# Revisiting Reference Materialization Techniques for Object Query Processing

Quan Wang[1] David Maier[1] Leonard Shapiro[2]
[1]Oregon Graduate Institute of Science and Technology
[2]Portland State University
[1]{quan, maier}@cse.ogi.edu
[2]len@cs.pdx.edu

## Abstract

Resolving object references, or *reference materialization*, is a fundamental operation in object query evaluation. Existing reference materialization techniques fall into two categories: pointer-based and value-based. We identify several drawbacks of existing techniques, and propose a hybrid technique that combines the advantages of each category. This algorithm relaxes the limitations of value-based techniques, while preserving much of their performance advantage over pointer-based techniques. The hybrid technique shows even stronger performance advantages in when moving from single-valued to collection-valued attributes. We also show how to enhance the performance of value-based techniques on collection-valued attributes when inverse relationships are available. Both the hybrid and enhanced value-based techniques can be easily incorporated into rule-based query optimizers, using transformations we present.

The contribution of this paper is twofold. First, we exhibit a new technique for reference materialization that performs well when no existing algorithm is applicable and efficient, and works with single-valued and collection-valued attributes. Second, we provide two enhancements to value-based techniques, which help adapt current work on path expression processing to the setting of collection-valued attributes. Initial experimental results using a commercial object-oriented database show that both the hybrid approach and the value-based enhancements can achieve an order of magnitude speedup over current algorithms.

The initial motivation for our work was optimization and evaluation of object-oriented query languages, particularly OQL [ODMG]. However, the key features we have concentrated on, references and collection-valued attributes, are present in object-relational products and the SQL:1999 proposal [EM99].

## 1 Introduction

Most object and object-relational models include the notion of reference attributes, also called object-valued attributes. Thus, resolving referenced objects, or *reference materialization*, becomes an essential operation in object query evaluation. In reference materialization, one object, containing a reference to a second object, is brought together in memory with that referenced object. Existing reference materialization techniques can be classified into two categories. The first kind, called *pointer-based techniques***,** retrieve referenced objects by converting references (if necessary) to disk addresses and retrieving the appropriate pages. Pointer-based techniques include assembly [KM89], pointer-based hash, pointer-based nested-loops, pointer-based sort-merge [SC90], and partition-merge [BCK98]. The second kind, the *value-based techniques*, attempt to avoid "pointer chasing" on disk by instead performing joins between objects with references and the objects being referenced (using an object's identifier as an implicit attribute) [BMG93]. Thus, whereas pointer-based techniques treat references as physical pointers, valued-based techniques regard references as just another kind of logical value, allowing the application of conventional join-processing technology to object query evaluation. In the remainder of this paper, we will generally shorten "reference materialization" to simply "materialization".

We observe some limitations of existing techniques. Pointer-based techniques can be inefficient, particularly when there is sharing of object references. Value-based techniques generally do better, but require the existence of a type extent. A *type extent* (or simply *extent*) is an explicit collection containing all the objects of certain type. Most object-oriented database systems do not automatically maintain extents, and even in some object-relational systems, they are optional. (While SQL:1999 requires `REF` columns to be scoped to a single table, object-relational products aren't as restrictive.) Besides these limitations, the behavior of materialization techniques when attributes are collections of references has not been studied extensively. Since one of our main interests in object query processing is *collection-valued attributes* (CVAs) [Rama et al. 98], we need to know which materialization techniques are appropriate with CVAs, and possibly modify those techniques to be more suitable in that case.

We address these issues in this paper. The remainder of Section 1 defines the terminology we use throughout. Section 2 evaluates existing materialization techniques. Section 3 proposes the hybrid technique to address some limitations of those techniques. Section 4 examines the various techniques in the setting of CVAs, and notes some problems with the value-based and hybrid approaches. Based on these observations, Section 5 presents two enhanced value-based algorithms for CVA materialization. Section 6 validates the arguments and analysis presented in the earlier sections via performance experiments. Section 7 and 8 discuss related work, draw conclusions, and outline future work.

Below is a simple database schema that is used throughout this paper for examples.

- **Student** (name: string, dept: Department, advisor: string, status: string, Core:{Course}, Takes:{Course})
- **Course** (title: string, dept: Department, Instructors: {Professor}, Participants: {Student})
- **Department** (name: string, head: Professor, Majors:{Student}, Offerings:{Course}, Faculty: {Professor})
- **Professor** (name: string, dept: Department, specialty: string, salary: int, Teaches: {Course})

This schema defines four types of objects, for students, courses, departments, and professors in a university database. Our convention is that single-valued attributes, such as *dept*, begin with a lowercase letter, and collection-valued attributes, such as *Instructors*, begin with uppercase. When the type of an attribute is lowercase, it indicates a scalar-valued attribute that is assumed to be stored in the object itself. Uppercase types indicate object-valued attributes, which are always assumed to be represented as references. In particular, all the collection-valued attributes in our example have elements that are object references. Figure 1 shows some object instances for the example schema. Here, nodes represent objects; edges stand for attributes; dashed edges represent non-CVA attributes.

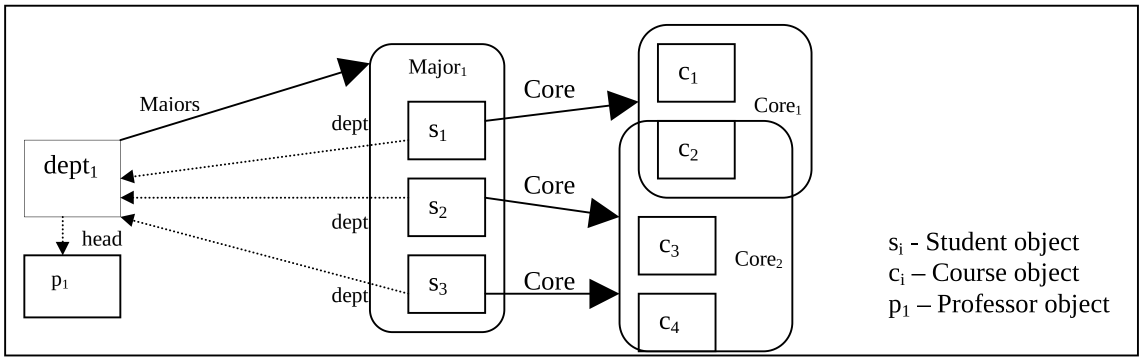


Figure 1: An instance of the example schema

An *attribute instance* is the value of an attribute. For example, in Figure 1, object $p_1$ is an instance of attribute *head* for object $dept_1$; $Major_1$ is an instance of the CVA *Majors*. *CVA elements* are the members of a CVA instance. In Figure 1, CVA instance $Major_1$ contains three *Student* objects, $s_1$, $s_2$, and $s_3$, which are the CVA elements of $Major_1$. By convention, a single-valued attribute instance is called a *child* of the hosting object. A CVA element is called a *child* of the object hosting the corresponding CVA instance. In Figure 1, $p_1$ is a child of $dept_1$; $s_1$ is also a child of $dept_1$. Note that the terminology of parent and child is relative to the attribute that concerns us in each setting. Object A can be a child of object B, and a parent of object C. Also, object A can be a child of B as well as a parent of B, when different attributes are considered.

Now we introduce terminology that refers to a particular database state for a schema. A *shared* attribute is an attribute via which several objects refer to the same attribute instance. The *fan-in* of a shared attribute is the average number of parent objects that refer to each distinct attribute instance. An *overlapping* CVA is one where different CVA instances have elements in common. In Figure 1, *Core* is a shared CVA, because two students, $s_2$ and $s_3$, refer to the same instance $Core_2$. The average number of students who have the same set of core courses is the fan-in of *Core*. Note that *Core* is also an overlapping CVA, because, two *Core* instances, $Core_1$ and $Core_2$, contain the same element, $c_2$. Single-valued attributes can be shared too. For instance, in Figure 1, the *dept* attribute of *Student* is a shared attribute as three student objects all refer to $dept_1$.

Types R and S have a *m:n relationship* if there exists attribute *x* in R and attribute *y* in S satisfying three conditions: (1) a R object refers to *n* S objects on average through *x*, and a S object refers to *m* R objects on average through *y*; (2) any R object referring to a S object is also referred to in turn by this S object; (3) any S object referring to a R object is also referred to by this S object. (Conditions (2) and (3) mean that *x* and *y* are *inverse attributes*.) For instance, in Figure 1, *Department* and *Student* have a 1:3 relationship through attributes *Majors* and *dept*, assuming each department averages three students, as illustrated. As another example, *Student* and *Course* might have a 4:30 relationship through *Takes* and *Participants*, if, on average, a student takes four courses, and a course has an enrollment of 30 students. Apparently, in a relationship, if *m* is one, *y* is a single-valued attribute. Otherwise, it is a CVA. Similarly for *n* and *x*.

The *density* of a type extent relative to a query is the ratio between the number of objects of the type that qualify for the query and the cardinality of the extent. For example, the density of the *Department* extent could be less than one in a query that accesses the major department of each student, if some departments have no majors.

