The Effects of Mutability on Querying

Mitch Cherniack*
Brown University,
Providence, RI 02912
Phone: +1 (401) 863-7600
email: mfc@cs.brown.edu

Eui-Suk Chung†
M.I.T. Laboratory for Computer Science,
Cambridge, MA 02139
Phone: +1 (617) 253-3058
email: euisuk@lcs.mit.edu

Abstract

The ability of an object to mutate is one of the defining characteristics of Object-Oriented Databases (OODBs). But the impact of object mutability on querying is typically ignored. Presumably, this is because queries are usually assumed free of side-effects, and therefore objects are effectively immutable while they are queried. But mutability affects other object behaviors aside from how they change, such as how they are compared for equality, and if it is possible to detect that they are shared. In this paper, we show how these behaviors impact querying in general and query optimization and mutable set construction in particular.

The issues considered in this paper arose during our efforts to provide OQL-based [Cat96] query facilities for Thor [LJGS90, LAC+96]. Whereas OQL (and specifically, the ODMG object model) make specific assumptions about the mutability of sets, Thor had not adopted a policy for sets when this project began. In formulating a policy, we considered the impact of the ODMG policy on query efficiency and then considered how Thor could reconcile different assumptions about set mutability with OQL. The analysis of these issues is the focus of the paper. This analysis, though done with respect to Thor, has implications for any system that wishes to adopt the ODMG standard (as many have pledged to do).

Keywords: {Querying, Query Optimization, Object-Oriented Databases, Mutability}

1 Introduction

One of the tenets of object-oriented data models requires that an object's existence be independent of its value [ABD+89]. This means that objects are potentially *mutable*, as mutation does not require that an object abandon its identity.

*Partial support for this work was provided by the Advanced Research Projects Agency under contract N00014-91-J-4052 ARPA order 8225, and contract DAAB-07-91-C-Q518 under subcontract F41100.

†The author is currently visiting MIT. His home organization address is: Ericsson Utvecklings AB, BOX 1505, 125 25 Älvsjö, Stockholm, SWEDEN. Email: EuiSuk.Chung@uab.ericsson.se. The author was supported by a grant from the Swedish Marcus Wallenberg Foundation for Scientific Research and Education.

Though an important feature of object-oriented databases, as far as we know the impact of mutability on querying has not been considered before. One can be lulled to believe that mutability has no effect on querying, as queries are typically assumed to be free of side-effects, and therefore mutable objects are effectively immutable during a query's evaluation. However, whether or not an object is mutable affects other aspects of the object's behavior aside from its ability to change.

- *Equality:* Two objects should be considered equal only if they are indistinguishable. Mutable objects should be indistinguishable even by mutation (i.e., *identical*), as otherwise, sets that contain these objects can have unintuitive behavior such as a cardinality that changes as a side-effect of member mutation [CZN95]. On the other hand, immutable objects (such as literals or tuple objects) may be compared using other means (such as *keys* for tuples or memberwise equality (*extensionality*) for immutable sets).

- *Sharing:* Representations of both mutable and immutable objects can be *shared* (i.e., referenced by multiple objects), but only sharing of mutable objects can be detected at the user level. Sharing of immutable objects is undetectable because detection requires viewing the effects of mutation [LG86].

Equality and sharing influence the correctness of transformations performed during query optimization, and the ease by which new mutable sets denoted by queries can be constructed.

In this paper, we explore the impact of object mutability on querying. We begin in Section 2 by describing the context that forced consideration of this issue; our efforts to provide OQL-based [Cat96] query support for Thor [LAC+96]. Thor's object model did not, until recently, include support for sets. The ODMG object model, which underlies OQL's semantics, makes specific assumptions regarding set mutability. We consider the impact of these assumptions on querying in Section 3, and then propose ways that OQL and Thor could be reconciled if Thor were to adopt a model that differed from that of ODMG. This discussion, though specific to sets, applies equally well with respect to other collection types such as lists and bags. In Section 4, we describe relaeted work and we conclude in Section 5.

2 The Thor Query Project

This section describes our plan to provide OQL-based [Cat96] query support for Thor [LAC+96, LCD+94]. In Section 2.1, we describe the components of the integration project and present an architecture that reveals how these components will interact. In Section 2.2 we introduce the integration issues that arose during initial work on the project.

2.1 A Loosely-Coupled Architecture

Thor is an object-oriented database system (OODB) that provides access to persistent objects across distributed and heterogeneous platforms. Currently objects are accessible by navigating from externally visible *root* objects. The goal of the integration project described is to provide an associative query facility for Thor, to complement its navigational access capability. ODMG-93 [Cat96] has been proposed as a standard for OODB object definition, manipulation and retrieval, and includes a declarative object query language (OQL). OQL's semantics are defined in terms of the ODMG object model which has much in common with Thor. We settled on OQL as a query language in recognition of it as an emerging standard.

