Experimental Evaluation of the SMX Strategy for Incremental Materialized Path View Maintenance *

Elke A. Rundensteiner
Department of Computer Science, Worcester Polytechnic Institute
Worcester, MA 01609-2280, USA
rundenst@cs.wpi.edu

Harumi A. Kuno
Hewlett-Packard Laboratories, Hewlett-Packard Company
Palo Alto, CA 94304-1126, USA
harumi.kuno@hp.com

Yijing Zhou
Microsoft Corporation
Redmond, WA 98052-6399, USA
yijingz@microsoft.com

Abstract

In this paper we address the problem of incrementally maintaining object-oriented database (OODB) views formed using path queries. We demonstrate that traditional index organizations, designed for query support, are not well suited for this task. As a solution, we introduce a new *Satisfiability Indicating Multi-Index (SMX)* organization, which maintains partial information indicating whether or not a given endpoint satisfies the query predicate rather than what the exact value of the endpoint is. We have implemented SMX as well as competing index structures in the MultiView OODB system over Gemstone, and conducted experiments on this system. The results of our evaluations indicate that the SMX dramatically improves upon the performance of traditional indices.

Keywords: Incremental view maintenance, path queries, data warehousing, view materialization, and object-oriented databases.

*This work was supported in part by the NSF RIA grant #IRI-9309076, the NSF NYI grant #IRI 97–96264, the NSF CISE Instrumentation grant #IRIS 97–29878, and the NSF grant #IIS 97–32897. Dr. Rundensteiner would like to thank industrial sponsors, in particular, IBM for the IBM partnership award and for the IBM corporate fellowship for one of her graduate students. Harumi A. Kuno is also grateful for support from the NASA Graduate Student Researchers Program during her Ph.d. studies at University of Michigan.

1 Introduction

Recent advances in information technology have issued a new set of challenges to the database community. There is a growing need for strategies that provide the means to cache and use query results, for mechanisms that support customized interfaces to shared data, and for the integration of such mechanisms with the powerful constructs of the object-oriented programming model. For example, the need for improved access to diverse data sources has spurred a recent interest in supporting queries across multiple information sources in a transparent fashion (e.g., *data warehouses* and *digital libraries* [24, 1]). Materialized database views are a recognized means of achieving such interoperability among applications, allowing applications to benefit from the powerful flexibility of view technology while minimizing the performance penalties traditionally associated with views. However, the fact that updates must be propagated to affected materialized views limits the variety of queries that can be used to define materialized views.

We have previously discussed the problem of view materialization in the context of object-oriented databases (OODBs) and proposed algorithms that exploit object-oriented characteristics in order to provide the incremental maintenance of materialized virtual classes created using the standard view query operators [12, 15, 16, 17]. In this paper we address the problem of the incremental maintenance of materialized virtual classes formed using selection queries on aggregation paths (or short, *path query views*). To the best of our knowledge, only two other research groups have addressed this topic. Kemper et al.'s work on *function materialization* addresses the problem of precomputing function results [7, 8]. Konomi et al. discuss a solution to supporting a type of join class that is formed along the aggregation graph [9].

Traditional indexing techniques are not well suited for this task of maintenance. The indexing needs of the path query view problem are unique in that because the contents of the materialized view are cached and can be queried directly, the primary use for a supplemental index is for the propagation of updates rather than for query processing. Because traditional index organizations are tailored for use during general query processing (i.e., primarily for data retrieval), they are not optimized to evaluate path instantiations with regard to a static predetermined predicate condition such as would be associated with a path query view. Furthermore, traditional index organizations do not distinguish between single-valued and multi-valued attributes, and thus do not account for the fact that multi-valued attributes enable a single object at the head of a path to be associated with multiple instantiations of the path, any number of which could satisfy the specific path query predicate. This means that if an updated path involves a multi-valued attribute then the aggregation hierarchy of the object at the head of the path must be completely re-calculated in order to determine whether or not that object participates in an alternative instantiation that fulfills the view query predicate despite the update.

As a solution, we introduced a new *Satisfiability Indicating Multi-Index (SMX)* organization that is specifically tailored to handle the issues of path query view maintenance [14, 20]. SMX maintains partial information indicating whether or not a given endpoint satisfies the query predicate rather than the exact values of endpoints. This strategy offers a number of benefits. (1) Instead of traversing all instantiations in which an object participates to their endpoints, with the SMX organization at most two path positions forward must be examined in order to determine whether or not the endpoint of an instantiation fulfills a given path query predicate. (2) The SMX index structure only needs to be updated when the validity of an object's instantiation (in terms of the query predicate) changes. (3) Instead of having to fully traverse all instantiations of an object to identify whether or not it participates in any alternative instantiations (due to multi-valued attributes) that affect its membership in a path query view, with the SMX organization we only need to check at most one forward reference.

We have implemented SMX and competing index structures in our object-oriented view system *MultiView* , which is built on top of the commercial Gemstone OODB [10, 19, 12, 17, 5]. In this paper, we report upon experimental studies we have conducted evaluating the performance of SMX and other traditional index structures both in terms of materialized view generation and continued maintenance under updates. This paper extends the work introduced in [20], by providing a more complete and formal treatment of the SMX solution, including Lemmas; by discussing the experimental setup and implementation in more detail; and by reporting the results of additional experiments that give further insight into the advantages and disadvantages of the SMX solution. Our results indicate that the SMX dramatically improves upon the performance of traditional index structures.

In Section 2, we review the *MultiView* object model. Section 3 presents three issues with the maintenance of path query views, including a discussion of the limitations of utilizing traditional index organizations to address these problems. We describe SMX in Section 4. Sections 5 and 6 present our experimental setup and performance results. We discuss related work in Section 7, and present conclusions in Section 8.

2 The *MultiView* Model and System

In this section, we briefly review the basic object model principles of the *MultiView* system. More details are given in [18] and [12]. Let O be an infinite set of **object instances**, or short, **objects**. Each object $O_i \in O$ consists of state (**instance variables** or attributes), behavior (**methods** to which the object can respond), and a **unique object identifier**. The domain of an instance variable can be **constrained** to objects or sets of objects of a specific class. If an instance variable is constrained to sets of objects, then we say that the instance variable is **multi-valued**. Because our model is object-oriented and assumes full encapsulation, access to the state of an object is only through **accessing methods**. 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 **classes**. We use the term **type** to indicate the set of applicable property functions shared by all members 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 **extent**(C_i).

We use both class type and class extent to determine subsumption relationships. For two classes C_i and $C_j \in C$, C_i is a **subtype** of C_j, denoted $C_i \preceq C_j$ if and only if (iff) (**properties**(C_i) $\supseteq$ **properties**(C_j)). All properties defined for a supertype are **inherited** by its subtypes. Similarly, C_i is a **subset** of C_j, denoted $C_i \subseteq C_j$, iff $(\forall o \in O)((o \in C_i) \Rightarrow (o \in C_j))$. C_i is a **subclass** of C_j, denoted C_i *is-a* C_j, iff $(C_i \subseteq C_j)$ and $(C_i \preceq C_j)$. C_i is a **direct subclass** of C_j if $\nexists C_k \in C$ s.t. $k \neq i \neq j$, C_i *is-a* of C_k, and C_k *is-a* of C_j.

An **object schema** is a rooted directed acyclic graph $G = (V, E)$, where the finite set of vertices V corresponds to classes $C_i \in C$ and the finite set of directed edges E corresponds to a binary relation on $V \times V$ representing 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 the set containing all instances and an empty type description.