In a typical object-oriented database system (such as GemStone [Gem96]) or storage manager (such as Shore [Carey et al. 94]), the value of a reference attribute is an *object identifier* (*OID*). OIDs might be physical or logical. That is, they might or might not suggest the physical location of the objects they identify. The actual location of an object with a logical OID is indicated by its physical object identifier (*PID*), which is  obtained by consulting an OID-to-PID mapping table, called an *object table*, using the OID as key. We assume a collection object (such as might be used for an extent) or a CVA instance contains a collection of OIDs identifying the element objects. Note that there are other ways of storing CVAs, for instance, using a table that contains the relationships between parents and children, or storing CVA elements within parents [Rama et al. 98]. These other representations, in most cases, can be regarded as special cases of the one we assume.

## 2 Existing Materialization Techniques

In algebraic query optimization, a *logical expression* (or simply *expression*) is an algebraic tree consisting of *logical operators*, such as select and join. A rule-based optimizer accepts an expression as input, then transforms it into its equivalent expressions using transformation rules. *Physical expressions* are obtained by implementing the logical operators in an expression using specific algorithms, such as merge join or hash join. The goal of optimization is to pick the most efficient algorithms among all the physical expressions implementing expressions equivalent to the original query. Relational algebra, being value-based, does not have an operator for resolving object references. To explicitly indicate the resolution of inter-object references in logical expressions, the Open OODB query optimizer [BMG93] introduced the *materialize* operator. One way to view the *materialize* operator is that it brings referenced objects into scope, so that succeeding operators can access them. Figure 2(1) is an expression that consists of a *materialize* (*M*) operator for attribute *r.a* , where *r* ranges over the output of expression *R*. The result of this expression can be regarded as pairs of objects <*r*, *s*>, where *s* is the object whose OID is *r.a*.

Pointer-based techniques directly implement the materialize operator using pointer-based physical algorithms such as assembly [KM89], pointer-based hash, pointer-based nested-loops, pointer-based sort-merge [SC90], and partition-merge [BCK98]. Among these algorithms, sort-merge and hybrid-hash are popular and competitive [SC90]. The hybrid-hash algorithm would perform the logical materialize operator in Figure 2(1) as follows. First, *r* objects from *R* are partitioned using the OIDs of their *r.a* instances. Then, the objects in each partition are iterated, while the PIDs of *r.a* instances are looked up in the object table. The purpose of the first partitioning is to avoid random accesses to the object table, thereby reducing I/O cost if the entire table won’t fit in memory. Third, the objects from *R* are partitioned again using the PIDs of *r.a* instances. Finally, the objects in each of these partitions are iterated, using the PIDs to locate and fetch the *r.a* instances themselves. Notice that the first partitioning can be dispensed with if the OIDs are physical to begin with, or can be used to directly compute physical addresses.

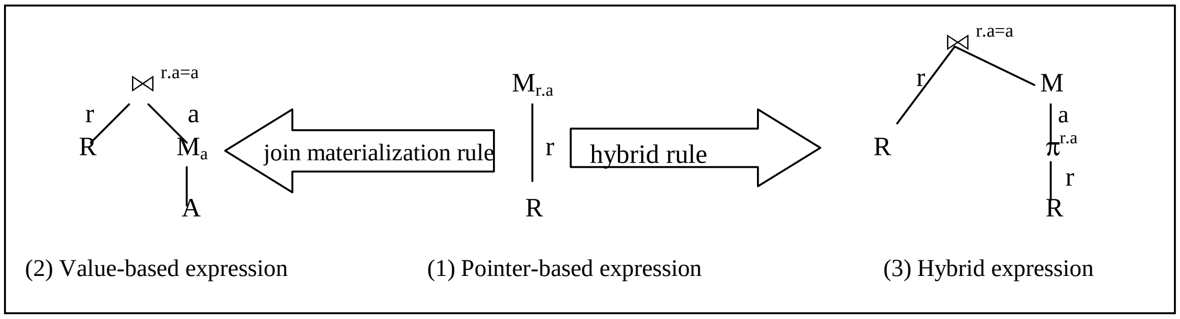

r.a=a
r
a
R
M_a
A
join materialization rule
M_r.a
r
R
hybrid rule
r.a=a
r
M
a
R
π r.a
r
R
(2) Value-based expression
(1) Pointer-based expression
(3) Hybrid expression

Figure 2: Alternative materialization techniques and rules to derive them

Although Figure 2(1) gives rise to pointer-based expressions when implemented directly, it can be transformed to derive other materialization techniques. Figure 2 illustrates the join-materialization rule [BMG93] that transforms a *materialize* into a *join* between *R* and a *materialize* on *A*, where *A* is the appropriate type extent for *r.a*. Assume that a type extent is a collection of OIDs, the introduced *materialize* brings *A* elements into memory in a pointer-based fashion. As mentioned before, we use term *value-based technique* to refer to materialization accomplished by join with an extent. We use the term *value-based expression* to refer to an expression that uses the value-based technique, for example, Figure 2(2). While the materialization operator can be implemented by a single physical algorithm, value-based expressions sometimes require the combination of several operators to materialize references. Nevertheless, we will sometimes loosely refer to part of an evaluation plan that performs materialization as a value-based algorithm, even when it comprises several operators.

Value-based techniques have many advantages. First, if the extent collection is appropriately ordered, these techniques sequentially fetch referenced objects, thus avoiding the effort in reordering object accesses that a pointer-based algorithm usually has to perform. Second, restrictive operations on referenced objects, such as selections, can be evaluated earlier in a value-based expression than in a pointer-based expression. Third, if the attribute to be materialized is shared, then operations on referenced objects are performed once for each object in a value-based expression, but multiple times in a pointer-based expression. Fourth, the value-based technique converts materialization into join, enabling the application of the conventional join re-ordering techniques to object query optimization. Fifth, the join predicate in a value-based algorithm only compares OIDs, thus the object table may be visited less frequently than in with a pointer-based algorithm. (Note that object table access is not totally avoided, as we assume the extent is a collection of OIDs that must be converted to PIDs. However, each OID is converted just once.)

However, value-based techniques have their shortcomings. First, they require the presence of appropriate extents. Second, if an extent is sparse relative to a query, a value-based algorithm may be inefficient because of all the inapplicable objects in that join operand. In contrast, pointer-based techniques are not constrained by extents, and apply in any circumstance. Also they carry out materialization with a single operator, thus simplifying query optimization.

## 3 The Hybrid Technique

The previous section suggests that existing materialization techniques are not sufficient to provide efficient solutions for all cases of materialization. In this section, we present a hybrid technique that combines both pointer-based and value-based techniques. This technique relaxes the limitations of value-based algorithms, while retaining most of their performance advantage over pointer-based algorithms.

The hybrid technique is illustrated by Figure 2(3), an expression equivalent to Figure 2 (1). This expression performs a *join* between *R* and instances of the *r.a* attribute. The right operand is produced by two operators: *projection* gathers all the OIDs of *r.a* instances, while *materialize* resolves them in a pointer-based fashion. This combination of operators simulates a type extent. It also serves two other purposes. First, the *projection* separates children from parents, so that the succeeding *materialize* (and possibly other operators) processes only children, reducing the amount of data handled by those operators. (Note that this aspect mainly affects any copying an operator may have to do between its inputs and outputs.) Second, more importantly, the *projection* eliminates duplicate OIDs, thus minimizing the input cardinality to the successive operators. We use term *hybrid expression* to an expression employing this hybrid technique in some form. Again, we sometimes use *hybrid algorithm* loosely to refer to the portion of an evaluation plan that implements the hybrid technique. Figure 3(3) illustrates a materialization process using the hybrid algorithm.

It might seem that the hybrid technique does all the work of both the pointer- and value-based techniques, as it is both dereferencing pointers and performing a join. However, in general, it is chasing fewer pointers than pointer-based methods and computing a smaller join than value-based methods. The hybrid technique does not always win out over the others, but our experimental results will show that in some instances it is much better than the other two. Also, the hybrid technique does introduce a repeated sub-expression, which we will discuss later.

Both hybrid and value-based techniques perform a *join*, however, they differ in *join* operands. Hybrid algorithms access referenced objects not through their extent, but through a collection of OIDs gathered on the fly. This difference implies two advantages. First, the hybrid technique is not limited by availability of type extents. Second, hybrid techniques will be more efficient even when an extent exists, if that extent is sparse for the query. Figure 3(2) and (3) illustrate the data flows for the value-based and hybrid expressions in Figure 2. Since extent *A* contains objects, such as $a_3$, that are not referenced by any object in *R*, the value-based expression will have a larger right join operand, thus a higher join cost than the hybrid expression.

