The VLDB Journal manuscript No.
(will be inserted by the editor)

Enhanced Abstract Data Types in Object-Relational Databases

Praveen Seshadri

Computer Science Department, Cornell University, Ithaca, NY 14853, USA. e-mail: praveen@cs.cornell.edu

Received April 15, 1998 / Revised June 15, 1998

Abstract. The explosion in complex multi-media content makes it crucial for database systems to support such data efficiently. This paper argues that the "blackbox" ADTs used in current object-relational systems inhibit their performance, thereby limiting their use in emerging applications. Instead, the next generation of object-relational database systems should be based on Enhanced Abstract Data Type (E-ADT) technology. An E-ADT can expose the *semantics* of its methods to the database system, thereby permitting advanced query optimizations. Fundamental architectural changes are required to build a database system with E-ADTs; the added functionality should not compromise the modularity of data types and the extensibility of the type system. The implementation issues have been explored through the development of E-ADTs in *Predator*. Initial performance results demonstrate an order of magnitude in performance improvements.

Key words: Object-relational database – Extensibility – Database types – Query optimization

1 Introduction

We are witnessing an explosion in the volume and complexity of digital information that people wish to access and analyze. If a DBMS is to appeal to application developers, it should support complex data types like geographic entities, chemical and biological structures, financial time-series and multi-media objects. Further, the level of functionality and performance should be comparable to special-purpose systems. Many relational database vendors are currently building "object-relational" extensions to support complex data. However, thus far, object-relational databases (OR-DBMSs) have played no more than a peripheral role in managing new complex content. Why is this?

Current object-relational databases (OR-DBMSs) model complex content as "blackbox" abstract data types (ADTs), with procedural methods that can be used within SQL queries. The primary advantage of this approach is modular extensibility. However, while the DBMS optimizes relational operations like joins, there is hardly any optimization for ADT methods. Clearly, the very best relational query optimizer cannot help a query much if most of the execution cost occurs in expressions involving ADT methods.

We argue that databases should support Enhanced Abstract Data Types (E-ADTs) which expose some part of the semantics of their methods to the DBMS. This allows the system to choose any specific implementation of each method, and importantly, *to optimize expressions involving a combination of method invocations*. For example, *Sharpen(Clip(Image, Region))* is likely to be more efficient than *Clip(Sharpen(Image), Region)*. Essentially, E-ADT methods in an SQL query are treated declaratively, rather than procedurally. Section 2 motivates the E-ADT concept and explains the possible benefits.

While E-ADTs are a simple abstract design paradigm, it is a non-trivial task to build a system that supports them. Several questions arise: What is the right architecture? What are the right components and mechanisms to use? What are the appropriate interfaces between components? We have addressed these questions through implementation in *Predator*, an object-relational database system that we have built at Cornell. Several E-ADTs have been developed for complex types like images, audio, video, rasters, polygons, etc. Our implementation (Section 3) demonstrates that E-ADTs can be implemented in full-fledged database systems, and our experiments (Section 4) suggest the possibility of dramatic performance improvements.

This paper makes three contributions: (1) It argues that the next generation of object-relational databases should be based on E-ADTs , (2) It describes the type system implementation in *Predator*, as an instantiation of the E-ADT design paradigm. (3) It defines new research opportunities that arise as a result of the use of E-ADTs.

Correspondence to: Praveen Seshadri

1.1 Background

Most OR-DBMSs support type extensibility; new data types can be added to the system without changes to the existing code. The basic technology used is that of Abstract Data Types (ADTs), which was adapted from programming language concepts [Gut77,LZ74] to databases in the 1980s [SRG83,Sto86] in the Postgres [SRH90] system. Object-oriented database systems and persistent programming languages also developed similar ideas, but we will focus here on object-relational database systems. The OR-DBMS maintains a table of ADTs, and new ADTs may be added by a database developer. Each ADT implements a common internal interface through which the system can access and manipulate values of that type. The internal interface includes functions for the storage and indexed retrieval of values. Each ADT can also declare primitive methods for manipulating or querying values of the type. For example, an ADT for images might provide methods *Sharpen(I), Clip(I, Region)*, and *Overlay(I1, I2)*. Complex data types like images usually define a large number of primitive methods, that can be composed to form meaningful expressions. For example, *Overlay(Sharpen(I1), Clip(I2, {0, 0, 100, 200}))* is an expression over images *I1* and *I2*. Such expressions can be embedded within an SQL query, thereby providing an expressive query capability over image data. Libraries of primitive methods for each ADT are sometimes called "datablades" (Informix), "data extenders" (IBM), or "data cartridges" (Oracle).

There are two characteristics of the current support for ADTs that we should note.

- *(1)* Each ADT, along with its methods, is built modularly, so that it can be added to or removed without affecting the rest of the system. The modularity and extensibility of the type system are essential features of an OR-DBMS.
- *(2)* Each ADT is a "blackbox" that reveals little more than the names of the methods supported. Each method is merely a name and type signature for a function, often written in C or C++[1]. Existing systems also allow some simple semantics about methods to be defined: does it have side effects or not?, what is the cost of the method?, etc.