As a first step, we chose a loosely-coupled approach to integration. This approach maintains a distinction between OQL and programming languages that are used to access Thor objects, much like ODMG approaches language bindings to C++ and Smalltalk. (The alternative, tightly-coupled approach would integrate OQL within the languages themselves.) Loose coupling allows integrated components (such as the optimizer, query parser, Theta compiler and object retrieval modules) to be incorporated as is, making the interfaces between them responsible for ensuring smooth interaction. Once we have a better understanding of interaction issues, we will move to a tighter integration. In the meantime, we will construct a working query facility without disrupting existing component implementations. Note that the loosely coupled approach forces us to delay the type-checking and optimization of queries until run-time.

Figure 1 shows an overview of the loosely-coupled system architecture we propose to implement. The interface between the client program and Thor is called a *veneer* [Bog94, LAC+96]. Veneers provide a number of services related to transaction management and data translation (between programming languages and Thor). In order to support queries, veneers have been augmented with functions to transport queries and their results between clients and Thor.

A query is issued by a client and is passed to Thor as a string (1). A query dispatcher receives the query and invokes a translator that maps the query into a KOLA expression (2). (KOLA [CZ96, Che95] is a query algebra that will be used to underlay the optimizer's representations of queries.) The translator parses and type checks the query and produces a KOLA tree (3), using type information obtained from the schema library (9). The KOLA tree is then passed to the optimizer (4), which performs both source-to-source (algebraic) transformations and mappings to physical plans. The algebraic optimizer takes a KOLA tree (4a) and returns an equivalent tree (4b) that is more efficient to evaluate. The physical plan generator uses the optimized tree (4c) to produce concrete execution plans (4d, 5). Finally, the plans are evaluated by the evaluator (6) and a result (actually a handle to the result object) is sent back to the client (7,8).

The current prototype implements a subset of OQL (e.g. we provide no support for bags or lists) and does not yet perform query optimization. Rather, the evaluator evaluates the KOLA tree directly. We expect to have an algebraic optimizer based on KOLA in the near future. A physical optimizer as well as index support [Hwa94] will be added at a later point.

2.2 Integration Issues

OQL semantics are defined in terms of the ODMG object model. Consequently, any OODB that wishes to provide OQL-based querying must either adopt the ODMG model, or define its own model and decide on ways to reconcile between the two. Reconciliation would be required everywhere translation occurs between the two object models. For example, objects

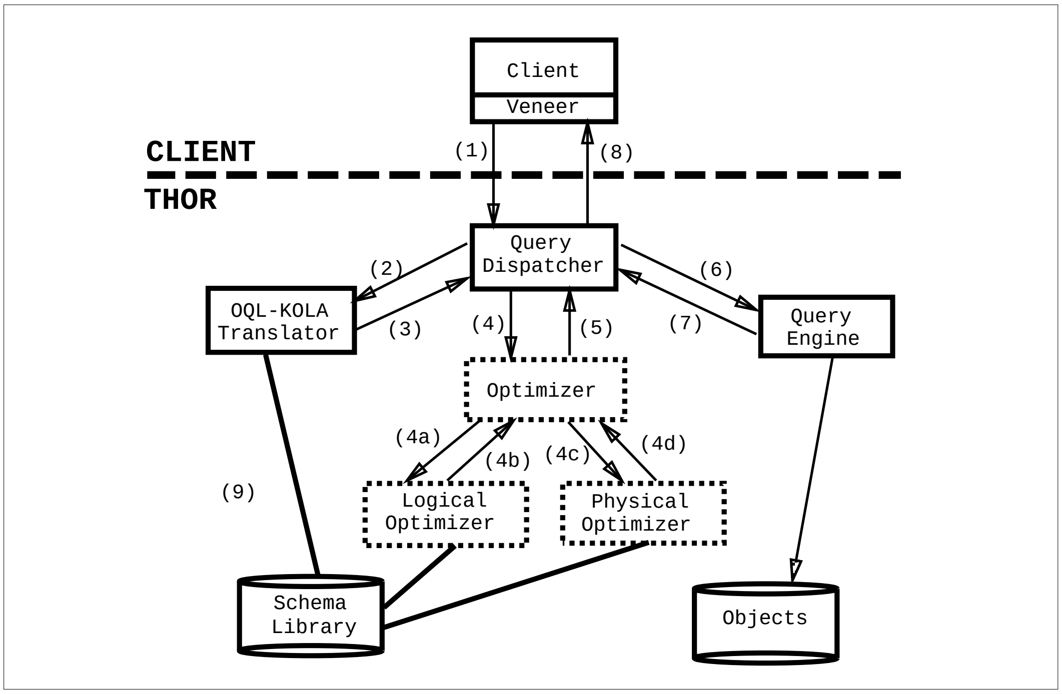

Figure 1: An Overview of Query Processing in Thor

generated by OQL queries and made persistent must be given interpretation in the object model of the OODB. Similarly, objects in the OODB must be interpreted in terms of the ODMG object model when passed to OQL queries as input.