An **aggregation path** P_i is defined as $C_{i,1}.A_{i,2}.A_{i,3} \ldots A_{i,n}$ where $C_{i,1}$ is the path's source class, $A_{i,2}$ is an instance variable of $C_{i,1}$, and $\forall A_{i,k}, 1 < k \leq n, A_{i,k}$ is an instance variable of the class to which instance variable $A_{i,k-1}$'s values are constrained. We use the term **instantiation** of path P_i to refer to a sequence of objects $O_1, O_2, \ldots, O_n$ s.t. O_1 belongs to class $C_{i,1}$, O_2 belongs to class $C_{i,2}$, etc., and $\forall k$ s.t. $1 < k \leq n$, the value of O_{k-1}'s $A_{i,k}$ instance variable refers to O_k. We identify an **instantiation** of subpath $P_i(j,k), j \leq k$ of path P_i as a sequence of objects $O_j, O_{j+1}, \ldots, O_k$ s.t. O_j belongs to class $C_{i,j}$, O_{j+1} belongs to class $C_{i,j+1}$, etc., and $\forall l$ s.t. $j < l \leq k, O_{l-1}$'s $A_{i,l}$ instance variable refers to O_l. Given an instantiation of a path $P_i = C_{i,1}.A_{i,2}.A_{i,3} \ldots A_{i,n}$, we call the object in the 1^{st} position (e.g., the object from class $C_{i,1}$) the **head** and the object in the n^{th} position (e.g., the object from class $C_{i,n}$) the **endpoint** of the instantiated path. An object $O_j, 1 \leq j \leq n$, can participate in multiple **instantiations** of a path P_i.

Virtual classes are defined by the application of a query operator to one or two classes that restructures the source classes' type and/or extent membership. *MultiView* provides a virtual-class-forming algebra that includes the following operators: difference, hide, intersect, join, refine, select, and union [18, 11]. These queries determine the methods, instance variables, and extent of the virtual classes. The join operator can be *object-generating*; all other operators are *object-preserving*.

Let Q be the set of all possible queries. We constrain a **query** $Q_i \in Q$ used to define a virtual class to correspond to a single algebra operation, and refer to the query $Q_i \in Q$ that defines a virtual class, $VC_i \in VC$, as **query**(VC_i). We identify three types of predicates used in virtual-class defining queries. **Class membership predicates** (intrinsic to hide, union, intersect, refine, and difference queries) are predicate terms that depend upon the classes to which an object belongs [1]. **Value predicates**, used by select and join queries, are predicate terms constraining instances based on the values of their local instance variables. In addition, our select operator supports the formation of virtual classes using **path queries** (queries that refer

[1] Set operations are typical 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.

to a value along an object's aggregation path). A **path query**, which consists of a path and a value predicate upon the endpoint of that path, takes the form $PQ_i = C_{i,1}.[\Theta]A_{i,2}.[\Theta]A_{i,3}\ldots[\Theta]A_{i,n}\theta value$, where if attribute $A_{i,j}$ is a multi-valued attribute then the quantifier $\Theta \in \{\exists, \forall\}$ indicates whether the multi-valued attribute should be handled in an existential or universal manner, and the comparison operator θ is defined for $C_{i,n}$ [2].

Given an instantiation of a subpath $P_i(j,n)$ $O_j, O_{j+1}, \ldots, O_n$ s.t. O_j belongs to class $C_{i,j}$, O_{j+1} belongs to class $C_{i,j+1}$, etc., and $\forall l$ s.t. $j < l \leq k, O_{l-1}$'s $A_{i,l}$ instance variable refers to O_l and $A_{i,j+1}$ is a single-valued attribute, if O_n satisfies the predicate $O_n\theta value$ then we say that the subpath instantiation is a **satisfying subpath instantiation**, or short, that it is **satisfying**. If $A_{i,j+1}$ is a multi-valued attribute and $\Theta = \forall$, then we say that the subpath instantiation is **satisfying** if and only if all the objects that serve as object $O_{i,j}$'s values for attribute $A_{i,j+1}$ participate in **satisfying subpath instantiations** of subpath $P_i(j+1,n)$. The extent of a virtual class that is defined by a path query PQ_i contains all objects $O_1 \in C_{i,1}$ that satisfy PQ_i. In the current paper we assume that virtual classes are materialized, and refer to a single virtual class that is defined using a path query as a **path query view** (PQV).

3 Path Query View Maintenance

3.1 Example Schema

Figure 1 shows an initial aggregation schema composed of four classes. *Person* has an attribute, `age`, which is constrained to the *Number* class. *Person* also has a multi-valued instance variable—`cars`, which associates each person with the set of cars they own. *Car* has one instance variable, `maker`, which is constrained to the class *Company*. *Company* has two instance variables: `stockPrice`, which is constrained to the *Number* class; and `owner`, which is constrained to the *Person* class.

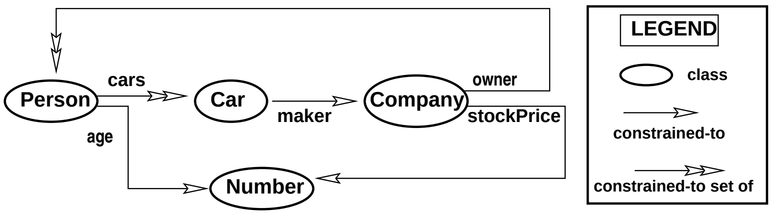

Figure 1: The example schema.

Figure 2 shows instances of the base classes and the aggregation relationships between them. For example, the *person1* object is an instance of the *Person* class,

[2]Typically, $\theta \in \{=, <, \neq, \leq, >, \geq\}$.

car1 is an instance of the *Car* class, and *company1* is an instance of the *Company* class. The *person1* object has both *car1* and *car2* as values for her `cars` attribute.

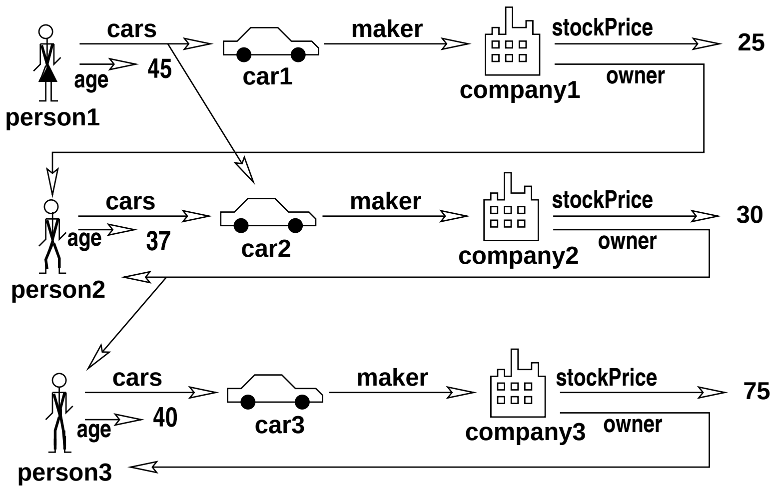

Figure 2: Initial base instances and aggregation relationships for Figure 1.

Suppose that we were to define a virtual class *PathSelect1* for the schema in Figure 1 using the (existential) path query `select from Person where [:person | person.∃cars.maker.stockPrice < 40]`. Those instances of the *Person* class whose `cars` instance variable includes an instance of *Car* that has a `maker` value whose `stockPrice` instance variable has a value less than 40 qualify to belong to the *PathSelect1* class. As shown in Figure 3, the initial extent membership of the *PathSelect1* class contains the *person1* and *person2* objects. Figure 4 depicts the index structures created for our example schema under the traditional multi-index (MX), path index (PX), and nested index (NX) organizations.

3.2 Path Query View Maintenance Issues

We now outline issues that must be addressed when maintaining the extents of materialized path query views.

Determining instantiation validity. First we must determine whether or not the original and new endpoints of the path instantiations in which the updated object participates fulfill the path query. The traditional index organizations shown in Figure 4 facilitate the identification of head objects of instantiations, but not their endpoints. For all three index structures, we must thus traverse each of the updated object's path instantiations to their original and new endpoints in order to determine if the update affects whether or not that instantiation satisfies the path query predicate.