This is the current state-of-the-art, which we refer to in this paper as the "blackbox" ADT approach. The E-ADT paradigm presented in this paper relaxes the second characteristic (blackbox ADTs) while retaining the first characteristic (modular ADTs).

Several commercial database systems have added support for blackbox ADTs. In practice, while many simple ADTs are added by database users, the important complex ADTs (like images) and their method libraries are usually written by experienced system developers. As the standardization of methods on these complex ADTs occurs[2], one can expect more efforts to be directed at building efficient implementations of them.

2 Motivation

E-ADTs are motivated by a simple observation: methods on complex ADTs can be expensive (for example, *Sharpen(Image)*). In fact, the cost of ADT methods often dominates the overall execution cost of a query. Query processing and optimization (which in the past has focused almost exclusively on relational operations) should now attempt to reduce the cost of ADT methods. There are two issues to consider:

- What are possible optimizations on ADT methods? What does one hope to achieve?
- How can the optimizations be applied systematically? Can this be achieved in a practical database system?

This section discusses the first issue, and develops a framework of design constraints for a reasonable implementation. Together, this constitutes the E-ADT paradigm. The actual implementation in *Predator* is discussed in the following section.

2.1 A Motivating Example

Consider an OR-DBMS based on blackbox ADTs, and assume that an image ADT has been added with the methods *Sharpen(I)* and *Clip(I, Region)*. While image data is stored on disk in a compressed format (like JPEG), the ADT methods are implemented on an uncompressed main-memory image data structure (like an RGB array). Consequently, the image argument of any method is converted to its main-memory uncompressed form before a method is invoked on it.

An earth scientist maintains a table of geographic data, each entry having a satellite image and several other columns. An SQL query posed against this database may include an expression over the images:

```
SELECT Clip(Sharpen(G.Photo), 0, 0, 100, 200)
FROM GeoData G
WHERE G.Region = 'arctic'
```

The query asks for a sharpened portion of each image of the arctic region. The cost of this query is dominated by the methods on the images. We now describe the evaluation of this query using the blackbox ADT approach. For every data entry corresponding to the arctic region, the Photo attribute is retrieved from disk and decompressed into a main-memory image. The *Sharpen* method is then applied to it, and the resulting image is written to a compressed disk-resident form (although it may not actually be flushed to disk). The *Clip* method is then applied with the intermediate result image as its input. This input image is decompressed to a main-memory form, it is clipped to the desired dimensions, and the resulting image is compressed and written out to disk. Current OR-DBMSs like Illustra [Ill94] and Paradise [DKL+94] use essentially this approach with some individual modifications; we mention these variations at length in Section 5.

[1] SQL-3 will allow functions to be written using SQL too

[2] The SQL3 Multi-Media standards group is currently defining methods for full-text, spatial data and general mathematical data. Future data types to be standardized include still-images, still-graphics, animation, full-motion video, audio, seismic data, and music.

One could improve this execution strategy as follows:

- It is unnecessary for *Sharpen* to compress and write its result to disk. Instead, it could be passed uncompressed directly in memory to *Clip*. This requires that the methods not be evaluated in isolation; instead, the system should recognize that these methods are part of a larger image query expression. This is a change to the blackbox ADT approach, which treats methods as isolated black-boxes[3].
- *Sharpen* is an expensive method, whose cost depends on the size of the image. It would be cheaper to evaluate the equivalent expression *Sharpen(Clip(G.Photo, {0, 0, 200, 100}))*. By performing *Clip* early, there can be significant reductions in the cost of *Sharpen*. This requires that the entire image query subexpression be "optimized". The optimizations could be applied heuristically in this case, but in general, a cost-based manner approach is preferable. To the best of our knowledge, no existing OR-DBMS can apply such optimizations in a heuristic manner, let alone in a cost-based manner.
  Note that this expression may not be strictly equivalent to the original expression at the boundary pixels. Similarly, many "equivalence" transformations may lose accuracy over floating point data. This is a legitimate concern for certain data types and applications, and may not be very relevant to the user in other cases, including this query. In the rest of the paper, we assume that only "valid" transformations are being applied, as dictated by the data type and application developer.
- If *Clip* is applied before *Sharpen*, the entire image does not need to be retrieved from disk. Only the appropriate portion of the image is needed[4]. Performing such optimizations requires a further enhancement to the system: the ADT methods should control the retrieval of the underlying ADT values.

An obvious question would be: why not require that the user write the query in an efficient manner? The answer is two-fold: (1) the choice of the most efficient expression may depend on costs and statistics that users may not be aware of, (2) queries are often created automatically, through GUIs, involving the merging of several views. It is, therefore, quite likely that inefficient E-ADT method expressions will exist in queries. In this example, it is possible that one user defined a view that created "sharpened" images, and another user generated a Clip() query over this view by simply drawing a rectangle on a map.