The object models assumed by OQL and Thor are largely compatible but there is potential disagreement with respect to the mutability of sets. ODMG supports both mutable and immutable sets. At the time of the project's conception, Thor had yet to commit to a policy for sets. We had a chance to consider two approaches. We first investigated the implications of ODMG's approach with respect to querying efficiency. We later considered how Thor could provide support only for mutable sets, while still accepting OQL queries. Both analyses follow.

3 The Impact of Mutability on Querying

We begin in Section 3.1 by describing the ODMG approach to sets and set mutability. We show how the ODMG approach affects query optimization and the construction of mutable sets from immutable sets. We show that mutable set construction is problematic given this approach, and in Section 3.2, we consider ways that an alternative approach to Thor sets could be reconciled with ODMG's approach, to permit OQL-based querying.

3.1 The ODMG Approach to Sets

OQL's semantics depend heavily on the ODMG data model, which takes the following approach to sets:

- Sets can be either mutable or immutable.
- Mutable sets are constructed through calls to explicit constructors. Construction can initialize the contents of a mutable set using the members of an immutable set.
- Queries of the form `select distinct` ... `from` ... `where` ..., (which are hereafter referred to as `select distinct` queries) return immutable sets.

Mutable sets act as queryable containers whose members can vary over time. Immutable sets make OQL a superset of SQL, whose relation-based semantics can be expressed in terms of immutable sets and bags of tuples.

Until recently, Thor had not established a policy regarding set mutability. We considered adopting ODMG's approach to simplify the integration of OQL and Thor. But first we wanted to consider the impact of this approach on the efficiency of querying. As far as we know, the issues raised here have not been addressed before.

3.1.1 Mutability and Query Optimization

Query optimization requires *transforming* a query into an alternative form that lends itself to efficient evaluation. Transformations must be correct, in that a transformed query should return an object that is *equal* to what would have been returned by the original query.[1] In other words, it should not be possible for a user to probe the object returned by a query to detect that a transformation has taken place.

Mutability restricts what can be called a correct transformation. To illustrate, in Figure 2 we show a transformation expressed in terms of OQL. We use the notation $Q_1 \stackrel{\rightarrow}{=} Q_2$ to denote a transformation where Q_1 and Q_2 are equivalent queries. This transformation is correct in the relational world (where all sets/relations are considered immutable), and is useful as it eliminates the need to iterate through the members of `A`. However, the transformation is incorrect if `A` is a mutable set. According to OQL semantics, the query on the left constructs a new immutable set containing the elements belonging to `A`. This result can be distinguished from the result returned by the query on the right by mutating `A`. If the effects of mutation are visible in the set returned by the query, then the transformation has occured.

This transformation would be incorrect, even if OQL's `select distinct` queries constructed new mutable sets. This is because the semantics of queries depend on the objects that share them. The transformation would replace a query that constructs a new set with a query that returns the named set, `A`. Whether or not the transformation took place can be

[1] In fact, transformation correctness need not be based on equality if the query concerned constructs a new mutable object. In this case, *similarity* (as opposed to identicalness) of results suffices to establish correctness. This issue is considered in [dBG92] and elsewhere.

```
select distinct x      ⊒   A
from x in A
```

Figure 2: A Query Transformation Rule over OQL

detected by mutating A and viewing the effects of the mutation in the set returned by the query. If the effects are visible, the transformation has occured.

Transformation correctness is restricted not only by sharing, but equality. Mutable objects should be equal only when identical. Mutable object constructors are thus nondeterministic, as two successive calls of a mutable object constructor (as in, `Person (name: ''John'')` which constructs a new Person object) result in unidentical and hence unequal objects.