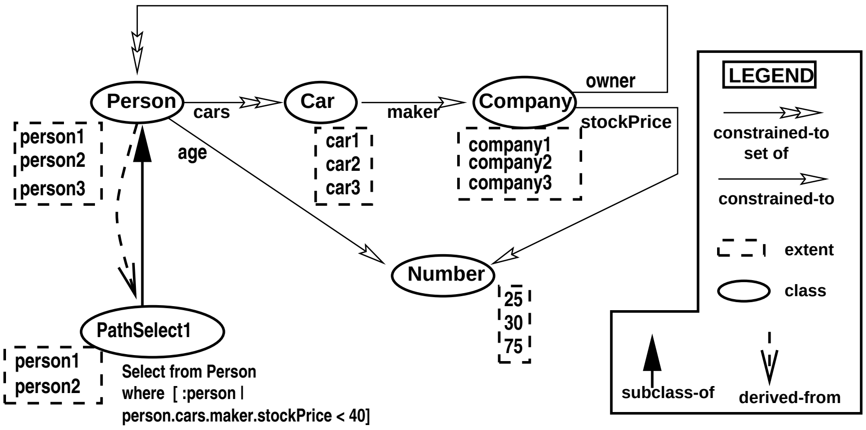

Figure 3: The *PathSelect1* is derived from the base schema using a path query.

NESTED INDEX

KEY	VALUE
25	{person1}
30	{person1, person2}
75	{person3}

PATH INDEX

KEY	VALUE
25	{person1.car1.company1}
30	{person1.car2.company2, person2.car2.company2}
75	{person3.car3.company3}

MULTI-INDEX STRUCTURES

KEY (Car)	VALUE (Person)
car1	{person1}
car2	{person1, person2}
car3	{person3}

Index on Person.cars

KEY (Company)	VALUE (Car)
company1	{car1}
company2	{car2}
company3	{car3}

Index on Car.maker

KEY (Person)	VALUE (Company)
25	{company1}
30	{company1}
75	{company2}

Index on Company.stockPrice

Figure 4: Traditional index organizations for the path `person.cars.maker.stockPrice`.

For example, if the *car2* object from Figure 2 were to change the value of its `maker` instance variable to refer to *company3* instead of *company2*, then (with the PX, NX, and MX organizations) we would have to traverse both the *company2* and the *company3* objects' instantiations of the *PathSelect1(2,4)* subpath and determine that while the original endpoint of *car2*'s path instantiation was '30' (which does fulfill *PathSelect1*'s query predicate), the new endpoint is '75' (which does not fulfill *PathSelect1*'s query predicate).

Finding head objects. If the original and new endpoints of a given path instantiation differ in that one satisfies the query predicate but the other does not, then we must identify the head object of the path instantiation. The membership of this head object in path query views based on the path instantiation is potentially affected by the update.

Given a path $P_i = C_{i,1}.A_{i,2}.A_{i,3} \dots A_{i,n}$ and a modified object O_j of class $C_{i,j}$, we use the PX and NX organizations by first finding the endpoint objects of instantiations involving the updated object by traversing the object's instantiations of the $P_i(j,n)$ subpath (as already done in step 1), then looking these endpoints up in the index structure. With the PX organization, we can then scan the paths retrieved from the PX, determine which involve O_j, and thus identify the head objects of the instantiations in which O_j participates. However, because the NX structure does not include any path information, if multiple heads are associated with a given endpoint then we must traverse these heads' instantiations of the $P_i(1,j)$ subpath forward in order to identify which instantiations (and thus head objects) involve object O_j. We can use the reverse references provided by the MX organization to avoid such forward traversals and instead identify the head objects of O_j's instantiations of the $P_i(i,j)$ subpath by performing lookups in the $j-1$ indices of the $C_{i,1}$ through $C_{i,j+1}$ classes.

For example, if the *car2* object from Figure 2 were to change the value of its `maker` instance variable to refer to *company3* instead of *company2*, then with the PX and NX organizations we would first traverse the *company2* object's instantiation of the $P_i(3,5)$ subpath and find that the original endpoint is *30*. We would then look up *30* as a key in the index. Because multiple values are associated with that key, under the PX organization we would examine each path to identify the one in which the *car2* object participates and thus determine that both the *person1* and *person2* objects are potentially affected by this update. However, with the NX organization, we must traverse all the instantiations of each object associated with the key of '30' in order to determine which objects' aggregation hierarchies involve the *car2* object. With the MX organization, we traverse backwards through all multi-index structures (e.g., we could look up the *car2* object in the *Person.cars* index and identify that both the *person1* and *person2* objects are potentially-affected head objects).

Identifying alternative instantiations. The presence of even one multi-valued attribute in the path of the query greatly increases the cost of evaluating the effects of updates using any traditional index structure. If an update changes whether or not an instantiation fulfills a path query's predicate and the path includes at least one multi-valued attribute, then we must determine if each involved head object participates in any alternative instantiations that cause the head object membership in the path query view to remain the same despite the update. With the PX, NX, and MX organizations, this means that in addition to the cost of identifying each head object, we must also traverse all of that head object's path instantiations completely forward in order to determine if it participates in any alternative *satisfying instantiations*.

For example, if the *car2* object were to update its `maker` instance variable to remove the reference to *company3* instead of *company2*, then although we could use traditional index structures to identify that the *person1* object is potentially affected by the update, we would have to traverse all of the path instantiations of the *person1* class, in particular *person1.car1.company1.25*, in order to determine that *person1* still satisfies *PathSelect1*'s query predicate and thus should not be removed from the extent of the *PathSelect1* class.

4 SMX Solution

4.1 The SMX Structure

From the above discussion, we can identify three characteristics of the path query view problem. First, we use the supplemental index structures at the time of *updates* instead of queries. Second, because our focus is on updates rather than queries, we do not need to know exact endpoints of updated instantiated subpaths—we really need only to know whether or not these endpoints satisfy the path query view predicate. Finally, we need to be able to determine whether or not, due to multi-valued attributes, head objects participate in alternative path instantiations that affect the impact of the update on a head object's membership in the path query view.

Our *Satisfiability-Indicating Multi-Index* (SMX) exploits these characteristics to maintain path query views in an efficient and incremental fashion. The fundamental principle of the SMX solution is that we do not need to know the exact endpoint of path instantiations—we only need to know whether or not that endpoint satisfies the query's predicate. SMX therefore extends each multi-index (MX) entry with a *satisfiability indicator* (or $SatInd$ for short) that indicates whether the key value object participates in any instantiations with an endpoint that fulfills the query predicate. Because the satisfiability of a path instantiation is determined by its endpoint object, we can consider satisfiability to be a transitive property in that if we know for all objects $O_j \in$ class $C_{i,j}$ whether or not O_j leads to endpoints that satisfy the path query predicate, then we also know for any object $O_{j-1} \in C_{i,j-1}$ that refers to O_j as the value for its $A_{i,j}$ attribute whether or not O_{j-1} leads to endpoints that satisfy the path query predicate.

Lemma 1 *Given any object $O_{j-1} \in C_{i,j-1}$ that refers to members of class $C_{i,j}$ as the value for its $A_{i,j}$ attribute, O_{j-1} leads to endpoints that satisfy the path query predicate if and only if those particular members of class $C_{i,j}$ lead to endpoints that satisfy a path query predicate.*

By Lemma 1, we can initialize the SMX index recursively. First, for each endpoint object O_n of class $C_{i,n}$ that serves as a value for the $A_{i,n}$ attribute of at least one object O_{n-1} of class $C_{i,n-1}$ (i.e., the last component of the path), we store $SatInd(O_n)$, which indicates whether or not O_n satisfies the path query predicate, i.e., $O_n \theta value$ evaluates to true. Next, for each object O_j of class $C_{i,j}$ that serves as a value for the $A_{i,j}$ attribute of at least one object O_{j-1} of class $C_{i,j-1}$ (i.e., the $C_{i,j-1}.C_{i,j}$ component of the path), we store $SatInd(O_j)$, which indicates either (1) (if the predicate expression's quantifier for the $C_{i,j-1}.C_{i,j}$ component is *existential*) whether or not $\exists O_{j+1}$ s.t. O_j's $A_{i,j+1}$ attribute value is set to O_{j+1} and $SatInd(O_{j+1})$ is true; or (2) (if the component's quantifier is *universal*) whether or not $\forall O_{j+1}$ s.t. O_j's $A_{i,j+1}$ attribute value contains O_{j+1}, $SatInd(O_{j+1})$ is true.

