query$, including cost of retrieving the object from C_i's extent.
$Cost(Query, C_i)$	Cost of applying $Query$ to the extent of class C_i.
$Cost(Update, C_i)$	Cost of applying an $Update$ to an object of class C_i.
$predComps(VC_i)$	# of predicate components in virtual class VC_i's query.
NV	# virtual classes in schema.
NSV	# selection virtual classes in schema.
$NSup(C_i)$	# of classes in the superclass hierarchy of C_i.
$NSub(C_i)$	# of classes in the subclass hierarchy of C_i.
NR	# of virtual classes registered for the updated property.
NUR	Number of virtual classes registered for the updated property that are subsumed by another registered class that is unaffected by the update and that does not contain the updated object in its extent.
$PAdd(VC_i, O_j, U_k)$	Probability that the update U_k causes the object O_j to gain the type of virtual class VC_i.
$PDrop(VC_i, O_j, U_k)$	Probability that the update U_k causes the object O_j to lose the type of virtual class VC_i.
$NDer(C_i)$	# classes directly or indirectly derived from class C_i
$TADer(C_i, C_{i+1}, \ldots C_{i+n})$	Average # classes in the union of the derivation hierarchies of classes $C_i \ldots C_{i+n}$ that need to be evaluated after any given update.
$VC_i.Query$	Query that defines virtual class VC_i.

PROPOSITION 3. *Using the derivation-ordering technique to control update propagation along the derivation hierarchy, an update will be evaluated in terms of any given class at most once.*[11]

4 COST ANALYSIS OF UPDATE PROPAGATION TECHNIQUES

In order to evaluate the impact of our optimization techniques analytically, we developed formal cost models for the costs of querying and updating materialized and nonmaterialized views, which we present below. For our analysis we consider each update operation to be applied to a single object. Unless stated otherwise, we characterize each query to perform a selection upon a single property. Each query thus requires an iteration through the view's extent retrieving a property from each object. For nonmaterialized views, we assume that query optimization will be performed during the query substitution process. Table 2 summarizes the parameters used in our models.

4.1 Cost of Querying a Materialized Virtual Class

If the view is materialized, then we do not need to compute the extent of the view in order to query it. Accessing an object from the extent of a materialized view involves only the cost of retrieving an object from GemStone (disregarding costs incurred by object-slicing). Thus the cost of performing the query upon the view extent does not involve any costs additional to the cost of performing the actual query. The time to query a materialized virtual class VC_i's extent, regardless of the number of classes from which it is derived or the complexity of the query that defines it, is thus:

$$Cost(Query, VC_i) := ScreenCost(Query, VC_i) \times size(extent(VC_i)). \quad (1)$$

11. A proof of this proposition is presented in [33].

4.2 Cost of Querying a Nonmaterialized Virtual Class

If the view is not materialized, then the cost of querying the view includes the cost of computing the view's extent in addition to the cost of performing the query. Thus each query upon a virtual class's extent can incur significant storage and computation costs. The cost of querying the extent of a nonmaterialized virtual class VC_i that is derived from a single source class SRC is roughly (discounting query optimization). Note that if $VC_i.Query$ is a select or join query, then the selectivity of the query ($f(VC_i.Query)$) will vary according to the specific values of objects in the database; if $VC_i.Query$ is a hide query then the $f(VC_i.Query)$ will be 100 percent (because all of the objects in the source class are members of the virtual class); and if $VC_i.Query$ is an intersect, implicit join, or union query then $f(VC_i.Query)$ will vary according to the specific type memberships of objects in the database.

$$\begin{aligned} Cost(Query, VC_i) := {} & ScreenCost(VC_i.Query, SRC) \times \\ & size(extent(SRC)) + \\ & ScreenCost(Query, VC_i) \times \\ & f(VC_i.Query) \times size(extent(SRC)) \end{aligned} \tag{2}$$

If a queried class is a nonmaterialized virtual class that is derived from other nonmaterialized virtual classes, then either each of the source classes in the queried class's derivation chain must be re-materialized or else the query must be modified (and optimized) so as to take the class's derivation history into account. Assuming that the selectivity of the view is immaterial because (due to query optimization) the current query is combined with the view-defining query, the cost of computing the extent of a class VC_i derived from a source class SRC_i (possibly itself a nonmaterialized virtual class) can be expressed recursively:

$$ExtentCost(VC_i) = \begin{cases} ScreenCost(VC_i.Query, SRC_i) & \text{if } SRC_i \text{ is} \\ \times\ size(extent(SRC_i)) & abaseclassor \\ & mat.virtualclass; \\ ExtentCost(SRC_i) + & \text{otherwise} \\ ScreenCost(VC_i.Query, SRC_i) & \\ \times\ size(extent(SRC_i)) & \end{cases} \tag{3}$$

4.3 Cost of Propagating Updates to Materialized Classes

When a virtual class is materialized, then we must maintain its extent in the face of updates to the database. After the update, all database objects' class memberships must concur with the class generalization relationships, and each database object should be a member of exactly those virtual classes whose queries it fulfills. The cost of an update to an object thus includes the cost of propagating the update to all affected materialized virtual classes, in addition to the cost of accessing the updated property. Because our update optimizations are tailored to combinations of classes, we focus on models with a varying number of virtual classes.

4.3.1 Cost of Propagating State-Altering Updates

Given an object and an update that modifies the value of one of the object's instance variables, the set of all materialized virtual classes derived using predicates (these classes are potentially directly affected) and the classes derived from those virtual classes (the classes derived from potentially directly affected classes) are the only classes that are potentially affected by this update. Any class that is derived using set-membership queries (e.g., hide or union) cannot be directly affected by the update (because the update changes the object's state rather than the object's set-membership). Without optimization (i.e., without registration or pruning), the cost of performing an update and propagating it to all affected materialized views is in the worst-case:

$$\begin{aligned} Cost(Update, obj) := {} & PerformCost(Update, obj) \\ & + \sum_{i=1}^{NSV} (ScreenCost(VC_i.Query.class(obj)) \\ & + (P\,Add(VC_i, obj, Update) \times N\,Der(VC_i) \\ & \times AddCost(VC_i, obj)) \\ & + (P\,Drop(VC_i, obj, Update) \times N\,Der(VC_i) \\ & \times DropCost(VC_i, obj))) \end{aligned} \tag{4}$$

Registration allows us to avoid evaluating virtual classes that are derived using queries that do not involve the updated property. With the optimization of distributed registration, we can avoid calculating the update's impact on any virtual classes registered for the updated property that are subsumed by another registered class that is unaffected by the update and that does not contain the updated object in its extent. Furthermore, the derivation ordering optimization we will process any virtual class at most once. With these optimizations, the cost performing and propagating the update becomes:

$$\begin{aligned} Cost(Update, obj) := {} & PerformCost(Update, obj) \\ & + \sum_{i=1}^{NR-NUR} [ScreenCost(VC_i.Query.SCR_i) \\ & + P\,Add(VC_i, obj, Update)\ Add(VC_i, obj) \\ & + P\,Drop(VC_i, obj, Update) \times Drop(VC_i, obj)] \\ & + \sum_{j=1}^{TADer(VC_1 \ldots VC_{NR-NUR})} [ScreenCost(VC_j.Query.SCR_j) \\ & + P\,Add(VC_j, obj, Update)\ Add(VC_j, obj) \\ & + P\,Drop(VC_j, obj, Update) \times Drop(VC_j, obj)] \end{aligned} \tag{5}$$

4.3.2 Cost of Propagating Class-Membership Updates

Given a class C_i and an update that adds/removes an object from the extent of C_i, then in addition to performing the update, which includes adding/removing the object from the extents of the superclasses or subclasses of C_i, we must also update the extent membership of any virtual class that is directly or indirectly derived from C_i or any of its superclasses/subclasses (superclasses if the update is a type addition, subclasses if the update is a type removal). Due to the self-canceling propagation problem, without optimization this process incurs a potential worst-case cost of (in the case of a type addition):

$$\begin{aligned} Cost(AddType, C_i) :=\ & PerformCost(AddType, C_i) \\ & + \sum_{j=1}^{NSup(C_i)} \sum_{k=1}^{NDer(Cj)} \\ & [ScreenCost(VC_k.Query, class(obj)) \\ & + (P\ Add(VC_k, obj, Update) \times N\ Der(VC_k) \\ & \times AddCost(VC_k, obj)) \\ & + (PDrop(VC_k, obj, Update) \times N\ Der(VC_k) \\ & \times DropCost(VC_k, obj)))]. \end{aligned} \quad (6)$$

or (in the case of a type removal):

$$\begin{aligned} Cost(DropType, C_i) :=\ & PerformCost(AddType, C_i) \\ & + \sum_{j=1}^{NSup(C_i)} \sum_{k=1}^{NDer(Cj)} \\ & [ScreenCost(VC_k.Query, class(obj)) \\ & + (P\ Add(VC_k, obj, Update) \times N\ Der(VC_k) \\ & \times AddCost(VC_k, obj)) \\ & + (P\ Drop(VC_k, obj, Update) \times N\ Der(VC_k) \\ & \times DropCost(VC_k, obj)))]. \end{aligned} \quad (7)$$