Mutable object constructors can be invoked in OQL queries, as in a query that constructs new Person objects for every record in a birth registry:

```
select distinct Person (name: y.name)
from y in Birth_Reg
```

This query assumes the schema of Table 1. During optimization, queries that invoke mutable object constructors must be treated differently from those that do not, as we illustrate with the transformation of Figure 3.

```
select distinct f (y)
from y in (select distinct g (x)    ⊒   select distinct f (g (x))
           from x in A)                 from x in A
```

Figure 3: A Query Transformation Rule over OQL

This transformation is useful as it transforms a query to one that does not require construction and iteration through an intermediate set result. Provided that operator `f` is deterministic, the transformation is correct. But if `f` is nondeterministic, then the transformation may be incorrect as shown below:

```
select dist Person (name: y)
from y in (select dist x.name    ⊒   select dist Person (name: x.name)
           from x in Birth_Reg)      from x in Birth_Reg
```

The original query constructs a set of people with unique names, as duplicate names are removed from the intermediate result. But, the transformed query could construct a set that has two people with the same name, and therefore is not equivalent. In special cases the transformation is correct (e.g., when `g` is *injective* (one-to-one) as in a key attribute). In general though, this transformation must be conditional on whether or not `f` is nondeterministic. Thus, mutable object equality restricts opportunities for optimization by invalidating transformations that were correct over relations.

Type	Attributes	
`Person`	`name:`	`String`
	`age:`	`Integer`
`BirthRecord`	`name:`	`String`
	`father:`	`Person`
	`mother:`	`Person`

Name	Type
`Birth_Reg`	`immutable_set [BirthRecord]`

Table 1: An Example Schema

3.1.2 Mutability and Mutable Set Construction

Often it is useful to store query results in mutable sets. OQL permits queries to construct mutable sets from immutable sets that are used to initialize their contents. Unfortunately, potential sharing of immutable object representations makes it very difficult to implement this operation efficiently.

We considered two alternative representations of sets that are illustrated in Figure 4. In the first (a), a mutable set is represented by a pointer to an immutable set, which itself is based on a data structure such as an array or linked list. In this case, construction of a mutable set in terms of an immutable set simply requires allocating a new pointer. But this representation has serious drawbacks because representations of immutable sets may be shared. Sharing of immutable objects is undetectable at the user level. This can be exploited by the underlying system to improve performance, as immutable objects can be replicated at will without concerns for coherence. This means that immutable objects might not be shared in some cases (e.g, immutable objects might be replicated across a network to cut down on network accesses) but shared in others (e.g., a large immutable object might be shared at a given site to save space). Unfortunately, this means that multiple mutable sets may point to the same immutable set and therefore, even simple mutations of any one of these sets (such as inserting a new element) cannot be performed in place. Rather, a new immutable set representation must be constructed; a potentially costly operation given the potential size of sets.

A second implementation (b) has all sets represented by data structures such as arrays or linked lists, with additional bits used to differentiate mutable from immutable sets. In this case, mutation can occur in place and is inexpensive. But construction of mutable sets from immutable sets requires shallow copying and is costly. In certain cases the immutable set can be coerced to be mutable (by resetting the "mutability bit"), but only if no object points to the immutable set. This occurs for sets constructed by queries, although only certain queries construct new sets (e.g., `select distinct` queries usually do, but queries such as `Joe.children` do not). Optimization can interfere, as the transformation of Figure 2 could replace a query that constructs a new set with one that returns an existing set.[2] Thus, intelligent mutable set construction under this implementation would require knowledge of

[2]It should be remebered that the transformation of Figure 2 is correct if the set `A` is immutable. The difference in terms of *implementation*, between the two queries, is that the first constructs a new set representation whereas the second does not.

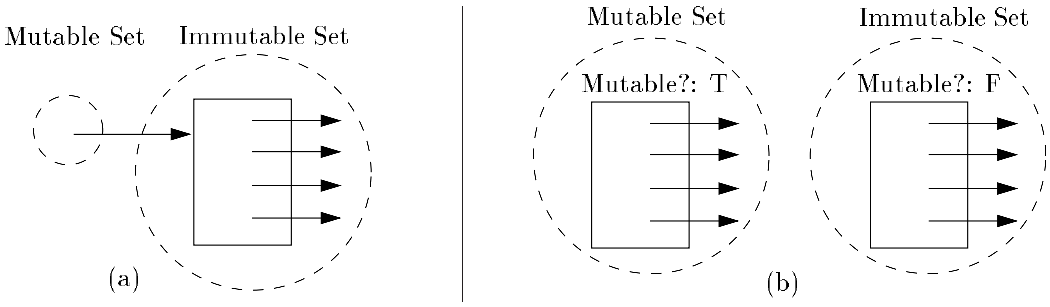

Figure 4: Two Possible Implementations of Sets

the underlying system's policies for replicating immutable objects and query optimization, and likely would still require shallow copying of immutable sets in some cases.

In short, the coexistence of mutable and immutable sets makes OQL expressive but introduces complications into query optimization and evaluation. Optimizers must consider whether input sets are mutable or immutable before applying transformations. Similarly, queries that construct mutable objects must be transformed differently than queries that construct immutable objects. Queries that construct mutable sets from immutable sets appear to be inherently inefficient, as only careful analysis reveals when copying is not required. This is because the immutable sets may be shared under the covers as a way of improving system performance.