Compared to pointer-based techniques, the hybrid technique inherits the following advantages from the value-based techniques. First, it allows restrictive operations on referenced objects to be performed earlier. For instance, if a query has the predicate *a.x=4*, a *selection* can be pushed down to restrict the right *join* operand. Second, when the attribute to be materialized is shared, any operation on the referenced objects is performed once for each instance in a hybrid algorithm, but multiple times in a pointer-based algorithm. In Figure 3 (1), $a_1$ is mentioned by two *r* objects. As a result, $a_1$ has to be resolved twice. In contrast, in Figure 3 (3), duplicate OIDs are eliminated, thus no redundant data is delivered to *materialize*. Third, in a pointer-based expression, for instance, Figure 2 (1), *materialize* accepts parent attributes as input, while only the OIDs of attribute instances are actually needed. In a hybrid expression, for instance, Figure 2 (3), *materialize* only accepts the OIDs, thus substantially shrinks the width of input data. As some physical algorithms for *materialize* move input data between disk and memory, reducing the amount of input data size may lower I/O costs. Figure 3 (1) and (3) illustrate the difference in input sizes of the two *materializ*e operators. Note that the difference in width is equal to the number of attributes in the objects of *R*. We will see further benefits of the hybrid approach when we consider materialization of CVAs in Section 4.

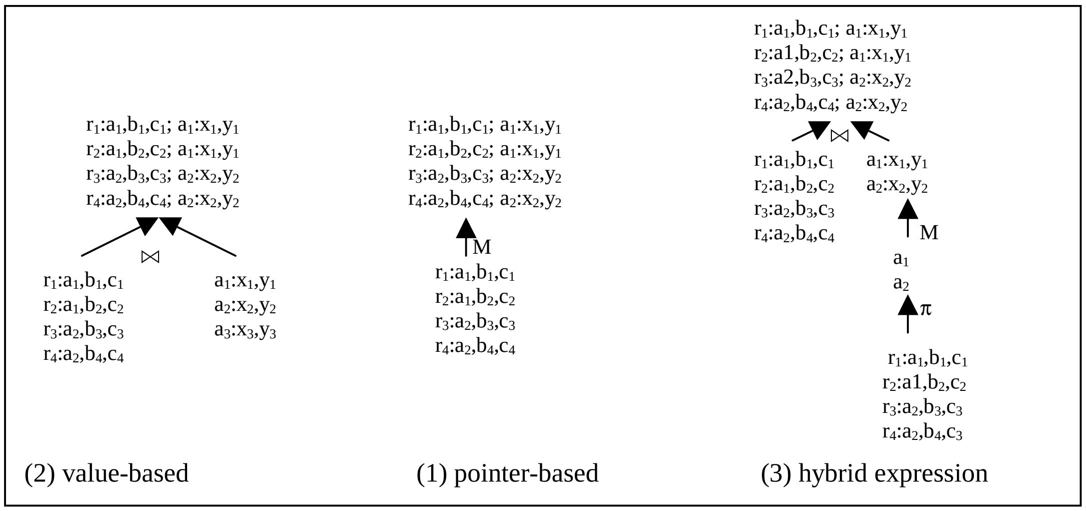


Figure 3: The data flows of the three expressions in Figure 2

While hybrid algorithms often have good costs, they are of little use unless they can be produced during query optimization. We can use the hybrid rule, illustrated in Figure 2, to generate a hybrid expression from a pointer-based expression. Like the join materialization rule, the hybrid rule transforms a *materialize* into a *join*, and a *materialize*, but with an extra *projection*. We use the example query below to illustrate the generation of a hybrid expression, and contrast it with other alternatives.

**Query 1:** Find all faculty who specialize in Mathematics and earn more than their department heads:

```
select f
from Faculty as f
where (f.specialty = ‘math’) and (f.salary > f.dept.head.salary) .
```

Three expressions for Query 1 appear in Figure 4. The pointer-based expression contains three *materialize* operators that successively fetch *Professor*, *Department*, and *Professor* objects. Note that, besides the reference attributes, the objects in base collection *Faculty* also need to be materialized ($M_f$), since we assume a collection stores only the OIDs of its elements. In general, we assume such a materialization uses pointer-based algorithms. The other two *materialize* operators, $M_{f.dept}$ and $M_{f.dept.head}$, each resolves some objects multiple times, due to the sharing of *f.dept*. Transforming the pointer-based expression using the join materialization rule yields the value-based expression. Alternatively, a hybrid expression, Figure 4(3), can be generated by applying the hybrid rule to $M_{f.dept}$, then pushing down the succeeding *select* and *materialize*. Obviously, the value-based and the hybrid expressions do not have the problem of the pointer-based expression that resolves one object several times. However, the value-based expression is less efficient than the hybrid one when only a few departments have professors specializing in Mathematics, in which case, the effort of materializing most departments is useless work.

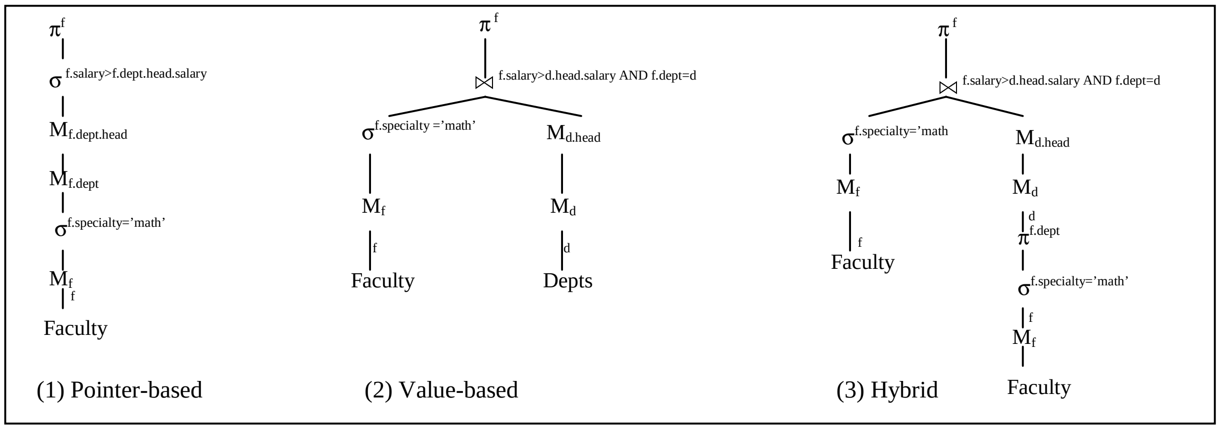


Figure 4: Expressions for Query 1

The presence of object sharing is an important factor motivating the hybrid approach. Object sharing occurs both directly, because of shared attributes, and indirectly, as the result of flattening an overlapping CVA, as demonstrated by the example query below.

**Query 2:** Return all CS courses and their participants:

```
select struct (c: c, p: p.name)
from Courses as c, c.Participants as p
where c.dept  = 'CS'.
```

To evaluate this query, a pointer-based expression first filters *Courses*, then flattens *c.Participants*, giving <*c, OID(p)*> pairs, and finally materializes individual participating students. Note that *c.Participants* is an overlapping CVA, since several courses may enroll  the same student. Therefore the result of flattening *c.Participants* will contain multiple references to some students. This sharing degrades the performance of the succeeding *materialize* operator, which will consequently resolve some objects multiple times.

Hybrid expressions do incur certain overheads in evaluation, from the need for *projection* and *join,* and the introduction of common sub-expressions. We designed the hybrid rule to explicitly duplicate expressions, rather than storing one to a temp and reusing the result. The reason is that other transformations may convert different occurrences of a common sub-expression by modifying one or both of them. For instance, in the hybrid rule of Figure 2, the *projection* operator might be pushed down into the underlying expression *R*, so that it outputs only the needed attributes. Alternatively, operators above the *join* might be pushed down towards the left occurrence of *R*. Note that such transformations can make the original common sub-expressions distinct, but may in the process generate new common sub-expressions. Based on the cost of a common sub-expression, an optimizer can choose to store and reuse its result, or evaluate it twice. Therefore the hybrid rule needs to be used in a cost-based framework, so that a hybrid algorithm is chosen only if its benefit dominates its overheads. For simplicity, the remaining paper assumes a hybrid expression represents both possibilities.

The introduction of a repeated sub-expression can also increase the cost of the query optimization process, as well, due to increasing number of operators, and more importantly, possible duplicated effort from optimizing the same sub-expression twice. However, previous work [GM93, CG94] successfully avoided duplicate optimization of common sub-expressions using directed acyclic graphs (DAGs) representation of expressions and memo structure.

## 4 Materializing Collection-Valued Attributes

The pervious section deals with materialization techniques in the setting of single-valued attributes. In this section, we explore the ramifications of the three techniques in the context of CVAs. CVAs first appeared as nested relations in the non-first-normal form ($NF^2$) relational models [SS86], and subsequently appeared in object-oriented and object-relational models. The presence of CVAs poses many challenges to conventional query optimizers. For instance, classic indexing and hash algorithms often do not apply to queries involving CVAs. In term of materialization, the techniques developed for single-valued attributes should be re-evaluated for CVAs. We demonstrate that, when applied to CVAs, value-based algorithms become inefficient, while hybrid algorithms are more competitive, but still have some shortcomings.