[3] Section 5 describes how Paradise uses caching to support this without visibly changing the ADT approach.

[4] When compression is used in storing the images, it is not always possible to retrieve a random portion of an image. For example, with the popular JPEG compression technique, the entire image needs to be retrieved sequentially. There are other compression techniques and variants of JPEG that allow a selective portion of the image to be retrieved and decompressed. [DKL+94] discusses these issues in more detail.

2.2 Optimization Categories

To summarize our motivation, the blackbox ADT approach used in existing OR-DBMSs violates a basic principle of database systems: *queries should be declarative.* The textual representation of an expression should not specify an evaluation plan. When the ADT expression is treated declaratively, the combination of different simple optimization strategies can lead to performance improvements of an order of magnitude. Similar improvements can be applied to query expressions involving other complex data types. Several commonly used data types have libraries of methods that form an algebra which is amenable to optimizations. Prominent among these are the multimedia types like images, audio and video, and the geographic types like points, polygons and rasters. Optimization opportunities also exist for mathematical data types like matrices, financial types like time-series, and chemical structures like molecules. The specific optimizations depend on the operators and domain-specific semantics of each data type (in this paper, we do not explore them in detail). However, we can identify four broad categories of optimizations:

- *Algorithmic:* Using different algorithms for each method depending on the data characteristics. For example, the best algorithm to use for the *Multiply* method on two matrices depends on the sizes of the matrices and the amount of memory available.
- *Transformational:* Changing the order of methods or even replacing one or more methods by other methods. The motivating example shows how *Clip* can be applied before *Sharpen*. In fact, the entire equational theory of the data type can be augmented with cost information to specify transformational optimizations.
- *Constraint:* Pushing physical constraints through the expression. The constraints may involve selecting a portion of the data, specifying a certain data resolution, or requiring a particular physical property. Consider the expression on images: *ChangeResolution(Sharpen(I), Res)*. The knowledge of the desired resolution can be used to ensure that *Sharpen* is applied to a low resolution image.
- *Pipelining:* There are two categories of pipelining techniques, both of which recognize that a sequence of operations/methods need to be applied to a large object and optimize the sequence. The simple form of pipelining used in the example keeps intermediate results in memory (without compressing them or writing them to disk). For large data types like audio and video, it may not be possible to fit an entire uncompressed object in memory. In this case, the only reasonable way to access the data is to iterate over it in smaller blocks (i.e. to treat it as a stream and to treat the operators as stream transducers). When a sequence of methods has to be applied to such a large object (for example, *IncrTreble(DecrVolume(Audio))*), stream pipelining the methods is clearly better than generating entire intermediate results for each method.

As an analogy, consider how a "Relation" ADT would be supported in a blackbox fashion. The relational algebra primitives would be the ADT methods. Queries on relations would be formed syntactically as relational algebra expressions and executed in the order specified, without any query optimization! This is clearly unacceptable – relational queries should be declarative. The choice of join algorithms is an Algorithmic optimization. The choice of join order, and selection pushdown are Transformational optimizations. The use of "interesting order" in join optimization is a Constraint optimization. Pipelined join execution is a Pipelining optimization. We know that these query optimizations can greatly improve performance. Likewise, in next-generation applications, ADT expressions that dominate query execution cost should be treated declaratively and optimized.

2.3 A Framework to Apply the Optimizations

It is not sufficient to observe that these optimizations are possible. There must also be an architectural framework to specify and apply them in a correct and cost-based manner. At the same time, the type system should remain extensible, so that new types can be added modularly. We now develop design requirements by examining the motivating example in more detail:

- The best algorithms used for each method can depend on the size of the input image, the amount of memory available, the storage format of the image, and the values of arguments to the method.
- For any complex expression involving multiple image methods, there are several possible evaluation plans.
- Deciding between different plans requires cost-based optimization of the image expression.
- The image expression may be evaluated several times during the course of the query. Its evaluation plan should be chosen before the query starts executing, because it may be unreasonable to repeatedly explore the options at runtime. Additionally, of course, run-time modifications of the plan are possible.
- The cost of the chosen plan is used by the SQL optimizer to "place" the evaluation of the expression at the appropriate position in the join evaluation tree [Hel95,CS96].
- In order to perform compile-time optimization, collective meta-information like the storage formats and size statistics need to be maintained over all the pertinent images (in this case, over all images in the GeoData table).
- The meta-information maintained and the optimizations to be applied are specific to each individual ADT.

2.4 The E-ADT Design Paradigm

An Enhanced Abstract Data Type (E-ADT) enhances the concept of blackbox ADTs in database systems to improve the performance of query processing. The E-ADT paradigm has the following components:

- The methods of every E-ADT should be declarative (rather than procedural) expressions.
- The combinations of E-ADT methods should be declarative expressions.
- The optimization and execution of these declarative expressions should involve the semantics of the E-ADT.
- The modularity and extensibility of the database type system should not be compromised.