With our optimizations of derivation-ordered propagation and pruning, we will process any given class at most once and will not process classes subsumed by demonstrably unaffected classes at all. Our optimized propagation cost is thus (if the update is a type addition):

$$\begin{aligned} Cost(AddType, C_i) :=\ & PerformCost(AddType, C_i) \\ & + \sum_{j=1}^{TADer(C_1 \ldots C_{NSup(C_i)})} [PAdd(VC_j,\ obj,\ AddType) \\ & \times ScreenCost(VC_j.Query, class(obj))] \end{aligned} \quad (8)$$

or (if the update is a type removal):

$$\begin{aligned} Cost(DropType, C_i) :=\ & PerformCost(DropType, C_i) \\ & + \sum_{j=1}^{TADer(C_1 \ldots C_{NSub(C_i)})} [PAdd(VC_j,\ obj,\ AddType) \\ & \times ScreenCost(VC_j.Query, class(obj))] \end{aligned} \quad (9)$$

4.4 Nonmaterialized Update Costs

If a view is not materialized, then we do not need to propagate an update to an object's type or instance variable values to the view. The cost of an update to an object is thus simply the cost of accessing and modifying the updated property, namely

$$Cost(Update, obj) := PerformCost(Update, obj). \quad (10)$$

5 Performance Evaluation

The goal of our performance evaluation is to quantify the impact of our optimization techniques upon update propagation times while taking into account the overhead of our object-slicing implementation. While object-slicing does have a large effect upon the performance of the *MultiView* system, the experiments presented in this section are designed to demonstrate the benefit of our strategies independent of object-slicing. In fact, our propagation algorithms should work even more efficiently in the absence of an object-slicing architecture.

5.1 Experimental Environment

We have successfully completed a prototype implementation of the *MultiView* model as described in this paper using the commercial GemStone OODB system.[12] This includes the object-slicing representation as well as the incremental update propagation algorithms as outlined in this paper. Readers are referred to our other papers for a description of the implementation [36], [33], [45]. The experiments described below were carried out using *MultiView* on a Sun4m running SunOS 4.1.3 with 32 megabytes of memory. We employ a 10,000 object implementation of the OO7 benchmark schema for our tests [9]. Because neither the OO7 benchmark nor any other existing benchmark includes any tests for evaluating the performance of object-oriented view systems (materialized or otherwise), we used the OO7 benchmark as a basis for creating our own tests. Fig. 7 shows the three class subschema of the OO7 Benchmark's example database that is central to our experimental models.

5.2 Cost to Perform Basic Operations

The basic parameters of our cost formulae as used in our experiments are listed in Table 3. GemStone employs an optimized proprietary hash table lookup algorithm to find objects. The cost of a single object retrieval in the GemStone system depends upon whether the object being referenced is in the cache. The initial access to an object in a session is similar to a B-Tree lookup. Subsequent accesses are cached to provide increased performance [52].

Object-Slicing Overhead. As mentioned in Section 2.3, our object-slicing implementation considerably impacts performance. In order to isolate the impact of our propagation techniques from the effects of the object-slicing architecture, we tailored our experimental models to minimize the involvement of the object-slicing implementation. For the purposes of the ensuing discussion, however, readers should be aware that the fact that the object-slicing architecture distributes each object's state, access to local instance variables is cheaper than in traditional architecture, and access to inherited instance variables is more expensive (in the absence of clustering). An in-depth experimental evaluation of the object-slicing architecture is presented in [34].

5.3 Evaluating the Costs and Benefits of Materialization

The benefit of view materialization is improved times for queries on virtual classes, due to not having to reperform

12. More specifically, we use GemStone version 4.1.3, using the SmalltalkDB interface.

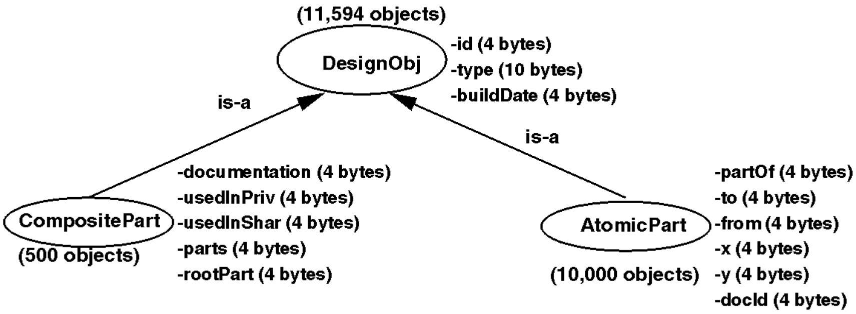

Fig. 7. Subschema of OO7 database.

TABLE 3
COST PARAMETERS OF THE BASE GEMSTONE SYSTEM USED IN ANALYSIS

parameter	definition (Cost in ms for an object to . . .)
$oid.retrieve$	follow an oid reference to locate an object.
$obj.class$	retrieve an object's class object.
$obj.access$	perform a "get" accessing method.
$obj.update$	perform a "set" accessing method.
$iterate(number)$	iterate through all the objects in a collection of size $number$.

the query that defines a virtual class. The cost of view materialization is that updates to objects belonging to classes on which the virtual class is based must be evaluated in order to determine their effect upon the extent of the materialized virtual class.

5.3.1 The Benefit of Materialization to Queries on Views

The first experiment demonstrates the basic benefit of materialization. Fig. 8 compares the cost of an aggregation query that computes the size of the extents of materialized and nonmaterialized *Select* virtual classes, each derived from a source class with a 10,000 object extent. The figure compares the cost of computing the size of the extent of a virtual class 1,000 times while varying the complexity of the virtual class's selection predicate. In this experiment we isolate the cost of the retrieval from the cost of any queries we might perform upon the retrieved extent. We chose this aggregate query because it allows us to ignore the cost associated with accessing and returning the result objects. This graph shows that we can compute the size of the extent of a materialized virtual class in constant time (near 0 milliseconds), independent of the complexity of the query defining the class.[13]

5.3.2 The Impact of Materialization on Update Queries

In order to compare the cost of materialization versus the benefit of a decreased query time, we focus upon queries and updates in the presence of an isolated selection view, *Select1*, directly derived from the base class *AtomicPart* (see Fig. 9). *Select1* is derived using the query:

AtomicPart createSelectClass:
    #Select1 query: [:ap | (ap x) < 435696658],

where approximately 1,000 of the 10,000 *AtomicPart* instances have randomly generated values for x that fulfill this query's predicate. For this model, we perform three series of tests, varying the parameters $predComps(VC_i)$ (number of predicate terms in the view predicate), k (number of update operations), and f (view selectivity).

In order to find out the ratio of the number of updates per query associated with the cost of materialization, we measured the time to perform k updates and 1 query where $0 \leq k \leq 5{,}000$, on a schema with the isolated view *Select1* with fixed selectivity $f(Select1.query) = 0.1$. The query and update we used are listed below:

- **Update1:** ((*AtomicPart* extent) at: (*aRandomNumber*)) x: *aNumber*

  where *aRandomNumber* is a random number between 1 and 10,000, and *aNumber* is either 1 or 2081201866, depending on the previous value of the object's x property. In this way we guarantee that every update results in a change in the *Select1* class's extent.

- **Query1:** *Select1* select: [:s1 | s1 id < 1088092823]

  For our database, this query results in the selection of approximately 500 objects from the 1,000 objects in the extent of *Select1* (the variance is because each update changes the extent of *Select1*). In order to minimize the affect of our object-slicing architecture, we are querying upon a property that is inherited by both that *AtomicPart* and *Select1* classes, namely *id* (as opposed to x) from Fig. 10.

