Relational Query Optimization with Enhanced ADTs

Praveen Seshadri
Cornell University
praveen@cs.cornell.edu

Abstract

Object-relational queries access large complex data types and expensive methods of those data types. In earlier work, we modeled complex types as “Enhanced ADTs” (E-ADTs) and demonstrated the resulting performance improvements when implemented in the PREDATOR system. This paper explores the opportunities for further improvements through interactions between relational query optimization and E-ADT optimization. We identify four broad categories of optimization opportunities and study specific examples in each of these categories. These examples span query rewrite, indexing, aggregation and join optimization. Our conclusion is that non-trivial interactions exist between E-ADTs and relational queries, and that special optimization techniques are necessary to achieve good performance. These techniques have been prototyped in PREDATOR, and we present experimental results that demonstrate their effect.

1. Introduction

Current first-generation object-relational database systems (OR-DBMSs) model complex data types as “blackbox” ADTs. The database system has little semantic information about the data type or its methods. Consequently, the performance of queries that use the ADTs can suffer. For example, consider the query below:

```
SELECT S.picture.sharpen().crop(0,0,0.1,0.1)  FROM  SatelliteData S
```

The *S.picture* field belongs to an image ADT with *sharpen()* and *crop()* methods. An OR-DBMS based on blackbox ADTs would apply the expensive *sharpen()* method to each large image and then apply the *crop()* method to the result to extract the desired small portion (a tenth of the height and width). The Enhanced ADT (E-ADT) design paradigm described in [2] allows complex data types to specify some semantics associated with each method so that optimizations can be applied. In this example, the developer of the image E-ADT can specify that the *sharpen()* and *crop()* methods commute. A system like PREDATOR [5] that supports E-ADTs also considers an alternative evaluation strategy: the small region of interest can be cropped from each image and then sharpened. The lower-cost strategy is chosen for actual evaluation. In effect, this corresponds to treating the ADT method expressions in a declarative fashion (rather than the procedural fashion dictated by blackbox ADTs). The performance impact is very significant. This is a synopsis of our previous work; note that its focus was on applying optimizations within method expressions (like *sharpen().crop(…)*).

This paper expands the E-ADT paradigm to explore optimizations on the boundary between relational query execution and the execution of the method expressions. We develop four broad categories of optimization opportunities and study specific techniques that fall into these categories. The proposed techniques have been prototyped in PREDATOR, and we use performance experiments to measure their effect. The relational components affected include the query rewrite engine, the join enumeration mechanism, the use of indexes, the processing of aggregates, the placement of predicates, and in effect, the entire process of query planning.

The topic of this research arises out of two conflicting design requirements. On one hand, it is desirable that the type system of the DBMS be totally modular and extensible. This means, for example, that all support for an image data type must be totally encapsulated within a module that can be added or removed without affecting the rest of the system. This motivation has dominated the blackbox ADT design of today's first-generation OR-DBMS engines. On the other hand, it is desirable to achieve efficient query execution, which blackbox ADTs do not accomplish. Our earlier work on E-ADTs improved the performance of individual ADT method expressions; this accelerates certain queries by orders of magnitude. However, while this is clearly an improvement over blackbox ADTs, there are further optimization opportunities in the interaction between method expressions and the relational query engine. The primary conclusion of the paper is that it is possible to further extend the E-ADT paradigm to improve query performance, while still maintaining the modularity of the type system.

The contribution of the paper is not a single idea or technique. Rather, it is the identification of the opportunity for optimization across E-ADT boundaries, and the development of practical mechanisms to exploit this opportunity. A collection of practical query processing and optimization techniques have been explored while implementing support for complex data types in PREDATOR. They arise because the E-ADT design paradigm allows the database system to treat complex data types in a "smart" manner. However, instead of viewing them as an ad-hoc collection of optimization hacks, we develop broad categories of optimizations into which these techniques fall. We will focus on object-relational systems in this paper, since our implementation has been in PREDATOR; however, this research is equally applicable to object-oriented databases.

Paper Outline

The outline of the paper is as follows: We begin with a review of the current techniques for relational query optimization in the presence of complex data types. We also present an overview of PREDATOR and contrast its support for complex data types with similar functionality in commercial OR-DBMSs. In the rest of the paper, we describe different issues of interest through examples, suggest possible techniques to optimize performance, and present experiments to demonstrate the effects of these techniques.

2. Background

Current OR-DBMS Technology

Most object-relational database systems (a.k.a "universal servers") model complex data types as blackbox ADTs. The methods of an ADT are characterized by the type signature and by a cost function. The Postgres[10] project suggested that ADT methods could be used to allow standard indexes to be created on complex data types. For instance, a B+-tree index can be built on any arbitrary data type, as long as an ordering method is provided. This idea has recently been generalized for a variety of tree-based indexing methods [11]. As an orthogonal feature, most database systems allow indexes to be created on method/path expressions (for instance, an index on *SatelliteData.picture.height()*).

The cost of a method is specified by the developer of the ADT. It is either a constant or is parameterized by the sizes of the arguments to the method. Borrowing a common example from the Postgres project, an image ADT may have a method called *snowcover()* that estimates the percentage of snowy terrain in a satellite photograph. A possible query would be:

```
SELECT  * FROM SatelliteData S WHERE S.year > "1994" AND S.picture.snowcover() > 30
```

Assume that the *snowcover()* method has been assigned a cost of 5000 units. In this case, the relational query optimizer can identify the predicate involving this method as an "expensive predicate", and can choose to apply it after the other predicate (*S.year > "1994").* One expects that the expensive predicate will therefore be applied fewer times, resulting in a cheaper query execution. The query optimization strategy becomes more complicated if the query involves joins in addition to expensive predicates. Each predicate plays a role in reducing the cardinality of the tables involved in joins. However, the result of a highly selective join can have a lower cardinality than the input relations. With an expensive predicate, the desire to apply the predicate early (to make subsequent joins cheaper) conflicts with the possible benefits of executing the join first. If the results of predicates can be cached, the delayed execution of expensive predicates becomes more attractive. A number of different query optimization strategies have been suggested to deal with this issue ([7,6]). It appears that the simple strategy of delaying the application of expensive predicates until the end of query execution performs well in most cases([7]).