These complications led us to consider an alternative model for sets in Thor. In the next section, we explore how we could reconcile a Thor data model that provided support only for mutable sets, with the ODMG model that permits sets to be immutable also.

3.2 An Alternative Approach: Are Immutable Sets Necessary?

An object model for Thor that only provides support for mutable sets would resemble the object models of GemStone [BOS91] and ObjectStore [LLOW91] (although query languages for these systems are far less expressive than OQL). Queries in this model would return mutable sets directly, and therefore mutable sets would never be constructed from immutable sets. The underlying data model would be simplified, while still permitting sets to be used as containers whose membership can vary over time.

In this section, we analyze techniques for reconciling OQL queries (whose semantics are such that they can construct immutable sets) with a version of Thor that supports only mutable sets. No apparent difficulty arises when one restricts queried sets to be mutable. But we must contend with the immutable sets that are constructed by OQL queries which, if made persistent in Thor, must be made mutable.

3.2.1 Approach 1: Redefine OQL Semantics to Construct Only Mutable Sets

The first approach we considered was to redefine OQL for our purposes to construct mutable sets whenever its ODMG-based semantics had it construct immutable sets. For example,

```
count (select distinct (select distinct p.age
                        from p in f)
       from f in Families)
```

Figure 5: A Problematic OQL Query: A Count of Age Sets

`select distinct` queries would be redefined to construct new mutable sets rather than immutable sets. Obviously, this approach requires changing the semantics of OQL. The issue is the degree to which this is true. That is, would the only difference between OQL queries evaluated over ODMG-compliant databases and over Thor be the mutability of sets that result, or would this change have more pervasive repercussions on the semantics of the rest of the query language? We demonstrate that the latter is true by way of an example query over *nested* sets.

The problematic query is shown in Figure 5 (`Families` is a mutable set of mutable sets of Person (Table 1) objects). The query is nested because the (*inner*) query that computes a set of ages is computed as part of the (*outer*) query over `Families`. The result of the overall query will be the number of distinct *age sets* of families in `Families`, where a family's age set is the set of ages of its members. Suppose that

`Families = { {Joe, Mary}, {Jack, Jill} }`,

where `Joe` and `Jack` are both 40 years old and `Mary` and `Jill` are both 35 years old. In this case, the result of this query over Thor will be different than the result of this query as determined by OQL's ODMG-based semantics. The interpretation of sets constructed by queries as mutable makes this query return the value 2 because the two sets constructed within the inner query will be unequal (as they are mutable sets constructed at different points during query evaluation and hence not identical) despite being extensionally equal. On the other hand, OQL's ODMG-based semantics assumes that the sets constructed during the inner query are immutable and therefore equal. The ODMG version of this query would therefore return the value, 1. The simple adjustment of OQL semantics making queries construct mutable rather than immutable sets impacts even those queries that do not construct sets.

3.2.2 Approach 2: Interpret Sets Constructed by OQL Queries to Be Mutable

There is a subtle difference between this approach and the last. The previous approach redefined OQL semantics to make all sets *constructed* by queries mutable. This approach leaves OQL semantics alone and interprets all sets *returned* as a result of queries as mutable.

The nested query of Figure 5 was problematic because of the interpretation of the results of the inner query as mutable sets. This meant that the duplicate elimination performed as part of generating the outer query used checks of identity to compare inner query results.

With this solution, we propose that OQL semantics be preserved in their entirety. OQL queries still generate immutable sets. (Thus, the age set query of Figure 5 will count the set, `{40, 35}` just once.) However, all immutable sets *returned* by queries are coerced to be mutable. The semantics of queries is preserved up to mutability, because the semantics of OQL is unchanged.

Note that this approach fails to preserve OQL compatibility in an *embedded* setting. It is possible to write code that evaluates the query of Figure 5 in steps. In this case, intermediate results (i.e., results of inner queries) are interpreted as mutable and the cardinality of the set that contains the intermediate results will be 2 rather than 1. Arguably, this discrepancy in query results is less serious than the one shown in Section 3.2.1. The previous discrepancy was problematic because it meant that software components specifically designed for OQL (as in OQL-based query optimizers) would not be usable in a Thor setting. However, the discrepancy described here does not occur because the results of queries have changed, but because of what happens when query results are used as expressions within a programming language. Therefore, an OQL-based query optimizer would still be of use even if the sets returned by OQL queries were interpreted as mutable.

Finally, it should be pointed out that this solution does not reintroduce the complications of constructing mutable sets from immutable sets described in Section 3.1.2. That is, *all* immutable sets returned by a query to Thor can be coerced to mutable sets without requiring copying. This is because any immutable set that is returned by a query **must** be a newly constructed set and therefore can be safely coerced. All existing sets that are returned as a result of querying will be mutable, for they will be Thor sets.

