Using Object-Oriented Principles to Optimize Update Propagation
to Materialized Views *

Harumi A. Kuno and Elke A. Rundensteiner
Dept. of Elect. Engineering and Computer Science
Software Systems Research Laboratory
The University of Michigan, 1301 Beal Avenue
Ann Arbor, MI 48109-2122
e-mail: kuno@umich.edu, rundenst@eecs.umich.edu
fax: (313) 763-1503
phone: (313) 936-2971

University of Michigan Technical Report # CSE-TR-252-95

Abstract

View materialization is known to be a valuable technique for performance optimization in relational
databases, and much work has been done addressing the problem of consistently maintaining 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. For example, we can
use the subsumption relationships between classes to identify branches of classes to which we do not
need to propagate updates. Similarly, we can 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. We describe a number of techniques that take advantage of such data model differences
to optimize the maintenance of materialized views in the object-oriented model. We have successfully
implemented all proposed techniques in the MultiView system, which provides updatable materialized
classes and virtual schemata on top of the GemStone OODBMS. In this paper, we also 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.

Keywords: Incremental view maintenance, object-oriented views, distributed registration, classification, view updatability.

---

* This work was supported in part by the NSF RIA grant #IRI-9309076, NSF NYI grant #IRI 94-57009, and 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.
1 Introduction

Views have been established as an effective tool by which database programmers can virtually restructure a schema so that it meets the needs of specific applications. Typically, the contents of a view are derived using the view's stored query, in which case each query upon the extent of a view must be translated into a query upon the view's source classes. In relational systems, this is known as query modification [9].

View materialization, i.e., the maintenance of derived extents of views as opposed to recomputing them upon demand, has been demonstrated in the relational model to improve the performance of queries. If the view is materialized, then its contents must be maintained in the face of updates to the view's sources. While a number of researchers have proposed view systems in the context of object-oriented databases (OODBs) [1, 11, 17, 27, 24, 25, 26], little work has been done on the support of materialized OODB views. Here at the University of Michigan, we are developing MultiView, a view management system capable of supporting updatable materialized object-oriented views [24, 16, 25, 26, 19].

In a previous paper [19], we introduced our implementation (using object-slicing techniques) of the MultiView model for object-oriented view materialization. In this paper, we present the results of our research regarding the optimization of update propagation. First, we identify potential inefficiencies that can affect propagation algorithms (e.g., the propagation of updates to irrelevant derived classes and the propagation of self-canceling updates). Second, we propose new techniques to propagate updates efficiently using object-oriented principles. Third, we discuss experimental studies we have conducted to evaluate the performance impact of our optimizations on the MultiView prototype.

We draw upon several techniques from previous relational work in the area. For example, [6] perform incremental view maintenance using production rules that are triggered by update operations. [3, 4] test modified tuples to see if they fulfill view predicates. We also use these strategies of triggering incremental view maintenance and filtering irrelevant updates. However, our current work exploits unique features of the object-oriented paradigm to adapt these solutions to the OODB view materialization problem. The generalization hierarchy structure makes the subsumption relationships between classes readily apparent. We thus propose an update propagation algorithm that identifies branches of classes to which we do not need to propagate updates. In addition, because we integrate virtual classes into the global schema as full-fledged citizens, we can use encapsulated interfaces combined with the fact that any unique database property is inherited from a single location to provide a "registration" service. Our registration strategy allows virtual classes to register their interest in specific properties and be notified upon modification of those properties. We introduce in this paper an optimized distributed registration technique that exploits knowledge about the subsumption relationships between classes to distribute the registrations along the generalization hierarchy and avoid unnecessary propagation.

We present a performance analysis that assesses the costs and benefits of our materialization techniques. Because we have implemented the MultiView system $^1$, we were able to evaluate actual implementations of each proposed update propagation algorithm (rather than just simulations). Specifically, we analyze the performance of non-materialized views, incrementally updated materialized views, and optimized incrementally-updated materialized views. 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 non-materialized views. The results of our experiments validate our cost models.

We begin by discussing work related to ours in Section 2. In Section 3, 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 4 to develop efficient update propagation techniques. In Section 5 we derive cost formulae and compare the performance of each of the proposed algorithms in terms of representative models. Finally, in Section 6 we present conclusions and propose future work.

2 Related Work

Relational and object-oriented systems share a common motivation for view materialization: the goal of improving query performance. 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. [3, 4] tests modified tuples to see if they fulfill view predicates, thereby detecting irrelevant and autonomously computable updates. This resembles our solution of filtering irrelevant updates by exploiting the generalization hierarchy

---

$^1$ The current MultiView prototype has been built on top of Servio Corporation's GemStone OODB product.
and the derivation structure. The system provided by [6] performs incremental view maintenance using production rules that are triggered by update operations. Similarly, we override generic-update operations with type-specific update operators for virtual classes.

However, there are significant differences between view materialization in OODBs and relational systems.

- Tables in relational databases (RDBs) are arranged in a “flat” fashion in the schema. There is no information about the subset or subtype relationships contained between tables in the schema. 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. Furthermore, update propagation techniques can be designed to exploit the object membership information that is implicit in the hierarchical structure in order to terminate propagation as soon as appropriate.

- Views in RDBs refer to virtual tables that are formed by applying query operations to base tables. Because an OODB schema is organized into a generalization hierarchy, OODB views include both virtual classes (formed by applying query operations to base classes) and virtual schema (formed by applying query operations to a base or view schema).

- Attributes in RDBs are identified only as the names of columns in tables. If two relational tables each have an attribute with the same name, there is no guarantee that the two attributes refer to the same property. Each property in the MultiView model is defined at a single class and using property inheritance shared by other classes. Database users can thus unambiguously identify whether or not two properties represent the same instance variable or method. In addition, when a predicate term in a virtual class’s query invokes a given property, we can identify exactly which class defines that property, enabling us to localize the effects of updates.

- Attributes in RDBs are associated with simple values. Although the simple value can be a foreign key referring to a row in another table, RDB attributes cannot contain sets or other complex data structures. Instance variables in OODBs, on the other hand, can contain references to actual objects or sets of objects. OODBs thus contain aggregation relationships that can be used in view formation queries.

- 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 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.

- Data in RDBs exists as values in tables. Thus when virtual tables in RDBs are materialized, the materialized tables typically contain copies of data values from the base tables. On the other hand, data in OODBs exists as objects that can be uniquely identified and then accessed using unique object identifiers (oids). Thus when virtual classes in OODBs are materialized, the materialized classes can contain references to the relevant objects rather than value duplicates (copies) of the original objects. This not only saves space, but also significantly simplifies the view update problem. Furthermore, regardless of which properties are “hidden” by a “project” query, objects belonging to a virtual class can be easily associated with their base incarnations. In fact, this provides us with a basis for the arbitrary restructuring of an object’s look and feel without compromising its identity.