In order to improve the performance of queries with expensive methods, caches can be introduced to hold the method results. Each time an expensive method needs to be evaluated, the cache is first searched to see if the pre-computed result from a previous invocation is available. A variant of this approach caches the boolean result of predicate invocations[8]. Caching has an impact only if a method/predicate has multiple invocations with identical parameters. In this situation, an alternative approach is to sort the input relation on the parameters (and maintain a cache of size 1).

Finally, our focus has been on methods without side-effects – indeed, since the methods are embedded within SQL, a declarative language, it is difficult to reason about or optimize methods with side-effects. All the same, commercial database systems allow methods to specify a "scratch memory area" which persists across method invocations within a query.

This is essentially the state-of the-art in commercial (and research) OR-DBMS systems. Recently, it has been recognized that there is an opportunity to exploit the semantics of the ADT methods to derive further optimizations. While some research focused on specific situations in which these semantics could be used ([12,13]), a comprehensive framework of "Enhanced ADTs" or E-ADTs was presented in [2].

Enhanced ADTs and PREDATOR

E-ADTs represent a second-generation design paradigm for database systems with extensible type systems. In traditional systems, most of the optimization and query processing effort is directed at relational data. ADTs and their methods are essentially opaque "blackbox" additions to the system, which are handled as described in the previous section. The fundamental idea behind E-ADTs is that each data type is treated as a first-class entity in the database system, thereby enhancing its contribution to optimization and efficient performance. As a first step towards this design paradigm, [2] suggested that expressions involving the concatenation of E-ADT methods should be treated as composite sub-expressions that could themselves be optimized. For example, *S.picture.sharpen().crop(0,0,0.1,0.1)* can be optimized to apply the *crop()* method first and to keep intermediate results uncompressed in memory. Some optimizations are heuristic, while others are cost-based and depend on statistics collected on the underlying data (in this case, perhaps the expected size of *S.picture*).

The PREDATOR database system is an implementation of a full-fledged OR-DBMS based on the E-ADT paradigm[5]. Each data type exposes optimization semantics through a set of well-defined interfaces; for many data types, these semantics are specified through transformational rules. SQL queries can have embedded E-ADT method expressions. The optimization and processing of SQL queries follows traditional techniques. The first stage is heuristic query rewrite [14], which attempts to merge views and sub-queries into a single query block. In the next stage, a query plan is generated for each query block using a cost-based query optimizer. As we have seen in this section, the relational optimizer needs an estimate of the cost of ADT sub-expressions in order to choose a good plan. Consequently, the E-ADT sub-expressions are first optimized to provide an execution cost estimate. If the E-ADT optimizations were defined as transformational rules, the optimization of the sub-expression corresponds to the application of the rules by a rule engine. Once all sub-expressions have been optimized, the entire query block is optimized using standard optimization techniques. PREDATOR uses a variant of the KBZ optimization algorithm [9].

3. Optimization Categories

In this paper, we examine the interaction of E-ADT optimization with relational query optimization. There are two broad categories of optimization opportunities, each with two subcategories:

1. Relational optimizer helps the E-ADT optimizer be more effective:
   - By providing more opportunities for optimization.
   - By providing more information (statistics and costs) for optimization.
2. E-ADT optimizer helps the relational optimizer be more effective.
   - By providing more opportunities for optimization.
   - By providing more information (statistics and costs) for optimization.

The figure below shows this space of options as a grid. By the end of the paper, we will fill in this grid with examples of optimization techniques in each quadrant.

	More opportunities	More information
SQL Opt. Helps E-ADT		
E-ADT Helps SQL Opt.		

As a canonical application domain used throughout this paper, we will consider an application that manipulates satellite photographs. For convenience, we assume that the photographs are represented in the compressed JPEG format (rather than the more arcane AVHRR format, which PREDATOR does also support). In most experiments, the schema used is *SatelliteData(Id integer, Time integer, Band integer, Photo jpegimage)*. We fix the cardinality of the SatelliteData table at 100. There are 20 distinct time values, and for each time value, there are 5 tuples, each with a distinct *Band* in the range of 0 through 4.The average size of each image is 250KB compressed, and 8MB decompressed. In many experiments, the relation cardinalities may appear small – however, one should remember that expensive predicates are being applied to these tuples, thereby increasing the total cost of the query. In real applications, although the overall cardinalities are much larger, we expect simple predicates to reduce the number of tuples to which expensive methods need to be applied. All experiments were conducted on a 200MHz uniprocessor PC with 64MB of main-memory running PREDATOR on top of Windows NT.

4. Relational Optimizer Helping E-ADT Optimization

The relational optimizer can improve the performance of queries by enhancing the effect of the E-ADT optimization. It can create more opportunities for E-ADT optimization through the use of query block rewrite techniques. It can provide more information for E-ADT optimizations by helping with statistics for method expressions involving aggregates.

1. The effect of E-ADTs on index creation and index matching.
2. The use of E-ADT information in join optimization.

4.1 Relational Query Rewrite

The query rewrite phase is important for relational queries involving views or subqueries (i.e. multiple query blocks) --- most non-trivial applications use such queries. Typically, a rule engine fires transformation rules that modify the query in a heuristic fashion [14]. PREDATOR implements a simple version of the query rewrite engine following the techniques developed in the Starburst project[14]. A particularly common rule merges two query blocks into a single block. Query rewrite is important because typical cost-based query optimizers work at the granularity of a single query block. Consequently, if multiple query blocks can be merged into a single block, the resulting query plan may be much more efficient. In this section, we show how query rewrite can lead to complex E-ADT method expressions that need to be optimized. The reader may have wondered if the example used in the introduction is a realistic query --- indeed, we claim that such queries are commonplace precisely because of the use of views in applications, and because of query rewrite.