For our first test, we ran *Update*1 k times and *Query*1 once for values of k from 0 to 5,000. Fig. 10 shows the average number of milliseconds to perform transactions while varying k. The results of this test indicate that with an isolated selection view of view selectivity $f(Select1.Query) := 0.1$,

13. Because a materialized virtual class's extent is maintained as a single instance of type Set, accessing the class's extent only needs to retrieve this single Set object. Of course the time needed to perform iterative operations upon the content of the extent set depends upon the number of objects in the set.

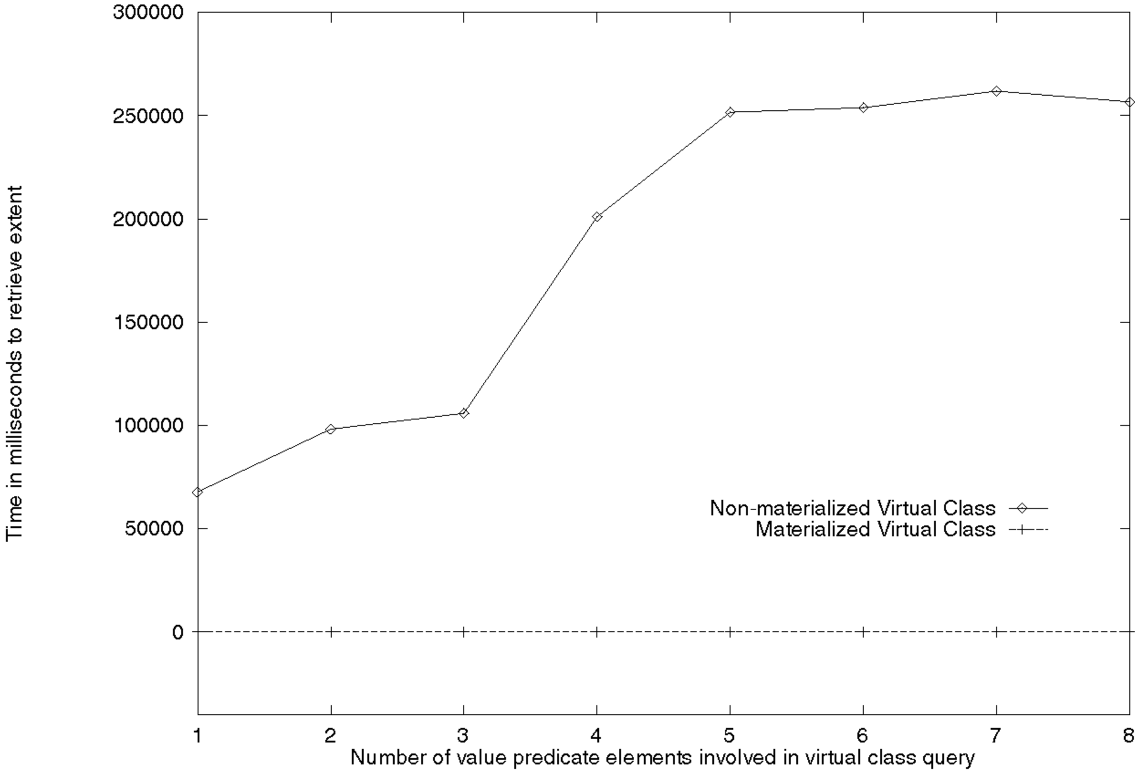

Fig. 8. Retrieval costs for materialized and nonmaterialized virtual class extents.

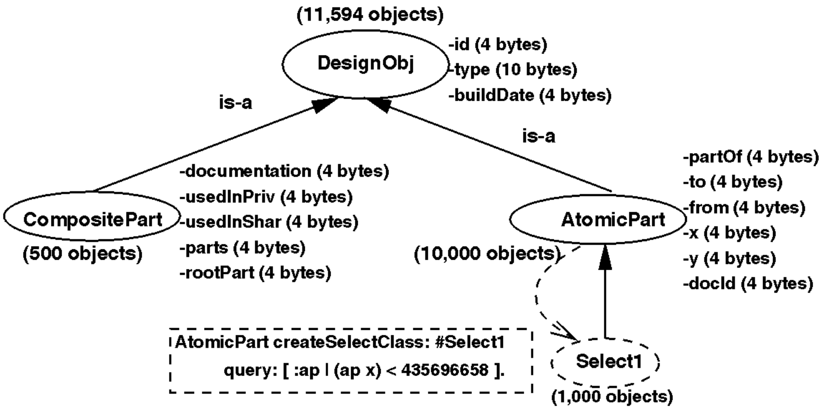

Fig. 9. The *Select1* virtual class is derived from *AtomicPart*.

an update impact probability of 1, and a query selectivity of *f*(*Query*1) := 0.05, the cost of performing a single query on the nonmaterialized view is counter-balanced by the cost of propagating approximately 3,000 updates to the materialized view.[14]

5.3.3 The Effect of View Predicate Selectivity

We next focus on the effect of view predicate selectivity by fixing the update probability *P* at 90 percent and varying the view selectivity factor *f* from 1 percent to 100 percent. Our goal in this experiment was to evaluate the impact of view selectivity upon materialization performance. Fig. 11 shows the results of performing a fixed transaction consisting of nine random updates and one query upon a simple selection view *Select2* derived from the *AtomicPart* class while varying the selectivity factor of the view from 0 (no objects fulfill the view predicate) to 1 (all 10,000 objects fulfill the view predicate). The selectivity factor of *Query2* is fixed at 1 percent of the 10,000 objects in *AtomicPart*.

- **Query2:** ((*Select2* extent) select:
  [:item | (item id) < 2081201856]

14. Although space does not permit us to present a discussion here, readers are referred to our performance evaluation of object-slicing in [34], which discusses the impact of inheritance upon performance in object-slicing. In the following sections, we try to isolate our experiments from the effects of object-slicing by querying upon inherited properties.

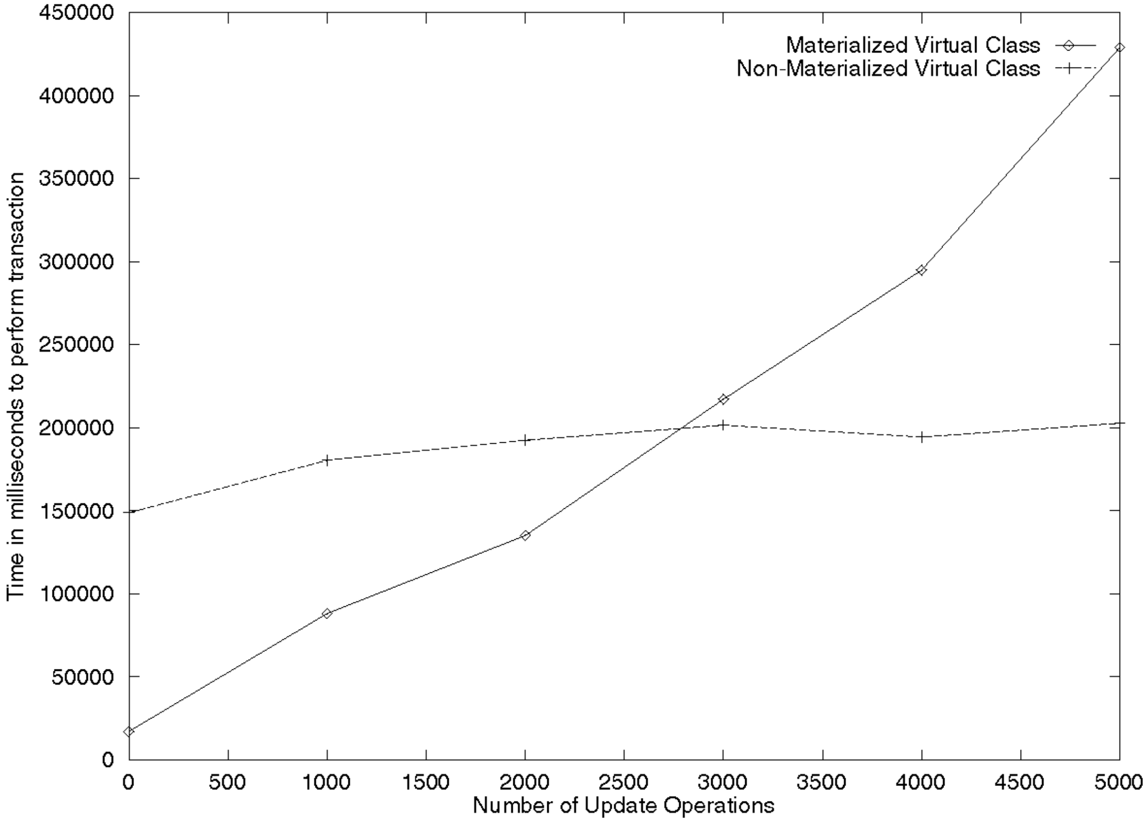

Fig. 10. The benefit of materialization in performing a single query balances the cost of propagating approximately 2,750 updates.