Only a few published papers address issues of view materialization in OODBs. [10] provide a view materialization model in which updates are propagated by use of change files, representing histories of design sessions. However, [10] duplicate objects (including identifiers) for virtual classes rather than merely storing references to objects. [14] address maintaining consistency for a particular type of join class formed along an existing path in the aggregation graph. Our work instead focuses on the exploitation of the structure of the schema hierarchy and derivation dependency graph in order to reduce update propagations. Our research is also unique in studying incremental updates in the context of the object-slicing paradigm.

Although the object-slicing techniques underlying the current implementation of MultiView can be compared to mechanisms used in role modeling approaches [28, 21, 8], no other research in the literature discusses

---

2 Note that some materialized RDB views may be implemented using techniques such as view indices that resemble membership materialization.
the application of the object-slicing paradigm regarding the support of object-oriented views. The role system proposed by Gottlob et al. was implemented using techniques similar to object-slicing [8]. This system, like ours, is implemented in Smalltalk. The difference between [8] and our implementation is that [8] is a role system, whereas we are developing a view system. Role systems strive to increase the flexibility of objects by enabling them to dynamically change types and class membership. View systems, on the other hand, enable users to restructure the types and class membership of complete classes—based on content-based queries. For example, unlike [8], we do not permit entities to occur several times in the same type of role. Also, the [8] system does not permit the derivation of new virtual classes, and thus does not address issues related to view management.

3 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.

3.1 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($C_i$) \(^3\). 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 $C_i \subseteq C_j$ 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 $(\forall o \in O)(o \in C_i)$ implies $(o \in C_j)$. Thus \( \text{globalExtent}(C_i) = \bigcup \text{localExtent}(C_j) \forall C_j \subseteq C_i \). $C_j$ 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 $(C_i \not\subseteq C_j)$. $C_i$ is a direct subclass of $C_j$ if $\exists C_k \in C$ 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 $V_i$ to $V_j$ represents the relationship $C_i$ is-a $C_j$. 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.

3.2 Object-Oriented Views

We now extend our basic model to include views. The set of all classes, $\mathcal{C}$, 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. They can be added dynamically to the schema throughout the lifetime of the database. The virtual-class-forming operations supported by the current implementation include the following algebra operators: hide, refine, select, union, intersect, join, and difference. These queries determine the methods, instance variables, and extent of the virtual classes. The join operator is object-generating; all other operators are object-preserving.

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. \(^4\) Value predicates, used by select queries, are predicate terms

\(\text{1 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 [13, 12]. Furthermore, the proposed ODMG standard [2] recently formulated by several key OODB vendors also follows this approach.}

\(\text{2 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.}
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 formed by applying a hide query to a source class is membership-dependent upon that source class. Similarly, a virtual class formed by applying a select query that uses some instance variable’s 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. We do not replicate objects that belong to materialized virtual classes, but instead store references to them. 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 Sections 4 and 5).

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 [16, 26] 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 4.

### 3.3 Object-Slicing Representation

Introducing virtual classes and schemata requires the underlying data model to support the following features:

- **Multiple classification** If an object instance qualifies for membership in two virtual classes, then it should belong to both even if no subscription relationship exists between the virtual classes.

- **Dynamic reclassification** Furthermore, 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.

- **Dynamic restructuring** 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.

- **Multiple inheritance** Certain virtual classes, such as those formed by intersect queries, must inherit properties from both their source class types.

With the exception of the IRIS functional DBMS [7], 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 and do not support the features identified as required for view support. We thus could not directly use existing OODB technology to implement MultiView. As described elsewhere [22, 19], we adopt an object-slicing technique to provide a means of supporting these required features [20]. 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 provides a solution that 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, \( O_{i_{con}} \), and one implementation object for each class \( C_j \in C \) to which the object belongs, \( O_{i_{im \neq C_j}} \). A conceptual object...

---

5 In fact, as we will describe in the following subsection, we employ an object-slicing technique for object materialization.
consists of a tuple, \(<\text{implObjects}, \text{OID}>\), where \text{OID} is the unique, system-generated object-identifier of the conceptual object, and \text{implObjects} is the set of implementation objects that are linked to the conceptual object. An implementation object is a tuple \(<\text{OID}, \text{oid}, \text{class}, \text{state}>\) where \text{OID} is the object-identifier of the linked conceptual object, \text{oid} is the object-identifier of the implementation object itself, \text{class} is the class of which the implementation object is a direct instance, and \text{state} corresponds to the values of the local instance variables stored for the given object. Each implementation object \(O_{\text{impl},i,j}\) is an object instance of the database class \(C_j \subseteq C\) it represents. Conceptual objects are object instances of a special system class, \text{ConceptualObject}, rather than of a user-defined class.

For example, Figure 1 depicts a schema composed of two base classes, \text{Cat} and \text{HouseCat}, and two virtual classes, \text{LightCat} (derived from a selection query upon the \text{Cat} class) and \text{NamedLightCat} (derived by refining \text{LightCat} to add a new instance variable, \text{name}). The right-hand side of the figure illustrates a single instance of the \text{HouseCat} class with instance variables \text{height} = 9", \text{weight} = 71bs, and \text{owner} = Fred. Because the instance fulfills the membership requirements of \text{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.

![Diagram of object slicing](image)

Figure 1: Example of object-slicing.

Because a single real-world object is now represented using multiple database objects, we define a number of functions to operate on objects in the model, including object creation, equality comparison, etc. For example, we say that two objects are \textit{equal} if they are linked to the same conceptual object. Similarly, given an object \(O_i \in O\) and a class \(C_j \subseteq C\), we define functions \text{MakeImp}(O_{\text{impl},i},C_j)\) and \text{DeleteImp}(O_{\text{impl},i},C_j)\) to create and destroy implementation objects of class \(C_j\) for object \(O_{\text{impl},i}\).

Classification of virtual classes, as shown in Figure 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 \textbf{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.\footnote{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.}

This promotion, called \textbf{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 \textbf{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 [19].

The object-slicing technique, combined with the characteristic of \textbf{single point of inheritance} and \textbf{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 \text{LightCat} virtual class in Figure 1 directly shares the \text{height} attribute with the \text{Cat} class—without duplicating its value. Consequently, regardless of whether the update is specified via the \text{Cat} or the \text{LightCat} class, the same copy of
of the height attribute will be modified. Since object-slicing requires an object to possess implementation objects for any class to which it belongs, an object-slicing view implementation is effectively materialized in that the implementation objects representing an object’s membership in virtual classes are actual object- instances of those virtual classes. The object-slicing technique is thus subject to the intrinsic problem of view materialization – how to update materialized views so as to keep their extents consistent with the rest of the database. Details on implementation issues in our model such as method resolution, inheritance mechanisms, etc., can be found elsewhere [15, 22].

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 [15], 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 representation models. This work was done in the context of regular base classes only [15]. 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.