Materializing a CVA means bringing the elements of the CVA instance into memory. Some algorithms can accepts CVA instances directly as input, and output collections of elements [BCK98]. Most algorithms handle only flat data, in which case references to CVA elements are accepted and resolved [SC90]. The first class of algorithms is useful when the algebra of an optimizer [SS86] can directly manipulate CVA elements without flattening them. More often, an algebra mainly supports relational operators, in which case the second class of algorithms is more desirable. Therefore, we assume *materialize* accepts and outputs flat data (tuples of scalar values and OIDs). The example query below illustrates how CVAs are materialized using different techniques.

**Query 3**: Return pairs of student and course title where the course is among the core requirements for the student:

select struct(s:s, c:c.title)
from Students as s, s.Core as c.

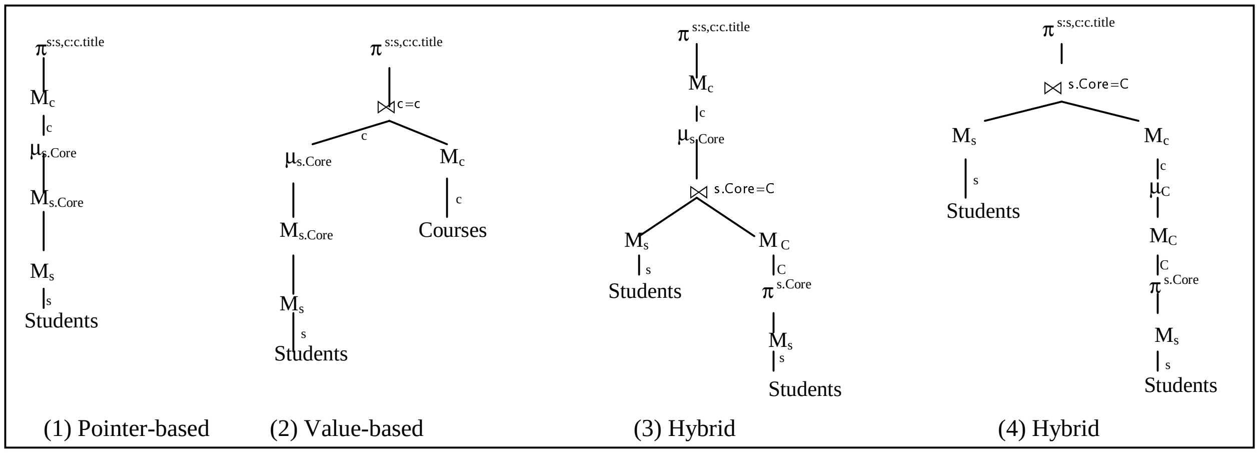


Figure 5: Expressions for Query 3

Expression (1) in Figure 5 is pointer-based. In this expression, CVA instances of *s.Core* are first materialized, then flattened by *unnest* [FT83], which outputs pairs consisting of a *Student* object and a *Course* OID. The OIDs are resolved by the succeeding *materialize*. Expression (2) is value-based, obtained from Expression (1) by applying the join materialization rule to the upper *materialize*. In this expression, the left join operand provides a stream of *Student* object-*Course* OID pairs. The right operand provides a stream of *Course* objects. The streams are then joined.

Note that it is usually impractical to apply the join materialization rule to *materialize* on the CVA instances themselves, for example, $M_{s.Core}$, because the CVA instances (that is, the collections themselves) generally have no appropriate extent. The hybrid rule, however, does make sense for such operators. One can get Expression (3) from Expression (1) by applying this rule to $M_{s.Core}$. Further, one can get Expression (4) from Expression (3) by pushing down *unnest* and *materialize*. Note that, in both hybrid expressions, join predicate is an identity check on CVA instances, rather than on CVA elements as in a value-based expression.

In the context of CVAs, the hybrid technique has two advantages over the value-based approach. First, it applies to any CVA query, while value-based techniques are subject to the availability of appropriate type extents. Second, a hybrid expression in general has a smaller left join operand, because it does not flatten CVA instances in there, while a value-based expression does, as illustrated by Expressions (4) and (2) in Figure 5. This difference often makes a hybrid expression superior to its value-based counterpart, which is the opposite to what happens for single-valued attributes. We will see in the experimental results that value-based expressions are often slightly better than hybrid-based expressions, due to smaller overheads.

Compared to pointer-based techniques, the hybrid technique also has two advantages in the context of CVAs. First, the *projection* operator in the hybrid rule separates children from parents. This separation removes parent attributes from the input to the succeeding *materialize*, while the corresponding *materialize* in a pointer-based expression accepts both parent attributes and element OIDs. Note that parent attributes will be highly redundant in the pointer-based case, because they are duplicated in order to be paired with each element OID in the CVA during flattening. The second advantage is that a hybrid expression eliminates duplicate CVA instances, thus reducing input cardinalities to relevant operators. (We are assuming here that collections have their own identifiers that can be compared. We are not proposing to detect the case where two distinct collection instances happen to contain the same set of elements.) A pointer-based expression, however, has no such mechanism. Shared CVAs may occur in a database, as illustrated by the previous query. They may also be present in views, or intermediate query results, as illustrated by the example query below.

**Query 4** : Return all professors who advise Ph.D students:

```
select distinct f
from Depts as d, d.Majors as s, d.Faculty as f
where s.status = 'Ph.D' AND f.name = s.advisor
```

Expression (1) in Figure 6 is pointer-based, where two CVAs, *d.Majors* and *d.Faculty,* are successively flattened and materialized. Expression (2) is value-based and materializes *d.Faculty* using a *join*. Expression (3) is hybrid, derived from Expression (1) by applying the hybrid rule to $M_{d.Faculty}$, and then pushing down the succeeding operators. Expression (4) is also hybrid, but derived from Expression (1) by transforming $M_{d.Majors}$.

Expression (1) performs an operation similar to Cartesian product between *d.Majors* and *d.Faculty* (creating a tuple for every combination of student and professor in the same department), thus producing large intermediate results. Expression (2) has a larger left join operand than Expression (3) or (4). Therefore, Expressions (3) and (4) are superior to Expressions (1) and (2). Expression (4) apparently incurs less overhead than Expression (3) in terms of common sub-expressions. However, if the predicate *s.status='Ph.D'* removes many *Department* objects (because many departments do not have Ph.D candidates), Expression (3) is better due to the smaller right join operand.

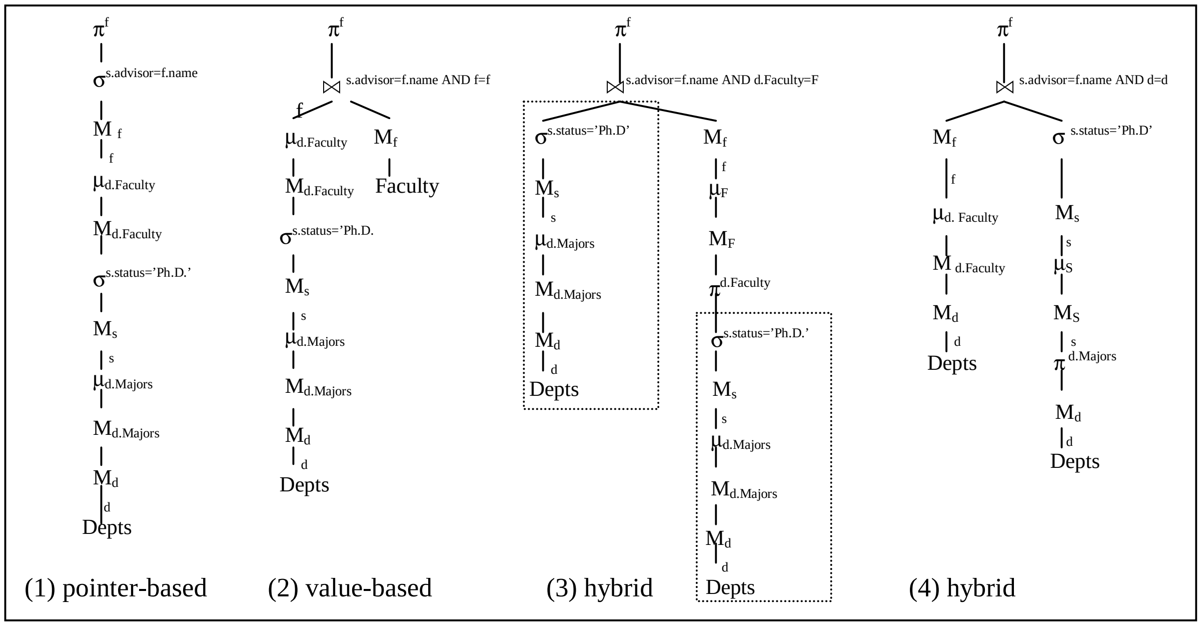


Figure 6: Expressions for Query 4

There are other ways of obtaining Expression (3) than by using the hybrid rule. For instance, a query interpreter may translate Query 4 into Expression (3) directly, by creating the same sub-expression for each occurrence of a CVA. A potential problem is that generating other alternatives such as Expression (1) and (2) is then difficult, whereas these alternatives may lead to promising expressions. Another way of generating Expression (3) is using the transformation rule $A = A \bowtie A$. This rule bears some resemblance to the hybrid rule. However, the hybrid rule is motivated by the observation that duplicating expressions in the presence of CVAs often gives favorable expressions, thereby we regard it as containing more useful guidance than the $A = A \bowtie A$ rule. While $A = A \bowtie A$ applies to any expression, the hybrid rule applies only to *materialize* operators. For instance, the $A = A \bowtie A$ rule can generate numerous alternatives, among which few perform well, while among four alternatives that the hybrid rule may generate, at lease two (Expressions (3) and (4)) are promising ones. Furthermore, the hybrid rule can only be applied once at a given point in an expression, where as the $A = A \bowtie A$ rule can be applied an unlimited number of times at the same location, requiring some additional mechanism to bound its use.