Example 1:

```
    CREATE VIEW TrueData AS
            (SELECT  S.time, S.band, S.picture.sharpen() as pic FROM SatelliteData S);
 Query: SELECT time, band, pic.crop(0,0,0.1,0.1) FROM TrueData WHERE pic.height() > 5;
```

The view *TrueData* is defined by a DBA or domain expert who is building a database application. The actual query would be generated automatically by a naïve user, probably through a GUI tool or by filling in fields of a form. This user does not know that TrueData is really a view and not a stored table – in fact, the user probably does not even understand SQL. When query rewrite is applied to this query, the resulting query looks like:

```
SELECT S.time, S.band, S.picture.sharpen().crop(0, 0, 0.1, 0.1)
FROM SatelliteData S where S.picture.sharpen().height() > 5;
```

There are two E-ADT expressions in this query. In the WHERE clause, *S.picture.sharpen().height() > 5* can be reduced (using E-ADT optimizations) to *S.picture.height() > 5*, since the *sharpen()* method does not change the height of the image. This is much cheaper to evaluate and is no longer considered an "expensive" predicate (we explore this topic further in Section 6). As we discuss in Section 5, this E-ADT optimization may also allow the use of an indexed access path. In the SELECT clause, the image semantics tell the system that *S.picture.sharpen().crop(0,0,0.1,0.1)* is equivalent to *S.picture.crop(0,0,0.1,0.1).sharpen()*, which is probably cheaper. Our insight from this example is:

- *Insight :* Relational query rewrite "creates" E-ADT method expressions. This is an example of the relaqtional optimizer creating opportunities for E-ADT optimization.

4.2 E-ADT Aggregate Optimizations

Example 2:

```
CREATE VIEW WholeData AS
      (SELECT  S.time, iavg(S.picture) as pic FROM SatelliteData S GROUPBY S.time);
```

Query: SELECT time, pic.deres(0.2) FROM WholeData WHERE time > "1/9/95";

This query is logically identical to a query in the Sequoia benchmark ([4]). Several images are taken at the same time in different frequency bands. *Iavg()* is a domain-specific aggregate method that creates a composite image from the individual images. The view *WholeData* is created by scientists managing the data. The external users of the database see this view as the conceptual schema. The query asks for a lower resolution picture. When the query blocks are merged, the result looks like:

```
SELECT S.time, iavg(S.picture).deres(0.2) FROM SatelliteData S
WHERE S.time > "1/9/95" GROUPBY S.time
```

Note that the SELECT clause specifies an aggregate, and the *deres()* method is applied to the result of the aggregate (the extension to SQL syntax used here is supported in PREDATOR). This is an E-ADT method expression involving an aggregate method. Can it be optimized? We describe the optimization possibilities next.

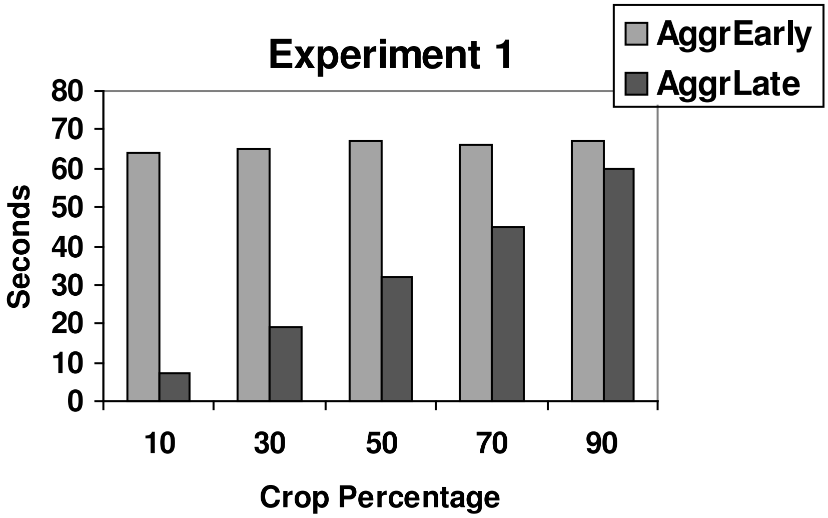
As we presented in Example 2, complex data types can specify aggregate methods in addition to scalar methods. An aggregate method is specified by three component functions for initialization, accumulation, and termination. Our earlier work on E-ADTs did not consider the optimization of expressions involving aggregate methods. Here is a simplified version of the query from Example 2:

```
SELECT  S.time, iavg(S.picture).deres(0.2) FROM SatelliteData S GROUPBY S.time
```

The naïve strategy following the textual query specification is to compute i*avg()* on all the images in a group, and then to lower the resolution of the result. However, the image E-ADT may indicate that this query is semantically equivalent to the following:

```
SELECT  S.time, iavg(S.picture.deres(0.2)) FROM SatelliteData S GROUPBY S.time
```

The difference in evaluation strategy is that *deres()* is first applied to each image in a group, and the results are accumulated to form the aggregate result. This is the form used to specify this query in the original Sequoia benchmark[4]. Which strategy is cheaper? It depends on the costs of *iavg()* and *deres()* and the number of members in each group. The costs of the methods depend on the sizes of the images. However, the estimate of the number of members in each group is not available to the E-ADT. It needs to be passed from the relational optimizer. The goal of the following experiments is to demonstrate that E-ADT aggregate optimizations can have significant impact. Let us call the two query variants AggrEarly and AggrLate respectively.

