The Satisfiability-Indicating Multi-Index Organization for Maintaining Materialized Path Query OODB Views [*]

Harumi A. Kuno Elke A. Rundensteiner
Dept. of Elect. Engin. and Computer Science, Software Systems Research Laboratory
The University of Michigan, Ann Arbor, MI 48109-2122
e-mail: `kuno@umich.edu, rundenst@eecs.umich.edu`

University of Michigan Technical Report CSE-TR-302-96

Abstract

Materialized database views allow applications to benefit from the powerful flexibility of views while minimizing the performance penalties traditionally associated with views. However, the need to maintain materialized views in the face of updates limits the variety of queries that can be used to define them. In this paper we address the problem of incrementally maintaining OODB views formed using path queries. Traditional index organizations are not well suited for this task. 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 during the propagation of updates rather than during query processing. 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 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 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. This new structure offers a number of benefits. (1) At most one path position forward must be traversed to determine whether or not the endpoint of an instantiation of the path 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) No more than one path position forward must ever be traversed in order to identify whether or not a given object participates in any alternative instantiations that fulfill a given path query predicate. In addition to proposing this new index organization, we also present cost models and analytic evaluations comparing the performance of the SMX organization to those of the multi, nested, and path index organizations with regards to calculating the effects of updates upon views. The results of our evaluations indicate that the SMX dramatically improves upon the performance of traditional index structures with respect to the problem of path query view maintenance.

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, NSF NYI grant #IRI 94-57609, 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

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* [19, 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 [11, 13, 14]. 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 the topic of maintaining materialized path query views in object-oriented databases. 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 [10]. Readers might also note that in [12] we discussed the path query view problem and proposed an initial technique for the maintenance of such views; however, none of the techniques proposed in the current work have been previously introduced.

In this paper, we explore the utilization of traditional path index structures for facilitating the incremental maintenance of path query views, but find that traditional indexing techniques are not well suited for this task. 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 introduce a new *Satisfiability Indicating Multi-Index (SMX)* organization that is specifically tailored to handle the issues of path query view maintenance. The SMX organization 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. The results of our evaluations indicate that the SMX dramatically improves upon the performance of traditional index structures with respect to the problem of path query view maintenance.

Although we focus on an object-oriented model in this current work, our solution is directly applicable to the relational context. Insofar as queries can be performed over multiple tables that are joined, select-project-join (SPJ) views are the traditional counterpart to path query views. Maintaining materialized SPJ views is a well-studied problem in the relational world [5, 17, 3]. Gupta and Blakeley present formal partial-information-based view maintenance techniques that infer knowledge about the state of the underlying base relations using local information (such as the view definition, the update, the current view materialization, and varying amounts of base relation replicas) [5]. 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 [17]. Although the path query view problem that we address in this chapter is more restricted than the general SPJ problem in that we do not allow free variables in the predicate expression, our solution is unique and dramatically improves upon the performance of traditional treatments of the path query view problem.

We begin in Section 2 by briefly reviewing the *MultiView* object model and formally describing the characteristics of path query views. In Section 3 we present three problems involved with the maintenance of path query views, including a discussion of the limitations of utilizing traditional index organizations to address these problems. As a solution, we propose the SMX organization in Section 4. We introduce cost models comparing the SMX organization to the traditional multi-index (MX), nested index (NX), and path index (PX) organizations in Section 5, and use these cost models to perform analytic evaluations which we examine in Section 6. Finally, we discuss related work in Section 8, and present our conclusions and future work in Section 9.

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 [16] and [11]. 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 [16, 13]. 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 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

[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.

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

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 with the SMX Organization

In this section, we present an example of a path query view and describe issues involved in the incremental maintenance of such views. We discuss the limitations of using traditional index organizations to address these issues, then introduce our *Satisfaction-Indicating Multi-Index* solution.

3.1 Example Schema

Figure 1 shows an initial aggregation schema composed of four classes that we use as a basis for the remainder of this discussion. *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.

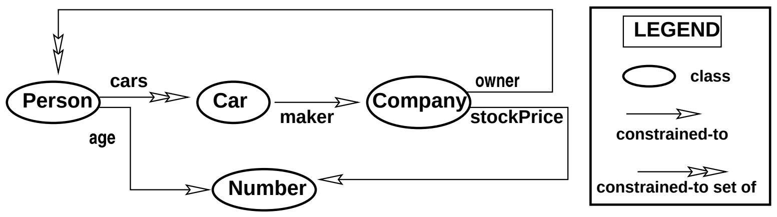


Figure 1: The example schema.

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, *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. The *car1* object has the *company1* object as a value for `maker`, the *company1* object has a `stockPrice` of 25, etc.

Now 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 that would be created for our example schema under the traditional multi-index (MX), path index (PX), and nested index (NX) organizations. We confine our current discussion to non-inherited forms of these index organizations for the sake of simplicity. However, note that our treatment could easily be extended to address the inherited forms.

3.2 Path Query View Maintenance Issues

The problem we address is how to maintain the extents of materialized path query views in the face of updates anywhere along their paths. In order to solve this problem, we must address the following issues.

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,

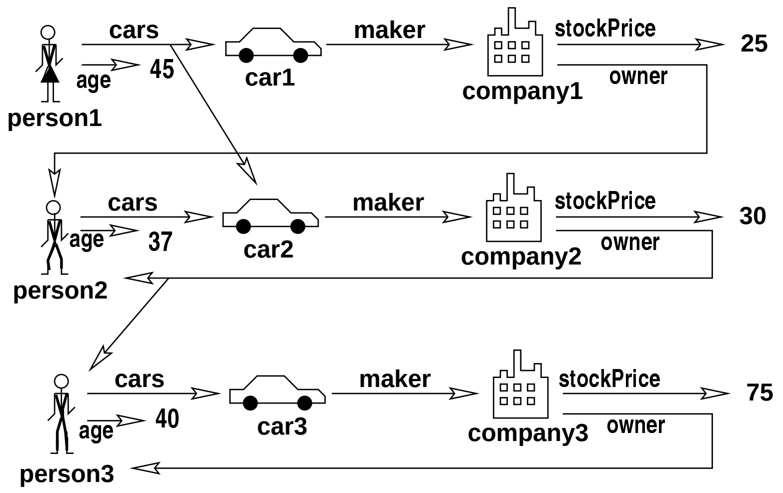


Figure 2: Initial base instances and their aggregation relationships for the schema in Figure 1.

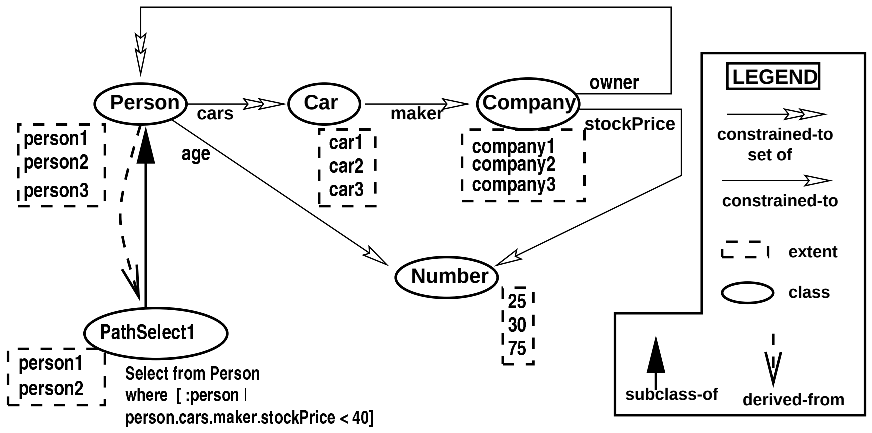
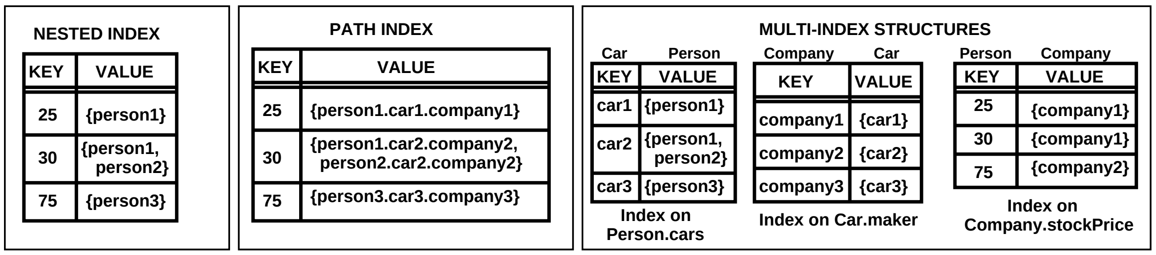


Figure 3: The *PathSelect1* is derived from the base schema using a path query.


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

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

Index on Person.cars

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

Index on Car.maker

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

Index on Company.stockPrice

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

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.

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 $P_i = C_{i,1}.A_{i,2}.A_{i,3}\ldots A_{i,n}$ 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:

1. We use the supplemental index structures at the time of *updates* instead of queries.
2. Because of this, 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.
3. 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) organization 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. The SMX organization 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 path query predicate of `person.cars.maker.owners.age` $<$ `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). However, due to space limitations, we discuss the processing of only the delete operation in depth. The strategies for maintaining the other operations are similiar.