4 Minimizing Update Propagation for Object-Oriented Views

4.1 General Goals

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, if an object gains the type of a class that has a union class derived from it, then one direct effect of that update might be that the object gains the type of the union class (i.e., a new implementation object of the union 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. Two, an operation that modifies the value of an object’s instance variable potentially directly affects classes that are value-dependent upon a given instance variable.

For example, given the schema shown in Figure 1, 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 was 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. Together, these invariants provide the basis for the specification of the semantics of update propagation. 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.

- **Invariant 1**: 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 invariant 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.

- **Invariant 2**: An object is a member of a virtual class \( VC_i \in VC \) iff it fulfills the conditions of \( \text{query}(VC_i) \).

The 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, 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.

Proof. Suppose, contrary to fact, that there were a class $C_j$ that is not directly-affected by the update itself, is not derived from any class that is directly-affected by the update, but should be affected by the propagation of the update. Because none of the values of a modified object’s instance variables can change during the propagation of an update, the membership of $C_j$ must be membership-dependent upon some class $C_k$ whose set membership changed during the update propagation. In order for one class to be membership-dependent upon another, a derivation relationship must exist between them. Thus $C_j$ must be in $C_k$’s Derived-from-Sub-Graph. Similarly, $C_k$ in turn is derived from some other class $C_{k+1}$ whose set membership changed during the update propagation. By transitivity, $C_j$ must have been derived from some directly-affected class $C_i$. Contradiction. □

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 branches that can be discarded. Finally, there may be some overlap among the classes in the Derived-from-Sub-Graphs of directly-affected classes, and we must avoid self-cancelling update propagation.

Our update optimization process addresses these problems in four steps:

1. First, we identify the set of classes that are potentially directly-affected by the update.
2. Utilizing information contained in the generalization hierarchy, relevance registration, etc., we eliminate from this set those classes that are not directly-affected by the update.
3. 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-cancelling update propagation.

4.2 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 v.s. add-type) determines the nature of the classes that are potentially directly-affected by the update.

- Set-membership operators, e.g., add-type, 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.

4.2.1 Classes Directly-Affected by Set-Membership Updates

If the update is an operation upon the set-membership of a class (e.g., the addition or removal of a type), then the set of potentially directly-affected classes contains all classes directly derived from the target class, along with all subclasses 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 3, each class maintains a set of all the virtual classes that are directly derived from it.

Not all of these classes are actually directly-affected. However, our generalization hierarchy structure makes the subsumption relationships between classes readily apparent. We can use this information to identify and eliminate classes that are not directly-affected.

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. The schema is assumed to be maintained in main-memory for the sake of the efficiency of this algorithm.
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. $C_i$ is-a $C_j$ 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, we can prune the set efficiently by avoiding the evaluation of the updated object’s membership regarding any class whose super or subclasses has already been evaluated without change. For example, suppose that an object belongs to some class $C_i$. This means that the object must also belong to all superclasses of $C_i$ (i.e., all $C_j$ s.t. $C_i$ is-a $C_j$). If the object now gains the type of some subclass $C_k$ s.t. $C_k$ is-a $C_i$, then the object must belong or be added to all superclasses of $C_k$. Because object’s type memberships were consistent before the update, we know that if the object belongs to $C_i$ then we do not have to evaluate its membership in any class $C_j$ s.t. $C_i$ is-a $C_j$.

4.2.2 Instance-Variable Value Updates

In order to be able 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. We are aided to this end by our data model’s injunction that each property must reside at a single class that defines the property and acts as its point of inheritance. 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.

For example, Figure 3 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 eleven pounds and the value of its height instance variable is less than ten 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. Figure 3(b) 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 (Figure 3(c)), then we can readily identify exactly which simple selection classes are immediately affected by the update.

When a modification to an instance variable occurs, not all of the classes that have registered an interest in the modified property will be directly-affected by the update. Using the example in Figure 2, suppose that the weight of an instance of Cat were updated from 10 to 8 lbs. In that case, only the LightCat class is directly-affected by the update. Because all classes in our model are arranged in a generalization hierarchy, we can optimize our registration process to take advantage of subsumption relationships. We now propose the optimization of the registration process using the notion of distributed registration. When a class $C_i$ receives a registration for a class/property pair $<VC_j/p_k>$, it first searches its existing class/property registrations. If a previous registration for the given property $<VC_k/p_k>$ exists such that the registered class $VC_k$ either subsumes or is subsumed by the new class $VC_j$, the registrations are re-aranged so that the class that defines the property possesses the registration for the superior class, say $VC_k$ and that class possesses the registration for the subclass (e.g., $VC_j$).

The registration structures shown in Figure 3 can thus be distributed along the generalization hierarchy, as illustrated in Figure 2. Figure 2(a) depicts the registration of the SmallCat, LightCat, and the VeryLight 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. Figure 2(b) illustrates how with just simple registration, an update to a Cat’s weight from 12 to 14 lbs could be evaluated in terms of both the LightCat and VeryLightCat classes. If, however, distributed registration is used (as illustrated in Figure 2(c)), 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 of LightCat, the propagation would terminate and the VeryLightCat class would not be processed.

In summary, the principles of our registration strategy are:

- When a virtual class is created, its predicate is parsed and the class is registered with each property involved in the query specification. Each registration is sent to the class that serves as the point of inheritance for the property involved. Because all classes in our model are arranged in a generalization hierarchy, we optimize our registration process to take advantage of subsumption relationships. The registration structures can thus be distributed along the generalization hierarchy. Distributing the registration structures allows us to avoid evaluating the update in terms of classes that are obviously unaffected by the update (i.e., classes derived from classes to which the object did not belong either before or after the update).

- When an object is updated, the update method \(^8\) triggers a notification function that propagates the update to all virtual classes that have registered an interest in the value of the updated property.

- When a virtual class has been informed, via the notification class method, that an object in which it has a potential interest has been modified, it then evaluates the update. Should the update warrant that the object gain, keep, or lose the type of the class, then the class will send a message notifying those dependent upon it of the change. Otherwise it does not propagate the change across its derivation hierarchy and the update propagation terminates.

**Proposition 2** The distribution of registration over the generalization hierarchy ensures that we will identify exactly the set of simple select classes that are directly-affected by the modification of an instance variable.

---

\(^8\) For the sake of simplicity, we currently make the assumption that instance variables reside at the same location as their update methods.
Proof. Suppose, contrary to fact, that there were a simple select class that is directly-affected by the modification but not identified. Any simple select class that is directly-affected by the modification of a property must refer to that property within the query defining the class. In that case, the class would have registered its interest in the property at the time of its creation. The only way the class would not be notified of the modification would be if its registration were located with a superclass that was not affected by the modification and whose query the object does not fulfill. If the object does not possess the type of the superclass, it cannot possibly belong to the subclass. Similarly, if the object does not gain the type of the superclass during the update, then it cannot possibly gain the type of the subclass. In that case, the class is not directly-affected by the modification. Contradiction.