The last criterion (modularity) leads to a design requirement that the new capabilities or enhancements for complex data types be encapsulated behind standard interfaces.

Specifically, in addition to providing the standard ADT functionality, each E-ADT may support one or more of the following enhancements through a uniform internal interface.

- *Query Operators and Optimization:* An E-ADT can provide an optimization interface that will translate a method expression into a query evaluation plan in its own evaluation algebra.
- *Query Evaluation:* An E-ADT can provide routines to execute the optimized plan.
- *Catalog Management:* An E-ADT can provide catalog routines so that schema information can be stored and statistics maintained on values of that E-ADT .
- *Storage Management:* An E-ADT can provide multiple physical implementations of values of its type. Some existing OR-DBMS systems already support this feature.

3 E-ADTs in *Predator*

Predator implements an architectural framework in which E-ADT optimizations may be applied. While the low-level implementation details are described in [Ses97], this paper describes the high-level system design. Current OR-DBMSs do not perform such optimizations; further, they lack the framework to do so.

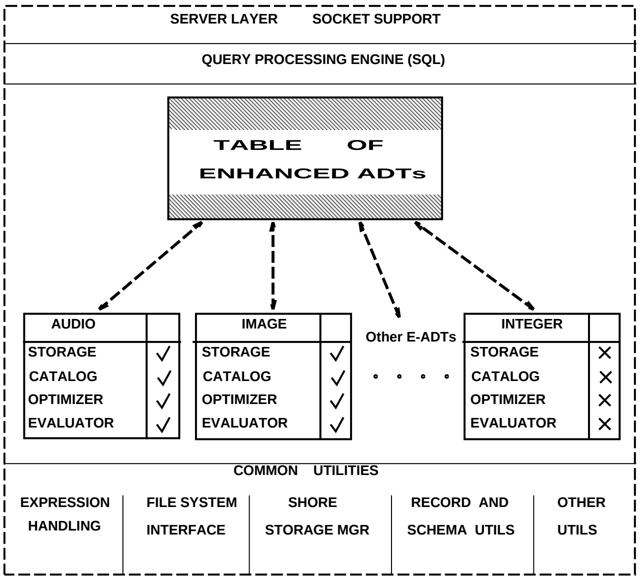

Fig. 1. *Predator* Type System Architecture

Predator is a client-server OR-DBMS in which the server maintains a loosely-coupled collection of E-ADTs.

A simplified high-level picture of the server is shown in Figure 1. The server is built on top of a layer of common database utilities that all E-ADTs can use. An important component of the utility layer is the SHORE Storage Manager [CDF+94] library, which provides facilities for persistent storage, concurrency control, recovery and buffer management. The core of the system is a main-memory table in which E-ADTs are registered. An SQL engine uses the E-ADTs in processing relational queries (multiple query processing engines are implemented, but we will focus on the SQL engine).

Some of the basic types like integers do not support any enhancements. The figure shows two E-ADTs that do support enhancements: audio and image. A complex value like an image can be a field within a relational tuple. The type "relation" is also modeled as an E-ADT, thereby allowing nested relations to be supported. Every data type can specify "meta-information" that is used to further define values of that type. All values of a particular attribute of a relation share the same meta-information. In the motivating example, all the satellite images share the same meta-information. There are two components of the meta-information, and they serve distinct purposes:

- *Type constraints*: The meta-information may specify constraints on the actual values of the data type that may occupy this attribute. For instance, an image attribute may be constrained to accept JPEG images only. An attribute of a time-series data type may be constrained to span a specific temporal range. This portion of the meta-information is analogous to schema information; a relation E-ADT would specify the tuple schema using its meta-information. In fact, the type specification of an attribute requires both the type identifier and the meta-information that constrains the type.
- *Statistics*: The optimization of an E-ADT method may depend on properties of its arguments. For example, the choice of algorithm for an image method may depend on the size of the image. Statistics are maintained for all the E-ADT values that share the same meta-information. As mentioned in Section 2, the E-ADT optimizations need to be applied before the query is executed; consequently, collective statistics are needed. When statistics are collected for the entire relation, each attribute's E-ADT is given the opportunity to maintain its own specialized statistics for its values. These statistics could be as simple as average object size, or arbitrarily complex.

An important feature of the system design is that E-ADTs are modular, with all E-ADTs presenting an identical internal interface. This serves a dual purpose:

- The development of optimizations for each E-ADT can proceed independently. The extensibility of the system would be compromised if each E-ADT had to be aware of all the other E-ADTs in the system.
- The interaction between the SQL query processing engine and the individual E-ADT expressions is critical for good performance. Because of the uniform internal interface of E-ADTs, the design of this interaction is independent of the details of each E-ADT.