Suppose that there exists a virtual path virtual view 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 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 the SMX organization 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

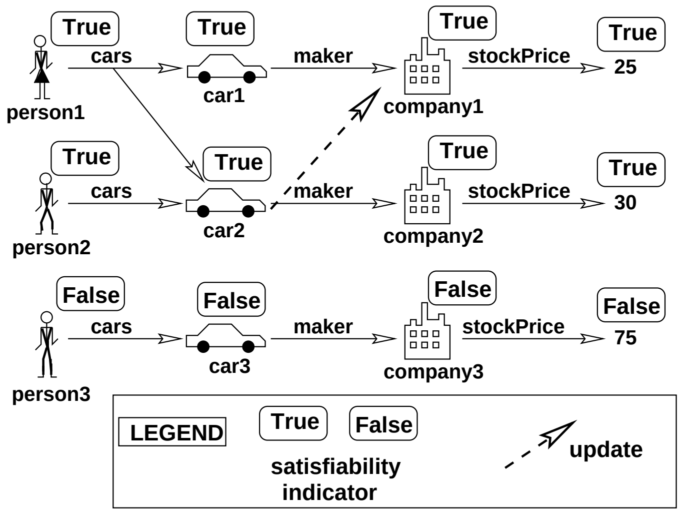


Figure 5: *Satisfiability Indicators* allow us to maintain path satisfiability information incrementally.

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.

SATISFACTION-INDICATING MULTI-INDEX STRUCTURES

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

Index on Person.cars

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

Index on Car.maker

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

Index on Company.stockPrice

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

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 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 Cost Models

In this section, we describe the cost of determining the effect of the creation, deletion, and modification of an object O_j along the aggregation path $P_i = C_{i,1}.A_{i,2}.A_{i,3} \ldots A_{i,n}$, where O_j belongs to class $C_{i,j}$ and $1 \leq j \leq n$ (see Table 1). We assume that the index structures are organized as B^+-trees, that each non-leaf node is stored on its own page, and that the leaf-nodes of each index are stored on their own pages. For the sake of simplicity, the formulas described below assume that the modified object participates in only one path query view and a given class appears at most once in the query's path. We will discuss extensions to our SMX solution that allow us to relax these assumptions in Section 7. We calculate cost in terms of the number of page accesses, and assume that a page contains objects of only one class. Table 2 presents the system parameters, adapted from the work of Choenni et al. [4] and Korth and Silberschatz [9], used in the following equations. Table 3 lists the path-query-view-specific parameters used in our equations.

parameter	definition
$CRIR_X$	Cost of retrieving an index record with organization X .
$CMIR_X$	Cost of maintaining an index record with organization X.
$CPropCreate_X$	Cost of determining the effect an object creation operation has on a virtual class's membership with organization X.
$CPropDelete_X$	Cost of determining the effect an object deletion operation has on a virtual class's membership with organization X.
$CPropMod_X$	Cost of determining the effect an object modification operation has on a virtual class's membership with organization X.

Table 1: Operation cost parameters.

5.1 Single Index Record Operations

The structure of the SMX organization is similar to that of the MX organization, in that the SMX index fulfills the function of an MX index by associating objects of a given class with those that refer to them. The difference is that a leaf node in an SMX organization is slightly larger because it stores more information. The cost of retrieving or maintaining a single index record in an SMX organization is thus very similar to that of performing the corresponding operation under an MX organization. As other researchers have previously discussed and contrasted the cost of performing basic functions using traditional index organizations [2, 4], we confine our model of the costs of performing basic operations to a comparison between the SMX and MX operations.

5.1.1 Storage Costs

Each SMX index record extends the MX index record with additional information as described in Section 4. Each leaf node used to store an index record from the SMX organization thus needs additional storage space for the *satisfiability indicator* associated with each object. Given a path query view PQV_i whose path P_i includes a class C_j for which we are building an index, the SMX organization's entry for an object of class C_i extends a typical MX entry with a *satisfiability indicator* $SatInd$. We can express the relationship between the sizes of the SMX and MX leaf nodes using Equation 1. $SatInd$ can be implemented by a 1 bit boolean value.

parameter	definition
h	Average height of B+ tree used to store an index structure.
p	Page size.
fr_X	Fraction of the record retrieved with organization X if leaf-node index record occupies more than one page.
fm_X	Fraction of record accessed during maintenance with organization X if leaf-node index record occupies more than one page.
ln_X	Average size of a leaf-node of an index with organization X.
OS_k	Average size of an object belonging to class C_k.
$SatInd$	Size of the structure used for the satisfiability indicator (1 bit).
$PathCtr$	Size of the structure used for the path position indicator.
oid	Object identifier size.
$ObjRet$	Average cost of an object retrieval (for an object of class C_l, this is $\lceil OS_l/p \rceil$).

Table 2: Cost model system parameters.

parameter	definition
$CheckType_k$	Cost of checking if an object is a member of class C_k.
$MultVal(i,j)$	Indicates whether or not path query PQ_i's $A_{i,j}$ attribute is multi-valued (0 or 1).
$SubPathMultVal(P_i(j,k))$	Indicates whether or not a subpath $PQ_i(j,k)$ includes any multi-valued attributes (0 or 1).
$Existential_{i,j}$	Indicates whether the quantifier associated with the $C_{i,j}.C_{i,j+1}$ component of path query PQ_i's quantifier is existential (0 or 1).
$Universal_{i,j}$	Indicates whether the quantifier associated with the $C_{i,j}.C_{i,j+1}$ component of path query PQ_i's quantifier is universal (0 or 1).
$NumRef_k$	Average number of objects referring to each member of class C_k as value for attribute A_k.
$NumVal(i,k)$	Average number of objects referred to as value of possibly multi-valued attribute $A_{i,k}$ by an object $O_{k-1} \in C_{i,k-1}$.
$NumEndpoints(i,j,n)$	Average number of distinct endpoints of instantiated subpaths $P_i(j,n)$ of path P_i in which object O_j participates.
$NumHeadpoints(i,j)$	Average number of distinct head objects of instantiations of path P_i in which an object O_j of class $C_{i,j}$ participates.
$NumValidBranch(i,j)$	Average number of instantiations of path P_i in which an an object of class $C_{i,j}$ that fulfills the given path query PQ_i participates.
$NumAffBranch(i,j)$	Average number of instantiations of path P_i that are affected by a modification to the $C_{i,j}.A_{i,j+1}$ attribute of an object $O_j \in C_{i,j}$.
$TraverseBranches(i,k)$	Average cost of traversing all instantiations that originate from an object O_k of class C_k at the head of subpath $P_i(k,n)$ (1 + $\sum_{l=k+1}^{n}(\prod_{m=k+1}^{l} NumVal(i,m)) * ObjRet$).
$ProbNoRef(i,k)$	Probability that (previous to the update) an object of class $C_{i,k}$ is not referred to as a value of attribute A_k by any object $O_{k-1} \in C_{i,k-1}$.
$ProbAff_{i,j}$	Probability that an update to an object O_j of class $C_{i,j}$ will change the value of $SatInd(O_j)$.
$ProbSat_{i,j}$	Probability that an update to an object O_j of class $C_{i,j}$ will change the value of $SatInd(O_j)$ from $false$ to $true$.
$ProbUnSat_{i,j}$	Probability that an update to an object O_j of class $C_{i,j}$ will change the value of $SatInd(O_j)$ from $true$ to $false$.

Table 3: Path query parameters.

$$\boxed{ln_{SMX} = ln_{MX} + SatInd.} \tag{1}$$

5.1.2 Retrieving and Maintaining an Index Record

Assuming that indices for both MX and SMX organizations are stored using B+ trees, retrieving an index record with either organization requires us to traverse the index from the root of the B+ tree to the appropriate leaf node. The cost $CRIR_X$ of this operation is:

$$\boxed{CRIR_X(h, fr_X) = \begin{cases} h & \text{if } (ln_X \leq p) \\ h - 1 + \lceil fr_X * ln_X / p \rceil & \text{otherwise} \end{cases}} \tag{2}$$

Similarly, the cost $CMIR_X$ of maintaining a single index record from either the MX or SMX organization is defined as:

$$\boxed{CMIR_X(h, fm_X) = \begin{cases} h + 1 & \text{if } (ln_X \leq p) \\ h - 1 + \lceil 2 * fm_X * ln_X / p \rceil & \text{otherwise} \end{cases}} \tag{3}$$

5.2 Path Query View Maintenance Operations

For the purposes of the following comparisons, suppose that there exists a virtual path virtual view 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.

5.2.1 Creation of a New Object

When a new object O_j of class $C_{i,j}$ is created, we consider the creation operation to also include the setting of the instance variable values for the new object. Because we treat creation and modification as independent operations, when a new object is created, no existing object could possibly refer to O_j as a value for any of its attributes. Therefore, when a new object O_j of class $C_{i,j}$ is created, unless $C_{i,j}$ is the class at the head of path P_i (i.e., if $j \neq 1$), O_j cannot participate in a full instantiation of path P_i and thus won't affect the membership of PQV_i. Note, however, that O_j's subpath $P_i(j, n) = O_j.A_{i,j+1} \ldots A_{i,n}$ instantiations could result in an end value O_n s.t. $O_n\theta value$.