When an update takes place, we can use the SMX index to look up whether or not an object O_j leads to an endpoint that satisfies the path query predicate (i.e., $SatInd(O_j)$). If a value of true is associated with O_j in the index, then O_j leads to an endpoint that satisfies the path query predicate; otherwise (if a value of false is associated with O_j's record) it does not. If O_j does not already have an index record associated with it because no object previously referred to it, then we must look up the O_{j+1} objects referred to by O_j as values for attribute $A_{i,j+1}$ in the $C_{i,j}.A_{i,j+1}$ index. Because of the entries corresponding to the $O_j.O_{j+1}$ relationship, we are guaranteed to find these objects in the index, and thus no other traversal is needed. This concept is summarized in the following lemma.

Lemma 2 *If the* satisfiability indicator *value of an updated object $O_j \in C_{i,j}$ does not change as the result of an update, then the* satisfiability indicator *value of any object $O_{j-1} \in C_{i,j-1}$ that refers to members of class $C_{i,j}$ as the value for its $A_{i,j}$ attribute will not change as a result of the update.*

For example, Figure 5 shows the *satisfiability indicators* for the objects from Figure 2. The initial path instantiations of the *person1* and *person2* objects fulfill the *PathSelect1* class's predicate of `person.cars.maker.owners.stockPrice` $<$ `40` (introduced in Section 3). Now suppose that the *car2* object were updated as shown in Figure 5, changing the value of its `maker` instance variable to refer to *company1* instead of *company2*. We can compare the satisfiability indicators associated with the *company2* and *company1* objects, thus determining that because both lead to satisfying endpoints, the update will not affect the satisfiability of *car2*'s path instantiations and nothing needs to be done. Figure 6 shows the SMX index structures that correspond to the objects and classes shown in Figure 5.

4.2 Incremental Processing Strategy

We can maintain the SMX structures efficiently under all three update operations (create, delete, and modify). Suppose that there exists a virtual class PQV_i defined by the query $PQ_i = C_{i,1}.A_{i,2}.A_{i,3} \ldots A_{i,n} \theta value$, and that some object O_j belonging to class $C_{i,j}$, $1 \leq j \leq n$ is updated. The *satisfiability indicators* can be maintained in an

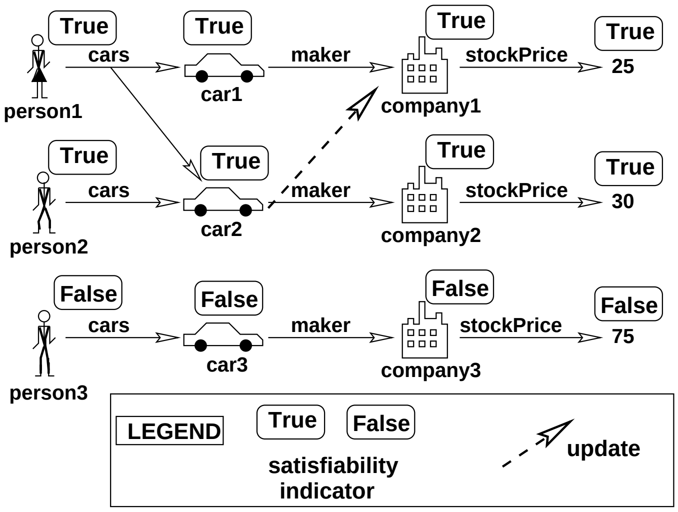

Figure 5: *Satisfiability Indicators* allow us to maintain path satisfiability information incrementally.

incremental fashion; when object O_j is updated, we can use the index to determine the old and new values for $SatInd(O_j)$. If $SatInd(O_j)$ changes as a result of the update, then we must iteratively traverse backwards through the multi-index components of SMX to update the *satisfiability indicators* of O_j and the objects that directly or indirectly refer to O_j in their path instantiations until either (1) we reach an object at the head of the path, in which case this object's membership in PQV_i could potentially have to change, or else (2) we find an element whose $SatInd$ is already set to the correct value.

Note that if $C_{j-1}.A_j$ is a multi-valued attribute, then potentially we might have to check all other values for the updated attribute in order to determine whether not the modified attribute value fulfills the quantifier. For example, if the quantifier were *existential* ($\Theta = \exists$), $SatInd(O_{new})$ were false, and $SatInd(O_{old})$ true, then we would have to check to make sure that no other value for $O_{j-1}.A_{i,j}$ has a positive $SatInd$ before we could determine that $SatInd(O_{j-1})$ should be reset to false. Similarly, if the query predicate were *universal* ($\Theta = \forall$) and $SatInd(O_{new})$ were true and $SatInd(O_{old})$ were false, then all other values of the multi-valued attribute must be checked in order to confirm that their *satisfaction indicators* are positive before we could determine that $SatInd(O_{j-1})$ should be reset to true.

Knowing whether or not a given instance participates in a satisfying instantiation also allows us to process object deletion updates efficiently. Suppose that we are given a path query $PQ_i = C_{i,1}.[\Theta]A_{i,2}\ldots[\Theta]A_{i,n}\theta value$, a deleted object $O_j \in C_{i,j}, 1 < j \leq n$, and an object $O_h \in C_{i,1}$ that is the head object for a satisfying instantiation of P_i in which O_j participates. Further suppose that $\exists$ a multi-valued

SATISFACTION-INDICATING MULTI-INDEX STRUCTURES

Car → Person

KEY	VALUE
car1	{True, {person1}}
car2	{True, {person1, person2}}
car3	{False, {person3}}

Index on Person.cars

stockPrice → Company

KEY	VALUE
25	{True, {company1}}
30	{True, {company2}}
75	{False, {company3}}

Index on Company.stockPrice

Company → Car

KEY	VALUE
company1	{True, {car1}}
company2	{True, {car2}}
company3	{False, {car3}}

Index on Car.maker

Figure 6: *Satisfaction-Indicating Multi-Index* (SMX) organization.

attribute $A_k, 1 < k \leq j$ along the subpath $PQ_i(1, j)$, with a quantifier Θ.

Lemma 3 *Given a deleted object $O_j \in C_{i,j}$, if the quantifier for the j^{th} component of the path query PQ_i is universal (i.e., $C_{i,j-1}.\forall A_{i,j}$), then any object O_h at the head of an instantiation of the queried path that did not satisfy PQ_i before the update will now satisfy PQ_i due to the deletion of O_j if and only if (iff) $\forall O_l \in C_{i,j}, l \neq d$, O_l participates in an instantiation of P_i headed by O_h that is satisfying, (i.e., if all remaining instantiations headed by O_h satisfy PQ_i). Similarly, if the quantifier for the j^{th} component of the path query PQ_i is existential (i.e., $C_{i,j-1}.\exists A_{i,j}$), then any object O_h at the head of an instantiation of the queried path that satisfied PQ_i before the update will satisfy PQ_i after the deletion of O_j if and only if (iff) $\exists O_l \in C_{i,j}, l \neq d$, s.t. O_l participates in an instantiation of P_i headed by O_h that is satisfying.*

It follows from Lemma 3 that if an object O_h satisfies PQ_i before the deletion of O_j and the quantifier for the j^{th} component of the path query PQ_i is universal, then all instantiations of P_i headed by O_h satisfy PQ_i.