Similarly, suppose, contrary to fact, that there were a class that is incorrectly identified as being directly-affected by the modification. In order to be identified in this manner, the class must have registered its interest in the property. In order to be identified as directly-affected, the class must have registered interest in the modified property and the object’s membership in the class must have been evaluated as changed. In that case, the class is by definition directly-affected by the modification. Contradiction.

4.3 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 update propagation by propagating the update to the classes derived from the directly-affected classes. We optimize propagation of the update to the directly-affected classes and the classes derived from them using two techniques:

Identify branch termination conditions. Under certain conditions, propagation to branches of the derivation or generalization hierarchy can be terminated. For example, 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.

Eliminate self-cancelling update propagation. The propagation process could involve updates that cancel out each other’s effects. We call this the self-cancelling update propagation problem because it could lead to repetitive evaluations of the object in the context of a single class, sometimes with self-cancelling results.

Branch termination conditions. If the value of an instance variable of the object is modified, we do not need to propagate the update to any class that depends upon the value of that instance if that class is subsumed by another class for which the object does not qualify to belong with either its original or new instance variable values. For example, because the VeryLightCat class is subsumed by the LightCat class, if an instance of the Cat class shown in Figure 4 were to change the value of its weight instance variable from 12 to 10 lbs, then our update algorithm should determine that because the instance remains unqualified for membership in the LightCat class there is no need to evaluate the instance’s status in the VeryLightCat class.

Self-cancelling propagation. Figure 4(a) presents a schema vulnerable to the self-cancelling problem. Figure 4(b) 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-cancelling propagation problem could extend to a potentially huge set of classes to the detriment of efficiency.

In order to overcome these two problems, we propose the following solution. First, we assign a unique derivation number to each class in the database, indicating its global derivation depth in the schema. A class’s derivation number can be calculated once during its lifetime (at class creation time) and because we don’t replace existing derivation relationships with new ones, it will never change. All base classes have a derivation number of “0”. All derived classes are assigned the derivation number of the maximum of the derivation numbers of its source classes, plus one. Through transitivity we are guaranteed that if class \( C_k \in \text{Derived-from-Sub-Graph}(C_j) \), then \( \text{DerNum}(C_k) > \text{DerNum}(C_j) \). Using this derivation number framework, we 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-cancelling update propagations by ensuring that if \( \text{DerNum}(C_k) \)
Figure 4: An example schema where propagation order can lead to self-cancelling propagation.

< DerNum(C_i), then a modified object’s membership in C_k can be evaluated only before the object’s membership in C_i has been evaluated.

For example, Figure 4(c) shows how this strategy avoids the pitfall illustrated in Figure 4(b). 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.

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

**Proof.** Derivation-ordering ensures that a modified object will never be evaluated in terms of a virtual class C_k before it is evaluated in terms of one of C_k’s source classes and, vice-versa, that once a class C_k has been evaluated, none of its source classes will subsequently be evaluated or modified during the propagation process. The state of an object cannot change during the propagation process. Because the only reason an object would be evaluated regarding a given class more than once would be if the object’s qualification to be in the class change, the update will be evaluated in terms of any given class at most once. Furthermore, if neither of a class’s source classes are affected by the update, then the propagation process will terminate before evaluating that class. □

5 Performance Evaluation

5.1 Experimental Setting

**MultiView** supports the standard set of view definition operators (project, select, union, intersect, difference, and join) and a full complement of update operations on both base and virtual classes (creation, deletion, type gain, type loss, and modification of the value of an instance variable). Our optimization algorithms are tailored to improve the efficiency of update propagation in the presence of certain types of configurations of classes. For this reason, we chose to use both the traditional simple virtual classes and more complex combinations of virtual classes as representative models for our evaluative experiments. The parameters that our cost models use are summarized in Table 1.

For the sake of simplicity, we limit our experiments to the following basic models.

1. By varying the update probability (P), view predicate selectivity (f), and number of query predicate components (QC components(VC_i)) of an isolated selection view, we perform evaluations of the trade-off in basic costs/benefits between materialized and non-materialized views.

2. We combine multiple select classes in a single schema to study the effects of our registration strategy, varying the subsumption relationships of the select class predicates.
3. We use a varying number of virtual classes arranged in configurations subject to the *self-cancelling update propagation* problem to demonstrate the cost and benefit of our derivation ordering strategy.

<table>
<thead>
<tr>
<th>parameter</th>
<th>definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k$</td>
<td># update transactions.</td>
</tr>
<tr>
<td>$q$</td>
<td># query transactions.</td>
</tr>
<tr>
<td>$P$</td>
<td>Probability that a given operation is an update ($P = k / k + q$).</td>
</tr>
<tr>
<td>$f(Q)$</td>
<td>Selectivity of Query.</td>
</tr>
<tr>
<td>$\text{ExtentCost}(\text{class})$</td>
<td>Cost in ms to retrieve the extent of a base or materialized class; or calculate the extent of a non-materialized virtual class.</td>
</tr>
<tr>
<td>$\text{PerformCost}(\text{Update, obj})$</td>
<td>Cost in ms to perform the specified update upon the object.</td>
</tr>
<tr>
<td>$\text{size}(\text{extent}(C_i))$</td>
<td>Number of objects in class $C_i$'s extent.</td>
</tr>
<tr>
<td>$\text{AddCost}(\text{UntClass, obj})$</td>
<td>Cost in ms to add the virtual class's type to the object.</td>
</tr>
<tr>
<td>$\text{DropCost}(\text{UntClass, obj})$</td>
<td>Cost in ms to drop the virtual class's type from the object.</td>
</tr>
<tr>
<td>$\text{ScreenCost}(\text{Query, C_i})$</td>
<td>Cost in ms to screen one object from class $C_i$ for $\text{Query}$, including cost of retrieving the object from $C_i$'s extent.</td>
</tr>
<tr>
<td>$\text{Cost}(\text{Query, Class})$</td>
<td>Cost of applying $\text{Query}$ to the extent of class $\text{Class}$.</td>
</tr>
<tr>
<td>$\text{predComp}(\text{VC}_i)$</td>
<td># of predicate components in virtual class $\text{VC}_i$'s query.</td>
</tr>
<tr>
<td>$\text{extent}(\text{Class})$</td>
<td>Set of objects in specified class's extent.</td>
</tr>
<tr>
<td>$\text{N}$</td>
<td># virtual classes in schema.</td>
</tr>
<tr>
<td>$\text{NSV}$</td>
<td># selection virtual classes in schema.</td>
</tr>
<tr>
<td>$\text{NR}$</td>
<td># of virtual classes registered for the updated property.</td>
</tr>
<tr>
<td>$\text{NUR}$</td>
<td>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.</td>
</tr>
<tr>
<td>$\text{PAdd} (\text{VC}_i, O_j, U_k)$</td>
<td>Probability that the update $U_k$ causes the object $O_j$ to gain the type of virtual class $\text{VC}_i$.</td>
</tr>
<tr>
<td>$\text{PDrop} (\text{VC}_i, O_j, U_k)$</td>
<td>Probability that the update $U_k$ causes the object $O_j$ to lose the type of virtual class $\text{VC}_i$.</td>
</tr>
<tr>
<td>$\text{NSupC}(C_i)$</td>
<td># classes in superclass hierarchy of class $C_i$</td>
</tr>
<tr>
<td>$\text{NSubC}(C_i)$</td>
<td># classes in subclass hierarchy of class $C_i$</td>
</tr>
<tr>
<td>$\text{VC}_i$-Query</td>
<td>Query that defines virtual class $\text{VC}_i$.</td>
</tr>
</tbody>
</table>

Table 1: Model Parameters.