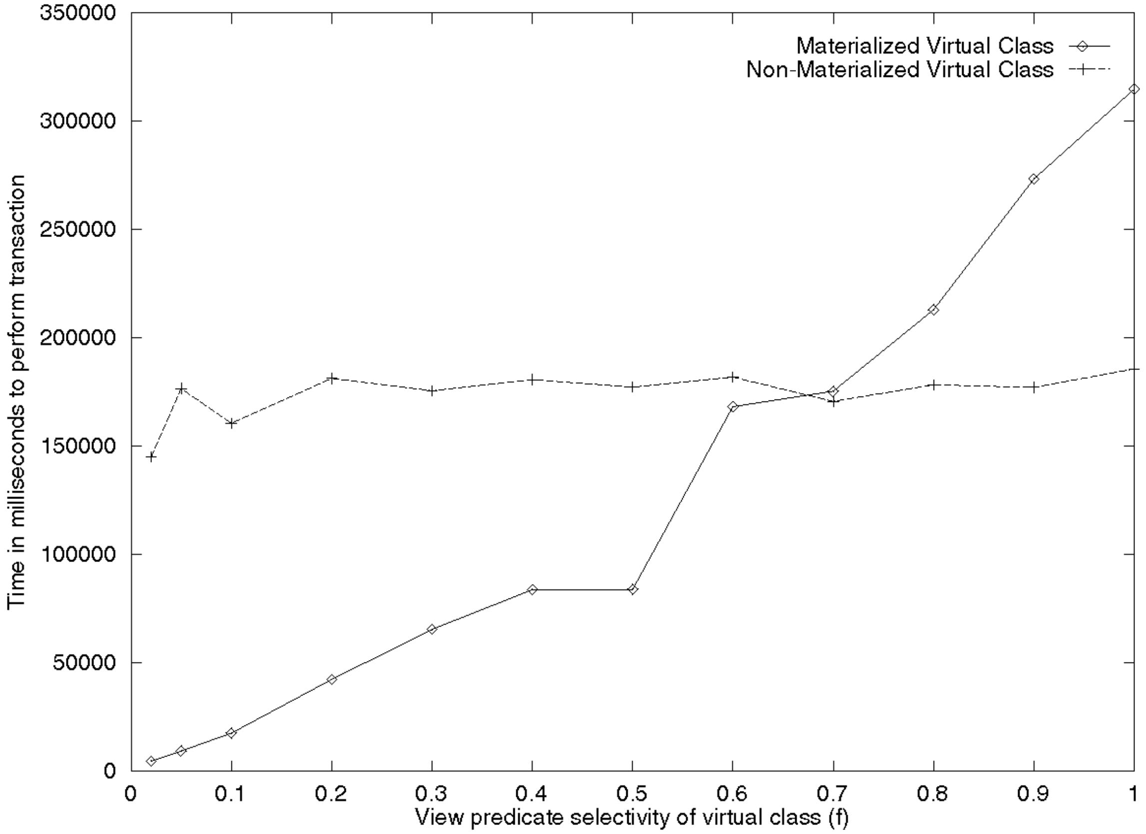

Fig. 11. Milliseconds to run nine updates (k = 9) and one retrieval (q = 1) on an inherited property while varying view selectivity (f).

From Equation 1 and Equation 2, we would expect that when query optimization is used during the query-rewriting process, the selectivity factor of the view has a greater impact on the cost of querying the materialized view than the nonmaterialized view. This is because as the selectivity of the view increases, the cost of performing the query upon the materialized view increases until it approximates the cost of performing the query upon a nonmaterialized version of the view, and the cost of propagating the updates comes to outweigh the benefit of materialization. This is reflected by the rise of the solid line in graph 11 (indicating the time in milliseconds to perform the updates and query on a materialized class). On the other hand, querying the nonmaterialized view involves one iteration upon the extent of the view's source class regardless of the view selectivity factor (assuming that query optimization is performed). The dotted line in the graph (indicating the time in milliseconds to perform the updates and query on a nonmaterialized class) reflects this fact by remaining fairly level.

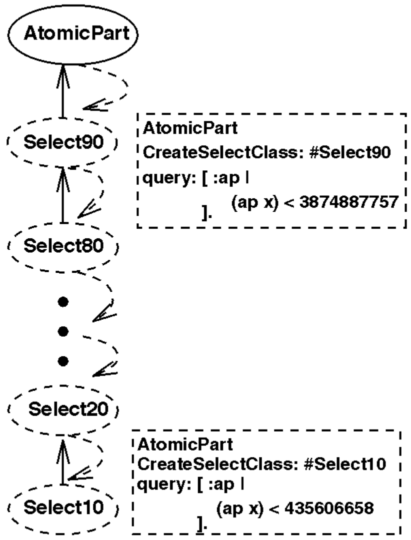

Fig. 12. A chain of selection classes derived from each other.

5.4 Evaluating the Hierarchical Registration Service

We next focused our evaluation on the modify operator, which being the most complex can result in an object gaining and losing the types of multiple classes. We evaluated the effects of our hierarchical registration strategy by combining *multiple select classes* in a single schema, varying the subsumption relationships of the select class predicates (Fig. 12).

In order to test the impact of the view hierarchy upon materialization propagation performance, we measured the time in milliseconds required to perform 10,000 updates in the presence of an increasing number of selection classes arranged in the subsumption hierarchy depicted in Fig. 12. *Select90* is derived from the *AtomicPart* class, *Select80* from the *Select90* class, etc. Because the benefit of the hierarchical ordering optimization lies in avoiding unnecessary evaluations, we fix the selectivity of the update at 0 (the update doesn't affect any updated objects' virtual class memberships) and define all virtual classes as dependent upon the updated property (x). Given an object that does not belong to the uppermost selection class, we expect that any update that does not cause the object to gain the type of the uppermost selection class to be immediately terminated if hierarchical registrations are employed. Otherwise (if the registrations are not distributed along the subsumption hierarchy), all classes must be evaluated in terms of the update.

Fig. 13 shows the time needed in milliseconds to perform 10,000 updates in a schema with a varying number of selection classes, where *NUR* varies from 0 to 25. When no virtual classes are present, *NUR* = 0, when one virtual class is present, *NUR* = 1, and so on. We compare the performance of our implemented update propagation method that does not distribute registrations across the generalization hierarchy to the performance of one that does.

The benefit of distributing registrations is conspicuous for this sample scenario. The dotted line in Fig. 13 indicates the time in milliseconds needed to perform 10,000 updates when the registrations are not distributed along the subsumption hierarchy. The solid line indicates the time in milliseconds needed to perform the 10,000 updates when the registrations are distributed. As we would expect, when no or only one virtual class exists in the schema, the two algorithms require the same amount of time. However, as the number of virtual classes registered for the updated property increases, the nonhierarchical algorithm must check the updated instance against each of the virtual class's predicates. The algorithm with hierarchical registrations only needs to check the updated instance against the topmost virtual class's predicate.

5.5 Evaluating the Benefit of Derivation Ordering

We now examine the relative impact of registration and derivation level propagation on the self-canceling update propagation scenario. In order to calculate the relative cost and benefit to algorithms of compensating for this pitfall, we devised a test in which we measured the time to update 1,000 objects of the *AtomicPart* in the presence of an increasing number of virtual classes arranged in a manner vulnerable to the self-canceling update propagation problem. We evaluated the impact of our propagation optimization techniques by implementing four propagation algorithms that employ combinations of our two main optimization techniques (derivation-ordered propagation and interest registration).

Fig. 14 organizes the four algorithms we implemented in terms of the optimizations they perform. Algorithm *both* and Algorithm *only-reg* use relevance registration to avoid propagating updates to irrelevant derived classes. Algorithm *both* and Algorithm *only-ordering* both update in order of derivation level, so they are not subject to self-canceling update propagation.