Lemma 4 *Given a deleted object $O_j \in C_{i,j}$, if the quantifier for the j^{th} component of the path query PQ_i is universal (i.e., $C_{i,j-1}.\forall A_{i,j}$), then any object O_h at the head of an instantiation of the queried path that satisfied PQ_i before the update will continue to satisfy PQ_i despite the deletion of O_j iff $\exists O_l \in C_{i,j}, l \neq d$ and O_l participates in an instantiation of P_i headed by O_h (i.e., in this case, every instantiation headed by O_h satisfies PQ_i).*

It follows that if an object O_h did not satisfy PQ_i before the deletion of O_j and the quantifier for the j^{th} component of the path query PQ_i is existential, then none of the remaining instantiations of P_i headed by O_h satisfy PQ_i.

Lemma 5 *Given a deleted object $O_j \in C_{i,j}$, if the quantifier for the j^{th} component of the path query PQ_i is existential (i.e., $C_{i,j-1}.\exists A_{i,j}$), then any object O_h at the head of an instantiation of the queried path that did not satisfy PQ_i before the update will not satisfy PQ_i after the deletion of O_j.*

5 Experimental Setup

5.1 System Implementation

As described in [23], in order to evaluate the relative performance of the index structures in a real system implementation, we implemented the four index structures in the *MultiView* system using *GemStone*. *MultiView* provides a hierarchical registration service that supports the registration of virtual classes with other classes that they are dependent upon (derived from) and the notification of the former upon modifications performed on the latter [10, 12, 11, 17]. We now extend the system to register all classes along the path query PQ_j with their respective PQV_j view. Our platform is a Sun 4m running SunOS 4.1.3 with 32 megabytes of memory.

5.2 Test Data

While benchmarks for OODB performance evaluations exist (e.g., the OO7 benchmark [3]), none is designed to evaluate the performance of view systems [13]. Thus, for the purpose of evaluating materialized path query views, we designed a benchmark schema, shown in Figure 7, consisting of up to 12 classes named A, B, C, ... L, and M with A at the head and M at the end of the path, each containing one attribute constrained to the set of instances of the next class in the path. Each class includes an instance variable constrained to instances of the next class in the path. For example, class B has an attribute ccc, which is constrained to instances of class C. A path query can thus be expressed: *aaa.bbb.ccc...*

The factors in our test schema are designed to be changeable parameters so that we can conduct diverse tests with this schema and data set. We populated the database using randomly generated data (based on the Congruential method in James T. Smith [22]) so that it is fully under the control of the parameters we set, and thus can be shaped to explore all limitations and advantages of the index solutions. Our key parameters are presented in Figure 8.

6 Performance Studies

Although we conducted an extensive set of experiments [23], due to space constraints we present only a small subset of these at this time.

6.1 Experiment 1: Materialized v.s. Non-Materialized Views

Our first set of experiments evaluated the cost and benefit of path query view materialization by comparing the cost of performing a variable number (k) of updates and one query (returning the extent of the view). The results indicate that (using the SMX strategy to maintain the materialized view) given a path of length 7, view selectivity

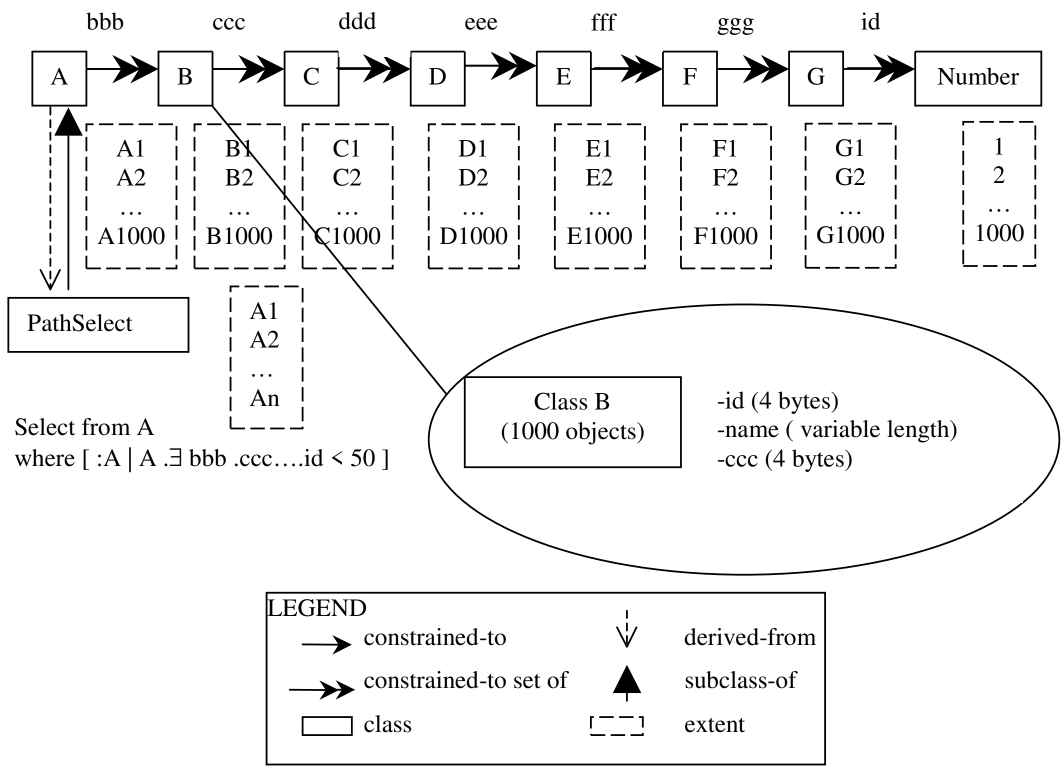

Figure 7: Benchmark Schema.

Symbol	Meaning	Domain	Default Value
ObjSize	Object size in byte	{8, 18, 28, 68, 108}	18 bytes
NumObjperClass	Number objects per Class	{100, 400,700,1000, 1300}	1000
ForkWidth	Fork (multi-valued attribute) width	{1, 2, 3, 4, 5, 6}	1
Fposition	Position of the fork in the path	{0, 1, 2, …, n}	The middle point
CvgRate	Converge rate	{100%, 85%, 70%, 55%, 40%, 25%, 10%, 1%}	100%
Cposition	Position of multi-reference links in the path	{0, 1, 2, …, n}	The middle point
Selectivity	Selectivity of the query	{ 10%, 20%, 40%, 60%, 80%, 100%}	10% of objects of base class are selected
PathLen	Number of classes participating in the path query	{3, 5, 7, 9, 11}	7
ObjUpdateRate	Percentage of objects in a class being updated	{100%, 80%, 60%, 40%, 20%, 1%}	100%
Uposition	Position where the update takes place	{0, 1, 2, …, n}	

Figure 8: Parameters of the Sample Database.

around 15%, 1000 forks of width 3 evenly distributed among 7000 objects, object size of 18 bytes, the cost of performing a single query on a non-materialized view is counter-balanced by the cost of propagating approximately 750 updates to a materialized view. This large update/query ratio indicates that materialization of path query views would benefit most applications.

6.2 Experiment 2: Varying the Length of the Path

In order to compare the performance of the SMX indexing strategy to other indexing strategies in the maintenance of materialized path query views, we measured the time needed to maintain a path query view in the face of updates while varying the path length from 3 to 11. We measured both the time for initializing the index and the time for a single update (averaged over 1000 updates). We consider only paths with single-valued attributes, which, as the next experiment demonstrates, is the worst case for SMX.

Figure 9 compares the cost of initializing the index structures. Given a path of length 5, SMX (and MX) takes four times longer than either NX or PX to initialize; given a path of length 11, the (S)MX index is 5.5 times more expensive.