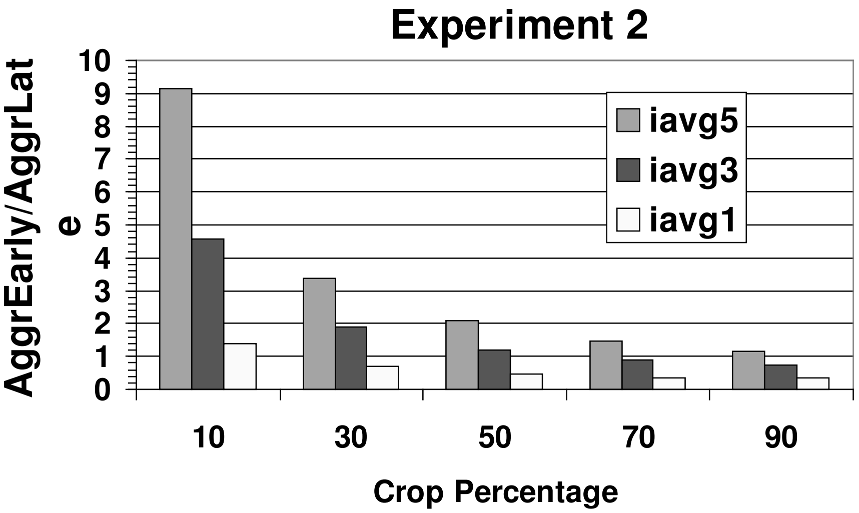


Experiment 1:

The experiment uses a slight variant of this query, but with a *crop()* method instead of *deres().* Along the X axis, we vary the percentage of the image that is cropped. On the Y-axis, we measure the query response time for AggrEarly and AggrLate. We expect that when the region cropped is small, it is more efficient to apply the *crop()* before the aggregate. However, as the region grows, we expect there to be less of a difference between the two. The graph shows, in fact, that AggrLate is always better for this query. The reason for this is that the work performed by *crop()* is linear in the size of the cropped image. The *iavg()* aggregate method is implemented as a pixel-level average image and performs work that is linear in the size of its input images. AggrLate is superior to AggrEarly because it is more effective at reducing the sizes of the image parameters to the expensive components of the methods. However, this does not imply that AggrLate is superior for all queries. The next experiment demonstrated this.

Experiment 2:

In practice, user-defined aggregate methods have diverse costs. We modify the previous experiment by varying the cost of the aggregate method. Instead of showing AggrEarly and AggrLate separately, we show their ratio for each aggregate method of different cost. The *iavg()* method used in Experiment 1 computed the average of all five images in the group. We consider two cheaper aggregate method, one which averages the first three images of a group, and the other, which just computes the (trivial) average of the first image of each group. The results show that for cheaper aggregates, AggrEarly is often a better strategy (in the parts of the graph where the ratio is less than 1). While we have run this experiment with cheap variants of aggregate methods, a similar result occurs if we fix the aggregate and vary the cost of the other method (in this case, *crop()*). In all these cases, PREDATOR is able to find the lowest-cost evaluation plan, since E-ADT optimizations are cost-based.



The size of each group helps determine the best evaluation plan. The effect of varying the group size is similar to the effect of modifying the amount of work performed in each aggregate method. The group size can be determined (or approximated) from the relational statistics gathered as part of the initialization step in optimizing a query block. It is important that this information be provided to the E-ADT optimization process.

5. E-ADT Helping Relational Optimization

An E-ADT can help the relational optimizer improve the performance of queries. It can create more opportunities for relational optimization in areas like index matching. Further, it can help the relational optimizer with join queries by providing statistics on memory use and argument locality.

5.1 Index Matching

Indexes can be created not just on attributes of a relation, but also on the results of methods of those attributes. For example, an index may be created on *SatelliteData.picture.height()* and can be used in the following query:

SELECT * FROM SatelliteData S WHERE S.picture.height() > 3000

While this functionality is supported in most existing OR-DBMSs, interesting issues arise when the indexed expression is more complex. In Example 1, the WHERE clause contains a predicate *S.picture.sharpen().height() > 3000*. If an index was explicitly created on the LHS of this expression, E-ADT optimizations can indicate that the expression is equivalent to *S.picture.height()* which is much cheaper to evaluate. This accelerates index creation time.by an order of magnitude (we omit experimental results demonstrating this optimization since the results are obvious). However, there is still one difficult issue: while some of the E-ADT optimizations including this one can be performed heuristically, other optimizations are based on statistics on the underlying arguments. These statistics do not exist if an index is created before the actual data is loaded into the table.

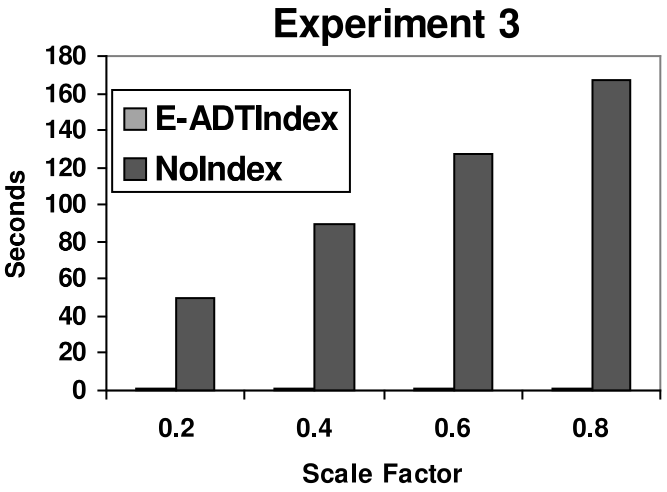
The more interesting issue is the use of indexes in query processing. Specifically, the query optimizer needs to decide if an existing index "matches" a predicate in a query, and therefore can be used as an access path. Assume that an index has been created on *SatelliteData.height()*. However, the query contains the predicate *S.picture.sharpen().height() > 3000* that reduces to *S.picture.height() > 3000*, which is an exact match for the index. An indexed access path can result in very much better performance when compared to a non-indexed evaluation strategy. In general, relational query optimizers go to great lengths to find indexed access paths. Our observation is that E-ADT transformations can help find additional indexed access paths, thereby dramatically altering the relational query plan. While the next example does demonstrate this, we do not use the specific query above. This is because on our small data set of 100 images, the absence of an index on *S.picture.height()* is not significant (i.e. the cost of performing a file scan and checking the *height()* is not significant).