For this test, we implemented the updates so that they alternately incremented and decremented the value of the x instance variables of the updated instances by one. The results of this test, displayed in Fig. 15, indicate that in the presence of the self-canceling update problem, an algorithm using only derivation ordering out-performs one using only registration. In summary, our tests indicate that although

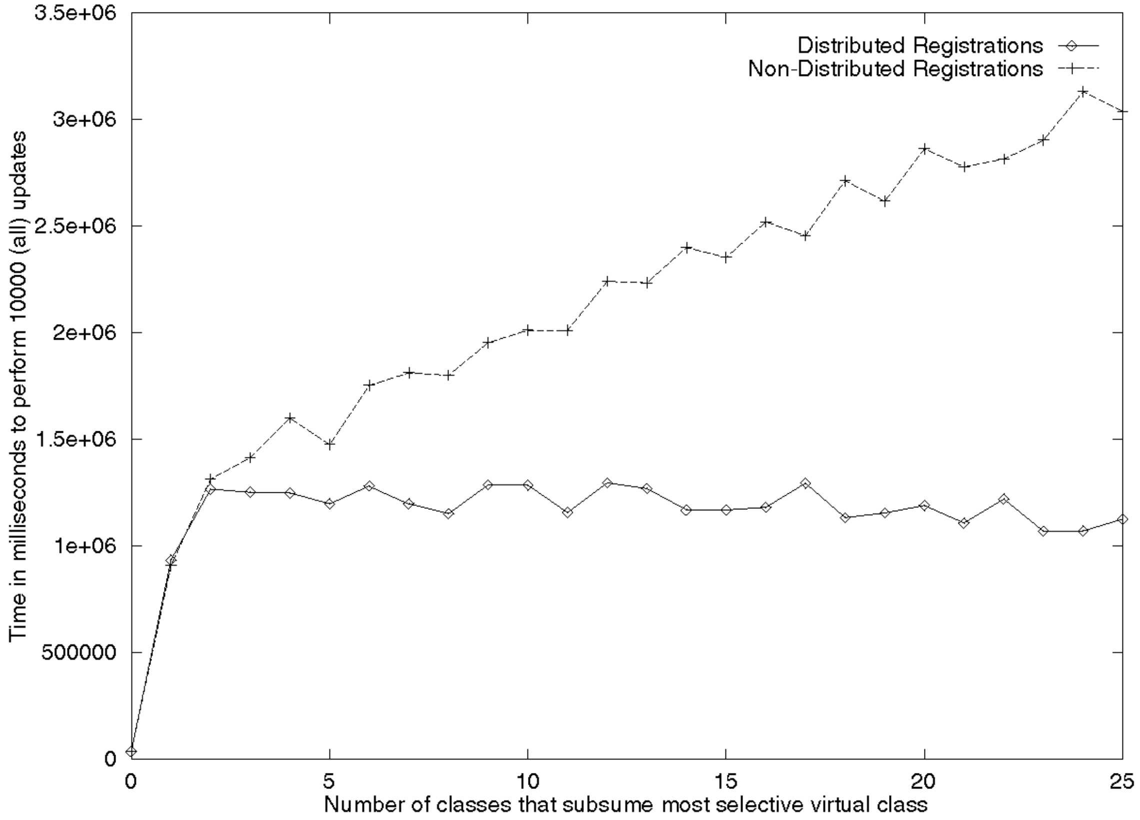

Fig. 13. Comparing the performance of hierarchical and nonhierarchical registrations while varying the height of the hierarchy.

registration alone significantly improves the time needed to propagate updates, an algorithm that employs registration and propagates in order of derivation number can significantly outperform one that employs only registration in the presence of even one class that is prone to the self-canceling update problem. Note that without derivation-ordering, update time increases exponentially with the number of self-canceling classes present. Because additional selection classes are added in order to create each self-canceling virtual class, the algorithm that uses derivation-ordering and without registration suffers a moderate performance loss as the number of self-canceling classes increases. However, the time of the algorithm using both derivation-ordering and registration is hardly affected at all by the increased number of self-canceling classes.

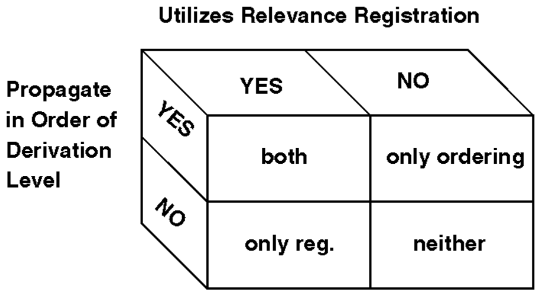

Fig. 14. Optimizations of the four algorithms.

5.6 Evaluating Branch Termination

As detailed in Section 3.4, when our update propagation algorithms determine that an object belongs to a given virtual class neither before nor after an update, we terminate evaluating that object's status for that particular branch of the class hierarchy. That is to say, we do not evaluate the updated object's membership in terms of any class that is a subclass of that given virtual class. This concept is distinct from hierarchical registration in that whereas hierarchical registration helps streamline the process of identifying which classes are directly affected by a modification of an instance variable's value, branch termination optimizes the more general process of propagating updates to classes derived from the directly affected classes.

In order to isolate the effects of this optimization, we used a schema similar to that shown in Fig. 12. Initially, none of the instances in the database qualify to belong to any of the Selection clases. For the actual experiment, we set the value of the *x* instance variable of all 10,000 instances of the *AtomicPart* class to values high enough that none of the instances qualify to belong to even the most general selection class while increasing the number of classes subsumed by the most general virtual class. We measured the time needed to perform the 10,000 updates with branch termination both enabled and disabled. As shown in Fig. 16, in the absence of branch termination the time needed to evaluate the update increases steadily, whereas if branch termination is enabled, the additional classes are pruned out of the set of classes that must be checked by the propagation procedure.

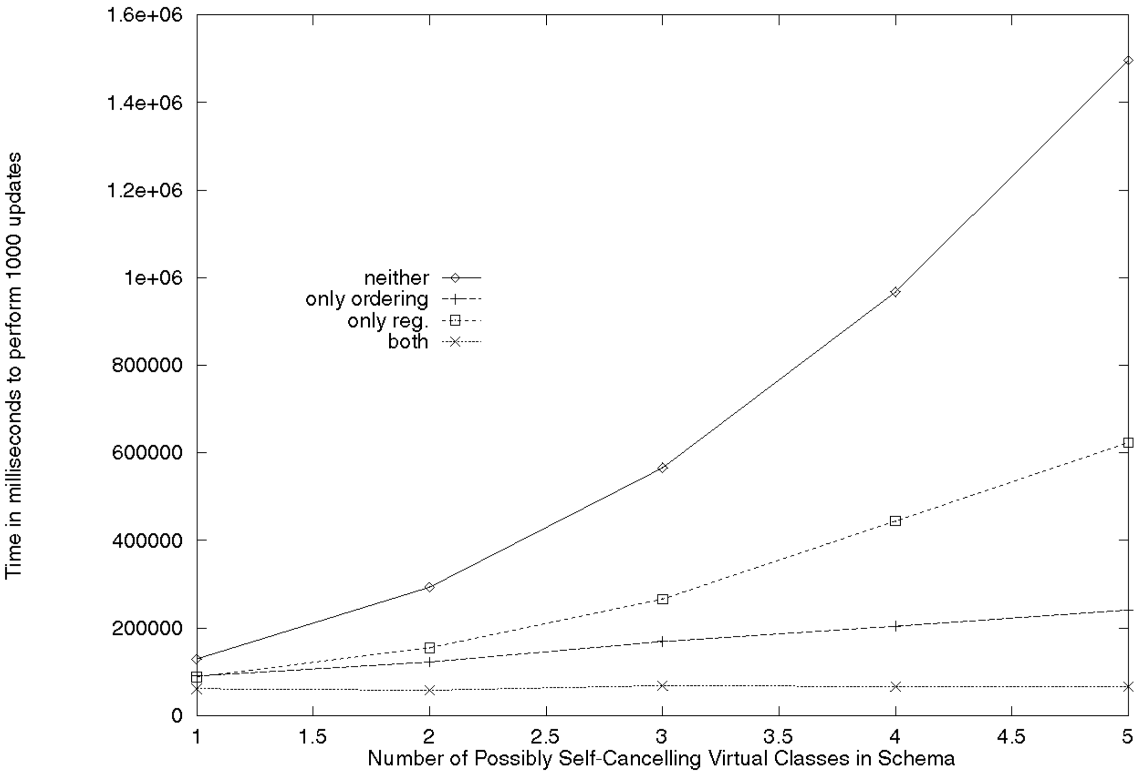

Fig. 15. Propagation in a schema with self-canceling updates.