Figure 10 compares the average cost of propagating an update using all four index structures. SMX is the fastest because it doesn't need to traverse forward to identify the exact value of either the original or new end point. MX, NX, and PX all must traverse forward to both new and original end points in order to determine whether or not the update affects the view. In addition, if multiple head objects are associated with the original endpoint, then NX must traverse forward from the head points to the update point in order to identify which were affected by the update. MX is the slowest, because it must also traverse backwards in order to identify affected head points. Note that while in Figure 10 PX performs as well as SMX, this is only true when no multi-valued attribute is present. SMX significantly outperforms PX when multi-valued attributes are introduced (as demonstrated in the next set of experiments).

6.3 Experiment 3: Forks and Multi-Valued References

According to the analytic model [10], multi-valued attributes (forks) and multi-references (converge) greatly increase the cost of maintaining the path query view in the face of updates. We validated that hypothesis by fixing the size of the database (to its default) and the length of the path (to 7), while varying the forking factor of the fifth path element's attribute. That is, given path aaa.bbb.ccc.ddd.eee.fff.ggg.hhh, we vary the fork width at 'eee' from 1 to 6. All other attributes are single-valued.

Figures 11 and 12 show the average time needed to perform updates before (at link 'ddd') and after (at link 'fff') the multi-valued attribute, respectively. The cost of maintaining the view using either the NX and PX increase quickly, because the affected head-points must be identified in order to maintain the index strucutres regardless of whether or not the view content is affected. As shown in Figure 12, SMX consistently

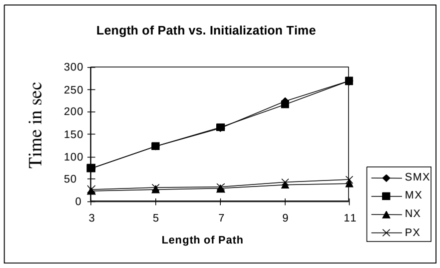

Figure 9: Time to initialize index structures (varying path length). NumObjPerClass=1000, ObjSize=18, No Fork, No multi-references, Update at the middle of path, Selectivity=20%

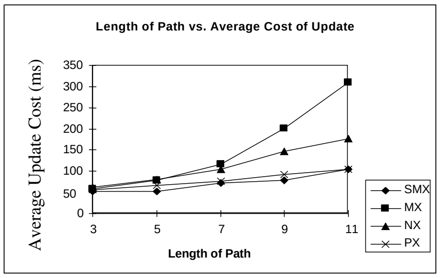

Figure 10: Update cost (varying path length). NumObjPerClass=1000, ObjSize=18, No Fork, No multi-references, Update at the middle of path, Selectivity=20%

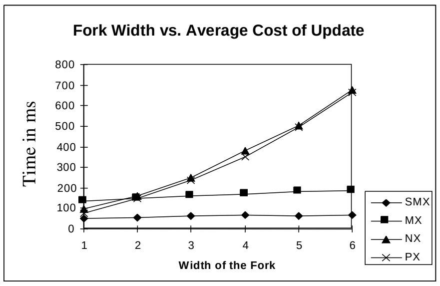

Figure 11: Average cost for one update before the fork (varying width of fork). NumObjPerClass=1000, ObjSize=18, PathLen=7, Fork position = eee, Selectivity=20%, Update position=ddd.

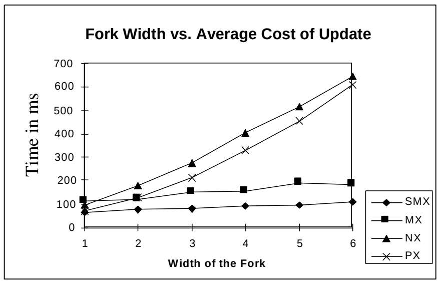

Figure 12: Average time cost for one update after the fork (varying width of fork). NumObjPerClass=1000, ObjSize=18, PathLen=7, Fork Position = eee, Selectivity=20%, Update position=fff.

outperforms MX by about 50 percent, because the MX strategy must traverse forward to both endpoints in order to determine whether or not the view is affected (whereas the SMX strategy can exploit the *SatInd* boolean for this purpose).

6.4 Experiment 4: Changing the Position of the Update

The position where the update takes place potentially impacts the performance of different index structures. For this experiment, we varyied the position of the update from the front to the end of the path (i.e., over positions bbb, ccc, ddd, eee, fff, ggg). We used a long path (PathLen=11) and default settings for all other values. We used no multi-valued attributes or multi-references.

As shown in Figure 13, both PX and SMX perform very well in the absence of

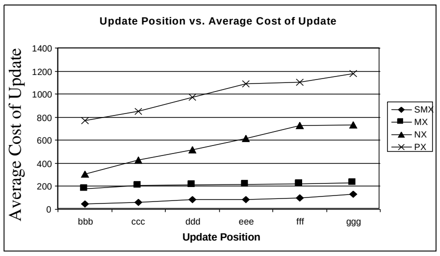

Figure 13: Average cost for update (varying position of update). NumObjPerClass=1000, ObjSize=18, PathLen=11, No Fork, Selectivity=20%.

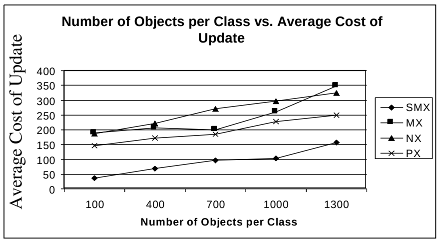

Figure 14: Average cost for update (varying the number of objects per class). SizeofObject=18, Selectivity=20%, PathLen=7, Fork width=3, Fork at eee, Update at ddd.

multi-valued attributes. According to the analytic model [10], the closer the updated object is to the head of the path, the higher the costs of traversing forward from it to its endpoints become. This will affect the traditional indices but not SMX. And the closer the updated object is to the end of the path, the lower the costs of traversing forward from the update site to its endpoints becomes. This benefits the traditional indices and would not lead to any advantage for SMX, because SMX never traverses forward. However, when the updated object is closer to the end of the path, then it becomes more expensive to use the multi-index structures to identify the heads of its instantiations. This increases the cost of evaluating updates with SMX and MX. In the case of MX, these two conflicting factors will compete, and the average cost of going one step forward v.s. one step backwards will be a determining factor.

6.5 Experiment 5: Varying the Number of Objects per Class

In this test, we vary the number of objects per class from 100 to 1300 while fixing the length of the path query to seven. The total number of objects in the database thus varies from 100*7 = 700 to 1300*7 = 9100 objects. There is a fork of width three in the middle of the path. Figure 14 shows the average cost of update at "ddd". All four lines go up as the number of objects increases. This is because as the total number of objects increases, the cost of looking up an individual object increases. Also all the index structures become larger, which makes the access more expensive.

6.6 Experiment 6: Changing the Position of the Fork

In this test we evaluate the impact of the position of the fork upon maintenance cost by moving a fork of width 4 along the path while observing the average time needed for an update at the middle point of the path (Class D). All other parameters are set to their default values, as shown in Figure 15.

The results of this test demonstrate a number of characteristics of the various index structures.

- The closer to the end of the path the fork is, the more expensive it is for the MX and SMX structures, because they must traverse backwards to identify the affected head points when the update affects the view.
- The closer to the end of the path the fork is, the more expensive it is for the NX structure, because it must traverse forward to identify the affected head points when the update affects the view.
- The closer to the head of the path the fork is, the more expensive it is for the traditional structures, because because they must traverse to the end of all identified paths to identify alternative endpoints.
- The presence of a fork at or before the point of update is more expensive for the traditional index structures (MX, NX, PX) because they must identify alternative instantiations in which the head objects might participate when the update affects the content of the view.
- The presence of a fork at or *after* the point of update is more expensive for the traditional index structures, because they must traverse to the end of all instantiated paths to determine the validity of old and new endpoints.
- The presence of a fork *at* the point of update is more expensive for the SMX structure, because it must examine the validity of all immediately linked objects.