Experiment 3:

We introduce a new method *NumPixels(color)* on images; it computes the number of pixels of the specified color by scanning through every pixel of the image. In this example, we assume that a B+-tree index has been built on *SatelliteData.picture.numpixels(red).* Consider a query of the form:

```
SELECT  count(*)  FROM ColorValues C, SatelliteData S
WHERE S.picture.rotate().numpixels("red") = C.numpixels;
```

Such a query could easily be constructed as a result of views and query rewrite (in fact, it is an analog of the *sharpen().height()* example that we have used in our discussion). The image E-ADT specifies the following transformation: *X.rotate().numpixels(Y)* → *X.numpixels(Y)*. By applying this transformation, the query has a indexed nested-loops evaluation plan with *ColorValues* acting as the outer relation. We vary the cardinality of the *ColorValues* table from 1 through 4 (even with this tiny table, the execution times are significant). In the absence of E-ADT optimizations, the indexed access path is not recognized as a viable alternative. This means that *numpixels("red")* needs to be computed for every pair of tuples from the two relations. As the results show, the indexed plan in the presence of E-ADT optimizations is virtually free, while the unindexed plan without E-ADTs is very expensive. While the results themselves are certainly not surprising, the issue to stress is that the interaction between E-ADT optimization and relational optimization is so significant even in such a simple query



This example demonstrates a simple instance of a problem in index matching that is difficult in the general case. How can the system decide whether the index built on expression I matches the predicate *E > 3000*? The two expressions need to be proved equivalent (for now, we only consider exact matches). The information available for such a proof is the equational theory of the E-ADT, usually specified by transformation rules. The standard technique from term rewriting theory is to find a normal form that both *E* and *I* reduce to. However, a unique normal form for E-ADT method expressions does not always exist, and even if it does exist, there may not be a good algorithm to find it. We are currently developing theoretical results to express the complexity of matching two arbitrary E-ADT expressions, in terms of the properties of the transformational rules.

In PREDATOR, the E-ADT transformations are meant as production rules to drive the optimization of E-ADT method expressions, not as equivalence specifications. Clearly, if E-ADT optimizations reduce *E* to *I,* then there is a match. Similarly, if the optimization of *E* leads to the same execution plan as the optimization of *I*, then there is a match. These are important special cases where it is easy to determine if an index can be used.

5.2 Handling Multiple Predicates

Our next issue deals with the treatment of methods that act as predicates in relational queries. While most research has concentrated on the treatment of expensive predicates, it has been implicitly assumed that the treatment of "cheap" predicates is straightforward. However, this is not true. Typical relational query optimizers utilize semantic information about the specific predicates used in SQL queries. A common example is the property of "column equivalence". If the WHERE clause of a query contains the predicates *A.x = B.x and B.x = 5*, the optimizer automatically introduces the predicate *A.x=5*. This is critical in order to find efficient access paths to the table *A* and can dramatically alter the cost of the query. To the best of our knowledge, most commercial RDBMS engines implement this optimizer feature for equality predicates (it is necessary even for the simple queries found in the Wisconsin benchmark[16]). However, consider what happens if the predicates involve methods of an ADT. Consider a query involving geometric data (like polygons). The WHERE clause may hold *A.x.contains(B.x) and B.x.contains(C.x)*. The semantics of geometric containment allow us to deduce that *A.x.contains(C.x)* should also hold. However, this clearly is not something that can be hardcoded into the relational query optimizer. In order to preserve the modularity of the system, any information about this data type must be associated with its E-ADT. A mechanism is needed to allow the E-ADT to specify such "derived" predicates. Such a mechanism is readily available in PREDATOR. Most E-ADTs in PREDATOR express their semantics in terms of transformation rules. We use the same rule mechanism to modify predicates from the WHERE clause. For example, the geometric data type would define the rule *"X.contans(Y) and Y.contains(Z) +→ X.contains(Z)"*. This allows the relational optimizer to derive the new predicate.

Let us examine a similar example related to the familiar image E-ADT. Since every image has a bounding box, we can define a *boxequals()* method on images that checks if two bounding boxes are identical. Note that this is not an expensive method, since the image does not need to be decompressed in order to find its bounding box. Consider a query of the form :

```
 SELECT *  FROM OldSatelliteData S, NewSatelliteData S1
WHERE S.picture.boxequals(S1.picture) and S1.picture.height() < 5000
```

In this case, the image E-ADT can specify an extra predicate *S.picture.height() < 5000*. This may be used to match an index built on *OldSatelliteData.picture.height()*, and this can significantly improve the query execution. We do not present performance numbers to demonstrate this, since the effects are similar to the later experiments demonstrating index matching.

- Insight : In queries that involve a boolean combination of E-ADT predicates, the semantics of the data type should be used to define new predicates. These new predicates may help the relational optimizer to find good execution plans.