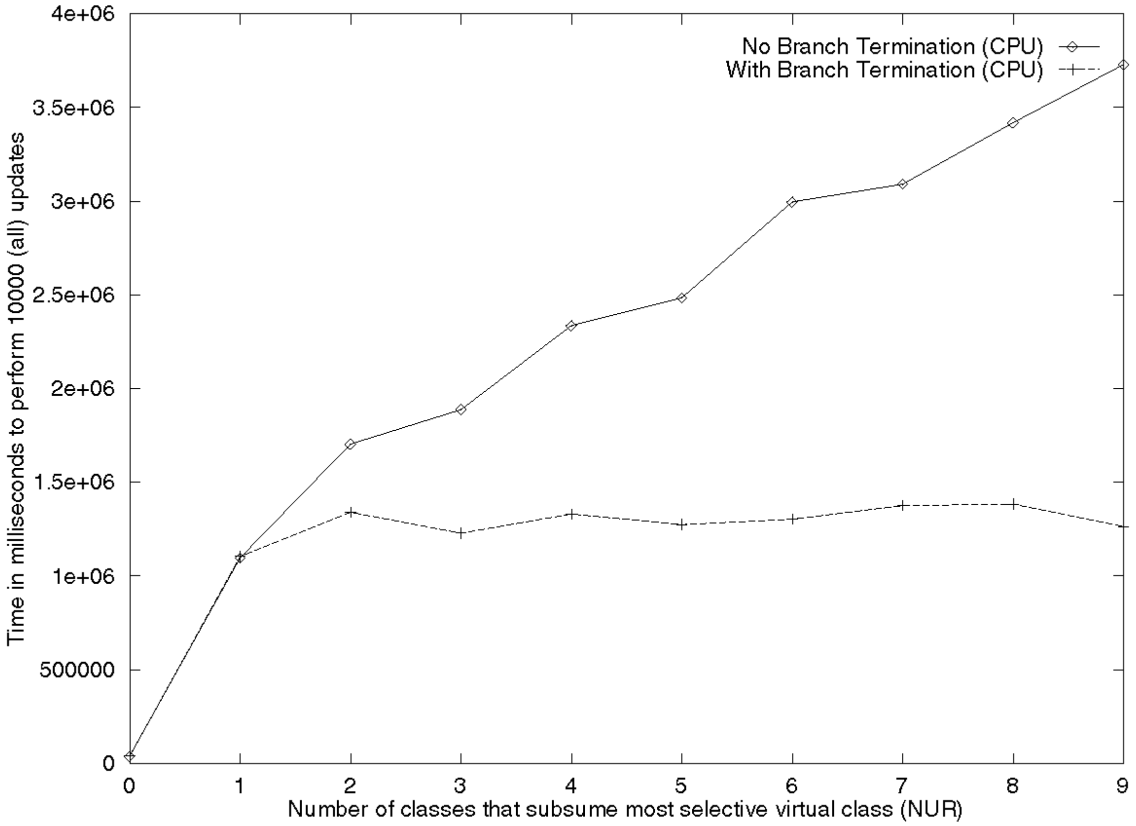

Fig. 16. Evaluating the effects of branch termination.

6 RELATED WORK

Below, we discuss work related to the materialization techniques described in this paper. We begin by comparing our techniques with the the considerable work done regarding the maintenance of materialized views in relational databases. Next, we review research in object-oriented view materialization. Finally, we conclude with a discussion of nonview-specific research into predicate-based topics (such as integrity constraint maintenance, predicate caches, rule based system, etc.) that relates to our optimization techniques.

6.1 Relational View Materialization

Relational and object-oriented systems share a common motivation for view materialization: the goal of improving query performance [5], [22], [24]. Although not much work has been done regarding OODB view materialization, object-oriented and relational systems both must address a number of common issues when designing a view materialization system, such as when to evaluate updates and how much to update. Our research on view materialization in OODBs borrows several techniques from the relational arena, but also extends them to exploit unique features of the object-oriented paradigm.

In particular, two aspects of our registration process are related to the work of preceding researchers:

1) we test modified objects to see if they fulfill the predicates of virtual classes that are created using queries that involve the modified property; and
2) Blakeley et al. test modified tuples to see if they fulfill view predicates, thereby detecting irrelevant and autonomously computable updates [5], [6].

However, unlike tables in relational databases (RDBs), both virtual and base classes in the *MultiView* model are arranged in an *integrated generalization hierarchy*. Information about the relations between the types and extents of OODB classes can thus be easily recognized by database users. We exploit the object membership information implicit in the hierarchical structure in order to perform further screening of affected classes.

The system provided by Ceri and Widom performs incremental view maintenance using production rules that are triggered by update operations [10]. Gupta et al. also present algorithms that use the view definition to produce rules that use the changes made to the base relations and the original materialized views to compute changes to the view [19]. Similarly, updates in the *MultiView* model trigger maintenance operations. However, the values of relational attributes are updated using generic queries, and thus update procedures must be implemented for the general case. On the other hand, because OODBs support *encapsulation*, updates to attributes are localized to specific accessing methods that are themselves defined at a single identifiable class. We can thus embed our update propagation methods into the accessing methods for the properties referenced by the predicates of materialized virtual classes. The accessing methods then serve as *update triggers*. Furthermore, we can exploit encapsulation by overriding generic-update operations with type-specific update operators for virtual classes.[15]

Gupta and Blakeley present formal partial-information-based view maintenance techniques that infer knowledge about the state of underlying base relations using local information (such as the view definition, the update, the current view materialization, and varying amounts of base relation replicas). They identify classes of materialized views according to the amount of information needed to maintain them in the face of updates [17], and demonstrate necessary and sufficient conditions for determining the amount of information needed to update a materialized select-project-join view. *MultiView*'s path query views, which are not discussed in this paper, are similar to the relational select-project-join view. However, for a discussion of our path query views (including a comparison between our work and that of others), readers are referred to [38].

6.2 Object-Oriented View Materialization

Only a few published papers address issues of view materialization in OODBs. Our research is unique in that we are the first to implement updatable materialized OODB views, and thus also the first to evaluate the performance of view materialization algorithms in the context of OODBs. We are also unique in our examination of incremental updates in the context of the object-slicing paradigm.

Hardwick and Downie focus on the use of materialized views to integrate related engineering applications [24]. They treat materialized views as versions that can be experimentally changed without impacting others. They use change files (histories of the design sessions) to represent updates to their materialized views/versions. With these change files, they use materialized views to produce a tool by which schema version integration can take place. In their prototype, updates are coordinated with human interaction. *MultiView*, on the other hand, propagates updates automatically and immediately.

Segev and Zhao propose a join pattern indexing technique for materialized rule-derived data that allows the identification of join completion without reading base objects/relations [55]. A join pattern relation is a precomputation of complete (all join attributes instantiated) and incomplete (not all join attributes instantiated) joins that satisfy the constraints of a rule. Join pattern indexing represents join relationships between existing data items/tuples.

Konomi et al. [32] use super-key classes to maintain consistency for a particular type of join class formed along an existing path in the aggregation graph. Superkey classes facilitate the incremental update and elimination of duplicates for materialized views produced by relational expressions that include projections. When join classes are first derived from a path of classes, Konomi et al. check the class schema for a super-key condition. If the super-key condition does not exist, then the class schema is transformed so that they will satisfy the super-key condition by adding extra classes and redundancies. The goal of their work is to permit the efficient deletion of objects. Our work instead

15. Again, note that similar ends could be achieved in the absence of encapsulation by implementing generic operators.

focuses on a general object algebra, and supports all standard update operations.

6.3 Evaluating Predicate Satisfaction

The crux of the view materialization problem is determining whether or not the state of data items satisfy predicate conditions. This is a general problem common to many areas of database research, including:

- deductive databases (which must determine if an updated data item violates a database assertion),
- active databases (which identify rules triggered by database updates),
- integrity constraint maintenance (how to assure that constraints are satisfied in the face of updates to the database), and
- predicate caches/predicate classes (which must maintain caches of data described by predicates).

We thus benefit from the efforts of researchers in these areas to address the problem of logically analyzing the potential effects of database updates upon predicate satisfaction.

In their short abstract, Hammer and Sarin propose a method for *perturbation analysis*—anticipating the potential effect of an update upon database assertions in deductive databases [21]. They provide an *assertion processor* that produces a set of *candidate tests* that indicate whether or not the update violates the assertion. They provide an algorithm for determining the least expensive candidate test. In principle, this is similar to our evaluation of state modification, in that before we begin to evaluate the actual predicate, we first take into account the nature of the modification and the type of the registered predicate comparison operator.