**Experimental environment.** We have successfully completed a prototype implementation of the MultiView model as described in this paper using the commercial GemStone OODB system. This includes the object-slicing representation, as well as the incremental update propagation algorithms as outlined in this paper. A description of the implementation is, however, beyond the scope of this paper. Instead, the reader is referred to our technical reports on this subject [17, 18, 22]. The experiments described below were carried out using MultiView on a Sun 4m running SunOS 4.1.3 with 32 megabytes of memory. We employ a 10,000 object implementation of the OOT benchmark schema for our tests [5]. Figure 5 shows the three class subschema of the OOT Benchmark’s example database that is central to our models.

For our analysis we consider each update operation to be the modification of a single object. Unless stated otherwise, we characterize a query upon a view as selection query on a single property. Each query thus requires an iteration through the view’s extent retrieving a property from each object. For non-materialized views, we assume that query optimization will be performed during the query substitution process.

**Materialized Query Costs.** 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 (independently from 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 $\text{VC}_i$’s extent, regardless of the number of classes from which it is derived, is thus:

---

*More specifically, we use GemStone version 4.0.1, using the Opal interface.*
\[
\text{Cost}(\text{Query, } VC_i) := \text{ScreenCost}(\text{Query, } VC_i) \times \text{size}(\text{extent}(VC_i)).
\]

**Non-Materialized Query Cost.** 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 non-materialized virtual class \( VC_i \) that is derived from a single source class \( SRC \) is roughly (discounting query optimization):

\[
\text{Cost}(\text{Query, } VC_i) := \text{ScreenCost}(VC_i, \text{Query, SRC}) \times \text{size}(\text{extent}(SRC)) + \text{ScreenCost}(\text{Query, } VC_i) \times f(VC_i, \text{Query}) \times \text{size}(\text{extent}(SRC))
\]

If the queried class is derived from other non-materialized 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. Discounting query optimization, the cost of computing the extent of a class \( VC_i \) derived from a source class \( SRC_i \) (possibly itself a non-materialized virtual class) can be expressed recursively:

\[
\text{ExtentCost}(VC_i) = \begin{cases} 
\text{ScreenCost}(VC_i, \text{Query, SRC_i}) \times \text{size}(\text{extent}(SRC_i)) & \text{if } SRC_i \text{ is base class} \\
\text{ExtentCost}(SRC_i) + \\
\text{ScreenCost}(VC_i, \text{Query, SRC_i}) \times \text{size}(\text{extent}(SRC_i)) & \text{otherwise}
\end{cases}
\]

**Materialized Update Costs.** When a view is materialized, then we must propagate all relevant updates to the view. 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. The cost of performing an update and propagating it to all affected materialized virtual classes is at a minimum (meaning with all possible optimizations):

\[
\text{Cost}(\text{Update, obj}) := \text{PerformCost}(\text{Update, obj}) + \sum_{i=1}^{N} (\text{PAdd}(C_i, obj, \text{Update}) \times \text{AddCost}(C_i, obj) + \text{PDrop}(C_i, obj, \text{Update}) \times \text{DropCost}(C_i, obj))
\]

Without optimization, the cost of performing an update and propagating it to all affected materialized views is in the worst-case:

\[
\text{Cost}(\text{Update, obj}) := \text{PerformCost}(\text{Update, obj}) + \sum_{i=1}^{N} (\text{ScreenCost}(VC_i, \text{Query}) + (\text{PAdd}(VC_i, obj, \text{Update}) \times \text{NSupC}(VC_i) \times \text{AddCost}(VC_i, obj)) + (\text{PDrop}(VC_i, obj, \text{Update}) \times \text{NSubC}(VC_i) \times \text{DropCost}(VC_i, obj)))
\]
unaffected by the update and that does not contain the updated object in its extent. With this optimization, the cost performing and propagating the update becomes:

\[
\text{Cost}(\text{Update, obj}) := \text{PerformCost}(\text{Update, obj}) + \sum_{i=1}^{N_R} N_C \cdot \left[ \text{PAdd}(VC_i, obj, \text{Update}) \times \text{Add}(VC_i, obj) \times N_{\text{SupC}}(VC_i) \right] + \left( \text{PDrop}(VC_i, obj, \text{Update}) \times \text{Drop}(VC_i, obj) \times N_{\text{SubC}}(VC_i) \right) + \text{ScreenCost}(VC_i, \text{Query}, SRC_i)
\]

(6)

**Non-Materialized 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

\[
\text{Cost}(\text{Update, obj}) := \text{PerformCost}(\text{Update, obj})
\]

(7)