3.1 Interaction between Data Types in Predator

Let us revisit the example query:

SELECT Clip(Sharpen(G.Photo), 0, 0, 100, 200)
FROM GeoData G
WHERE G.Region = 'arctic'

This query has an image expression embedded in the SELECT clause. How is this image expression handled? We present a simplified discussion. When the SQL engine parses the query, it passes the image expression to the image E-ADT , which performs type checking and returns a parse structure (ParseStruct). The ParseStructure specifies the type (and meta-information) of the result, but is otherwise opaque to the rest of the system. An arbitrarily complex parse structure could be used by the data type.

Query Optimization: The optimization of an SQL query uses cost-based techniques to search for a cheap evaluation plan from a large space of options. Typically, all sub-expressions need to be optimized before the SQL query can be optimized. To optimize the image subexpression, the following interface of the image E-ADT is invoked:

Optimize(in: ParseStruct, in: ArgPlans, out: PlanStruct);

Optimize() takes the ParseStruct generated by parsing the image expression, and evaluation plans for its arguments, and returns a generic PlanStruct that represents an evaluation plan for that expression. The meta-information associated with each of the inputs provides statistics that can be used in the optimization. The PlanStruct will only be interpreted by the image E-ADTand so can be arbitrarily complex. It could actually represent multiple plans, one of which will be chosen at run-time; however, the PlanStruct has a well-known Cost() interface which provides a single "average" cost estimate. The SQL optimizer uses this cost estimate to help determine the best plan for evaluating the SQL query. If the image expression is very expensive, it may be preferable to apply it as few times as possible within the SQL query plan.

Query Evaluation: The SQL query is evaluated based on its execution plan. While executing the SELECT clause, the image expression is evaluated through the following interface of the image E-ADT :

Evaluate(in: PlanStruct, in: ArgValues, out: ReturnValue);

Evaluate() is passed the plan for the image expression and the value of its argument (G.Photo). The image E-ADT executes the optimized image expression, and the return value is used to continue with the computation of the SQL query.

To summarize, the query is broken into components that correspond to method expressions of each E-ADT. Each method expression is treated as a declarative query and is optimized, and executed by its own E-ADT .

3.2 Implementation Specifics

We now describe some of the specifics of the *Predator* implementation of E-ADTs. We have been motivated by three requirements: (a) Simple data types should be just as efficient as in a system without extensible types, (b) New data types should be relatively simple to build, (c) If desired, new data types should have the flexibility to have arbitrarily complex enhancements (optimizations and evaluation algorithms). The interfaces described above are actually a subset of the entire set of interfaces that each E-ADT could provide. However, defaults are provided for most interfaces, so that a new data type need only define those interfaces that its developer chooses to extend. To avoid the overhead associated with abstract interfaces (virtual member function invocation in C++), simple data type like integers and floats are classified as "core" types and hard-coded through switch statements. The E-ADT design paradigm affects the lowest layers of the query processing engine. Every single expression needs to be "optimized" since some sub-component of it could involve an E-ADT expression. Further, the type and meta-information of the result of every expression need to be generated.

Specifying E-ADT Optimizations The ability of each data type to optimize its own method expressions is central to the E-ADT paradigm. As described in the previous section, each E-ADT can define code for an *Optimize()* method that performs this task. However, for most E-ADTs like images and audio, there are important optimizations (like moving a Clip() method ahead of other methods) that are very simple. Sometimes, these can even be applied on a heuristic basis without examining costs. Instead of requiring the developer to always write code to manipulate the expression trees, we envisage that the database engine would provide optimizer toolkits that would help with these tedious implementation issues. The data type developer would then be able to specify simple optimizations at a high level.

Predator provides a rule engine that interprets high-level transformation rules. Each rule takes the form

PreCondition (ExprTree) -> Transformation (ExprTree).

The *PreCondition* and *Transformation* can take the form of procedural code, or they can be specified by simple patterns. For example,

Clip(Sharpen(I), Box) -> Sharpen(Clip(I, Box))

or

Scale(Sharpen(I), Factor) & Factor < 1.0
-> Sharpen(Scale(I, Factor))

Essentially, this is a simple optimizer toolkit; more complex optimizer generators developed by others in recent research [GM93,**?**] could also be provided to the E-ADT developer. One requirement of the optimization mechanism is that all components of the system, including all E-ADTs, need to share a common cost model. The cost model used in the implementation involves a weighted combination of CPU and I/O costs, as in traditional relational optimizers.