3.3 The Design Decision for Thor

In the end, our analysis revealed that much work has to be done to allow mutable and immutable sets to coexist without introducing inherent inefficiencies into querying. Immutable objects can and likely will be shared to improve the performance of the system. But this means that copying of these sets will be required, either to mutate mutable sets if mutable sets are represented using immutable sets, or to construct mutable sets if they are not.

On the other hand, it appears that restricting sets to be mutable in Thor (and reconciling differences between OQL and Thor sets as described in Section 3.2.2) allows us to retain the query expressivity provided by OQL without introducing inefficiencies into set construction. Given these considerations, the current plan is for Thor to assume all sets as mutable, but for OQL query semantics to be preserved. Immutable sets constructed and returned by OQL queries will then be interpreted as mutable as was described in Section 3.2.2. This leaves us the flexibility of adding support for immutable sets at a later point should we decide that we need them in the model.

4 Related Work

Object-oriented databases offer many challenges to querying. Object complexity makes it possible to pose queries with lengthy path expressions or that are nested in ways not possible over relations. Path expression optimization has been considered in [KM90a, KM90b] and [Ber91]. Optimization of Object-Oriented nested queries has been considered in [CM93] and [CZ96]. Methods in queries make it necessary to reconsider ways to index [MS86, Hwa94] and to consider the order in which predicates should be evaluated on data [HS93, Hel94, KMPS94, SPMK95].

Many areas of querying impacted by objects have been well-studied, but as far as we know, mutability's impact on querying has not been considered before. Transformation rules over complex object bases have been proposed in [BK90] (which provides rules for multiple collection types), [Osb88] (which focuses on queries that construct new objects) and [SZ89] (which considers criteria for determining the correctness of transformations of queries over objects). None of these distinguish rules for queries over mutable and immutable inputs. The work of Shaw and Zdonik [SZ89] comes closest to ours in that they considering how sharing relationships must be preserved in generally applicable query transformations. However, the sharing relationships they address are those where the referencing objects (the "sharers") are objects retrieved or constructed by the query. The transformations of the kind shown in Figure 2 demonstrate that sharing **of** and not just **by** objects returned by the query must be considered.

The effect of nondeterminism on transformation correctness has been considered in [dBG92]. Nondeterminism due to object construction in queries has been considered in [AK92, Osb88]. These works formulate policies for when transformations involving queries that construct new objects are correct, with the idea that identical results need not be produced by both queries. Our discussion of nondeterminism illustrates the kind of rule that object mutability invalidates. That is, we do not address the general question of transformation validity in the presence of nondeterminism. Instead, we are concerned with finding those transformations that nondeterminism breaks.

Indistinguishability as the determining factor for equality was first proposed in the context of CLU [LG86]. Indistinguishability as a goal in and of itself motivated proposals to make mutable object equality tied to *identicalness* in [Bak93, GC94, GS93]. In [CZN95], we showed that equalities must hold only of objects that are permanently indistinguishable to ensure the reasonable behavior of sets that contain them.

Our work also addresses the issue of whether OODB object models should provide both immutable and mutable sets, or whether mutable sets alone are sufficient. Commerical systems have had to make the same kinds of choice, and can be classified accordingly to their decisions. Among the object models that support only mutable sets are those of GemStone [MOP85] and ObjectStore [LLOW91]. Both systems include their own proprietary query languages, but have pledged to adopt the OQL querying standard, and will thus have to consider the issues regarding reconciliation of mutability approaches discussed in Section 3.2. Object models providing both mutable and immutable sets include O_2 [BDK92] (which arguably had the most influence on the ODMG model) and EXTRA (Exodus) [CDG$^+$90]. These systems must consider the issues related to query optimization and mutable set construction discussed in Section 3.1. We also looked at other systems that have pledged to adopt the OQL standard (e.g., Objectivity/DB [Inc93], ONTOS [Ont93] and POET [Sof93]) but have yet to receive documentation about how they intend to address these issues.[3]

[3] To the reviewers: We have sent a letter to each of these companies asking them how they plan to interpret the immutable sets produced by OQL queries and will update this section when we have received answers.

5 Conclusions

The ability of an object to change is one of the defining characteristics of object-oriented databases. Despite this, the impact of mutability on querying appears not to have been considered before. Likely, this is because queries are usually assumed free of side-effects and therefore objects are effectively immutable while they are being queried. But the mutability of an object affects other object behaviors such as how objects are compared for equality, and whether or not sharing is detectable. These two properties impact the efficiency of querying.