## 5 Enhancing Value-based Algorithms

It has been noted that a value-based expression may be less efficient for CVAs than its hybrid counterparts, due to larger join input. In Figure 6, the value-based expression, Expression (2), has a large left join input, because CVA instances in the left join operand are flattened to allow evaluation of the join predicate. A hybrid expression avoids this problem by checking the identities of CVA instances rather than those of CVA elements in the join predicate. We attempt to overcome the problem in the value-based case by using the identities of parent objects in the join predicate. This modification is possible if there is a relationship with the appropriate inverse attribute. Notice that, in the example schema, *Professor* and *Department* have a *1:n* relationship via *dept* and *Faculty*. This connection implies that the predicate $f = f$ in Expression (2) can be replaced by $d = f.dept$, which makes $\mu_{d.Faculty}$ and $M_{d.faculty}$ redundant. Expression (5), in Figure 8, can be regarded as an alternative of Expression (2) in Figure 6 by replacing the join predicate and removing the unnesting operator. Apparently, Expression (5) overcomes the problem of large join input, and performs competitively among the alternatives.

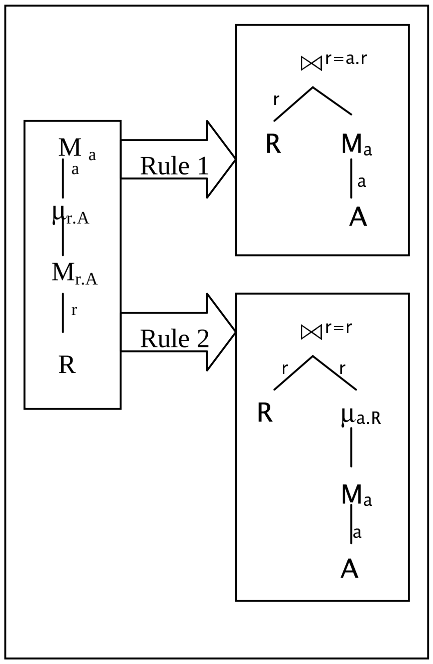


Figure 7: Enhancement Rules

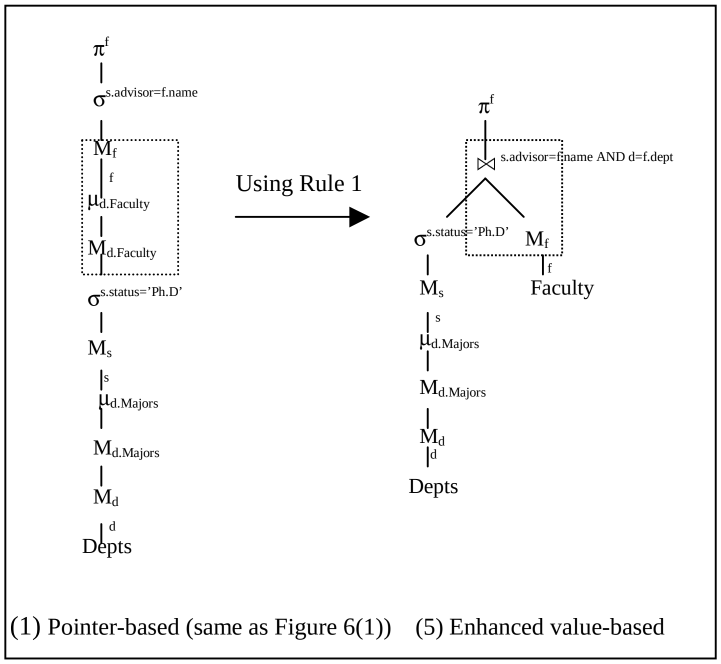


Figure 8: Another Expression for Query 4

Figure 7 illustrates an enhancement rule, Rule 1, which utilizes a *1:n* relationship to generate a value-based expression from a pointer-based expression. Assuming that *R* and extent *A* have a *1:n* relationship via attributes *a* and *R*, Rule 1 transforms an *unnest* operator and two *materialize* operators into a *materialize* and a *join*, using the parent identity as join condition. In Figure 8, this rule transforms Expression (1) of Query 4 into Expression (5).

Rule 1 improves value-based algorithms using *1:n* relationships. Extending this idea to *m:n* relationships gives rise to Rule 2, the second enhancement rule also depicted in Figure 7. Assuming an *m:n* relationship between *R* and *A*, Rule 2 transforms an *unnest* operator and two *materialize* operators into an *unnest*, a *materialize*, and a *join*, using the parent identity as a join condition. As with Rule 1, the motivation for Rule 2 is to reduce *join* costs by shrinking *join* operands. This is especially beneficial when the left join operand is much larger than the right one so that the effort of partitioning and iterating it dominates the overall join cost, as illustrated by the example query below.

**Query 5:** Returns student-course pairs where the student is a MBA candidate, and takes the course that is taught by his or her advisor:

```
select struct(c:c, p:p)
from Courses as c, c.Instructors as i, c.Participants as p
where p.advisor = i.name AND p.status='MBA'
```

For this query, Expressions (1) and (2) in Figure 9 both generate large intermediate results, by successively flattening two CVAs. Consequently, the top *materialize* in Expression (1) and the *join* in Expression (2) become very expensive. Consider the *join* in Expression (2). Assume neither operand fits in memory. Then the effort in partitioning and iterating the left operand dominates the join cost, because the left operand is much larger than the right one. Apparently, reducing the left operand helps lower join cost. Utilizing the *m:n* relationship between *Course* and *Student*, Rule 2 transforms Expression (1) into Expression (3), which has much lower join cost.