Path Index. In the case of the path index organization, if $C_{i,j}$ is at the head of the path ($j = 1$), we must traverse all of the new object O_j's instantiations of path P_i in order to determine the endpoint key values under which we must add O_j to the path index ($TraverseBranches(i, j)$). The path index must be updated once for each distinct endpoint of every path instantiation in which O_j participates (there are $NumEndpoints(i, j, n)$ such endpoints, so the cost is $NumEndpoints(i, j, n) * CMIR_{PX}$). If $C_{i,j}$ is not at the head of the path, then we do not need to update anything because O_j does not participate in a full instantiated path.

$$\boxed{CPropCreate_{PX} = \begin{cases} TraverseBranches(i, 1) & \text{if } (j = 1) \\ +NumEndpoints(i, 1, n) * CMIR_{PX} & \\ 0 & \text{otherwise} \end{cases}} \tag{4}$$

Nested Index. The cost of evaluating the creation of a new object with a nested index is identical to the cost of evaluating the creation operation with the path index organization.

$$\boxed{CPropCreate_{NX} = \begin{cases} TraverseBranches(i,1) & \text{if } (j=1) \\ +NumEndpoints(i,1,n) * CMIR_{NX} & \\ & \\ 0 & \text{otherwise} \end{cases}} \tag{5}$$

MX. When a new instance O_j of a class $C_{i,j}$ is created, if $C_{i,j}$ is not at the *head* of path $P_i = C_{i,1}.A_{i,2}\ldots A_{i,n}$, $1 < j \leq n$, then we need to update the $C_{i,j}.A_{i,j+1}$ index to add O_j to the entries for all values of $O_j.A_{i,j+1}$ (that is, update the multi-index to reflect the relationships between O_j and the objects referred to by $O_j.A_{i,j+1}$). This cost is $CMIR_{MX} * NumVal(i, j+1)$.

If $C_{i,j}$ is at the *head* of the path (i.e., $j = 1$) then in addition to updating the index structure for attribute $C_{i,j}.A_{i,j+1}$, we must also traverse object O_j's complete aggregation hierarchy in order to see if it has any endpoints that satisfy the query's predicate, thereby qualifying O_j for membership in PQV_i.

$$\boxed{CPropCreate_{MX} = \begin{cases} NumVal(i,j+1) * CMIR_{MX} & \text{if } (j=1) \\ +TraverseBranches(i,j) & \\ & \\ NumVal(i,j+1) * CMIR_{MX} & \text{otherwise} \end{cases}} \tag{6}$$

SMX. When a new instance O_j of class $C_{i,j}$ is created, if $C_{i,j}$ is not at the *head* of the path $P_i = C_{i,1}.A_{i,2}.A_{i,3}\ldots A_{i,n}$ (i.e., $j \neq 1$), then we need to update only the appropriate SMX index records, and do not require any object traversal operations. Because only one object is created at a time, the $A_{i,j+1}$ attribute of O_j must refer to an existing object O_{j+1} (or set of objects if $A_{i,j+1}$ is multi-valued) of class $C_{i,j+1}$. We can look up the satisfiability indicator of O_{j+1}'s instantiations of the subpath $P_i(j+1,n)$ at the same time that we update the index records for the values for $O_j.A_{i,j+1}$. The cost in this case is the cost of looking up and updating the index record associated with each O_{j+1} s.t. $O_j.A_{i,j+1}$ refers to O_{j+1} and of updating that index record to add O_j ($NumVal(i,j+1) * CMIR_{SMX}$). If an object O_{j+1} does not already have an index record associated with it because no object previously referred to it (i.e., when $ProbNoRef > 0$), then we must look up O_{j+1}'s values for $A_{i,j+2}$ in the $C_{j+1}.A_{j+2}$ index to get the correct *satisfiability indicator* value for O_{j+1} ($ProbNoRef(i,j+1) * NumVal(i,j+1) * NumVal(i,j+2) * CRIR_{SMX}$). Since we are guaranteed to find O_{j+2} in this index, no other traversal is needed. If $C_{i,j}$ is at the *head* of the path (i.e., $j = i$) then we can look up the satisfiability of O_{j+1}'s instantiation of the subpath $P_i(j+1,n)$ at the same time that we update the index record for O_{j+1}. Thus the cost is independent of $C_{i,j}$'s position in the path.

$$\boxed{CPropCreate_{SMX} = \begin{cases} NumVal(i,j+1) * CMIR_{SMX} \\ +ProbNoRef(i,j+1) * NumVal(i,j+1) \\ NumVal(i,j+2) * CRIR_{SMX} \end{cases}} \tag{7}$$

5.2.2 Deletion of an Existing Object

When an instance O_j of the class $C_{i,j}$ is deleted, if $C_{i,j}$ is at the *head* of the path (i.e., $j = 1$), we can determine whether or not O_j participates in a materialized path query view PQV_i by checking to see if O_j possesses the type of the virtual class (regardless of the type of index organization in use). The cost of handling the deletion in this case is then simply $CheckType_i$ plus the cost of updating the appropriate index structures.

If O_j is not at the head of the path ($j \neq 1$), then we must determine whether or not the change affects the membership of the path query view PQV_i. This task requires us to identify the object at the head of every instantiated path in which O_j participates. Once we identify the head objects, we can check to see if each head object possesses the type of the virtual class. Even if we identify that a head object possesses the type of the virtual class, however, if the path includes any multi-valued attributes between the head of the path and the deleted object, it is possible that the head object participates in alternative instantiations of the path that don't include O_j and that will thus allow it to continue its membership in PQV_i.

In summary, given a path query view PQV_i defined using path query $PQ_i = C_{i,1}.[\Theta]A_{i,2}.[\Theta]A_{i,3}\ldots[\Theta]A_{i,n}\theta value$ and a deleted object $O_j \in C_{i,j}, 1 < j \leq n$, the following tasks must be performed in order to maintain PQV_i.

- Identify every object $O_k \in C_{i,1}$ that is the head object of an instantiation of PQ_i's path involving O_j.
- Determine which of these head objects is a currently member of PQV_i.
- If subpath query $PQ_i(1,j)$ includes any multi-valued attributes, then the head objects potentially still satisfy PQ_i. Lemmas 3 through 5 state the conditions under which head objects can continue to satisfy PQ_i.

Path Index. If the deleted object O_j is at the head of the path (i.e., $j = 1$), then we could immediately check its type to determine whether or not it currently is a member of PQV_i and therefore satisfies PQ_i ($CheckType_i$). However, to update the PX structure, we must traverse each of O_j's instantiations forward from the deleted object O_j to every endpoint object O_n (i.e., $TraverseBranches(i,j) + NumEndpoints(i,j,n) * CMIR_{PX}$). Since we must do the traversal anyway, we can forgo the cost of checking the type of O_j.

If the deleted object O_j is not at the head of the path (i.e., $j \neq 1$), then we must also forward-traverse each of O_j's instantiations forward from the deleted object O_j to every endpoint object O_n and then look up each of these endpoints in the path index in order to identify the heads of the instantiations and update the index entries. The cost for this operation is $TraverseBranches(i,j) + NumEndpoints(i,j,n) * CMIR_{PX}$. In addition, for each head object O_h of an instantiation in which O_j participates (there will be $NumValidBranch(i,j)$ such head objects), if the subpath $P_i(1,j)$ includes a multi-valued attribute, then we may have to forward-traverse all instantiations of subpath $P_i(1,j)$ headed by O_h in order to determine whether or not O_h satisfies PQ_i by Lemma 3. The worst-case cost of this traversal is $TraverseBranches(i,1)$. Thus in the worst-case, the cost of determining the impact of a deletion and updating the path index structure is:

$$\boxed{CPropDelete_{PX} = \begin{cases} TraverseBranches(i,j) & \text{if } (j = 1) \\ +NumEndpoints(i,j,n) * CMIR_{PX} & \\ & \\ TraverseBranches(i,j) & \text{otherwise} \\ +NumEndpoints(i,j,n) * CMIR_{PX} & \\ +SubPathMultVal(PQ_i(1,j)) & \\ *TraverseBranches(i,1). & \end{cases}} \quad (8)$$

Nested Index. If the deleted object is at the head of the path (i.e., $j = 1$), then the cost of propagating the update with the NX organization is the same as it would be with the PX organization.

Otherwise ($j \neq 1$), then (in addition to the cost of $TraverseBranches(i,j) + NumEndpoints(i,j,n) * CMIR_{NX}$) if there is more than one path head associated with a given endpoint O_n in the NX index (if $NumHeadpoints(i,n) > 1$) then we must traverse all of the instantiations of each of these path heads to position j in order to identify which path instantiations actually involve the deleted object O_j ($NumEndpoints(i,j,n) * NumHeadpoints(i,n) * (TraverseBranches(i,1) - TraverseBranches(i,j))$). We must do this because whereas with the path index, which records the full instantiations associated with key, with the nested index only the head objects are stored and we thus cannot tell from the index which of these objects head instantiations that do not involve O_j. In addition, for each head object O_h of an instantiation in which O_j participates, if the subpath $P_i(1,j)$ includes a multi-valued attribute, then we must forward-traverse all instantiations of P_i headed by O_h in order to determine whether or not O_h satisfies PQ_i by Lemma 3. The worst-case cost of this traversal is $TraverseBranches(i,1)$.

$$CPropDelete_{NX} = \begin{cases} TraverseBranches(i,j) & \text{if } (j=1) \\ +NumEndpoints(i,j,n) * CMIR_{NX} & \\ & \\ TraverseBranches(i,j) & \text{otherwise} \\ +NumEndpoints(i,j,n) * CMIR_{NX} & \\ +NumEndpoints(i,j,n) & \\ *NumHeadpoints(i,n) & \\ *[TraverseBranches(i,1) & \\ -TraverseBranches(i,j)] & \\ +NumEndpoints(i,j,n) * CMIR_{PX} & \\ +SubPathMultVal(PQ_i(1,j)) & \\ *TraverseBranches(i,1). & \end{cases} \tag{9}$$

MX. If the deleted object is at the head of the path ($j = 1$) then we only need to update the index records of the $NumVal(i, j+1)$ objects that are directly referenced by $O_j.A_{j+1}$ (at a cost of $NumVal(i, j+1) * CMIR_{MX}$). We can also test the deleted object to see if it should be deleted from the extent of the materialized class, bringing the total cost to $NumVal(i, j+1) * CMIR_{MX} + CheckType_i$.