These issues arose in the early stages of an integration project to provide OQL-based querying for Thor. OQL assumes the existence of both mutable and immutable sets. Thor had not committed to a policy for sets when the integration project began. In this paper, we have presented our analysis that has helped to shape this policy. We considered the implications of the OQL approach and showed that mutability restricts query optimization and impedes the effecient construction of mutable sets from immutable sets. We then considered how to make all Thor sets mutable without disturbing the semantics of OQL queries.

Though inspired by a specific integration project, the analysis presented here has applicability to numerous systems. OODBs that support both mutable and immutable sets (including those that propose to adhere to the ODMG standard) must consider the potential impediments to query optimization, and inefficiencies in mutable set construction discussed in Section 3.1. OODBs that support only mutable sets and that wish to adopt OQL or SQL standards for querying must consider the reconciliation issues presented in Section 3.2. Any database that provides support for objects then, must consider the issues discussed here, concerning how an object's mutability determines the efficiency of its retrieval.

6 Acknowledgements

We would like to thank Barbara Liskov and Stan Zdonik for helpful comments on earlier drafts of this paper.

References

[ABD+89] Malcolm P. Atkinson, Franois Bancilhon, David J. DeWitt, Klaus R. Dittrich, David Maier, and Stanley B. Zdonik. Object-Oriented Database System Manifesto. In Won Kim, Jean-Marie Nicolas, and Shojiro Nishio, editors, *First International Conference on Deductive and Object-Oriented Databases (DOOD)*, pages 223–240, Kyoto, Japan, December 1989.

[AK92] Serge Abiteboul and Paris C. Kanellakis. Object identity as a query-language primitive. In Francois Bancilhon, Claude Delobel, and Paris Kanellakis, editors, *Building an Object-Oriented Database System: The Story of O_2*, pages 98–127. Morgan Kaufmann, 1992.

[Bak93] Henry Baker. Equal rights for functional objects or, the more things change, the more they are the same. *OOPS Messenger*, 4(4):1–26, October 1993.

[BDK92] Francois Bancilhon, Claude Delobel, and Paris Kanellakis, editors. *Building an Object-Oriented Database System: The Story of O_2*. Morgan Kaufmann, 1992.

[Ber91] Elisa Bertino. An indexing technique for object-oriented databases. In *Proceedings of the Seventh International Conference on Data Engineering*, pages 160–170, Kobe, Japan, April 1991.

[BK90] Catriel Beeri and Yoram Kornatzky. Algebraic optimization of object-oriented query languages. In S. Abiteboul and P. C. Kanellakis, editors, *Proceedings of the Third International Conference on Database Theory*, number 470 in Lecture Notes in Computer Science, pages 72–88, Paris, France, December 1990. EATCS, Springer-Verlag.

[Bog94] Phil Bogle. Reducing cross-domain call overhead using batched futures. Master's thesis, Massachusetts Institute of Technology, 1994.

[BOS91] Paul Butterworth, Allen Otis, and Jacob Stein. The GemStone object database management system. *Communications of the ACM*, 34(10):64–77, October 1991.

[Cat96] R. G. G. Cattell, editor. *The Object Database Standard: ODMG-93, Release 1.2.* Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1996.

[CDG+90] Michael J. Carey, David J. DeWitt, Goetz Graefe, David M. Haight, Joel E. Richardson, Daniel T. Schuh, Eugene J. Shekita, and Scott L. Vandenberg. The EXODUS extensible DBMS project: An overview. In Stanley B. Zdonik and David Maier, editors, *Readings in Object-Oriented Database Systems*, pages 474–499. Morgan Kaufmann Publishers, Inc., Los Altos, California, 1990.

[Che95] Mitch Cherniack. Form(ers) over function(s): The KOLA query algebra. Technical report, Brown University Department of Computer Science, May 1995. In preparation.

[CM93] Sophie Cluet and Guido Moerkotte. Nested queries in object bases. In *Proc. 4th Int'l Workshop on Database Programming Languages*, New York, New York, August 1993. Springer-Verlag.

[CZ96] Mitch Cherniack and Stanley B. Zdonik. Rule languages and internal algebras for rule-based optimizers. In *Proc. ACM SIGMOD Int'l Conference on Management of Data*, Montréal, Québec, Canada, June 1996. To Appear.

[CZN95] Mitch Cherniack, Stanley B. Zdonik, and Marian H. Nodine. To form a more perfect union (intersection, difference). In *Proc. 5th Int'l Workshop on Database Programming Languages*, Gubbio, Italy, September 1995. Springer-Verlag. To Appear.

[dBG92] Jan Van den Bussche and Dirk Van Gucht. Semi-determinism (extended abstract). In *Proceedings of the Eleventh ACM SIGACT/SIGMOD Symposium on Principles of Database Systems*, 1992.