Creating a New E-ADT An important concern is that the new enhancements might require a deep understanding of the workings of the DBMS, the internal interfaces, the cost metrics, etc. We have pushed much of this complexity into the Utility layer of *Predator*, and present suitable templates for data type developers. A developer takes the following steps to build a new data type like Image in *Predator*:

- Start with a template for a new E-ADT. The template contains a C++ header file with the appropriate class definition and member function declarations. There is also a C++ source file with stubs for all the member functions.
- Define Read() and Write() interfaces. This determines how the data wil be stored in the database, and involves very little actual new code. With images, the entire image would be loaded into the storage manager, and a handle to it (its OID) would be stored within a record.
- Specify the structure of meta-information, if any, that this E-ADT requires.
- Write code for each of the actual methods that will be supported on Images. This is usually the most time-consuming component, but existing libraries can often be used (for example, free JPEG libraries exist).
- Ensure that the method code is "registered" with the E-ADT ; this requires one line of code for each method.
- Specify individual transformational rules for optimization. Each rule has a precondition and an action.
- Register the rules with a rule engine. The utility layer pre-defines a simple rule engine class, so all that the E-ADT developer must do is register each rule (1 line of code each), and choose a search strategy (1 line of code).
- Add the new E-ADT to the Makefiles and register it in the system-wide table of E-ADTs. The system must then be recompiled. We could extend *Predator* to allow dynamic loading of the E-ADTs instead.

Of course, there are some other details involved, but they are neither difficult or time-consuming. One of our ongoing efforts is to build an E-ADT toolkit or "wizard" that will make it even simpler to develop new data types. The goal is that most of the specification must happen through a point-and-click interface. Currently, a data type like Image uses a standard template and requires a few hundred lines of code (in addition to the purely image-related algorithm code). All of the E-ADT enhancements are purely optional. The effort required to add a traditional ADT (with no E-ADT enhancements) to *Predator* is just the same as the effort required to add an ADT to a standard OR-DBMS.

4 Initial Performance Results

The purpose of this section is to demonstrate the orders of magnitude of performance improvements that arise from the E-ADT paradigm. We emphasize that the results are not unexpected. It is nonetheless interesting to observe the dramatic effects of simple optimizations. We use an image E-ADT as the primary example to demonstrate performance improvements. We also present one experiment demonstrating E-ADT support for audio data.

We use a simple data set containing 74 images of cars, compressed using JPEG. A *Cars* relation is created, with a name assigned to each image. Each image column value contains an *Id* for the image and its bounding box information. The size of the compressed images ranges from 23K to 266K, with an average of 65K. This is a relatively small amount of data, but it serves to demonstrate the issues involved. The average size of the uncompressed images in memory is 0.8MB representing more than a 10-fold increase from the size on disk. Standard JPEG libraries are used to perform compression and decompression. Experiments were run on a Sparc20 machine with 8 MB used as a database buffer pool (a small size to match the small data set). The buffer space was sufficient to hold the compressed data, and each individual uncompressed image easily fit in physical memory. The system is CPU-bound in all these experiments, since the image methods are expensive. The queries were chosen to be the simplest possible demonstrations of different optimizations. All queries were run several times and average execution times were recorded.

Experiment 1: The first experiment examines the effects of the Pipelining optimizations. We noted in Section 2 that in the ADT approach, each method reads (and decompresses) its input from a disk-based representation, and writes (and compresses) its output. We call this strategy DISK. An improvement is to pass intermediate results in their main-memory form. We call this strategy MEM. Finally, if there is a sequence of methods, we could pipeline their execution by establishing an image row iterator. We call this strategy PIPE. We expect PIPE to be significantly better than MEM only when the intermediate results are larger than main-memory. This is not the case for these images.

SELECT Height(Negative(C.picture))
FROM Cars C;

In the query above, *Negative()*, which inverts the pixel values, is a relatively cheap method. For all three strategies, *C.picture* must be read and decompressed. The DISK strategy compresses and writes the result out, whereas the MEM strategy does not. The *Height* method is applied so that the display time for the result does not distort the measurements. The height is obtained from the image bounding box, so the actual image is not required. We disable any optimizations that move *Height* ahead of other methods. We gradually vary the query by introducing additional invocations of *Negative()*. For instance, *SELECT Height(Negative(Negative(C.picture)))* requires one additional compression and decompression using the DISK strategy. The results are shown in Figure 2. Along the X-axis is the number of *Negative* methods in the SELECT clause. The Y-axis shows the execution time.

As the number of methods in the image expression increases, the effect of the compression and decompression at the image boundaries dominates. Consequently, the MEM strategy which avoids these unnecessary costs is significantly cheaper than DISK. PIPE performs only marginally better than MEM, since the intermediate results fit easily in memory. For the rest of the image experiments, the MEM strategy has been used as the default.