If the deleted object is not at the head of the path ($j \neq 1$), then we must delete O_j's index record in the $C_{i,j-1}.A_{i,j}$ index and update the index records in the $C_{i,j}.A_{i,j+1}$ index of the $NumVal(i, j+1)$ objects that are directly referenced by $O_j.A_{j+1}$ (at a cost of $(1+NumVal(i, j+1) * CMIR_{MX})$. We must also identify the objects at the head of path instantiations that involve the deleted object O_j. We can use the MX structures for this task by looking up O_j in the $C_{i,j-1}.A_{i,j}$ MX index to identify the $NumRef_j$ objects of class $C_{i,j-1}$ that refer to O_j, then looking up each of those objects in the $C_{i,j-2}.A_{i,j-1}$ MX index to identify the $NumRef_{j-1} * NumRef_j$ objects of class $C_{i,j-2}$ that refer to those objects, etc., for a cost of

$CRIR_{MX} * \sum_{l=1}^{j} (\prod_{m=2}^{l} NumRef_m)$. We then must test the head object of each instantiation to determine if it possesses the type of PQV_i and thus should possibly be removed from PQV_i ($CheckType_i * NumHeadpoints(i, j)$). If a head object does possess the type of PQV_i and the path contains multi-valued attributes, then by Lemma 3. in the worst case we must traverse the head object's full aggregation hierarchy to determine if it possesses an alternative instantiation that satisfies PQV_i's predicate despite the deletion of object O_j ($SubPathMultVal(PQ_i(1, j)) * TraverseBranches(i, 1)$).

$$CPropDelete_{MX} = \begin{cases} NumVal(i,j+1) * CMIR_{MX} + CheckType_i & \text{if } (j=1) \\ & \\ (1+NumVal(i,j+1)) * CMIR_{MX} & \text{otherwise} \\ +CRIR_{MX} * \sum_{l=1}^{j} (\prod_{m=2}^{l} NumRef_m) & \\ +CheckType_i * NumHeadpoints(i,j) & \\ +SubPathMultVal(PQ_i(1,j)) & \\ *TraverseBranches(i,1) & \end{cases} \tag{10}$$

SMX. With the SMX organization, if $j = 1$ then we must update the record of every object referred to by O_j as a value for $C_{i,1}.A_{i,2}$ in the $C_{i,1}.A_{i,2}$ index structure to reflect the deletion of O_j. The cost for this is $NumVal(i, 2) * CMIR_{SMX}$. We can immediately determine whether or not O_j participated in a satisfying instantiation by checking the satisfaction indicators associated with the updated records.

Otherwise, if $j \neq 1$ then in addition to updating the records the objects referred to by O_j as a value for $C_{i,1}.A_{i,2}$ ($NumVal(i, j+1) * CMIR_{SMX}$), we must also update O_j's record in the $C_{i,j-1}.A_j$ index structure to reflect O_j's deletion ($1 * CMIR_{SMX}$). Again, we can immediately identify whether or not O_j participated in a successful path instantiation by checking the satisfaction indicators associated with the updated records. If $C_{i,j-1}.A_{i,j}$ is a multi-valued attribute, then we must check to see if the deletion of O_j changes the satisfaction indicators of objects that referred to O_j. If $C_{i,j-1}.A_{i,j}$'s quantifier is existential and O_j participated in a path instantiation that satisfied PQV_i's query, or if $C_{i,j-1}.A_{i,j}$'s quantifier is universal and

O_j did not participate in a path instantiation that satisfied PQV_i's query, then by Lemma 3. for each object O_k that referred to O_j, we must now examine O_k's other values for $C_{i,j-1}.A_{i,j}$ to determine whether or not O_k now participates in a satisfying instantiation and update the satisfaction indicators if appropriate. This task requires us to look up each O_{j-1} object that refers to O_j in the $C_{i,j-2}.A_{i,j-1}$ index. For each of these O_{j-1} objects, if the quantifier for $C_{i,j-1}.A_{i,j}$ is existential (and $C_{i,j-1}.A_{i,j}$ is multi-valued) and $SatInd(O_{j-1})$ is positive, or if the quantifier for $C_{i,j-1}.A_{i,j}$ is universal and $SatInd(O_{j-1})$ is negative, then we must look up the satisfaction indicators of all the other objects that are referred to by O_{j-1} $((NumVal(i,j) - 1) * CRIR_{SMX})$. If the value of $SatInd(O_{j-1})$ should be changed as a result of the deletion of O_j, then we must update the index record for O_{j-1} and repeat this process for each object $O_k \in C_{i,l}, 1 \leq l \leq j$ that both indirectly refers to the deleted O_j object **and** has a satisfaction indicator that changes value as a result of the deletion. In the worst-case (assuming that we must do a full reverse traversal from j to 1), the cost for this task is $(1 + NumVal(i,j+1)) * CMIR_{SMX}$ to update the records of objects that refer to and are referred to by O_j, plus

$ProbAff_{i,j} * (\sum_{l=1}^{j}(\prod_{m=2}^{l} NumRef_m) * CMIR_{SMX}$ to update the records of objects that indirectly refer to O_j, plus

$SubPathMultVal(PQ_i(1,j)) * [\sum_{l=1}^{j} MultiVal(i,l)(\prod_{m=2}^{l} NumRef_m * NumVal(i,m) - 1)] * CMIR_{SMX}$ to recalculate the satisfaction indicators of path components involving multi-valued attributes. Otherwise, we do not have to update the satisfaction indicators. The cost of propagating a deletion using a SMX index organization is thus:

$$\boxed{CPropDelete_{SMX} = \begin{cases} NumVal(i,2) * CMIR_{SMX} & \text{if } (j = 1) \\ \\ (1 + NumVal(i,j+1)) * CMIR_{SMX} & \text{otherwise} \\ +ProbAff_{i,j} * [\sum_{l=1}^{j}(\prod_{m=2}^{l} NumRef_m)] & \\ *CMIR_{SMX} & \\ +SubPathMultVal(PQ_i(1,j)) & \\ *[\sum_{l=1}^{j} MultiVal(i,l)(\prod_{m=2}^{l} NumRef_m & \\ *NumVal(i,m) - 1)] * CMIR_{SMX} & \end{cases}} \quad (11)$$

5.2.3 Modification of an Instance Variable

When an instance O_j of class $C_{i,j}$ is modified so that its $A_{i,j+1}$ attribute is set to object O_{new} of class $C_{i,j+1}$ instead of object O_{old} of class $C_{i,j+1}$, then in order for the update to be propagated we must perform the following tasks:

1. We must identify whether O_{new} and O_{old} lead to equivalent endpoints in terms of whether or not they satisfy the query's predicate.

2. If the endpoints are not equivalent in terms of satisfying the query's predicate condition, then we must identify the head objects of all path instantiations that involve O_j and are thus potentially affected by the modification.

3. If P_i contains any multi-valued attributes, and the endpoints are not equivalent in terms of satisfying the query's predicate condition, then we must determine whether or not alternative instantiations exist that affect the impact of the update.

4. We must update the appropriate index structures.

Path Index. Whether or not O_j is at the head of the path, we must traverse all path instantiations of both O_{old} and O_{new} to their endpoint objects
(at a cost of $2 * TraverseBranches(i, j+1)$) so that we can update the path index records for these endpoints (at a cost of $2 * NumEndpoints(i,j,n) * CMIR_{PX}$). If a comparison of the new and old endpoints indicates that the update affects the instantiations in which O_j participates, then we must identify the objects at the heads of these instantiations. Regardless of O_j's position in the path, we can perform this task at the same time that we update the path index records. However, if subpath query $PQ_i(1,j)$ includes any multi-valued attributes, then we must determine whether or not the objects at the heads of affected instantiations participate in alternative instantiations not involving O_j, and what the impact of these alternative instantiations is. We thus must perform a forward traversal from each of these head objects in order to determine if it participates in any alternative instantiation of P_i (not involving O_j) that affects the head object's membership in PQV_i. The cost of this traversal is $NumAffBranch(i,j) * TraverseBranches(i,1)$.

$$CPropMod_{PX} = \begin{cases} 2 * (TraverseBranches(i, j+1) & \text{if } (j = 1) \\ +NumEndpoints(i,j,n) * CMIR_{PX}) & \\ & \\ 2 * (TraverseBranches(i, j+1) & \text{otherwise} \\ +NumEndpoints(i,j,n) * CMIR_{PX}) & \\ +SubPathMultVal(PQ_i(1,j)) & \\ *NumAffBranch(i,j) & \\ *TraverseBranches(i,1). & \end{cases} \tag{12}$$