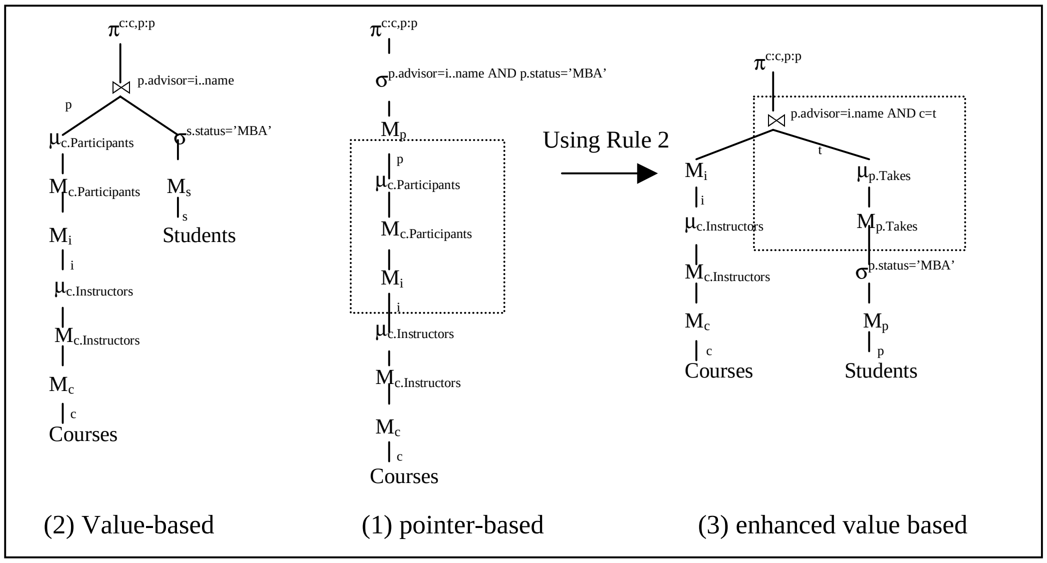


Figure 9: Expressions for Query 5

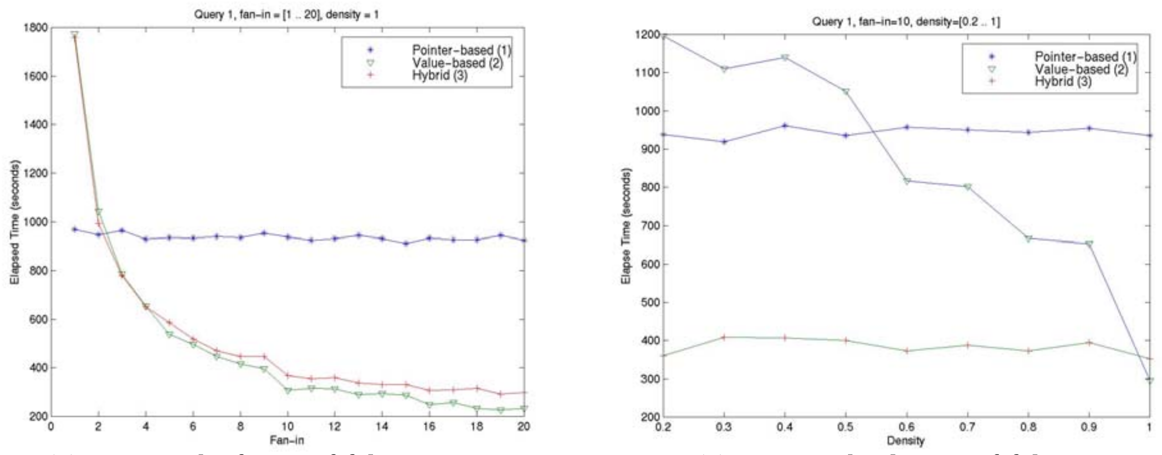
Following the same idea, one can improve hybrid algorithms using *m:n* relationships. Note that the hybrid technique is efficient when applied to a CVA that has *1:n* relationship with its children; and is less efficient when applied to a CVA that has *m:n* relationship with its children. Consider Expression (1) of Query 5 in Figure 9. If the hybrid rule is applied to $M_{c.Participants}$ in Expression (1), the result expression is not efficient because it does not eliminate duplicate student objects in the right join operand. On the other hand, if the rule is applied to $M_f$, the result expression is still inefficient because of a large left join operand. To solve this problem, one can apply an enhanced rule similar to Rule 2 on $M_p$, thus achieve an enhanced hybrid expression that performs comparably to the enhanced value-based expression,

## 6 Experimental Results

In this section, we present experimental results from implementing the hybrid and enhanced techniques. The experiments were conducted on a Sun SparcStation/20 and with the GemStone/S system, a commercial object-oriented database system [Gem96]. A query evaluator was developed on GemStone for these experiments. The evaluator supports such operators as *unnest*, pointer-based *materialize*, *join*, *projection*, and *select*. All the operators, except *unnest* and *select*, are implemented using hybrid hash algorithms. The operators are implemented in Smalltalk-DB, the DML of GemStone/S. If not stated otherwise, a hybrid expression evaluates common sub-expressions as many times as they appear. We present the experimental data for some of the queries mentioned in this paper. For each query examined, the alternative expressions representing different approaches were executed. The performance was evaluated under various sharing situations and extent densities. We measured CPU time, I/O amount, and total elapsed time. In most experiments, I/O and CPU costs were affected in the similar fashion. Therefore, we use elapsed time as our performance criterion in this presentation. The disk page size and buffer size in GemStone are 8K. We chose the sizes of our test databases to be relatively modest, so we could run a large number of tests. In most cases, the values shown are the result of one run. However, where two techniques had performance that was close together, the data points are averages of several runs, in order to see fine distinctions. We organize this section in two parts. First, we examine the performance of the three basic approaches, namely, the pointer-based, the value-based, and the hybrid techniques, for both single- and collection-valued attributes. Then, we evaluate the effect of the enhancement rules.

## 6.1 Evaluating Pointer-based, Value-based and Hybrid Techniques

We begin by demonstrating the performance of the pointer-based, value-based and hybrid techniques in the setting of single-valued attributes using Query 1. Three expressions in Figure 4 are executed when the fan-in or the density of *Department* objects varies. Recall the notion of fan-in and density in Section 1. For Query 1, fan-in means the average number of professors in the same department; the density is the ratio between the number of departments that have faculty member specializing in Mathematics and the total number of departments.

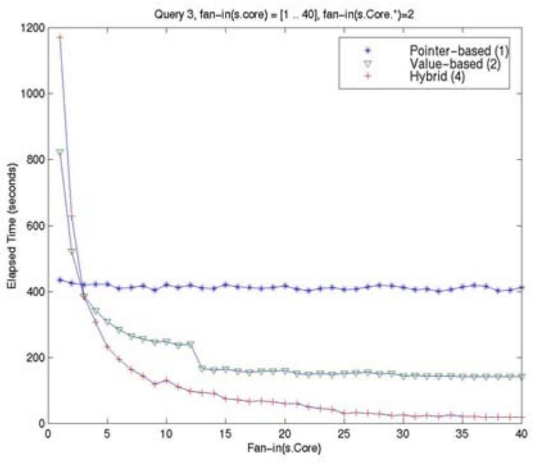


(1) Varying the fan-in of *f.dept* (2) Varying the density of *f.dept*

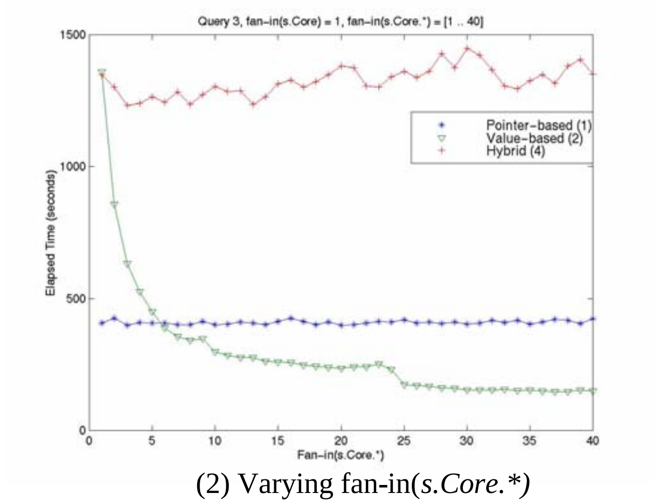
Figure 10: Query 1, elapsed time of the expressions in Figure 4

Figure 10 (1) illustrates the elapsed time of three expressions when the fan-in of *f.dept* changes. When the fan-in equals one, the pointer-based expression is the cheapest, while the other two expressions suffer from large join operands. However, once sharing occurs, the curves for the hybrid and value-based expressions drop quickly, while the one for the pointer-based expression remains high. At higher fan-in, it becomes obvious that the hybrid expression is more expensive than its value-based counterpart, though in a narrow margin, due to the overhead of an extra projection operator and common sub-expressions. Figure 10 (2), however, shows the opposite case, where the hybrid expression outperforms the value-based expression as the extent of *Department* becomes sparse. This figure contrasts the three expressions as the density of *Department* increases. The value-based expression is cheapest when the density is one, but degrades as the density goes down, and finally becomes the most costly. The other expressions are not affected by density change, since they don't use the extent.

Now let us evaluate the three techniques in the context of CVAs using Query 3. The test data consists of a total of 2048 *Student* objects, with 80 elements on average in each *s.Core* instance. Three expressions in Figure 5, Expressions (1), (2), and (4), are executed as the fan-in of *s.Core* instances or the fan-in of *s.Core* elements changes. The fan-in of *s.Core* instances, denoted as *fan-in(s.Core),* is the average number of students with the same core course requirement. The fan-in of *s.Core* elements, denoted as *fan-in(s.Core.*),* is the average number of *Core* collections that overlap on the same course.



(1) Varying fan-in(*s.Core*)

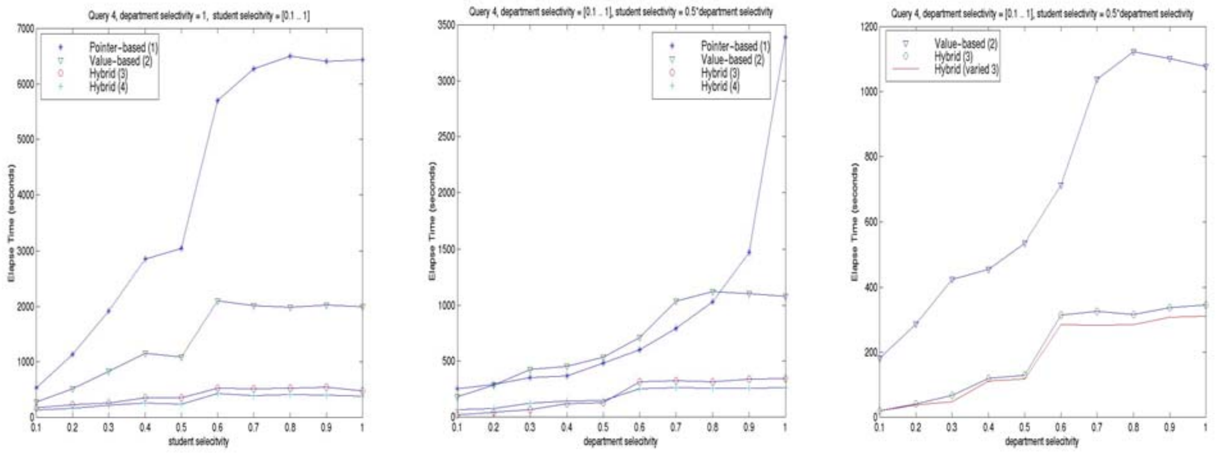


(2) Varying fan-in(*s.Core.*)*

Figure 11: Query 3, elapsed time of the expressions in Figure 5

Figure 11(1) varies the fan-in of *s.Core* instances, while the fan-in of *s.Core* elements is fixed as two. When the fan-in of *s.Core* instances is one, the hybrid and value-based expressions are more expensive than the pointer-based one. As the fan-in grows, the costs of the two expressions drop quickly. The decrease essentially stops when the fan-in passes 15, because, for both expression, as the previously dominating operands shrink to a certain degree, other cost factors that are independent of density change starts to dominate overall expression costs. In this figure the hybrid expression outperforms the value-based expression, because of the large left *join* operand in the value-based expression. Figure 10(1) and Figure 11(1) suggest that value-based algorithms generally have less overhead than hybrid algorithms when applied to single-valued attributes, but more when applied to CVAs.