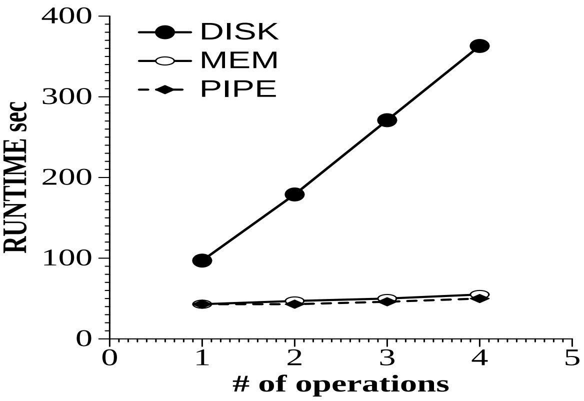

Fig. 2. Effect of Pipelining

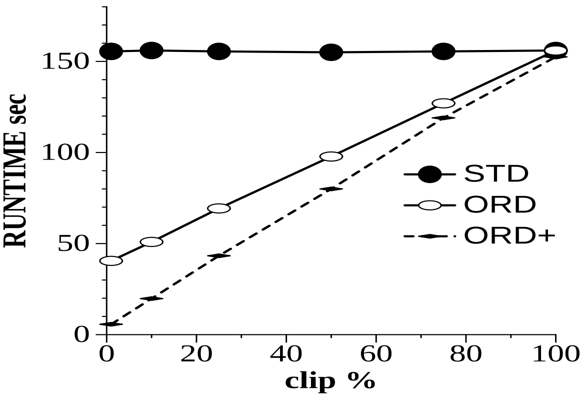

Fig. 3. Effect of Varying Clip Size

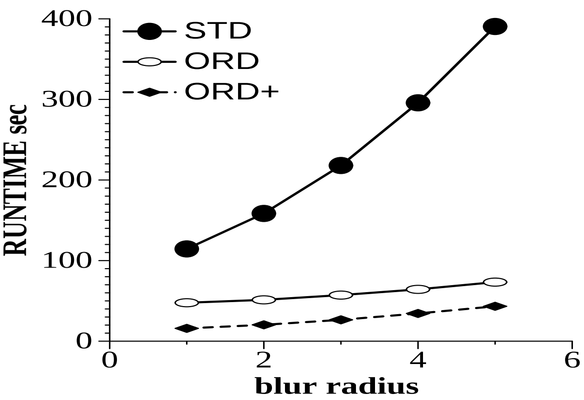

Fig. 4. Effect of Varying Blur Radius

Experiment 2: The second experiment examines the effects of Transformational and Constraint optimizations. The following query is used:

SELECT Height(Clip(Blur(C.picture, <radius>),
<region>))
FROM Cars C;

The result of a *Blur* method is an image in which every pixel's value is the average of the pixels within *radius* of it in the input image. *Blur* is an expensive method, especially when *radius* is large. In the standard ADT approach, the images will first be blurred, then clipped to the appropriate region. We call this the STD strategy. Using the transformational optimization of reordering, *Clip* could be performed before *Blur*. We call this the ORD strategy. Finally, if *Clip* is being performed first, it can use the region to constrain the retrieval and decompression of the images. We call this the ORD+ strategy.

Figure 3 shows how these strategies perform when the *Blur radius* is fixed at 2 and the size of the Clip region is varied as a percentage of the image size. Since blurring an image is expensive, both ORD and ORD+ perform very much better than STD when a small region needs to be clipped. Further, it is evident that ORD+ is also significantly better than ORD because it requires a smaller portion of each image to be retrieved an decompressed. As the size of the *Clip region* increases, the absolute difference between ORD and ORD+ decreases because the savings due to the Constraint optimization are reduced. The relative difference also decreases because *Blur* starts to dominate the cost.

Figure 4 fixes the *Clip region* at 10% of the image size, and varies the *Blur radius* from 1 to 5. As the *radius* increases, the cost of the blurring grows quadratically. For the STD strategy, since this is the dominant cost, the overall execution time also grows quadratically. In comparison, in ORD and ORD+, the clips are performed early. The difference between ORD and ORD+ is the difference in the retrieval costs; it is evident that the execution time of ORD is dominated by the retrieval and decompression time. Consequently, the exponential change in the *Blur* cost is not apparent.

Without Optimization	With Optimization
79.5 secs	0.58 secs

Fig. 5. Effect of Optimization

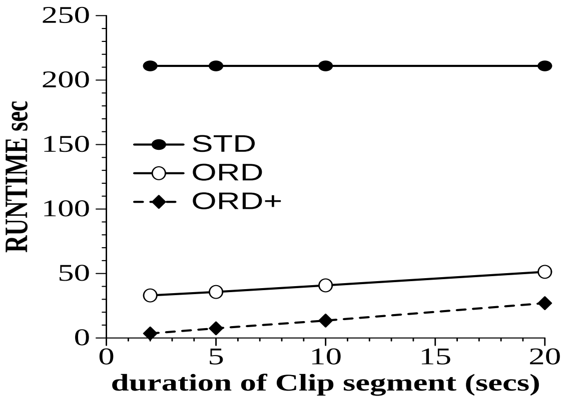

Fig. 6. Order and Constraint Optimization

Experiment 3: The third experiment demonstrates a very simple optimization. Consider the query below: it needs to find the area of overlap between pairs of images. There is really no physical requirement to compute the actual overlap of the images; their bounding boxes contain the necessary information to find the area of overlap. This can be thought of as either a Transformational optimization (changing *Overlap* to a different function), or a Constraint optimization (pushing the physical requirements into the computation of *Overlap*). The result is greatly improved performance. Figure 5 shows the difference.

```
SELECT Area(Overlap(C1.picture, C2.picture))
FROM Cars C1, Cars C2
WHERE C2.name = "Alfa_Romeo_8C_2300";
```