In fact, it is not always the case that new predicates will be created. An E-ADT can also specify semantic knowledge that can be used to reduce the number of predicates. A particularly common example is when two predicates are not mutually satisfiable. For example, *A.contains(B) and B.contains(A) → false*, (assuming that *A.contains(A)* is false).

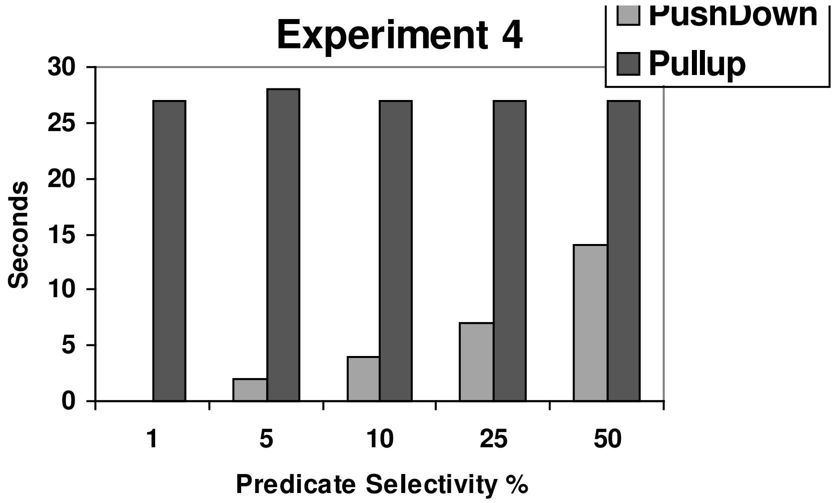
5.3 E-ADT Estimates for Relational Optimization

We now present three issues related to the effect of E-ADT method expressions on the cost-based optimization of select-project-join query blocks. Performance experiments present empirical evidence of their effect on query execution time.

Predicate Placement:

The E-ADT paradigm and its implementation in PREDATOR emphasize the **compile-time** optimization of E-ADT expressions. Why is it not sufficient to optimize each expression just before evaluating it? While there is some cost associated with this optimization, there are many cases where the cost may be low. On the other hand, since the actual parameters are available, exact information about them may be used in the optimization. As we describe below, a major motivation for this decision is the interaction between E-ADT expressions and relational queries. Since a relational query is optimized before the query is executed, performance considerations demand that E-ADTs also be optimized at compile-time (although this does not preclude further re-optimization at runtime).

The placement of predicates within relational query plans has been studied extensively. A heuristic that works well in practice is to apply "expensive" predicates after joins, and cheap



predicates before joins [7]. This work depends on knowledge of the cost of applying predicates. In PREDATOR, when a predicate involves an E-ADT method expression, its cost is not the additive cost of the individual methods involved: instead, it depends on how the E-ADT optimizes the expression. For example:

SELECT * FROM SatelliteData S, TimesOfInterest T
WHERE S.picture.sharpen().height() < %CONST% AND S.time = D.time

In the absence of E-ADT optimizations, the predicate *S.picture.sharpen().height() < %CONST%* is treated as an expensive predicate, since *sharpen()* is compute intensive. The Predicate PullUp algorithm (and, typically, the Predicate Migration algorithm) of [7] would use a query plan that first evaluated the join of S and D, and then applied the predicate. On the other hand, after E-ADT optimizations, the predicate is identified as cheap and can be applied early. How big an effect can this have?

Experiment 4:

The *TimesOfInterest* table has 1000 tuples and the selectivity of the join predicate is 0.05. Along the X-axis, we vary the selectivity of the "expensive" E-ADT predicate on the *SatelliteData* table. We observe the behavior of a nested-loops join plan with *SatelliteData* as the outer relation. We measure the execution time associated with two different evaluation algorithms: "Pushdown" uses E-ADT transformations at compile-time to recognize that *S.picture.sharpen().height()* reduces to *S.picture.height()*, which is not expensive. Consequently, the predicate is applied before the join. "PullUp" treats the predicate as expensive, but applies the E-ADT optimization at run-time. Any difference in costs between PullUp and PushDown is purely due to the choice of relatiional query plan. The results clearly show the unnecessary work performed in the join by the PullUp strategy.

- *Insight:* The placement of a predicate depends on its cost, which is altered by E-ADT optimizations. In general, an E-ADT can produce an arbitrary plan for a method expression. The plan has an associated cost, which can be significantly different from the cost of a naïve execution plan. It is essential that the estimated cost be available at compile-time. Consequently, E-ADT transformations should be applied **before** relational (cost-based) optimization of each query block, and E-ADTs should share the same cost model as the relational optimizer.

The execution plan for an E-ADT method expression can also indicate the expected usage of resources (importantly, main memory). Unlike simple predicates and expressions, E-ADT methods operate on very large complex data types that can consume significant memory. Some methods may write large results to disk through the database buffer pool, thereby displacing other contents of the buffer. This resource usage should be taken into account during join optimization. To the best of our knowledge, this issue has been ignored in other work related to the treatment of expensive methods.

The cost of join algorithms depends critically on the amount of buffer space available. For example, a nested-loops join algorithm exhibits sequential flooding when the size of the inner relation exceeds the size of the available buffer (with the LRU page replacement strategy). In the next experiment, we demonstrate that the presence of E-ADT methods being evaluated as part of the join (either as selections or projections) can significantly impacts the behavior of the join. We make this case using the simple function *scale(factor)* that reads in a compressed JPEG image and writes out a scaled version of it. The use of buffer space for this method interferes with the buffers being used for the join. Note that the buffer use in this case is very small (the average size of a JPEG image is 250KB). Objects of other large ADTs like audio and video are significantly larger and can cause more significant disruptions of memory usage patterns.