Figure 11(2) demonstrates the performance of Query 3 when the fanin of *s.Core* elements goes from 1 to 40, with the fan-in of *s.Core* instances fixed as one. Different data sets with various fan-ins are used in the test. Since there is no sharing among *s.Core* instances, the hybrid expression suffers from a large right join input, and becomes the worst algorithm throughout the test. The value-based expression, however, takes advantage of the smaller extent size as fan-in grows, and becomes the best as the fan-in increases. This experiment suggests that the hybrid approach is sensitive to the existence of object sharing.



(1) Varying student selectivity (2) Varying department selectivity (3) Varying department selectivity

Figure 12: Query 4, elapsed time for the expressions in Figure 6 and Figure 8

To examine the three techniques for a somewhat more complex CVA query, we execute the expressions for Query 4. The test data includes 300 departments, each with 30 faculty members and 60 students. The expressions in Figure 6 are executed as student selectivity or department selectivity changes from 0.1 to 1. *Student selectivity* is the ratio between the number of Ph.D students and the total number of students. *Department selectivity* is the ratio between the number of departments with Ph.D students and the total number of departments.

Figure 12(1) shows the elapsed time for the four expressions relative to student selectivity, with department selectivity fixed at one, meaning every department has some Ph.D students. As the selectivity increases, the costs of both the pointer-based and value-based expressions grows rapidly, while the hybrid expressions are quite stable at the lowest cost level. When the selectivity is high, the pointer-based expression suffers from large intermediate results, especially the input to the top *materialize* operator. Similarly, with high selectivity, the value-based expression incurs a high *join* cost due to a large left *join* operand. Both hybrid expressions demonstrate their superiority. Expression (3), however, costs slightly more than Expression (4), due to bigger common sub-expressions.

Figure 12(2) demonstrates the behavior of the four expressions as predicate *s.status=’Ph.D’* removes more and more departments. The time for these expressions is plotted against department selectivity. Student selectivity is linear in department selectivity with a factor of 0.5, which means, for each department that has Ph.D students, 50 percent of its students are selected. Different from in Figure 12(1), in Figure 12(2), Expression (3) outperforms Expression (4) when the selectivity is small, because ruling out many departments during selection help reduce the right *join* operand for Expression (3), but not for Expression (4). This experiment also shows that the pointer-based expression can outperform the value-based expression with small selectivity, even though it tends to diverge when selection becomes less restrictive.

Hybrid expressions contain common sub-expressions. To avoid evaluating a sub-expression twice, one can store and reuse its evaluation result. We apply this strategy to Expression (3) of Query 4, and illustrate its performance in Figure 12(3), where *hybrid (3)* stands for a algorithm derived from Expression (3) that evaluates the common sub-expression twice, and *hybrid (varied 3)* represents a algorithm that stores and reuses the intermediate result. The test is conducted using the same configuration as for Figure 12(2). The pointer-based expression is also shown for reference. At least in this case, storing and reusing results of common sub-expressions does not bring significant benefit.

## 6.2 Evaluating Enhanced Algorithms

We use Query 4 to evaluate the first enhancement rule, Rule 1. Figure 13 contrasts the value-based and enhanced value-based expressions (in Figure 6 and 8 respectively). The pointer-based expression is also included for reference. The test data includes 300 departments, each with 30 faculty members and 60 students. The elapsed time of the three expressions is shown with varied student selectivity. When the selectivity is small, the gap between the value-based and the enhanced value-based is minimal. As the selection becomes less restrictive, the gap grows rapidly, because the left join operand expands more quickly in the value-based expression than in the enhanced value-based expression. Exploiting a *1:n* relationship, the enhanced technique achieves significant improvement compared to the original value-base algorithms.

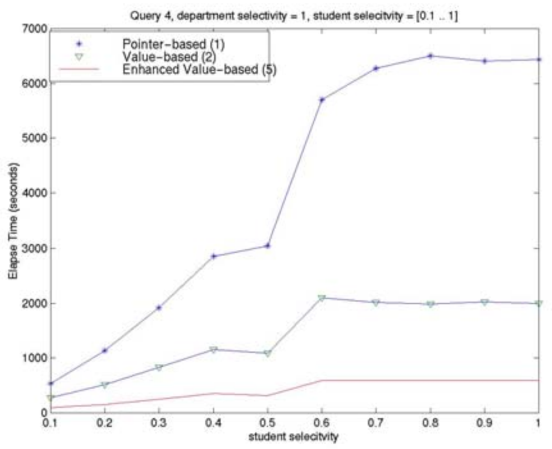


Figure 13: Query 4, elapsed time versus student selectivity while department selectivity equals one

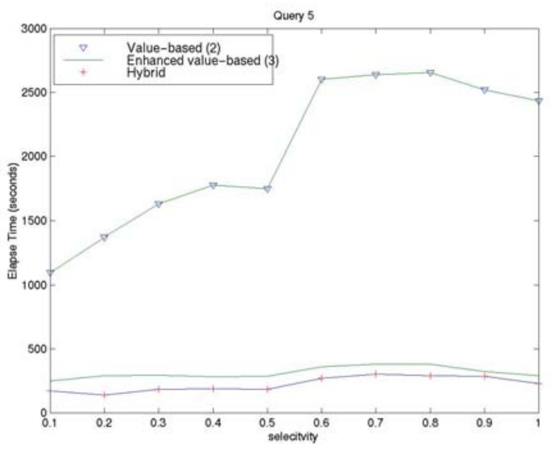


Figure 14: Query 5, elapsed time versus selectivity

Figure 14 evaluates the second enhancement rule, Rule 2, using three expressions for Query 5. Among the three expressions, value-based and enhanced value-based are illustrated in Figure 9. The hybrid expression, not depicted in the figure, performs *join* between the unnesting result of *c.Instructors* and a *select* operator which filters the unnesting result of *c.Participants* using the predicate *p.status='MBA'*. The test data includes 300 courses, each with 10 instructors and 40 students. In Figure 14, elapsed time of the three expressions is depicted as a function of the selectivity of the predicate *p.status='MBA'*. The value-based expression chooses the smaller operand, the right branch in Figure 9 (2), as the inner operand. Thus it is sensitive to selectivity change, while the other two expressions are barely affected, because they choose different inner operands that happen to fit in memory. This test demonstrates that, utilizing a *m:n* relationship, the enhanced technique can significantly improve value-based algorithms. We also conduct experiments with the same enhancement for the hybrid technique, which yielded similar results to those of Figure 14.

We see that the two enhancement rules can significantly improve the value-based technique using relationships. Thus, relationships captured as inverse attributes can be quite useful in evaluating queries with CVAs, and an optimizer can benefit from being aware of such relationships.

## 7 Related Work

Other materialization techniques not covered in this paper are indexing [MS86, BK89] and access support relations [KM90]. An index maps attributes to their parents. Retrieving parents through their entries in an index is itself a materialization process, where the techniques discussed in this paper apply. An index may also be considered an implementation of relationships (with all the inverse references gathered into one place), and thus can be used in the enhancement methods. Access support relations are a mechanism that stores the results of reference materialization ("materialization materialization"), which is complementary to the materialization process.

Path expression processing [ODE95, GGT96] aims at finding the optimal evaluation order for segments of path expressions, assuming that segments can be evaluated using value-based algorithms. Since original value-based techniques are not efficient for processing CVAs, the enhanced technique could play a key role in adapting existing path expression processing techniques to collection-valued attributes. Compared to the value-based techniques, one limitation of the hybrid technique is that it does not facilitate path expression re-ordering. Therefore, when processing a path expression, all the materialization techniques might be employed. For instance, the path expression can be cut into sub-paths. Within a sub-path, various algorithms can be applied, while between sub-paths, value-based algorithms are employed to facilitate reordering.