Hanson and Johnson propose a selection predicate indexing scheme for active databases that uses an *interval skip list* index that facilitates the efficient retrieval of all intervals that overlap a point [23]. This index structure provides an efficient manner in which to test a collection of predicates to identify which predicate conditions an object satisfies. Although *MultiView* does not currently use this index structure, it could prove beneficial if we were to adopt this scheme, in that it would facilitate the evaluation of updated objects. However, because *MultiView*'s registration process already filters the set of predicates that must be evaluated, we would need to perform a more detailed analysis to determine whether the benefit of creating an *interval skip list* for *MultiView* would exceed the cost in storage.

Gupta et al. use techniques from query containment to address the problem of identifying whether or not an updated object satisfies database constraints without performing a complete evaluation of the constraints [20]. This problem translates directly onto the view materialization problem. Gupta et al. classify query languages according to the types of constraints supported and the amount of information needed from the updated item to test the constraint. *MultiView* performs similar optimizations in our techniques for comparing class memberships and also in our update filter methods. Vielle et al. determine whether or not updates violate integrity constraints [58].

Finally, Keller and Basu propose a new type of cache called *predicate caches*, which are created by having client machines cache data from query results [29]. The contents of a client machine's cache can thus be described using a predicate expression, but must be maintained in the face of updates to server data. This is essentially the same problem addressed in view materialization, and Keller and Basu do not focus upon a solution to this problem, permitting client machines to use either automatically refresh cached data upon updates to the server or else refresh upon demand by a subsequent query.

7 Contributions and Future Issues

In this paper, we identify techniques by which we can exploit unique features of the object-oriented paradigm to optimize updatable materialized views. Our update algorithms are incremental, and perform selective notification to the set of classes determined to be **directly affected** by a given update. We use inherent object-oriented features, such as the integration of classes into a generalization hierarchy, encapsulation, and membership materialization, both to facilitate the identification of the classes that are **directly affected** by an update and also to optimize the propagation to the classes derived from classes **directly affected** by the update. New techniques we introduce include distributed registration, subsumption-based propagation termination, and derivation-ordered prevention of the self-canceling effect.

We have implemented a prototype (version 1.0) of the *MultiView* system [39], [36], which supports updatable materialized object-oriented views. Our system incorporates the incremental update propagation strategies introduced in this paper. We present a cost model and experimental results that demonstrate the performance gains derived from our proposed view materialization strategies. We have identified two potential inefficiencies to which update propagation algorithms are subject—the propagation of updates to irrelevant derived classes and the propagation of self-canceling updates. Our evaluation demonstrates that our techniques of *relevance registration* and *derivation-ordered propagation* successfully address these problems.

While object-slicing does have a large effect upon the performance of the *MultiView* system, the experiments presented in this paper are designed to demonstrate the benefit of our strategies independent of object-slicing. In fact, our propagation algorithms should work even more efficiently in the absence of an object-slicing architecture. In a related paper, we explore the use of clustering to ameliorate the overhead of the object-slicing technique employed by our implementation [34].

In the future, we plan to study more powerful query languages for view definition. We also plan to study issues related to the support of deferred updates and multiple (batched) updates for materialized views.

ACKNOWLEDGMENTS AND THANKS

This work was supported, in part, by the National Science Foundation under RIA Grant No. IRI-9309076, by NSF NYI Grant No. IRI 94-57609, and by the University of Michigan Faculty Award Program, Intel, and AT&T. Harumi Kuno is also grateful for support from the NASA Graduate Student Researchers Program.

A "thank you" goes to Young-Gook Ra, who helped design and implement the *MultiView* object-slicing architecture. Thank yous and appreciation also go to Pedro Jose Marron, Daryl Scott Cantrell, Matthew Jones, and Victor Ng, who implemented *MultiView*'s graphic interface. We are grateful to Dr. Umesh Dayal for suggesting that we further optimize our registration process to take into account the nature of the update operation and for pointing us toward relevant literature. Besides the persons listed above, we also would like to thank the other members of our University of Michigan Database Group, in particular Viviane Taube, Stacie Hibino, Woody Huang, and Nauman Chaudhry, for valuable discussions and suggestions throughout the development of the *MultiView* system.

REFERENCES

[1] S. Abiteboul and A. Bonner, "Objects and Views," *Proc. SIGMOD*, pp. 238–247, 1991.
[2] A. Amir and N. Roussopoulos, "Optimal View Caching," *Information Systems*, vol. 15, no. 2, pp. 169–171, 1990.
[3] E. Baralis, S. Ceria, and S. Paraboschi, "Conservative Timestamp Revisited for Materialized View Maintenance in a Data Warehouse," *Proc. SIGMOD Workshop Materialized Views: Techniques and Applications*, pp. 1–9, 1996.
[4] E. Bertino, "A View Mechanism for Object-Oriented Databases," *Proc. Third Int'l Conf. Extending Database Technology*, pp. 136–151, Mar. 1992.
[5] J. Blakeley, P. Larson, and F. Tompa, "Efficiently Updating Materialized Views," *Proc. SIGMOD*, pp. 61–71, 1986.
[6] J.A. Blakeley, N. Coburn, and P.-A. Larson, "Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates," *ACM Trans. Database Systems*, vol. 14, no. 3, pp. 369–400, Sept. 1989.
[7] S.K. Bratsberg, "Unified Class Evolution by Object-Oriented Views," *Proc. 12th Int'l Conf. Entity-Relationship Approach*, pp. 423–439, 1992.
[8] P. Breche, F. Ferrandina, and M. Kuklok, "Simulation of Schema Change Using Views," *Proc. Int'l Conf. and Workshop Database and Expert Systems Applications*, pp. 247–258, 1995.
[9] M.J. Carey, D.J. DeWitt, and J.F. Naughton, "The OO7 Benchmark," technical report, Univ. of Wisconsin-Madison, Jan. 1994.
[10] S. Ceri and J. Widom, "Deriving Production Rules for Incremental View Maintenance," *Proc. Int'l Conf. Very Large Data Bases*, pp. 577–589, Sept. 1991.
[11] Oracle Corp., "Oracle Express Objects," http://www.levelv.co.uk/Technology/oracolap.html#sect1-2, 1996.
[12] W. Du, S. Peterson, and M.-C. Shan, "Enterprise Workflow Architecture," *Proc. IEEE Int'l Conf. Data Eng.*, pp. 63–64, 1995.
[13] D.H. Fishman, "Iris: An Object Oriented Database Management System," *ACM Trans. Office Information Systems*, vol. 5, pp. 48–69, Jan. 1987.
[14] J.C. French, A.K. Jones, and J.L. Pfaltz, "Scientific Database Management," Technical Report No. 90-22, Univ. of Virginia, Aug. 1990.
[15] J.P. Gilbert, "Supporting User Views," *Computer Standards and Interfaces*, vol. 13, pp. 293–296, 1991.
[16] T. Griffin and L. Libkin, "Incremental Maintenance of Views with Duplicates," *Proc. PODS*, pp. 328–339, 1995.
[17] A. Gupta and J.A. Blakeley, "Using Partial Information to Update Materialized Views," *Information Systems*, vol. 20, no. 8, pp. 641–662, 1995.
[18] A. Gupta and I.S. Mumick, "Maintenance of Materialized Views: Problems, Techniques, and Applications," *IEEE Bull. Technical Committee on Data Eng.*, vol. 18, pp. 3–18, June 1995.
[19] A. Gupta, I.S. Mumick, and V.S. Subrahmanian, "Maintaining Views Incrementally," *Proc. SIGMOD*, pp. 157–166, 1993.
[20] A. Gupta, Y. Sagiv, J.D. Ullman, and J. Widom, "Constraint Checking with Partial Information," *Proc. PODS*, pp. 45–55, 1994.
[21] M. Hammer and S.K. Sarin, "Efficient Monitoring of Database Assertions," abstract, *Proc. SIGMOD*, p. 159, 1978.
[22] E.N. Hanson, "A Performance Analysis of View Materialization Strategies," *Proc. SIGMOD*, pp. 440–453, 1987.
[23] E.N. Hanson and T. Johnson, "Selection Predicate Indexing for Active Databases Using Interval Skip Lists," Technical Report No. TR94-017, Univ. of Florida, Gainesville, Apr. 1994.
[24] M. Hardwick and B.R. Downie, "On Object-Oriented Databases, Materialized Views, and Concurrent Engineering," *Proc. 1991 ASME Int'l Computers for Eng. Conf. and Exposition*, Engineering Databases: An Engineering Resource, 1991.
[25] S. Heiler and S.B. Zdonik, Object Views: Extending the Vision," *Proc. IEEE Int'l Conf. Data Eng.*, pp. 86–93, 1990.
[26] Itasca Systems Inc., *Technical Summary Release*, 2.0 ed., 1991.
[27] H.V. Jagadish, I.S. Mumick, and A. Silberschatz, "View Maintenance Issues for the Chronicle Data Model," *Proc. PODS*, pp. 113–124, 1995.
[28] H. Jakobsson, "On Materializing Views and On-Line Queries," *Proc. ICDT*, pp. 407–420, Oct. 1992.
[29] A. Keller and J. Basu, "A Predicate-Based Caching Scheme for Client-Server Database Architectures," *VLDB J.*, vol. 5, no. 1, pp. 35–47, Jan. 1996.
[30] J. Kiernan and M.J. Carey, "Extending SQL-92 for OODB Access: Design and Implementation Experience," *Proc. OOPSLA*, pp. 467–480, 1995.
[31] H.J. Kim, "Issues in Object-Oriented Database Systems," PhD thesis, Univ. of Texas at Austin, May 1988.
[32] S. Konomi, T. Furukawa, and Y. Kambayashi, "Super-Key Classes for Updating Materialized Derived Classes in Object Bases," *Proc. Int'l Conf. Deductive and Object-Oriented Databases*, pp. 310–326, July 1993.
[33] H.A. Kuno, "View Materialization Issues in Object-Oriented Databases," PhD thesis, Univ. of Michigan, Ann Arbor, June 1996.
[34] H.A. Kuno, Y.G. Ra, and E.A. Rundensteiner, "The Object-Slicing Technique: A Flexible Object Representation and its Evaluation," Technical Report No. CSE-TR-241-95, Univ. of Michigan, 1995.
[35] H.A. Kuno and E.A. Rundensteiner, "Materialized Object-Oriented Views in MultiView," *Proc. ACM Research Issues in Data Eng. Workshop*, pp. 78–85, Mar. 1995.
[36] H.A. Kuno and E.A. Rundensteiner, "The MultiView OODB View System: Design and Implementation," *J. Theory and Practice of Object Systems (TAPOS)*, special issue on subjectivity in object-oriented systems (extended version of Technical Report No. CSE-TR-246-95), 1996.
[37] H.A. Kuno and E.A. Rundensteiner, "The MultiView OODB View System: Design and Implementation," H. Ossher and W. Harrison, eds., *Theory and Practice of Object Systems (TAPOS)*, special issue on subjectivity in object-oriented systems, John Wiley, New York, 1996.
[38] H.A. Kuno and E.A. Rundensteiner, "The Satisfiability-Indicating Multi-Index Organization for Maintaining Materialized Path Query OODB Views," Technical Report No. CSE-TR-302-96, Univ. of Michigan, Ann Arbor, 1996.
[39] H.A. Kuno and E.A. Rundensteiner, "Using Object-Oriented Principles to Optimize Update Propagation to Materialized Views," *Proc. IEEE Int'l Conf. Data Eng.*, pp. 310–317, 1996.
[40] S. Lewis, *The Art and Science of Smalltalk*, Prentice Hall, 1995.
[41] M. Loomis, T. Atwood, R. Cattell, J. Duhl, G. Ferran, and D. Wade, "The ODMG Object Model," *J. Object Oriented Programming*, pp. 64–69, June 1993.
[42] C. Mohan, G. Alonso, R. Gunthor, and M. Kamath, "Exotica: A Research Perspective on Workflow Management Systems," *IEEE Bull. Technical Committee on Data Eng.*, pp. 19–26, Mar. 1995.
[43] O2 Technology, *O2 Views User Manual*, version 1 ed., Dec. 1993.
[44] R. Peters, M.T. Ozsu, and D. Szafron, "TIGURAT: An Object Model for Query and View Support in Object Database Systems," Technical Report No. TR 92-14, Univ. of Alberta, Canada, Oct. 1992.