Nested Index. The cost of identifying whether O_{new} and O_{old} lead to equivalent endpoints regarding whether or not they satisfy the query's predicate is the same with the NX as it is for the PX organization ($2 * TraverseBranches(i, j+1)$). If O_j is at the head of the path ($j = 1$), then we can then update the appropriate index structures ($2 * NumEndpoints(i, j, n) * CMIR_{NX}$). However, if O_j is not at the head of the path, then in addition to looking up the head objects associated with each endpoint object in the NX so that we can update them, if there is more than one path head associated with a given endpoint O_n in the NX index
(if $NumHeadpoints(i, n) > 1$) then we must traverse all of the instantiations of each of these path heads to position j in order to identify which path instantiations actually involve the modified object O_j ($NumEndpoints(i, j, n) * NumHeadpoints(i, n) * (TraverseBranches(i, 1) - TraverseBranches(i, j))$). In addition, if the endpoints of O_j's new and old instantiations indicate a change in satisfaction of PQ_i and the subpath $P_i(1, j)$ includes a multi-valued attribute, then for the head object O_h of each instantiation in which O_j participates (there will be $NumAffBranch(i, j)$ such head objects), we must forward-traverse all instantiations of subpath $P_i(1, j)$ headed by O_h in order to determine whether or not O_h still satisfies PQ_i by Lemma 3. The cost of this traversal is $NumAffBranch(i, j) * TraverseBranches(i, 1)$.

$$CPropMod_{NX} = \begin{cases} 2 * (TraverseBranches(i, j+1) & \text{if } (j = 1) \\ +NumEndpoints(i,j,n) * CMIR_{NX}) & \\ & \\ 2 * (TraverseBranches(i, j+1) & \text{otherwise} \\ +NumEndpoints(i,j,n) * CMIR_{NX}) & \\ +NumEndpoints(i,j,n) & \\ *NumHeadpoints(i,n) & \\ *(TraverseBranches(i,1) & \\ -TraverseBranches(i,j)) & \\ +SubPathMultVal(PQ_i(1,j)) & \\ *NumAffBranch(i,j) & \\ *TraverseBranches(i,1) & \end{cases} \tag{13}$$

MX. Identifying the new and old endpoints of the path requires the traversal of all branches from O_{new} and O_{old} ($2 * TraverseBranches(i, j)$). If the new and old endpoints indicate that the path query view must be updated, then we must

identify the objects at the heads of O_j's path instantiations, at a cost of $CRIR_{MX} * \sum_{l=1}^{j} \prod_{m=2}^{l} NumRef_m$. We must also update the $C_{i,j}.A_{i,j+1}$ MX index structure to reflect O_j's old and new values of $A_{i,j+1}$ ($2 * CMIR_{MX}$). Finally, if the subpath $P_i(1, j)$ includes any multi-valued attributes, and O_j's instantiations are affected, then we must traverse all instantiations of each associated head object in order to see if the head object participates in any alternative instantiations that negate the effect ($SubPathMultVal(PQ_i(1, j)) * NumAffBranch(i, j) * TraverseBranches(i, 1)$).

$$CPropMod_{MX} = \begin{cases} (2 * TraverseBranches(i,j)) + CRIR_{MX} * \\ \sum_{l=1}^{j} \prod_{m=2}^{l} NumRef_m \\ + SubPathMultVal(PQ_i(1,j)) * NumAffBranch(i,j) \\ * TraverseBranches(i,1). \end{cases} \quad (14)$$

SMX. When object O_j is modified so that its $A_{i,j+1}$ attribute is set to the object O_{new} instead of object O_{old} then the following tasks can be performed using the SMX organization:

1. Update the $C_{i,j}.A_{i,j+1}$ index and change O_{old}'s index record to remove O_j. When we perform this update, we can look at O_{old}'s *satisfiability indicator* to see whether or not the endpoints of O_{old}'s path instantiations satisfied the query's predicate ($1 * CMIR_{SMX}$).

2. Update the $C_{i,j}.A_{i,j+1}$ index to reflect the fact that O_j now refers to O_{new} ($1 * CMIR_{SMX}$). This could be done at the same time as the previous step, so as to reduce the extra cost. If no other object previously referred to O_{new} then we must look up $O_{new}.A_{i,j+2}$'s value(s) in the $C_{i,j+1}.A_{i,j+2}$ index in order to calculate what the *satisfiability indicator* for O_{new} should be ($ProbNoRef(i, j+1) * NumVal(i, j+2) * CRIR_{SMX}$).

3. If O_{old} and O_{new}'s *satisfiability indicators* differ in value, then we must iteratively 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 should change, or else (2) we find an element whose $SatInd$ is already set to the correct value. Note that if the query predicate is *existential* ($\Theta = \exists$) and $SatInd(O_{new})$ is false and $SatInd(O_{old})$ is true, or if the query predicate is *universal* ($\Theta = \forall$) and $SatInd(O_{new})$ is true and $SatInd(O_{old})$ is false, then all values of each examined multi-valued attribute $A_{i,k}$ must be checked. We can delineate this process as follows:

Procedure 1

```
For each $O_l$ s.t. $SatInd(O_l)$ changes as a result of the modification to $O_j$, do:
    Update $O_l$'s $SatInd$ in the $C_{i,l-1}.A_{i,l}$ SMX index. ($1 * CMIR_{SMX}$)
    If $PQ_i$ is existential and $O_j$'s instantiation ceases to
    satisfy the path query predicate as a result of the modification
    ($Existential_i * ProbUnSat_{i,j}$),
    or if $PQ_i$ is universal and $SatInd(O_j)$ changes to true
    as a result of the modification ($Universal_i * ProbSat_{i,j}$),
        then $\forall O_{l-1}$ s.t. $O_{l-1}$'s $A_{i,l}$ attribute refers to $O_l$ ($NumRef_l$)
            If the $A_{i,l}$ attribute is multi-valued ($MultVal(i,j)$)
                then look up all of $O_{l-1}$'s values for $A_{i,l}$
                in the $C_{i,l-1}.A_{i,l}$ SMX index ($NumVal(i,l)$)
                and evaluate whether or not $SatInd(O_{l-1})$ changes
                as a result of the modification to $SatInd(O_l)$.
                If $SatInd(O_{l-1})$ changes, then $SatInd(O_{l-1})$ changes
                as a result of the modification to $O_j$,
                and thus $O_{l-1}$ should be processed.
                Otherwise, $O_{l-1}$ should not be processed.
```

The total cost of propagating a modification with the SMX organization is thus:

$$CPropMod_{SMX} = \begin{array}{l} 2 * CMIR_{SMX} \\ + ProbNoRef(i, j+1) * NumVal(i, j+2) * CRIR_{SMX} \\ + ProbAff(i,j) * \sum_{l=j}^{1} (\prod_{k=j}^{l} NumRef_k) * [CMIR_{SMX} \\ + (Existential_i * ProbUnSat_{i,j} + Universal_i * ProbSat_{i,j}) \\ * NumRef_l * MultVal(i,j) * NumVal(i,l) * CRIR_{SMX}] \end{array} \quad (15)$$

6 Performance Evaluation

In order to compare the performance of the SMX index organization with that of traditional index organizations, we used the cost models presented in Section 5 to drive an analytic comparison of the organizations using a mathematical function plotting package. Because the SMX organization stores only 1 additional bit of information per data item over the MX organization, the costs of operating upon the index (i.e., storage, retrieval, insertion, and deletion of an index record) with the SMX organization can be considered to be nearly identical to those of the MX organization. Our evaluation instead focuses upon the comparative costs of calculating the effects of an update in the presence of a path query view with the various index structures.

Figure 7 presents factors that we identify as particularly relevant to our cost models. The columns of Figure 7 correspond to the types of index organizations we evaluate, and the rows represent parameters. The "faces" indicate the impact of the parameter upon the cost of calculating the effect of an update to object $O_j \in C_j$ with the corresponding index organization, given a virtual class based upon the path query $PQ_i = C_{i,1}.[\Theta]A_{i,2} \ldots [\Theta]A_{i,j} \ldots [\Theta]A_{i,n}\theta value$. In this section we present the results of our main analytic experiments that evaluate the effects of varying these parameters.

	PX	NX	MX	SMX
j->1	☹	☹	☹	🙂
j->n	🙂	🙂	☹	😕
n large	☹	☹	☹	😕
Fork (MVA)	☹	☹	☹	😕
Converge (mult. refs.)	😐	☹	☹	😕

LEGEND	
🙂	Reduces cost
😐	Cost unaffected
😕	Increases cost if inst.'s sat. changes
☹	Increases cost

Figure 7: Parameters that affect cost of maintaining PQV_i.

- **j $- >$ 1.** The closer the updated object is to the head of the path, the higher the cost of traversing forward from it to its endpoints becomes. This affects the traditional index organizations, but does not affect the SMX organization.

- **j $- >$ n.** The closer the updated object is to the end of the path, the lower the cost of traversing forward from it to its endpoints becomes. This affects the traditional index organizations, but does not affect the SMX organization. However, when O_j 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 all updates with the MX organization and increases the cost of evaluating updates where $SatInd(O_j)$ changes with the SMX organization.

- **n large.** As the path increases in length, both the cost of traversing forward to identify endpoints and the cost of traversing backwards using MX structures to identify headpoints increase. The increased cost of forward traversal increases the cost of evaluating all updates with traditional organizations. The increased cost of finding headpoints increases the cost of evaluating all updates with the MX organization, and increases the cost of evaluating those updates where $SatInd(O_j)$ changes with the SMX organization.