6.7 Experiment 7: Evenly Distributed Fork Scenario

Experiment 3 verified that the presence of a single multi-valued attribute (fork) can greatly increase the cost of maintaining traditional index structures in the face of updates. In this experiment, we evenly distribute 1000 forks of width 3 among all the 7000 objects in the database. Although forks are evenly distributed, links are randomly generated; it is thus possible that a path instantiation contain more than one fork. On the average, about 17% of the objects of each class were multi-valued. The update position was varied while other parameters are fixed to their defaults.

Comparing the results of these tests, shown in Figure 16, to those of Experiment 4 (which changed the position of the update in the presence of a single fork), the path is shorter, but a greater number of forks are present. As we might expect, given that the cost of updating the PX and NX organizations increases in the presence of multi-valued attributes (as previously demonstrated by Experiment 3), the SMX performs two to ten times faster on average per update. Because the 1000 forks are evenly distributed along the path, we do not see any peaks like those seen in Experiment 3, Changing the Position of the Fork.

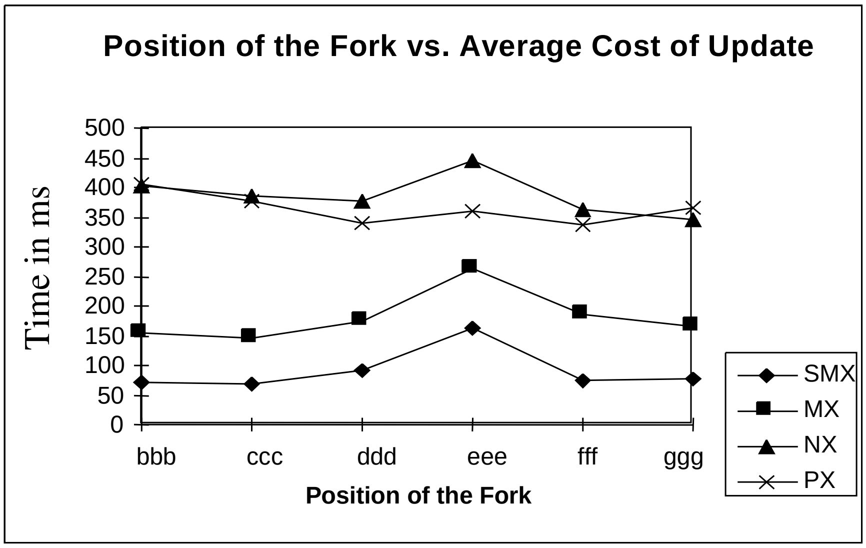

Figure 15: Average cost of update at eee (varying position of the fork). NumObjperClass=1000, ObjSize=18, PathLen=7, ForkWidth=4, Selectivity=20%, Update position=eee

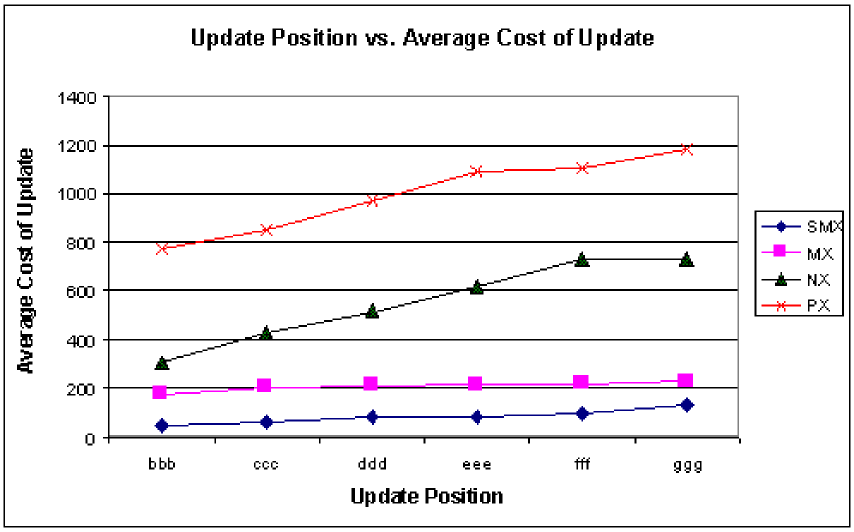

Figure 16: Update cost with forks evenly distributed along the path (varying update position) NumObjPerClass=1000, ObjSize=18, PathLen=7, Fork width=3, Selectivity=15-20%

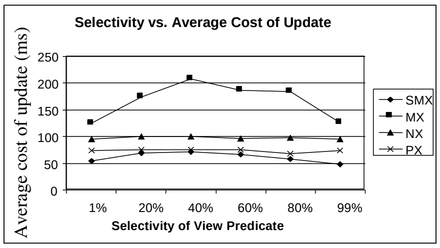

Figure 17: Average cost for update (varying selectivity of view predicate). NumObjperClass=1000, ObjSize=18, PathLen=7, No Fork, Update at ddd.

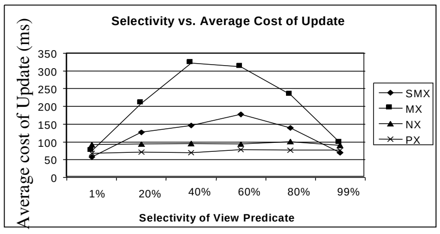

Figure 18: Average cost for update (varying selectivity of view predicate). NumObjperClass=1000, ObjSize=18, PathLen=7, No Fork, Update at ggg.

6.8 Experiment 8: Varying the Selectivity of the Path Query

The selectivity of the query used to define a path query view impacts both query and update times. The more selective the view predicate, the more materialized views will outperform non-materialized views when querying the view. The probability that the update changes the validity of the updated object (and thus requires updates to the materialized view's extent) depends on the selectivity of the query. If the selectivity of the query is X, then the probability that the update changes its status will be X * (1 - X) which reaches its maximum when X=50%.

Figure 17 compares the times needed to maintain the view in the face of updates while varying the view predicate selectivity from 1% to 99% (with 0% or 100% selectivity not meaningful) and fixing the other parameters at their default values. Regardless of the validity of the updated object, the PX and NX index structures must be updated. The SMX structure outperforms the other index organizations because it must be updated if and only if when the update changes the satisfaction indicator of an object; note that during the process of propagating the change forward, if we reach any object that is already set with the right satInd, we can stop. The MX organization is vulnerable whenever an object changes its validity, and thus the cost of the update peaks at 50% view selectivity (the highest probability of that an update impact view extent membership).

Figure 18 fixes the update point closer to the end of the path than Figure 17, which increases the cost of the reverse travesal used by MX and SMX to identify the head objects associated with updated objects. This is reflected in the fact that the "worst-case" 50% predicate selectivity performance using SMX and MX is higher than in Figure 17.

6.9 Discussion

Overall, the *multi-valued* attributes are a chief determining factor. For any fork width greater than 1, SMX performs 2-10 times faster than others. When no multi-valued or multiply-referenced attributes are involved, the performance of SMX is equivalent to PX. Because PX is faster to initialize than SMX, in this scenario it out-performs SMX.

Since SMX takes a longer time to initialize but helps us to update the view faster, the construction of SMX structure is only beneficial if there are *numerous update operations*. We found that when there are multi-valued attributes, the cost difference between initializing SMX and traditional indices can typically be balanced by the benefit of using SMX for about (0.01 * Total Number of Objects in Database) updates. When there are no multi-valued attributes, it takes 2 to 3 times more updates to balance the cost difference of initialization.

To find the head points when updating the view, SMX needs to traverse backward. When updates take place at a position close to the end of the path, SMX will need to traverse a longer distance backward. However, even in the worst case for SMX (no multi-valued attributes), on average, SMX was the fastest together with PX.