[GC94] Peter Grogono and Patrice Chalin. Copying, sharing and aliasing. In *Colloquium on Object-Orientation in Databases and Software Engineering*, Montreal, Quebec, 1994.

[GS93] Peter Grogono and Philip Santas. Equality and assignment in object-oriented languages. In *EastEurOOPe'93*, pages 191–201, November 1993.

[Hel94] Joseph M. Hellerstein. Practical predicate placement. In Richard T. Snodgrass and Marianne Winslett, editors, *1994 ACM SIGMOD International Conference on Management of Data*, Minneapolis, Minnesota, June 1994.

[HS93] Joseph M. Hellerstein and Michael Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *Proc. ACM SIGMOD*, pages 267–276, 1993.

[Hwa94] D. Hwang. *Function-Based Indexing for Object-Oriented Databases*. PhD thesis, Massachusetts Institute of Technology, February 1994.

[Inc93] Objectivity Inc. Objectivity database reference manual, 1993.

[KM90a] Alfons Kemper and Guido Moerkotte. Access support in object bases. In Hector Garcia-Moline and H. V. Jagadish, editors, *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, pages 364–376, Atlantic City, NJ, May 1990.

[KM90b] Alfons Kemper and Guido Moerkotte. Advanced query processing in object bases using access support relations. In Dennis McLeod, Ron Sacks-Davis, and Hans-Jrg Schek, editors, *16th International Conference on Very Large Data Bases*, pages 290–301, Brisbane, Australia, August 1990.

[KMPS94] Alfons Kemper, Guido Moerkotte, Klaus Peithner, and Michael Steinbrunn. Optimizing disjunctive queries with expensive predicates. In Richard T. Snodgrass and Marianne Winslett, editors, *1994 ACM SIGMOD International Conference on Management of Data*, Minneapolis, Minnesota, June 1994.

[LAC+96] Barbara Liskov, Atul Adya, Miguel Castro, Mark Day, Sanjay Ghemawat, Robert Gruber, Umesh Maheshwari, Andrew C. Myers, and Liuba Shrira. Safe and efficient sharing of persistent objects in Thor. In *SIGMOD 96*, 1996. To appear.

[LCD+94] Barbara Liskov, Dorothy Curtis, Mark Day, Sanjay Ghemawhat, Robert Gruber, Paul Johnson, and Andrew C. Myers. *Theta Reference Manual*. Programming Methodology Group Memo 88, MIT Laboratory for Computer Science, Cambridge, MA, February 1994. Also available at http://www.pmg.lcs.mit.edu/papers/thetaref/.

[LG86] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. MIT Press, 1986.

[LJGS90] Barbara Liskov, Paul Johnson, Robert Gruber, and Liuba Shrira. A highly available object repository for use in a heterogeneous distributed system. In Alan Dearle, Gail M. Shaw, and Stanley B. Zdonik, editors, *Implementing Persistent Object Bases, Principles and Practice, Proceedings of the Fourth International Workshop on Persistent Objects*, Martha's Vineyard, MA, USA, September 1990. Morgan-Kaufman.

[LLOW91] Charles W. Lamb, Gordon Landis, Jack A. Orenstein, and Daniel Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10):51–63, October 1991.

[MOP85] D. Maier, A. Otis, and A. Purdy. Development of an Object-Oriented DBMS. *Quarterly Bulletin of IEEE on Database Engineering*, 8(4), 1985. Also in Proceedings of the 1986 OOPSLA conference.

[MS86] David Maier and Jacob Stein. Indexing in an object-oriented database. In Klaus R. Dittrich and Umeshwar Dayal, editors, *1986 International Workshop on Object-Oriented Database Systems*, pages 171–182, Pacific Grove, California, 1986.

[Ont93] Ontos. Ontos reference manual, 1993.

[Osb88] S. L. Osborn. Identity, equality and query optimization. In K. R. Dittrich, editor, *Advances in Object-Oriented Database Systems:Proceedings of the 2nd International Workshop on Object-Oriented Database Systems*, number 334 in Lecture Notes in Computer Science, pages 346–351, Bad M¨unster am Stein-Ebenburg, FRG, September 1988. Springer-Verlag.

[Sof93] Poet Software. Poet programmer's and reference guide, 1993.

[SPMK95] Michael Steinbrunn, Klaus Peithner, Guido Moerkotte, and Alfons Kemper. Bypassing joins in disjunctive queries. In *21st International Conference on Very Large Data Bases*, Zurich, Switzerland, September 1995.

[SZ89] Gail M. Shaw and Stanley B. Zdonik. Object-oriented queries: Equivalence and optimization. In Won Kim, Jean-Marie Nicolas, and Shojiro Nishioe, editors, *Proceedings of the The First International Conference on Deductive and Object-Oriented Databases*, pages 264–278, Kyoto, Japan, December 1989.