- **Fork (MVA).** If due to multi-valued attributes, an object O_k ($1 \leq k \leq n$) can lead to multiple endpoints, then the costs of finding both headpoints and endpoints increases. This increases the cost of evaluating all updates with traditional organizations, but only increases the cost of evaluating those updates where $SatInd(O_j)$ changes with the SMX organization.

- **Converge (Mult. refs.).** If, due to multiple objects referring to a given object O_k as a value for their $C_{i,k-1}.A_{i,k}$ attribute ($2 \leq k \leq n$), a given object O_k can lead to multiple headpoints, then the costs of finding headpoints increases for the NX, MX, and SMX organizations. This increases the cost of evaluating all updates with the NX and MX organizations, but only increases the cost of evaluating those updates where $SatInd(O_j)$ changes with the SMX organization.

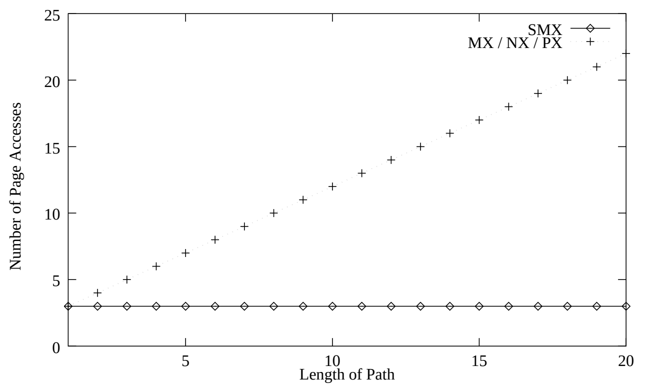


Figure 8: Creating a new head object while varying the length of P_i.

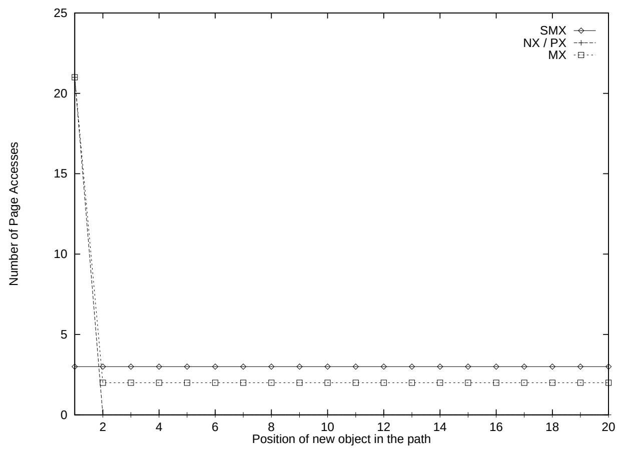


Figure 9: Creating a new object O_j while varying path positions (varying j).

6.1 Object Creation

If the new object is at the head of the path, then we would expect the SMX organization to outperform traditional index organizations because we can use the SMX *satisfiability indicators* to avoid traversing forward from the new object to its endpoints in order to determine whether or not it qualifies for membership in PQV_i. The cost of the forward traversal depends on both the length of the path and on the characteristics (number, position, fan-out) of multi-valued attributes included in the path. For the sake of simplicity, however, we constrain our current comparison to paths that do not include any multi-valued attributes.

Figure 8 compares the performance of the PX/NX, MX, and SMX organizations when calculating whether or not a newly-created object O_j qualifies to belong to a path query view PQV_i based on path $P_i = C_{i,1}.A_{i,2}.A_{i,3} \ldots A_{i,n}$, where O_j is at the head of the path (i.e., $j = 1$). The vertical axis of the graph indicates the number of page accesses required for this calculation. The horizontal axis of the graph indicates the length of P_i (i.e., the value of n). Because the creation costs for the PX, NX, and MX organizations are identical in this case, we plot them as a single function. As is illustrated by the graph, even in the absence of multi-valued attributes, the cost of calculating the effect of the creation remains constant with the SMX organization, while the cost increases in proportion to the length of the path with the other organizations.

Figure 9 compares the cost of calculating whether or not a newly-created object O_j qualifies to belong to PQV_i while varying the position of O_j in the path (j). As indicated by our cost model equations, if O_j is not at the head of the path, the object creation cannot possibly affect the membership of the path query view, and thus the only thing that must be done is to update the relevant index structures. This cost is constant for all of our index organizations—nothing needs to be done for the NX or PX structures (because no head object is affected), and the MX and SMX structures only need to update the $C_j.A_{j+1}$ index structure. Note that although the SMX is slightly more expensive because it must retrieve an extra index record for $O_j.A_{i,j+1}.A_{i,j+2}$'s value if no previously existing object referred to $O_j.A_{i,j+1}$, this expense is at most only one additional page access.

6.2 Object Deletion

If the path is simple, then with the NX and PX organizations, our primary cost is that of traversal forward from O_j to identify the associated endpoints. The primary cost of the MX and SMX organizations, on the other hand, is that of traversing backwards from O_j to identify its associated headpoints. Figure 10 compares the cost of evaluating the deletion of object O_j from a simple path of length 10 while varying the position of O_j from $j := 1$ to n. Note that there are two lines plotted for the SMX organization—one representing a 10% probability and one assuming a 20% probability that $SatInd(O_j)$ will change due to the deletion. Because the path is simple (no forks and no convergence), the costs of the PX and NX organizations are identical. As we would expect, the cost of evaluating the deletion with the PX and NX organizations decreases as j approaches n while the cost with the MX and SMX organizations increases. Because the SMX organization requires traversal only when $SatInd(O_j)$ changes, we can see that the lower that probability, the better SMX's performance.

If one of the attributes along the path, say $C_{i,5}.A_{i,6}$, is multi-valued, then the cost associated with the NX organization increases because now each endpoint is associated with multiple headpoints, each of whose instantiations must be traversed

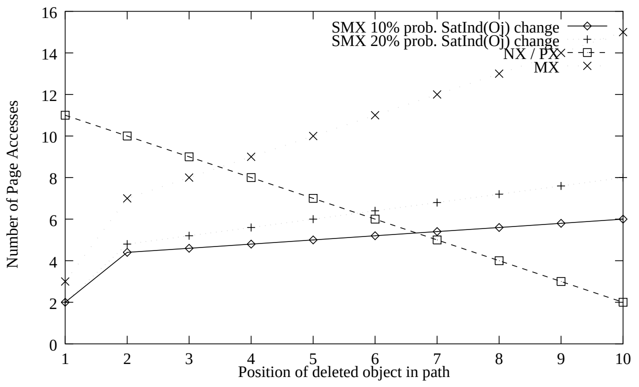


Figure 10: Deleting an object O_j from a simple path while varying j.

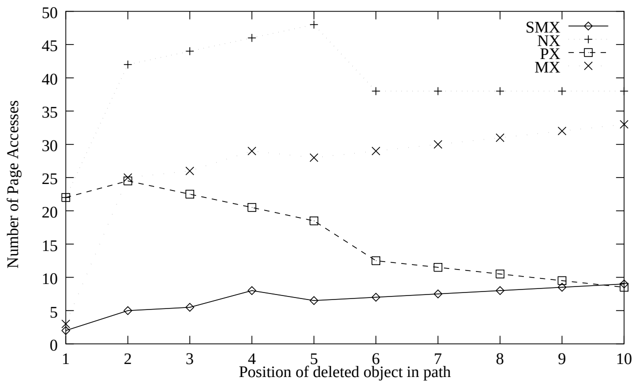


Figure 11: Deleting an object O_j from a path with one multi-valued attribute (forward fork) while varying j.

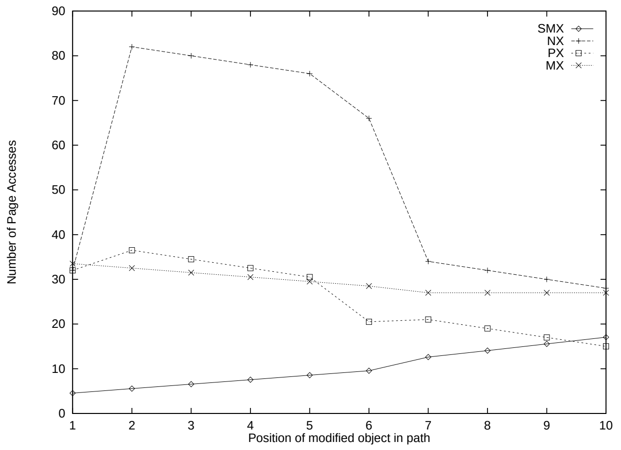


Figure 12: Modification (varying j).

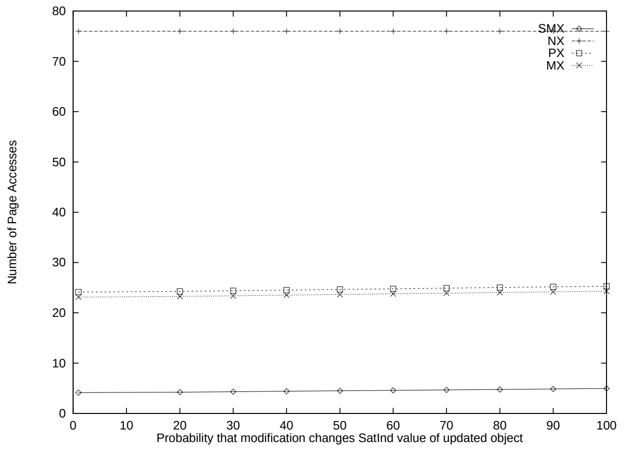


Figure 13: Modification (varying ProbUnSat).

to their jth element in order to identify which involve O_j. In addition, if $j < 6$, then the cost of evaluating the update with the traditional organizations increases because we must perform additional traversals to identify alternative paths that satisfy the predicate. Figure 11 compares the cost of evaluating the deletion if each object in $C_{i,6}$ has as a value for attribute $C_{i,5}.A_{i,6}$ an average of 2 objects from $C_{i,5}$, and a probability of 25% that the view is affected. Because the SMX organization eliminates the need for traversals to identify alternative paths, its performance is overall the best.