### 5.2 Model 1–Trade-off between Materialization and Non-Materialization

Our first model concerns an isolated selection view `Select1` directly derived from the base class `AtomicPart`, as shown in Figure 6:

```
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 components in the view predicate), `k` (number of update operations), and `f` (view selectivity).

![Figure 6: The Select1 virtual class is derived from AtomicPart.](image)

**Varying view predicate components** (`predComps(Select1)`). The first experiment demonstrates the basic benefit of materialization. Figure 7 compares the cost of an aggregation query that computes the size of the extents of materialized and non-materialized `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 zero milliseconds), independent of the complexity of the query defining the class.

**Varying number of updates** (`k`). In order to test the effects of update probability upon materialization, we measured the time to perform `k` updates and 1 query where `0 \leq k \leq 10000`, on a schema with the isolated view `Select1` with fixed selectivity `f(query(Select1)) = 0.1`. The query and update we used are listed below.

---

10 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.
Figure 7: Retrieval costs for materialized and non-materialized virtual class extents.

- **Update1:** \((\text{AtomicPart extent}) \times \text{aRandomNumber} \) \times \text{aNumber}\) where \(\text{aRandomNumber}\) is a random number between 1 and 10,000, and \(\text{aNumber}\) 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 \(\text{Select1}\) class’s extent.

- **Query1:** \(\text{Select1 select: [:s1 | s1 \text{ id} < 1088092823]}\) For our database, this query results in the selection of approximately 500 objects from the 1,000 objects in the extent of \(\text{Select1}\) (the variance is because each update changes the extent of \(\text{Select1}\)). In order to minimize the effect of our object-slicing architecture, we are querying upon a property that is inherited by both that \(\text{AtomicPart}\) and \(\text{Select1}\) classes, namely \(\text{id}\) (as opposed to \(x\)) from Figure 8.

For our first test, we ran Update1 \(k\) times and Query1 once for values of \(k\) from 0 to 10000. Figure 8 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 \(f(\text{Select1}) := 0.1\), an update impact probability of 1, and a query selectivity of \(f(\text{Query1}) := 0.05\), the cost of performing a single query on the non-materialized view is counter-balanced by the cost of propagating approximately 3000 updates to the materialized view.

![Figure 8](image)

Figure 8: Milliseconds to run \(k\) updates and 1 retrieval on an inherited property (varying \(k\)).

**Varying view predicate selectivity \((f)\).** We next focus on the effect of view predicate selectivity by fixing update probability \(P\) at 90% and varying the view selectivity factor \(f\) from 1 to 100%. Our goal in this experiment was to evaluate the impact of view selectivity upon materialization performance.

Figure 9 shows the results of performing nine random updates and one query upon a simple selection view \(\text{Select2}\) derived from the \(\text{AtomicPart}\) 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 the query is fixed at 1% of the 10,000 objects in AtomicPart.

**Query2:** \(((Select2 \text{ extent}) \ \text{select:} \item \item. \item \item \item \item \item < 2081201856)\]

As we would expect from Equations 1 and 2, 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 non-materialized view. Regardless of the view selectivity factor, querying the non-materialized view involves one iteration upon the extent of the view’s source class. The dotted line in the graphs (indicating the time in milliseconds to perform the updates and query on a non-materialized class) reflects this fact by remaining fairly level. If the view is materialized, however, then the query involves only those objects that fulfill the view selection predicate. Thus, as \(f(Select2)\) increases, the proportion of the total non-materialized query cost that is derived from the cost of actually querying the virtual class (as opposed to computing its extent) increases. This is reflected by the rise of the solid line in the graphs (indicating the time in milliseconds to perform the updates and query on a materialized class).

![Graph showing the impact of view selectivity on query performance](image)

**Figure 9:** Milliseconds to run 9 updates (k=9) and 1 retrieval (q=1) on a property while varying view selectivity (f).

### 5.3 Model 2—Evaluating the Distributed Registration Service

We next focused our evaluation on the modify operator, which being the most complex can result in an object gaining and losing multiple types. For Model 2 we demonstrate 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.

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 Figure 10. Select80 is derived from the AtomicPart class. Select90 is derived 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 (meaning that the update does not 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 would expect that any update that does not cause the object to gain the type of the uppermost selection class to be immediately terminated if distributed registrations are employed. Otherwise (if the registrations are not distributed along the subsumption hierarchy), all classes must be evaluated in terms of the update.

Figure 11 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 9. 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.

Even under such a small sample schema, the benefit of distributing registrations is conspicuous. The dotted line in Figure 11 indicates the time in milliseconds needed to perform 10,000 updates when the registrations are not distributed. 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 non-distributed algorithm must check the updated instance against each of the virtual class’s predicates. The algorithm with distributed registrations only needs to check the updated instance against the topmost virtual class’s predicate.

5.4 Model 3—Benefit of Derivation Ordering

In Model 3, we examine the relative impact of registration and derivation level propagation on the self-cancelling update propagation scenario. In order to calculate the relative cost and benefit to algorithms compensating for this pitfall, we devised a test in which we measured the time to update 1000 objects of the AtomicPart in the presence of an increasing number of virtual classes arranged in a manner vulnerable to the self-cancelling 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 as described in Section 4 (derivation-ordered propagation and interest registration).

Figure 12 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-cancelling update propagation.

For this test, we implemented the updates so that they alternately incremented and decremented the value of the instance variables of the updated instances by one. The results of this test, displayed in Figure 13, indicate that in the presence of the self-cancelling update problem, an algorithm using only derivation ordering out-performs one using only registration. In summary, our tests indicate that although 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-cancelling update problem. Note that without derivation-ordering, update time increases exponentially with the number of self-cancelling classes present. Because additional selection classes are added in order to create each self-cancelling virtual classes, the algorithm that uses derivation-ordering and without registration suffers a moderate performance loss as the number of self-cancelling 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-cancelling classes.
6 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-cancelling effect.

We have implemented a prototype (version 1.0) of the MultiView system [18], which supports updatable materialized object-oriented views. Our system incorporates the incremental update propagation strategies introduced in this paper. We present 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-cancelling 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 isolate our algorithmic optimizations from the impact of the object-slicing representation. They thus 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 [15].

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.