Experiment 5:

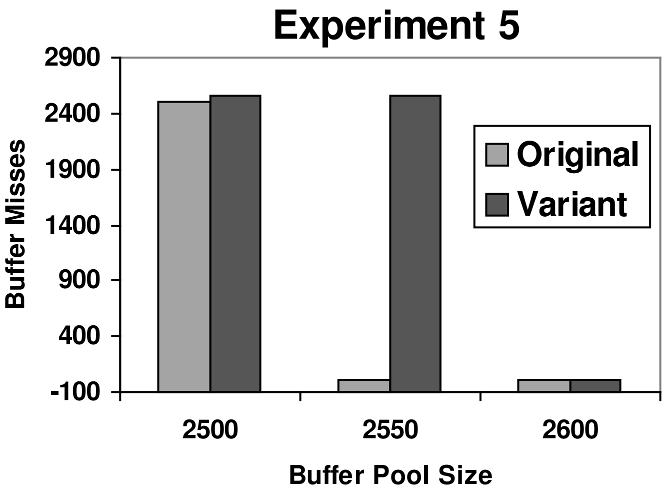
Our goal is to demonstrate that predictions of buffer usage are inaccurate when E-ADT methods are present. We use the following carefully tuned query:

```
Original Query: SELECT S.time, L.id  FROM SatelliteData S, LargeTable L
                WHERE S.id = L.id and D.id < 2
```

LargeTable contains 10,000 tuples, each of which is very wide, so that only four tuples fit on each database page. Consequently, the table is approximately 2500 pages large. We force a nested-loops join plan with *SatelliteData* as the outer relation. The condition *D.id < 2* ensures that there are exactly two outer-tuples; consequently, the inner relation is read exactly twice. We are interested in the number of buffer misses that occur on the second loop – if there are sufficient buffers, the pages of *LargeTable* should stay in the buffer. Such calculations are regularly made by query optimizers while costing various alternative join methods. We also consider a variant of the query shown below:

```
Variant:  SELECT S.time, L.id, S.picture.scale(0.5)  FROM SatelliteData S, LargeTable L
          WHERE S.id = L.id and D.id < 2
```

Along the X-axis, we vary the number of pages in the server buffer pool. Along the Y-axis, we measure the number of buffer misses when executing the second loop of the nested loops join query. As the figure shows, the original query exhibits sequential flooding with 2500 buffer pages, but sizes 2550 and 2600 result in no buffer misses (the entire inner relation fits in the buffer). In the variant, there are two invocations of the *scale()* method, each of which reads a JPEG image and writes out another JPEG image. Observe that the variant exhbits sequential flooding even with a buffer pool of size 2550. Clearly, the very minor use of buffer space has affected the performance of the join. Realistic queries will involve more than two invocations of such E-ADT methods, and the values may be much larger than these images.



- *Insight:* If E-ADTs reveal information about the memory usage of their methods, the relational query optimizer can use more accurate cost estimates, thereby resulting in significantly better plans.

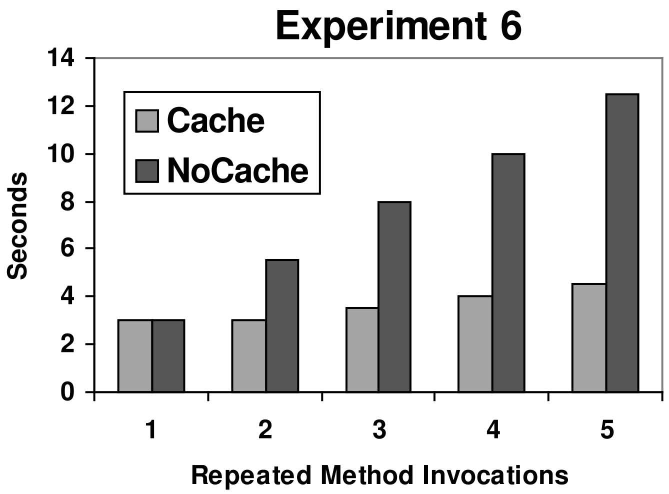
Argument Locality:

The results of method executions may be cached. Typically, these caches hold some small number of results, because each individual method result may be large. As we discussed earlier, method result caches are useful only when the method is repeatedly invoked with the same arguments. However, there is a more subtle form of caching that can play an important role in query execution performance. Consider the following query that matches each image in the SatelliteData table with a number of tuples from the AnalysisSpec table. Each AnalysisSpec tuple specifies a parameter which is passed to an analysis method of the image. For this example, we have chosen a simple analysis method that computes the number of pixels in the image that have the specified color.

```
SELECT S.picture.numpixels(A.color), A.id FROM SatelliteData S, AnalysisSpec A
WHERE S.id = A.id and  %selection_condition_on_A%
```

This query is expensive because the method *numpixels()* is invoked several times. The exact number of invocations depends on the selectivity of the WHERE clause predicates. There are 20 tuples in *AnalysisSpec* that altogether reference 4 distinct images (each image needs to be analysed on 5 different colors). The *numpixels()* method is never applied twice to the same image with the same color (because <*id, color*> is a key of *AnalysisSpec*). Consequently, it does not help to cache the results of this method. However, the image E-ADT can use its scratch space smartly to optimize query performance. Instead of caching the result of the method, it is useful to cache intermediate stages of the method computation. In this example, a large part of the cost of *numpixels()* is associated with decompressing the JPEG image. There are several invocations of the method on the same image (but with different colors). If all these invocations occur with some temporal locality, the decompressed image can remain in the scratch space cache. We demonstrate the effect of this in the following experiment.

Experiment 6:



Both relations are small enough to fit in memory, so our focus is on CPU costs. The image E-ADT maintains a srcatch-space cache that can hold one decompressed image. Along the X-axis, we vary the number of repeated calls to *numpixels()* for the same image (but with different color parameters). As the number of repeated calls increases, we expect caching to have an impact. The results do reflect the expected behavior.