6.3 Modification of Attribute Values

If $C_{i,5}.A_{i,6}$ is both multi-valued and has multiple-references to its values, then for traditional index organizations the cost of evaluating the modification of an object O_j so that its $A_{i,j+1}$ attribute is set to object O_{new} of class $C_{i,j+1}$ instead of object O_{old} of class $C_{i,j+1}$ involves both forward traversals to identify the endpoints of O_{old} and O_{new}'s instantiations, and, for NX and PX, extra traversals in order to handle multiple headpoints being associated with each endpoint. (We can use the MX indices to explicitly identify the headpoints that indirectly refer to an updated object.) The SMX organization is primarily affected by the cost of propagating the update when $SatInd(O_j)$ changes.

Figure 12 compares the cost of evaluating the modification of object O_j from a path of length 10 where $C_{i,6}.A_{i,7}$ is both multi-valued (2) and has multiple-references (2) to its values while varying the position of O_j from $j := 1$ to n. We assume a 50% probability that $SatInd(O_j)$ will change due to the modification and a 5% probability that O_{new} does not have a previously existing entry in an SMX index. As we would expect, Figure 12 demonstrates that even with only one minimal multi-valued attribute and only one minimal multiple-reference, the SMX organization outperforms the traditional index organizations for path queries (with up to 10 components in the path predicate) that involve only one minimal multi-valued attribute and even one class of objects that is minimally multiply-referenced.

Figure 13 fixes j at position 5 for a path query of length 10 and compares the cost of evaluating the modification of O_j while varying the probability that $SatInd(O_j)$ will change due to the modification from 0% to 100%. As demonstrated by the figure, even with the probability of a change set to 100%, because SMX avoids the need to perform full forward traversals, it out-performs the traditional index organizations.

7 Extensions to the SMX Organization

The SMX organization as described thus far offers a number of advantages. We can use the SMX index to determine whether or not a path instantiation satisfies a path query predicate without doing any forward traversals. The SMX organization facilitates the incremental maintenance of path query views, even if the path includes multi-valued attributes. SMX indices need to be maintained in the face of an update if and only if the update actually affects the status of the instantiation with regard to the predicate. In this section we present a number of extensions to our SMX organization. These extend the SMX organization to:

- handle multi-valued attributes in an efficient manner;
- take into account classes that appear in multiple positions in a path (e.g., cycles in the query's path);
- share index structures among multiple path query views.

7.1 Multi-Valued Attributes

As discussed in Section 4, given a path query PQ_i containing a multi-valued attribute $C_{i,j-1}.A_{i,j}$, if an object belonging to class $C_{i,j-1}$ changes one of its values for $C_{i,j-1}.A_{i,j}$ then in order to determine whether or not the updated object's *satisfaction indicator* should change value we might have to retrieve the *satisfaction indicators* for the updated object's other values for $C_{i,j-1}.A_{i,j}$. If the updated object's *satisfaction indicator* does change, then as we propagate the change backwards, we must similarly check the alternative values of any other multi-valued attributes encountered.

For example, Figure 14 depicts a path query view *PathSelect2* defined by the path query `Car.maker.∃owner.age` $<$ `50` (using the schema and instances from Figure 2). Initially all three cars belong to the *PathSelect2* class. If *person3* were to change his `age` from 40 to 55, then upon determining that $SatInd(person3)$ will change from *True* to *False*, we must retrieve the *satisfaction indicators* for all other values of *company2*'s `owner` attribute in order to determine the correct value for $SatInd(company2)$.

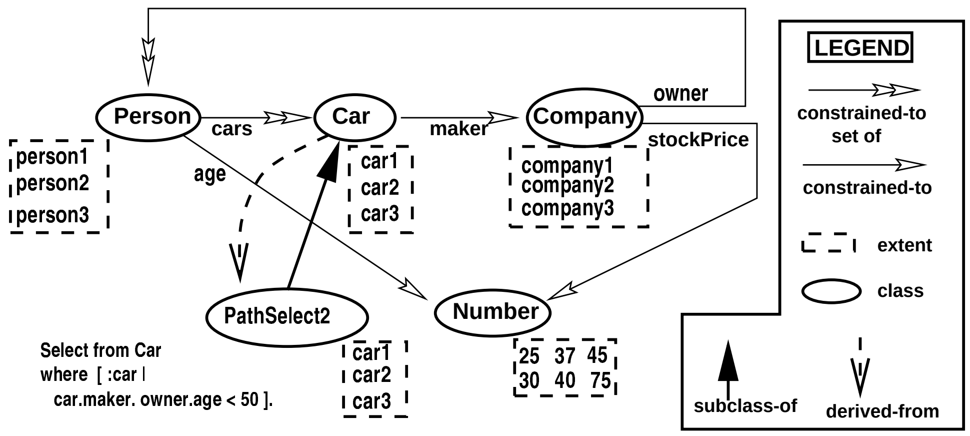


Figure 14: The *PathSelect2* class uses an existential quantifier.

We can avoid these additional retrievals if, instead of using a boolean bit, we represent the *satisfaction indicator* of an object belonging to class $C_{i,j}$ using a bit string that *counts* the number of positive *satisfaction indicators* referenced by the object's values for attribute $C_{i,j}.A_{i,j+1}$. For example, Figure 15 depicts the SMX index that would be constructed for the path query of Figure 14.

Company	Car
KEY	VALUE
company1	{1, {car1}}
company2	{2, {car2}}
company3	{1, {car3}}

Index on Car.maker

Person	Company
KEY	VALUE
person2	{1, {company2, company3}}
person3	{1, {company1, company2}}

Index on Company.owner

age	Person
KEY	VALUE
45	{1, {person1}}
77	{1, {person2}}
40	{1, {person3}}

Index on Person.age

Figure 15: Using counters to represent *satisfaction indicators*.

The advantage of extending the *satisfiability indicators* to "count" the number of *satisfying subpath instantiations* reachable from a given object is that we can thus avoid having to individually look up all the values of a multi-valued attribute. However, the disadvantage is that now we must update the value of the counter even if the status of the object has not changed in regard to whether or not it participates in a satisfying subpath instantiation. For example, if *person3* changes his age to 55, we could immediately identify that although this update would cause the *satisfiability indicators* for *company3* and *car3* to be decremented to 0, the *satisfiability indicator* for *company2* would only be decremented to 1. Because 1 is positive, we would be able to recognize that *SatInd(company2)* would remain positive without any retrieving any of *company2*'s other values for the multi-valued attribute `owner`. In addition, an increased amount of storage is needed to represent the *satisfiability indicators* as counters. However, these costs are reasonable if large multi-valued sets are expected as values for multi-valued attributes, because this strategy replaces the need to perform individual look-ups of the items in these sets with a look-up of a single number.

7.2 Incorporating Cycles

The SMX strategy as originally proposed requires that the query path be free from cycles, in that it associates each indexed object with only a single *satisfaction indicator*. However, a given path query $PQ_i = C_{i,1}.[\Theta]A_{i,2}.[\Theta]A_{i,3}\ldots[\Theta]A_{i,n}\theta value$ could include a given class/attribute $C_j.A_k$ in multiple positions. For example, Figure 16 depicts a simple schema in which a virtual class *PathSelect3* is defined using the query `Person.car.mechanic.car.pri` $<$ `1000`. We cannot use the SMX solution as presented to maintain this class because a given `Person.car` entry could lead to either a positive or negative *satisfaction indicator*, depending on its position in the path. E.g., the *satisfaction indicator* associated with *car1* should be positive if it appears as the second item in the path (`person1.car1.person2.car2.800`), but should be negative if it appears as the fourth item in the path (`person2.car2.person1.car1.5000`).

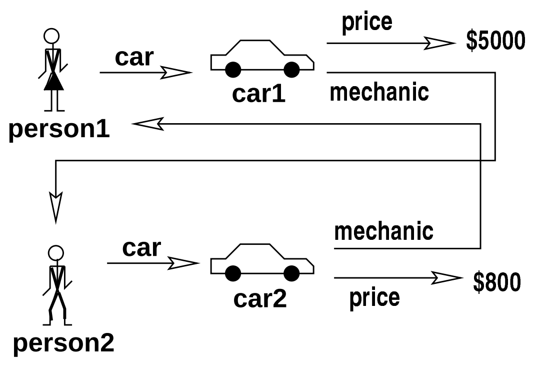
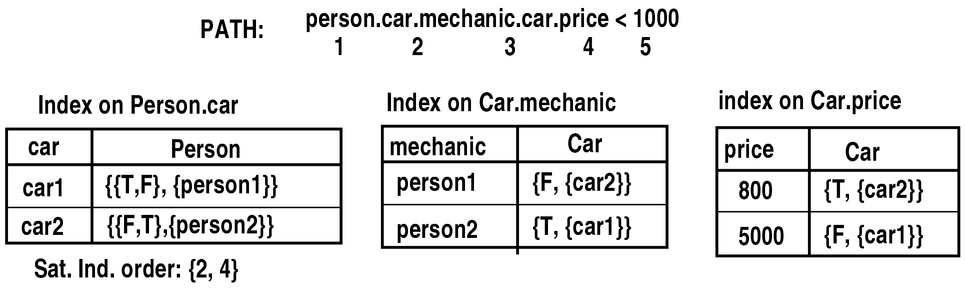

price
$5000
car
car1
mechanic
person1
mechanic
car
car2
price
$800
person2
PathSelect3 := select from Person where
[:p | p.car.mechanic.car.price < $1000]

Figure 16: The *PathSelect3* class includes the Person.car attribute in two positions.