References


A Basic Functions

For the purposes of the following algorithms, we employ a number of terms referring to the properties of classes in our system. We list below the functions used to access these properties.

- **DerivationNumber**($C_i$) returns the derivation number of the class $C_i$.
- **SourceClasses**($VC_i$) returns the source classes of the query that defines virtual class $VC_i$.
- **DerivedClasses**($C_i$) returns the set of classes derived from class $C_i$.
- **Extent**($C_i$) returns the set of objects belonging to class $C_i$’s extent.
- **DefiningClass**($property_y$) returns the class that serves as the single point of inheritance for property $property_y$.
- **Query**($VC_i$) returns the query that defines virtual class $VC_i$.
- **SelectRegistrations**($C_j$) returns the set of select registrations of class $C_j$ (a set of pairs $<p_k, C_i>$, where $p_k$ is a property and $C_i$ is a select class).
- **JoinRegistrations**($C_j$) returns the set of join registrations of class $C_j$ (a set of pairs $<p_k, C_i>$, where $p_k$ is a property and $C_i$ is a join class).
- **JoinProperty**($JoinQuery_i$) returns the property on which the natural join query $JoinQuery_i$ is formed.

We describe here the purpose, parameters, and results of functions used within the algorithms presented below.

- **Evaluate**($O_i, VC_i$) evaluates $O_i$’s state and type memberships and returns add, drop, or noChange, depending on whether or not $O_i$ qualifies to be a member of $VC_i$.
- **addToExtent**($O_i, C_j$) adds object $O_i$ to class $C_j$’s extent. In our object-slicing implementation, this entails the creation and appropriate linking of an implementation object of class $C_j$’s type.
- **DropFromExtent**($O_i, C_j$) removes object $O_i$ from class $C_j$’s extent. In our object-slicing implementation, this entails the deletion and appropriate unlinking of the implementation object of class $C_j$’s type.
- **AddRegistration**($RegistrationSet, aRegistration$) adds the given registration structure $aRegistration$ to the set of registration structures $RegistrationSet$, indexed by property name.
- **GetRegisteredClasses**($RegistrationSet, property_y$) returns the classes associated with the registration structures in the $RegistrationSet$ that have their $property$ value set to $property_y$.
- **addToDerivedClasses**($VC_i, C_j$) adds virtual class $VC_i$ to the set of classes associated as being derived from class $C_j$.
- **AddToArrayOfQueues**($anArrayOfQueues, anItem, n$) will add $anItem$ to the $n$th queue in $anArrayOfQueues$.
- **JoinObjectSet**($O_i$) returns the set of objects generated for join virtual classes that use object $O_i$ as a source object.

B Preparing the Schema for Update Propagation

Each time a new virtual class is created that will depend upon the value of instance variables (e.g., classes created using select or join queries), the schema must be prepared so that updates can be appropriately propagated to the new class. The various types of virtual classes require different information to be kept about them. The $SetUp(VC_i)$ function is called at the time of virtual class creation and prepares the schema according to the following principles.
• All virtual classes depend on the extent-membership of their source classes. For example, if an object gains the type of a source class, then it should also gain the type of any hide virtual classes formed from the class. For this reason, every class in the database schema has a derivedFrom class instance variable that is used to maintain references to the classes that are directly derived from the class. When a new virtual class is created, its name is added to the derivedFrom set of each of its direct source classes.

• Virtual classes formed from join or simple selection queries depend on the values of instance variables of individual objects. In order to maintain the information needed to propagate updates in a structured fashion, we define a Registration class. The instances of this class are used to record the registration of select and join classes with the classes that own attributes used in predicates. The class has two instance variables: className—the name of the class that is being registered, and property—the uniquely-identified property used in the predicate of the registered class. Every class in the database schema is associated with a set of these structures (possibly empty), indexed by property.

Function 1 SetUp(VCi)
begin Setup
For each class SRCj in SourceClasses(VCi)
    AddToDerivedClasses(VCi, SRCj).
    if (VCi is formed from a simple selection query)
        then
            for each property propertyk in properties(query(VCi))
                RegisterSelect(VCi, propertyk, DefiningClass(propertyk)).
            endfor.
    endif.
    if (VCi is formed from a join query)
        then
            RegisterJoin(VCi, joinProperty(query(VCi)), DefiningClass(propertyk)).
        endfor.
end Setup.

The macro functions used by the SetUp() function are described below. The RegisterSelect(VCi, propertyj, Classk) and RegisterJoin(VCi, propertyj, Classk) functions create registration structures for the new class and add them to appropriate registration sets of the classes on which they depend.

Function 2 RegisterSelect(VCi, propertyj, Classk)
begin RegisterSelect
    for each class Classj in GetRegisteredClasses(SelectRegistrations(Classk), propertyj)
        if subsumes(Classj, VCi)
            then
                RegisterSelect(VCi, propertyj, Classj).
            endif
    endfor.
    AddSelectRegistration(VCi, propertyj, Classk).
end RegisterSelect.

Function 3 AddSelectRegistration(VCi, propertyj, Classk))
begin AddSelectRegistration
    var tmpReg
    tmpReg := (Registration new).
tmpReg className: VC_i.
tmpReg property: property_j.
Add(SelectRegistrations(Class_k), tmpReg).
end AddSelectRegistration.

Function 4 RegisterJoin(VC_i, property_j, Class_k)
begin RegisterJoin
  for each class Class_j in GetRegisteredClasses(JoinRegistrations(Class_k), property_j)
    if subsumes(Class_j, VC_i)
      then
        RegisterJoin(VC_i, property_j, Class_j).
        exit.
      endif.
  endfor.
AddJoinRegistration(VC_i, property_j, Class_k).
end RegisterJoin.

Function 5 AddJoinRegistration(VC_i, property_j, Class_k)
begin AddJoinRegistration
  var tmpReg
  tmpReg := (Registration new).
  tmpReg className: VC_i,
  tmpReg property: property_j.
  Add(JoinRegistrations(Class_k), tmpReg).
end AddJoinRegistration.

C Identifying Directly-Affected Classes

Because the MultiView model assumes strict encapsulation, we embed our propagation method into each
updating method. When an update takes place in a prepared schema, the MultiView update propagation
method first identifies the classes directly affected by the update, then propagates the after-effects of the
update to the classes derived from the directly-affected classes. By directly-affected, we mean all classes
C_i \in \mathcal{C} \text{ s.t.} \text{ the modified object's membership in } C_i \text{ changes as a result of the update, and } \exists C_k \in \mathcal{C} \text{ s.t.} \text{ the modified object's membership in } C_k \text{ changes as a result of the update and } C_i \text{ is derived from } C_k.

The following functions identify the classes directly affected by updates. Once these classes have been
identified, we propagate the update to the classes derived from them in an efficient manner, as detailed in
Section D.

C.1 Identifying Classes Directly-Affected by a Type Change

When an object gains or loses a new type belonging to class C_j, it must also add or lose in an appropriate
manner the types of all classes related to C_j through either a derivation or subsumption relationship. Our
functions identify classes from the following three hierarchies that are directly-affected by the type change:

- The change must be propagated to all classes derived from C_j as indicated by the DerivedFrom
  relationship associated with C_j.
- If C_j’s type is added to an object O_i, then the change must also be propagated to all superclasses of
  C_j and all classes derived from them.
- If C_j’s type is dropped from an object O_i, then the change must also be propagated to all subclasses
  of C_j and all classes derived from them.

Given an object O_i \in \mathcal{O} and a class C_j \in \mathcal{C}, if O_i is currently not a member of C_j’s extent, but now is to be