Ross [Ross95] proposes decomposing parent and child objects at the beginning of materialization, then sorting and resolving the child references, and finally merging the parent and children attributes to give the materialization result. His algorithm minimizes the space taken by the sorting phase, and under certain circumstances, eliminates the need for partitioning. However, the benefit of decomposition is localized in that it only reduces the cost of materialization itself, while hybrid algorithms may benefit other operators as well.

Similar to join materialization [BMG94], the hybrid rule transforms an implicit join into an explicit one, but duplicates the input expression instead of using extents.

Braumandl et al. [BCK98] base a new pointer-based algorithm for CVA attributes on a storage model similar to GemStone and Shore. To our knowledge, their work is the first effort towards adapting pointer-based algorithms to CVAs. The enhancement algorithms we present can be considered an initial effort to adapt value-based algorithms in the same manner.

We consider set comparison too expensive for join predicates. Therefore, we did not include alternatives that use set predicates in materializing CVAs. However, Helmer and Moerkotte [HM97] demonstrate that set predicates can be implemented efficiently using superimposed signatures. Such techniques can be used together with the value-based and hybrid techniques for CVA materialization.

## 8 Conclusions and Future Work

We have extended the current body of work on materialization techniques in two dimensions. First, to address major shortcomings of existing techniques, we proposed the hybrid technique that gives rise to competitive materialization algorithms in many circumstances. Second, we present some enhancements for existing techniques when they are applied to CVAs.  Initial experiments with an evaluator written on a commercial object-oriented database confirm our informal analyses of various techniques.

More techniques for materialization and CVAs mean a larger search space for the query optimizer. Exhaustive search of this space might not always be desirable. Therefore we will do more analytical and experimental evaluation to develop guidance for optimizers to generate promising expressions quickly. For instance, among many materialize operators in an original expression, an optimizer should seek to choose the most effective places to fire the hybrid rule and enhancement rules. One strategy is to apply the hybrid rule where attributes are highly shared, or where the materialization input has not had the hybrid rule applied already. Of course, we also need to develop appropriate cost functions and statistics so that we can reliably choose a relatively inexpensive plan among the alternatives.

The work described in this paper is part of an ongoing effort to explore new techniques for efficiently processing CVA queries. We have noted three kinds of CVA query techniques, and classify them according to the algebra in which queries are represented and transformed. The *nested algebra approach* [SS86] allows bulk operators on CVAs to be performed and transformed within a nested projection operator. *The recursive algebra approach* [Colby89] includes complex *select*, *projection*, and *join* operators that can manipulate both parent objects and CVA elements. These operators simulate their relational counterparts, but accept and produce nested collections (collections with CVAs). The *flat algebra approach* [Steen95, Feg98] uses essentially the relational algebra to process CVAs queries. Since the relational algebra has no means of manipulating CVAs, CVA instances have to be unnested before being accessed by relational operators.

Besides the relational operators, an actual algebra may include *unnest* and *nest* [FT83]. The *nest* operator achieves an effect similar to what is provided by a group-by operation, except that *nest* may generate CVAs for a group instead of a single value.

The flat algebra approach is more compatible with conventional query processing frameworks, thus is adopted in much work on object query optimization [Cluet93, Feg98, Steen95]. However, we observe that each approach has its own advantages, and can outperform the others in certain circumstances. Based on this observation, our work on CVA query processing will be carried on in two directions. One is to explore evaluation techniques for these three approaches. Lack of efficient algorithms is a major factor impeding the adoption of the nested and recursive algebra approaches. Good materialization strategies may alleviate this problem. So far, most work on materialization (including this paper) assumes the flat algebra approach. However, Braumandl et al.[BCK98] demonstrate that adapting the pointer-based algorithms to the nested and recursive algebra approaches is possible. Other adaptation and techniques remain to be explored. The other direction for investigation is a combined query processing approach that integrates more than one existing approach. An appropriate algebra and corresponding transformations will be developed to generate a combined expression space, which will likely much larger than the space for any single approach. Thus, exhaustive search may become impractical. Therefore, effective search strategies, for instance, pruning techniques [Shapiro et al. 99], will also be investigated to limit the search effort required for optimizing queries.

## Acknowledgement

The first and second authors are supported by NSF IRI-9509955, IRI9118630, DARPA(BAAB07-91-C-Q513). The third author is supported by NSF IRI-9119446, DARPA(BAAB07-91-C-Q513) subcontract from Oregon Graduate Institute to Portland State University.

## Bibliography

[BCK98] R. Braumandl, J. Claussen, A. Kemper. *Evaluating functional joins along nested reference sets in object-relational and object-oriented databases*, 1998 VLDB.

[BK89] E. Bertino, W. Kim. *Indexing techniques for queries on nested objects*, in IEEE Trans. Knowledge and Data Engineering, June, 1989.

[BMG93] J. A. Blakeley, W. J. McKenna, G. Graefe. *Experiences building the Open OODB query optimizers*, 1993 SIGMOD.

[Carey et al. 94] M. J. Carey, D. J. DeWitt, M. J. Franklin, N. E. Hall, M. L. McAuliffe, J. F. Naughton, D. T. Schuh, M. H. Solomon, C. K. Tan, O. G. Tsatalos, S. J. White, M. J. Zwilling. *Shoring up persistent applications*, 1994 ACM SIGMOD.

[CB97] R. G. G. Cattel, D. K. Barry. *The object database standard: ODMG 2.0,* Morgan Kaufmann Publishers, Inc, 1997.

[CM93] S. Cluet, G. Moerkotte, *Nested queries in object basess*, Proc. Int. Workshop on Database Programming Languages, 1993.

[Colby89] L. Colby. *A recursive algebra and query optimization for nested relations*, 1989 ACM SIGMOD.

[CG94] R. L. Cole, G. Graefe. *Optimization of Dynamic Query Evaluation Expressions*, 1994 SIGMOD.

[EM99] A. Eisenberg, J. Melton. *SQL:1999, formerly known as SQL3*, SIGMOD Record, 18(8), March, 1999.

[Feg98]L. Fegaras. *Query Unnesting in Object-Oriented Databases*, 1998 ACM SIGMOD.

[FT83]P. C Fisher , S. J. Thomas. *Operators for non-first-normal-form relations*, Proc. IEEE Computer Software and Applications Conf., 1983.

[GM93] G. Graefe, W. J. McKenna. *The Volcano optimizer generator: extensibility and efficient Searches*, Proc. IEEE Conf. on Data Eng., Vienna, Austria, 1993.

[Gem96] *GemStone System Documentation*, Gemstone Inc., 1996.

[GGT96] G. Gardarin, J. R. Gruser, Z. H. Tang. *Cost-based selection of path expression processing algorithms in object-oriented databases*, 1996 VLDB.

[Gra93] G. Graefe. *Query evaluation techniques for large databases*, ACM Computing Surveys, vol. 25, No. 2, June, 1993.

[HM97] S. Helmer. G. Moerkotte. *Evaluation of main memory join algorithms for joins with subset join predicates*, 1997 VLDB.

[KMG] T. Keller, G. Graefe, D. Maier. *Efficient assembly of complex objects*, 1991 ACM SIGMOD.

[KM90] A. Kemper, G. Moerkotte. *Advanced query processing in object bases using access support relations*, 1990 VLDB.

[MS86] D. Maier, J. Stein. *Indexing in an object-oriented DBMS*, Proc. 1986 Int. Workshop on Object-Oriented Database Systems, 1986, IEEE CS Press.

[ODE95] C.Ozkan, A. Dogas, C. Everndilek. *A heuristic approach for optimization of path expressions*, Technical Report, Middle East Technical University, 1995.

[ODMG] The object database standard: ODMG 2.0, edited by R. G. G. Cattell, D. K. Barry, Morgan Kaufmann Publishers, 1997.

[Ross95] K. A. Ross. *Efficiently following object references for large object collections and small main memory*, Proceedings of the Fourth International Conference on Deductive and Object-Oriented Databases, December, 1995.

[Rama97] R. Ramakrishnan. *Database management systems*, McGraw-Hill, 1997.

[Rama et al. 98] K. Ramasamy, P. M. Deshpande, J. F. Naughton, D. Maier. *Set-valued attributes in O/R DBMS: Implementation options and performance implications*, to be published, University of Wisconsin, Madison, 1998.

[Shapiro et al. 98] L. Shapiro, D. Maier, K. Billings, Y. Fan, B. Vance, Q. Wang, H. Wu. *Safe pruning in the Columbia query optimizer*, www.cs.pdx.edu/~len/pruning.doc or pruning.doc.zip.

[Steen95] H. J. Steenhagen. *Optimization of object query languages*, Ph.D dissertation, Universiteit Twente, 1995.

[Ses98] P. Seshadri. *Query processing techniques for correlated queries*, IBM Technical Report RJ 10129 (95004), 1998.

[SC90] E. J. Shekita, M. J. Carey. *A performance evaluation of pointer-based joins*, 1990 ACM SIGMOD.

[SS86] H. J. Schek, M. J. Scholl. *The relational model with relation-valued attributes*, Information Systems, 11(2), 1986.