We can extend the SMX structure to handle classes that appear multiple times in a path query by ordering the *satisfaction indicators* of each entry according to path position and associating the index with a key correlating the order with path position. Figure 17 shows the SMX structures that would be associated with the instances of Figure 16. We would correspondingly extend our definition of the $SatInd()$ function to take an optional additional parameter of position in the path. If, for example, *person2* were now to change the value of its `car` attribute to *car1* instead of *car2* then we would be recognize that $SatInd(car1, 4)$ returns `false`, and thus that the *satisfaction indicator* of the *car1.person2* entry in the *Car.mechanic* index should be updated to *false*, which in turn causes the position 2 *satisfaction indicator* of the *person1.car1* entry in the *Person.car* index to change to `false`.



Index on Person.car

car	Person
car1	{{T,F}, {person1}}
car2	{{F,T},{person2}}

Sat. Ind. order: {2, 4}

Index on Car.mechanic

mechanic	Car
person1	{F, {car2}}
person2	{T, {car1}}

index on Car.price

price	Car
800	{T, {car2}}
5000	{F, {car1}}

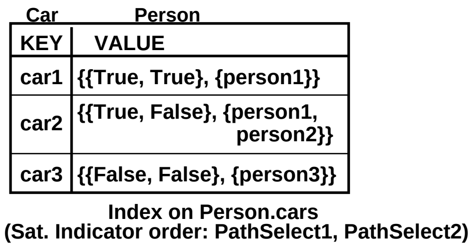
Figure 17: We can extend the SMX structure with an ordered list of *satisfaction indicators*.

7.3 Sharing SMX Index Structures

Given a class C_j and an attribute A_{j+1} of class C_j, $C_j.A_{j+1}$ could appear in multiple path queries. The relationship between any two of these path queries must be one of the following:

1. The two path queries could share the subpath $C_j.A_{j+1} \ldots A_n$ and end on an identical predicate on A_n, in which case the two views can share *satisfiability indicators* for the $C_j.A_{j+1} \ldots C_{n-1}.A_n$ indices.
2. The two path queries could share the subpath $C_j.A_{j+1} \ldots A_n$ and end on different predicates on A_n, in which case the two views cannot share *satisfiability indicators*, but the indicators can be ordered to exploit the subsumption of predicate conditions in such a way as to facilitate possible early termination of evaluation.
3. The two path queries could share the subpath $C_j.A_{j+1} \ldots A_k$, $j < k < n$, in which case the two views cannot share *satisfiability indicators*.

In order to maximize the sharing of index structures for cases 2 and 3, we propose to extend the SMX organization to associate multiple *satisfiability indicators* with each $C_j.A_{j+1}$ index entry. The disadvantage of such an extension is that it increases the storage space needed for each *satisfiability indicator*, but the advantage is that now the multi-index portion of the SMX structure can be shared amongst multiple path query views. For example, Figure 18 shows the SMX index for `Person.cars` that would result if we were to extend the schema in Figure 1 with an additional class *PathSelect2*, defined using the query `select from Company where [:company | company.∃owner.car = car1` (so that the two paths share the `Person.cars` index structure).



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

Index on Person.cars
(Sat. Indicator order: PathSelect1, PathSelect2)

Figure 18: Sharing the `Person.cars` SMX structure for two virtual classes.

8 Related Work

Project-Select-Join Views. Although traditional relational databases 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 object-oriented aggregation hierarchy. The relational equivalent of a path query is a select-project-join (SPJ) query, in that finding tuples from a different table that join with a given tuple is similar to the act of traversing a given object's attribute link.

Note that path query views, as defined in this paper, are less powerful than the more general select-project-join views. We do not yet support aggregate functions, and we assume constants in predicate comparison functions (as opposed to comparisons with other components of the path). However, in the future we hope to adapt our SMX solution to address these more complex problems. A common expense of relational SPJ and object-oriented path query views is the cost of finding a join match in a relational table / following a reference. Our SMX solution allows us to avoid such traversals.

A number of researchers have investigated the problem of maintaining materialized SPJ views. Gupta and Blakeley present formal partial-information-based view maintenance techniques that infer knowledge about the state of underlying base relations using local information (such as the view definition, the update, the current view materialization, and varying amounts of base relation replicas). They identify classes of materialized views according to the amount of information needed to maintain them in the face of updates [5], and demonstrate necessary and sufficient conditions for determining the amount of information needed to update a materialized select-project-join 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 [17]. A join pattern relation is a precomputation of complete (all join attributes instantiated) and incomplete (not all join attributes instantiated) joins that satisfy the constraints of a rule. Join pattern indexing represents join relationships between existing tuples. These join patterns are similar to our satisfaction indicators in that they facilitate the incremental maintenance of materialized rule-derived data by marking data records that satisfy rule constraints. Join pattern information more closely resembles a path index than a multi-index, and thus needs to be maintained in the face of all updates (unlike the SMX satisfaction indicators).

Shekita and Carey selectively replicate individual data fields in order to improve query processing performance by eliminating some functional joins [18]. Although the problem of maintaining the consistency of these replicated fields does not benefit from our satisfiability indicators because the replicated fields represent endpoint objects (as opposed to the head point objects materialized in our path query views), the task is similar to that of maintaining materialized path query views in that both tasks require a means of inverting the queried path in order to handle updates to references along the path. Shekita and Carey create special link objects to maintain inverse mappings that associate objects with the objects that reference them [18]. These link objects are like the entries in a multi-index, except that the database objects maintain references to the link objects and the link objects maintain references to the link objects that precede them along the aggregation graph. However, because such references are implemented as stored oids rather than as direct pointers, following one link object to another requires a retrieval of the link object by oid, which is equivalent to the retrieval of a multi-index record in our SMX solution.

Konomi et al. [10] 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 also considered the work of previous researchers who have compared index structures when designing our SMX organization. In particular, the work of Bertino et al. informed us of the costs and issues involved in the performance of basic operations with traditional index organizations [2, 4]. However, note that traditional indexing techniques have the goal of supporting the evaluation of queries. Our primary indexing goal, on the other hand, is to reduce the overhead of propagating an update on a single object to materialized path query views.

9 Conclusions and Future Work

To the best of our knowledge, ours is the first work to identify and address the specific needs of the path query view problem for object-oriented databases 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 index organizations. The SMX organization can be used to eliminate these forward traversals. We also present cost models and compare the performance of the SMX organization with regards to calculating the effects of updates upon views to that of the multi, nested, and path index organizations. The results of our evaluations indicate that the SMX dramatically improves upon the performance of traditional index structures with respect to the problem of path query view maintenance.

Future Work. Although the *MultiView* model currently supports multi-valued attributes, it does not yet support the definition of views using aggregation functions over multi-valued attributes (e.g., `sum`, `max`, or `min`). Support for such views would be a valuable contribution to *MultiView*'s functionality. Furthermore, although a number of researchers have studied the problem of maintaining materialized aggregation functions in relational databases [15, 6], to the best of our knowledge, this problem has not been examined in an object-oriented context. Because the object-oriented model provides for the definition of collection classes, an OO approach to this problem might extend the construct of the collection class with an instance variable that would store the result of the aggregation function (similar to a refine view) and an indicator (like our satisfaction-indicators from the path query view problem) that represents the status of the aggregation value instance variable.

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] J. Blakeley, P. Larson, and F. Tompa. Efficiently updating materialized views. *SIGMOD*, pages 61–71, 1986.

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

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

[6] V. Harinarayan, A. Rajaraman, and J.D. Ullman. Implementing data cubes efficiently. *SIGMOD*, page XXXX, 1996.

[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] W. Kim and J. Seo. Classifying schematic and data heterogeneity in multidatabase systems. *IEEE Computer*, pages 12–18, 1991.

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

[11] 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.

[12] H. A. Kuno and E. A. Rundensteiner. Augmented inherited multi-index structure for maintenance of materialized path query views. In *ACM Research Issues in Data Engineering Workshop*, March 1996.

[13] H. A. Kuno and E. A. Rundensteiner. The *MultiView* OODB view system: Design and implementation. In Harold Ossher and William Harrison, editors, *Accepted by Theory and Practice of Object Systems (TAPOS), Special Issue on Subjectivity in Object-Oriented Systems*. John Wiley New York, 1996.

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

[15] D. Quass. Maintenance expressions for views with aggregation. *Proceedings of the SIGMOD Workshop on Materialized Views: Techniques and Applications*, pages 110–118, 1996.

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

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

[18] E. J. Shekita and M. J. Carey. Performance enhancement through replication in an object-oriented dbms. *SIGMOD*, pages 325–336, 1989.

[19] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *SIGMOD*, 1995.