added to C_j’s extent, then \textbf{IdentifyAffectedByGainType}(O_i, C_j) will identify all classes directly-affected
by the change, including C_j itself.
Function 6 IdentifyAffectedByGainType(Oi, Cj)

begin IdentifyAffectedByGainType
  For each Ck s.t. Cj is-a Ck
    if Oi \notin extent(Ck)
      then
        AddToExtent(Oi, Ck)
        for each VCi \in DerivedFrom(Ck)
          if (Evaluate(Oi, VCi) == (\text{add} or \text{drop}))
            then
              mark VCi as directly-affected
            endif.
        endfor.
      endif
  endfor.
end IdentifyAffectedByGainType.

Given an object Oi \in O and a class Cj \in C, if Oi is currently a member of Cj’s extent, but is to be dropped from Cj’s extent, then IdentifyAffectedByDropType(Oi, Cj) will identify all classes directly-affected by the change, including Cj itself.

Function 7 IdentifyAffectedByDropType(Oi, Cj)

begin IdentifyAffectedByDropType
  For each Ck s.t. Ck is-a Cj
    if Oi \in extent(Ck)
      then
        DropFromExtent(Oi, Ck)
        for each VCi \in DerivedFrom(Ck)
          if (Evaluate(Oi, VCi) == (\text{add} or \text{drop}))
            then
              mark VCi as directly-affected
            endif.
        endfor.
      endif
  endfor.
end IdentifyAffectedByDropType.

C.2 Identifying Classes Directly-Affected by Creation or Deletion

When a new object-instance of a class C_i \in C is created for the first time, it is equivalent to creating a new empty conceptual object, then adding the type of C_i. Given a global database schema G = (C, E), the function create(C_i) for C_i \in C, will create a conceptual object, O_{i_{\text{new}}} for the new object O_i, then perform the following tasks:

Function 8 IdentifyAffectedByCreate(Oi, Cj)

begin IdentifyAffectedByCreate
  For each Ck s.t. Cj is-a Ck
    mark Ck as directly-affected
  endfor.
end IdentifyAffectedByCreate

Our algorithm for the Delete operation is straightforward. When an object is deleted, it completely ceases to exist in the context of the database. Thus we can simply delete the object’s conceptual object and all of
its implementation objects. Since a conceptual object keeps track of all occurrences of the object in both base and virtual classes, it is not necessary to search through the derivation hierarchy \(^{11}\). There will now be no references to the original object, so it is now deleted.

**Function 9 Delete**(\(O_i\))

\begin{verbatim}
begin Delete
  For each implementation object \(O_{impl_{C_i}} \in O_{conc}\)
    DropFromExtent(\(O_i, C_i\))
  endfor.
  Create a new object \(O_{deleted}\) of type DeletedObject.
  Swap the object-identifiers of \(O_{conc}\) and \(O_{deleted}\).
end Delete.
\end{verbatim}

The objects generated for join classes are associated with objects from the source classes of the join query. When an object is deleted, all objects that are associated with it via join queries must also be deleted. The function **DeleteJoinObjects**(\(O_i\)) will recursively delete all join virtual class instances that are associated with \(O_i\).

**Function 10 DeleteJoinObjects**(\(O_i\))

\begin{verbatim}
begin DeleteJoinObjects
  For each object \(O_j \in JoinObjectSet(O_i)\)
    DeleteJoinObjects(O_j).
    Delete(O_j).
  endfor.
end DeleteJoinObjects.
\end{verbatim}

**C.3 Identifying Classes Directly-Affected by a State Modification**

**Function 11** The function **TraverseRegistration**(\(O_i, \text{property}_j, \text{value}_{new}, C_k, \text{type}\)) calculates and returns the set of classes that have registered an interest in the modified property and potentially might be affected by the change. The \(\text{type}\) parameter defines the type of registrations to be traversed (i.e., Select or Join).

\begin{verbatim}
begin TraverseRegistration
  Initialize a set, Result, to hold the set of classes that have registered an interest in the modified property and potentially might be affected by the change.
  Evaluate \(O_i\)’s status in \(C_k\) in terms of both its current and new state.
  if (\(\text{type} == \text{Select}\))
    then
      If \(O_i\)’s status in \(C_k\) does not change and \(O_i\) is a member of \(C_k\)
        then
          for each \(C_i \in \text{GetRegisteredClasses(SelectRegistrations(C_k),property}_j)\)
            Result := Result \(\cup\) TraverseRegistration(\(O_i, \text{property}_j, \text{value}_{new}, C_i, \text{type}\)).
          endfor.
        endif.
    else if (\(\text{type} == \text{Join}\))
      then
        If \(O_i\)’s status in \(C_k\) does not change and \(O_i\) is a member of \(C_k\)
          then
            for each \(C_i \in \text{GetRegisteredClasses(JoinRegistrations(C_k),property}_j)\)
              Result := Result \(\cup\) TraverseRegistration(\(O_i, \text{property}_j, \text{value}_{new}, C_i, \text{type}\)).
          endfor.
        endif.
  endif.
\end{verbatim}

\(^{11}\)Since GemStone provides persistence through reference, any time an object in our database is “deleted,” we create a new object of a special \(\text{Deleted}\) type and swap identifiers.
end if.
If O_i's status in C_k changes and (type = Join)
then
Result := Result \cup \{C_k\}.
For each C_i \in GetRegisteredClasses(JoinRegistrations(C_k, property_j))
Result := Result \cup TraverseRegistration(O_i, property_j, value_{new}, C_i, type).
end for.
endif.
If O_i's status in C_k changes and (type = Select)
then
Result := Result \cup \{C_k\}.
For each C_i \in GetRegisteredClasses(SelectRegistrations(C_k), property_j)
Result := Result \cup TraverseRegistration(O_i, property_j, value_{new}, C_i, type).
end for.
endif.
Return Result.
end TraverseRegistration.

Because our model assumes complete encapsulation, when an update is made to the value of an object's instance variable property_j, the update must be made using the accessing methods for that property. Given an object O_i \in O, a property property_j, and a new value value_{new} for O_i's property_j instance, the function IdentifyAffectedByStateModification(O_i, property_j, value_{new}) will identify all classes directly-affected by the change, including C_j itself.

Function 12 IdentifyAffectedByStateModification(O_i, property_j, value_{new})
begin IdentifyAffectedByStateModification
Let C_j be the class that defines property_j.
Initialize the set RegisteredClasses := TraverseRegistration(O_i, property_j, value_{new}, C_j, Select) \cup TraverseRegistration(O_i, property_j, value_{new}, C_j, Select)
For each class RC_k \in RegisteredClasses
mark RC_k as directly-affected.
end for.
end IdentifyAffectedByStateModification

D Propagating Updates to Derived Classes

The function SetUpDirectlyAffectedClasses(O_i, updateType) prepares the QueueArray to propagate an update of type updateType to all classes marked as directly-affected and the classes derived from them. Propagation is performed in order of derivation number.

Function 13 SetUpDirectlyAffectedClasses(O_i, updateType)
begin SetUpDirectlyAffectedClasses
For each class C_k marked as Directly-Affected
Unmark class C_k as Directly-Affected
if Evaluate(O_i, C_k) == noChange and O_i \notin extent(C_k)
then
break for each
endif.
if updateType == AddType and Evaluate(O_i, C_k) == add
then
AddType(O_i, C_k)
else if updateType == DropType and Evaluate(O_i, C_k) == drop
then
DropType(O_i, C_k)
end for each.
end for each.
endif.
endif.
For each class $C_i \in \text{derivedFrom}(C_h)$
\text{AddToArrayOfQueues}(\text{anArrayOfQueues}, C_i, \text{derivationNumber}(C_i))
endfor.
endfor.
end \text{SetUpDirectlyAffectedClasses}.

The function \textbf{PropagateToDerivedClasses} uses the temporary data structure of an array of queues, \text{anArrayOfQueues}, to propagate the update to classes in order of derivation number.

\textbf{Function 14 PropagateToDerivedClasses}

\begin{verbatim}
begin PropagateToDerivedClasses
While anArrayOfQueues is not empty
do
  Remove a class $C_h$ from the first non-empty queue in anArrayOfQueues.
  if ((evaluate($O_i, C_h$) == correct) and ($O_i \not\in \text{extent}(C_h)$))
  then
    break while
  endif.
  if updateType == AddType
  then
    AddType($O_i, C_h$)
  else if updateType == DropType
  then
    DropType($O_i, C_h$)
  endif.
  endif.
For each class $C_i \in \text{derivedFrom}(C_h)$
  AddToArrayOfQueues(\text{anArrayOfQueues}, C_i, \text{derivationNumber}(C_i))
endfor.
endwhile.
end PropagateToDerivedClasses.
\end{verbatim}