7 Related Work

Project-Select-Join Views. Although traditional relational databases (RDBs) don't have complex attributes of the object-oriented variety, foreign keys allow tuples to include references to other tuples. Multiple tables can be "joined" using foreign keys to form a relational equivalent to the OO aggregation hierarchy. The relational equivalent of a path query is a select-project-join (SPJ) query. However, path query views, as defined in this paper, are less powerful than the more general SPJ views – and our solution is tuned to take advantage of this fact. A common expense of relational SPJ is the cost of finding a join match in a relational table after following a reference, which our SMX solution allows us to avoid such traversals.

A number of researchers have investigated the problem of maintaining materialized SPJ views in RDBs. Gupta and Blakeley [6] infer knowledge about the state of underlying base relations using local information (such as the view definition, the update, and varying amounts of base relation replicas). They identify classes of materialized views according to the amount of information needed to maintain them in the face of updates and demonstrate necessary and sufficient conditions for determining the amount of information needed to update a materialized SPJ view. Segev and Zhao propose a join pattern indexing technique for materialized rule-derived data that allows the identification of join completion without reading base relations [21]. Join patterns more resemble a PX than an MX, and thus need to be maintained in the face of all updates (unlike the SMX satisfaction indicators).

OODB View Materialization. The only previous work on materialized path query views for OODBs that we are aware of is that of Konomi et al. [9], who use superkey classes to maintain consistency for a particular type of join class formed along an existing path in the aggregation graph. Superkey classes facilitate the incremental update and elimination of duplicates for materialized views produced by relational expressions that include projections. Instead of using external index structures, the authors provide a procedure that transforms class schemas to add new classes that will allow it to satisfy the super-key condition and thus permit incremental updates of the join class. They do not address the more general problem of path query views, which is the focus of this paper, nor do they provide any cost models or performance analyses.

Function Materialization. The work of Kemper et al. on *function materialization* is closely related to OODB view materialization [7, 8]. The goal of function materialization is the precomputation and maintenance of function results. Similar to the SMX *satisfaction indicator* solution, Kemper et al. associate a "validity" value with each object that can serve as an argument to a function. However, our *satisfaction indicators* indicate whether or not the object can be used to reach endpoints that satisfy the path view predicate and allows us to avoid evaluating updates that don't affect view membership. The "validity" value, on the other hand, indicates whether or not the object has been updated and thereby invalidated the stored result. The goal of keeping "validity" values is to facilitate *lazy rematerialization* of the function result.

Indexing Techniques. We utilized the work by Bertino et al. regarding the performance of basic operations with traditional index organizations [2, 4].

8 Conclusions

To the best of our knowledge, ours is the first work to address the specific needs of the path query view problem for OODBs and to present a solution that is tailored to these needs. We introduce a new *Satisfiability Indicating Multi-Index (SMX)* organization, which maintains partial information indicating whether or not the endpoints reachable from an object satisfies the query predicate. We identify a number of tasks required to maintain materialized path query views that involve multiple forward traversals with traditional indices. SMX can be used to eliminate these forward traversals. We also discuss experimental studies we have conducted on a working system implementation in *MultiView* . Our results indicate that the SMX dramatically improves upon the performance of traditional index structures for path query view maintenance.

References

[1] E. Baralis and S. Ceria nd S. Paraboschi. Conservative timestamp revisited for materialized view maintenance in a data warehouse. *Proceedings of the SIGMOD Workshop on Materialized Views: Techniques and Applications*, pages 1–9, 1996.

[2] E. Bertino and P. Foscoli. Index organizations for object-oriented database systems. *IEEE Transactions on Knowledge and Data Engineering*, 7(2):193–209, April 1995.

[3] M. J. Carey, D. J. DeWitt, and J. F. Naughton. The OO7 benchmark. *SIGMOD*, pages 12–21, 1993.

[4] S. Choenni, E. Bertino, H. M. Blanken, and T. Chang. On the selection of optimal index configurations in OO databases. In *IEEE Int. Conf. on Data Engineering*, pages 526–537, 1994.

[5] V. Crestana, A. Lee, and E. A. Rundensteiner. Schema Version Removal: Optimizing Transparent Schema Evolution Systems. Technical Report CSE-TR-340-97, University of Michigan, Ann Arbor, 1997.

[6] A. Gupta and J.A. Blakeley. Using partial information to update materialized views. *Information Systems*, 20(8):641–662, 1995.

[7] A. Kemper, C. Kilger, and G. Moerkotte. Function materialization in object bases. *SIGMOD*, pages 258–267, 1991.

[8] A. Kemper, C. Kilger, and G. Moerkotte. Function materialization in object bases: Design, realization, and evaluation. *IEEE Transactions on Knowledge and Data Engineering*, pages 587–608, 1994.

[9] S. Konomi, T. Furukawa, and Y. Kambayashi. Super-key classes for updating materialized derived classes in object bases. In *Int. Conference on Deductive and Object-Oriented Databases*, pages 310–326, July 1993.

[10] H. A. Kuno. *View Materialization Issues in Object-Oriented Databases*. PhD thesis, University of Michigan, Ann Arbor, June 1996.

[11] H. A. Kuno and E. A. Rundensteiner. Incremental update propagation algorithms for materialized object-oriented views in *MultiView*. Technical Report CSE-TR-246-95, University of Michigan, Ann Arbor, June 1995.

[12] H. A. Kuno and E. A. Rundensteiner. Materialized object-oriented views in *MultiView*. In *ACM Research Issues in Data Engineering Workshop*, pages 78–85, March 1995.

[13] H. A. Kuno and E. A. Rundensteiner. New benchmark issues for object-oriented view systems. In *OOPSLA Workshop on Object-Oriented Database Benchmarking*, October 1995.

[14] H. A. Kuno and E. A. Rundensteiner. The satisfiability-indicating multi-index organization for maintaining materialized path query oodb views. Technical Report CSE-TR-302-96, University of Michigan, Ann Arbor, 1996.

[15] H. A. Kuno and E. A. Rundensteiner. Using object-oriented principles to optimize update propagation to materialized views. In *IEEE Int. Conf. on Data Engineering*, pages 310–317, 1996.

[16] H. A. Kuno and E. A. Rundensteiner. The *MultiView* OODB view system: design and implementation. *Theory and Practice of Object Systems (TAPOS), Special Issue on Subjectivity in Object-Oriented Systems*, 2(3), 1997.

[17] H. A. Kuno and E. A. Rundensteiner. Incremental maintenance of materialized object-oriented views in *multiview*: Strategies and performance evaluation. *IEEE Transaction on Data and Knowledge Engineering*, 10(5):768–792, Sept/Oct. 1998.

[18] E. A. Rundensteiner. *MultiView*: Methodology for supporting multiple views in object-oriented databases". In *18th VLDB Conference*, pages 187–198, 1992.

[19] E. A. Rundensteiner, H. A. Kuno, Y.-G. Ra, V. Crestana-Taube, M. C. Jones, and P. J. Marron. Demonstration paper: The MultiView project: Object-oriented view technology and applications. *SIGMOD*, page 555, 1996.

[20] E. A. Rundensteiner, H. A. Kuno, and Y. J. Zhou. Incremental maintenance of materialized path query views. In *Object-Oriented Information Systems*, September 1998.

[21] A. Segev and J. L. Zhao. Efficient maintenance of rule-derived data through join pattern indexing. In *Int. Conference on Information and Knowledge Management*, pages 194–205, December 1993.

[22] J. T. Smith. *C++ For Scientists and Engineers*. McGraw-Hill Publishers, 1991.

[23] Y. J. Zhou. Experimental evaluation of materializing path query views in multiview. In *Master's Thesis, WPI*, February 1998.

[24] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *SIGMOD*, pages 316–327, May 1995.