- *Insight :* The locality of method arguments can play an important role in the performance of queries. Traditional query optimizers have no opportunity to exploit this locality. However, knowledge of this locality should be used in cost estimates and also in determining interesting orders during query optimization.

On one hand, the locality created by the choice of relational query plan affects the cost of execution of the E-ADT expression. Conversely, the cost of execution of the E-ADT expression needs to be used to optimize the relational query. It appears that we have a paradox with respect to this issue. Luckily, there is a simple mechanism that provides the desired functionality. In the E-ADT paradigm as presented in [2], each method expression is optimized by its own E-ADT to produce the lowest-cost execution plan. Instead, we generalize this idea to allow a set of lowest-cost plans to be generated, where each plan corresponds to a specific temporal locality in the values of the arguments to the method expression. In this example, the image E-ADT would produce two plans, one for unordered parameters, and one for the case when duplicate image parameters are clustered together. This allows the relational optimizer to correctly cost the different join plans.

7. Conclusions

To summarize our contributions, we have explored several query optimization issues that arise from the interaction between E-ADTs and relational queries in which E-ADT expressions are embedded. We have categorized the possible optimizer interactions across the relation/E-ADT boundary and indentiifed four broad categoriies of insights into system design, each motivated by an example and by performance results. Our conclusion is that there are non-trivial interactions between complex data types and relational queries that manipulate them.

When the data types are "enhanced" with optimization capabilities, the effects on relational optimization are even more significant. By extending the E-ADT paradigm used in PREDATOR, we argue that each E-ADT can exchange information with the relational query optimizer without compromising the modularity of the type system. We demonstrate that in the absence of such information, the relational query optimizer may generate very inefficient plans. Indeed, a naïve OR-DBMS with blackbox ADTs can perform significantly worse than a system like PREDATOR, even on relatively simple queries that use methods of complex data types. The techniques studied in the paper are summarized by the figure below: We expect that further techniques will arise that will fall into one of the four categories of optimizations that span the relation/E-ADT boundary.

SQL Opt. Helps E-ADT	Query Rewrite	E-ADT aggregates
E-ADT Helps SQL Opt.	Index matching, new predicates	Memory usage, predicate placement, interesting orders
	More opportunities	More information

A weakness of this work is that there is no reason to believe that this is a complete set of insights (i.e. we do not know if we have completely explored the interactions between E-ADTs and the relational optimizer). However, the categorization of optimization opportunities is a strong initial step that will help with the understanding further techniques as they are developed. Further, we have demonstrated optimizations that result in significant performance improvements. By implementing these optimizations in PREDATOR, we have shown them to be practical (and in fact, relatively easy to implement in a system based on the E-ADT paradigm). The insights that lead to these optimizations should play an important role in the design of the next generation of database type systems and query processing engines.

References:

1. *Shoring up Persistent Objects*: M.J.Carey, et al., Proceedings of the ACM SIGMOD Conference on Management of Data, 1994.

2. *The Case for Enhanced Abstract Data Types*: P.Seshadri, M.Livny. and R.Ramakrishnan, Proceedings of the Twenty Third International Conference on Very Large Databases, 1997.

3. *Access Path Selection in a Relational Database Management System*: P.G.Selinger, M.Astrahan, D.Chamberlin, R.Lorie, and T.Price, Proceedings of the ACM SIGMOD Conference on Management of Data, 1979.

4. *The Sequoia 2000 Storage Benchmark*: M. Stonebraker, J.Frew, K.Gardels, and J.Meredith, Proceedings of the ACM SIGMOD Conference on management of Data, 1993.

5. *Enhanced Data Types in the PREDATOR Database System:* P.Seshadri, Submitted to the *VLDB Journal, 1998.*

6. *Optimization of Queries with User-Defined Predicates:* S.Chaudhuri and K.Shim, Proceedings of the Twenty Second International Conference on Very Large Databases, 1996

7. *Optimization and Execution Techniques for Queries with Expensive* Methods: J.M.Hellerstein*,* PhD Thesis, University of Wisconsin, 1995.

8. *Query Execution Techniques for Caching Expensive Methods:* J.M.Hellerstein and J.F.Naughton, Proceedings of the ACM SIGMOD Conference on the Management of Data, 1996

9. *Optimization of Non-Recursive Queries:* R.Krishnamurthy, H.Boral, and C.Zaniolo, Proceedings of the Twelfth International Conference on very Large Databases, 1986.

10. *The Implementation of Postgres:* M.Stonebraker, L.Rowe, and M.Hirohama, IEEE Transactions on Knowledge and Data Engineering, 2(1):125—142, March 1990.

11. *Generalized Search Trees for Database Systems:* J.M.Hellerstein, J.F.Naughton, and A.Pfeffer, Proceedings of the Twenty First International Conference on Very Large Databases, 1995.

12. *Semantic Query Optimization for Methods in Object-Oriented Database Systems:* K.Aberer and G.Fischer, proceedings of the Eleventh IEEE Conference on Data Engineering, 1995.

13. *Query Optimization in the Presence of Foreign Functions:* S.Chuadhuri and K.Shim, Proceedings of the Nineteenth International Conference on Very Large Databases, 1993.

14. *Extensible/Rule-Based Query Rewrite Optimization in Starburst:* H.Pirahesh, J.M.Hellerstein, and W.Hasan, Proceedings of the ACM SIGMOD Conference on Management of Data, 1992.

15. *Client-Server Paradise:* D.J.DeWitt, et.al., Proceedings of the Twentieth International Conference on Very Large Databases, 1994.

16. *Benchmarking Database Systems: A Systematic Approach:* D.Bitton, et.al., Proceedings of the Ninth International Conference on Very Large Databases, 1983.