[45] Y.G. Ra, H.A. Kuno, and E.A. Rundensteiner, "A Flexible Object-Oriented Database Model and Implementation for Capacity-Augmenting Views," Technical Report No. CSE-TR-215-94, Univ. of Michigan, 1994.
[46] Y.G. Ra and E.A. Rundensteiner, "A Transparent Schema Evolution System Based on Object-Oriented View Technology," *IEEE Trans. Knowledge and Data Eng.*, submitted for publication, 1996.
[47] E.A. Rundensteiner, "*MultiView*: A Methodology for Supporting Multiple Views in Object-Oriented Databases," *Proc. 18th VLDB Conf.*, pp. 187–198, 1992.
[48] E.A. Rundensteiner, "Tools for View Generation in OODBs," *Proc. Int'l Conf. Information and Knowledge Management*, pp. 635–644, Nov. 1993.
[49] E.A. Rundensteiner, "A Classification Algorithm for Supporting Object-Oriented Views," *Proc. Int'l Conf. Information and Knowledge Management*, pp. 18–25, Nov. 1994.
[50] E.A. Rundensteiner, H.A. Kuno, and Y. Zhou, "Incremental Maintenance of Materialized Path Query Views," *Proc. OOIS, Int'l Conf. Object-Oriented Information Systems*, 1998.
[51] B. Schiefer, "Supporting Integration and Evolution with Object-Oriented Views," Technical Report FZI-Report No. 15/93, Forschungszentrum Informatik (FZI), July 1993.
[52] D. Schneider, e-mail transmittal, Servio Customer Support query response, 1995.
[53] M.H. Scholl, C. Laasch, and M. Tresch, "Updatable Views in Object-Oriented Databases," *Proc. Second DOOD Conf.*, Dec. 1991.
[54] M.H. Scholl and H.J. Schek, "Survey of the Cocoon Project," *Objektbanken fur Experten*, Oct. 1992.
[55] A. Segev and J.L. Zhao, "Efficient Maintenance of Rule-Derived Data Through Join Pattern Indexing," *Proc. Int'l Conf. Information and Knowledge Management*, pp. 194–205, Dec. 1993.
[56] J. Shilling and P. Sweeney, "Three Steps to Views: Extending the Object-Oriented Paradigm," *Proc. OOPSLA*, pp. 353–361, Oct. 1989.
[57] K. Tanaka, M. Yoshikawa, and K. Ishihara, "Schema Virtualization in Object-Oriented Databases," *Proc. IEEE Int'l Conf. Data Eng.*, Feb. 1988.
[58] L. Vieille, P. Bayer, and V. Küchenhoff, "Integrity Checking and Materialized Views Handling by Update Propagation in the EKS-V1 System," Technical Report CERMICS No. 91.1, CERMICS-INRIA, June 1991.
[59] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom, "View Maintenance in a Warehousing Environment," *Proc. SIGMOD*, 1995.

Harumi A. Kuno received her BA degree from Yale University, and her MS degree and PhD degree from the University of Michigan. Her research was supported, in part, by a NASA Graduate Research Fellowship. She is currently with Hewlett-Packard in Santa Clara, California. Her current research interests include object-oriented databases, data warehousing, distributed database applications, and web databases. She is a member of the IEEE Computer Society.

Elke A. Rundensteiner received her BS degree (Vordiplom) from the Johann Wolfgang Goethe University, Frankfurt, West Germany; her MS degree from Florida State University; and her PhD degree from the University of California, Irvine. She spent several years as a faculty member in the Department of Electrical Engineering and Computer Science at the University of Michigan, and she is currently an associate professor in the Department of Computer Science at the Worcester Polytechnic Institute. She has been active in the database research community for more than 10 years, and her current research interests include object-oriented databases, data warehousing, database and software evolution, multimedia databases, distributed and web database applications, and information visualization. She has published more than 100 papers in these areas. Her research has been funded by a number of government agencies, including the National Science Foundation, the Advanced Research Projects Agency, NASA, the CRA, the DOT, and by such companies as IBM, AT&T, Informix, and GE. She has been a member of the program committees of such key database-field conferences as IEEE ICDE, ACM SIGMOD, and VLDB. She has received numerous honors and awards, including a Fulbright Scholarship, an NSF Young Investigator Award, an Intel Young Investigator Engineering Award, and an IBM Partnership Award. She is a member of the IEEE Computer Society.