Experiment 4: This experiment is similar to Experiment 2, except that it uses audio data. We use a data set of 73 musical pieces defined by a Music relation. Each tuple has a name and an audio field. The audio data is stored in uncompressed AU format. The average size of each audio value is 1.13MB. Obviously, all the data does not fit in memory, making the optimizations even more important. Three functions on audio data are used; *Clip* extracts a time segment, *Echo* adds an echo effect, and *Rate* specifies the audio rate property. We would like to measure the effect of performing *Clip* ahead of *Echo* and vice-versa. The *Rate* method is used as a cheap way to reduce the display time of the result; as with *Height* in the image experiments, we disable optimizations that would apply *Rate* ahead of other methods.

SELECT Rate(Clip(Echo(M.song, <delay>),
<segment>))
FROM Music M;

We examine three evaluation strategies. The STD strategy does not reorder the methods. The ORD strategy pushes *Clip* ahead of *Echo*, and ORD+ ensures that *Clip* retrieves exactly the appropriate time segment from the audio data. All three strategies are optimized to pass intermediate results in main-memory. In the experiment, we vary the size of the clipped segment from 2 seconds to 20 seconds. The results are shown in Figure 6.

The performance of the STD strategy is dominated by the retrieval of the data from disk and *Echo* being applied to it. The size of the *Clip* segment makes little difference to it. The ORD strategy performs much better by limiting the amount of work that *Echo* needs to do. ORD+ improves on ORD by reducing the amount of data transferred from disk. While these results may look similar to those of Experiment 2, there are important differences. For the image experiment, the advantage of ORD+ was a reduction in the decompression needed; for this audio experiment, ORD+ wins by reducing the disk I/O needed.

Summary of Experiments: Each of the demonstrated optimizations can result in very significant improvements. Their combination clearly leads to orders of magnitude in performance gains. It is clear that even for these extremely simple queries, the E-ADT optimizations are radically more effective than the blackbox ADT approach. In practice, queries tend to be more complex than our examples, and the effects of optimization are even greater.

5 Related Work

Postgres, Illustra, and Informix: The issue of support for ADTs in relational database systems was first explored in [SRG83] and [Sto86]. This led to the development of the Postgres research DBMS [SRH90] and its commercial version, Illustra [Ill94]. The Postgres project explored issues dealing with the storage and indexed retrieval of ADTs. It also stressed that functions associated with ADTs could be expensive, and that special relational optimization techniques are necessary when such functions are present [Hel95]. The basic ADT approach described in Section 2 corresponds closely to Illustra's support for ADTs. The results of every ADT method are written to disk, and no inter-method optimizations are considered [Ols96]. While Illustra does have a rule engine, it is not used to apply optimization rules. Currently, Illustra's technology has been integrated with the Informix Universal Server and extended to exploit parallelism. Several modifications are being made to improve the evaluation of ADT expressions [Ols96]. The main improvements allow functions to retain the results in main-memory, or to present an iterator interface that helps pipelined execution as well as parallel execution. Transformational and Constraint optimizations are not supported.

Illustra did provide a slightly declarative semantics to ADT expressions. Users can specify simple commutative transformations to apply to the arguments of a function. For example, *Overlay(I1, I2) = Overlay(I2, I1)*. However, the system does not have the ability to apply these rules in a cost-based manner. Further, important optimizations like pipelining of methods are not supported. It is conceivable that this functionality could be extended to allow each ADT to specify an expressive collection of transformation rules, along with cost formulas and control strategies for rule application. To make the cost formulas meaningful, meta-information would need to be maintained for each ADT. These extensions basically require that Illustra adopt much of the E-ADT paradigm.

The Paradise System: Among current research systems, the Paradise client-server DBMS [DKL+94] is developing ADT extensions for the parallel execution of methods on spatial, geographic, and scientific data. This work concentrates on issues of scalable parallelism and the use of tertiary storage for large ADTs. The importance of parallelism for ADTs arises from the large size of complex data types, and the high cost of methods on them. The approach used is to partition large objects into "tiles" and define functions to work on one tile at a time. From the E-ADT viewpoint, parallelism is yet another benefit of declarative ADT expressions. Paradise also reduces the overhead of passing results between functions by allowing each ADT to manage its own main-memory buffer of data; temporary results do not have to be written to disk. This approach has limitations for large data types like audio and video that may not fit in memory. Paradise does not support the re-ordering of ADT methods in either a heuristic or a cost-based manner. However, the tiling of objects allows a Clip function to simply retrieve the appropriate tiles rather than the whole image.

Table 1. Comparison of Supported ADT Optimizations

Optimization	Illustra	Informix	Paradise	*Predator*
Algorithmic	No	No	No	Yes
Transform	No	No	No	Yes
Constraint	No	No	Partial	Yes
Pipelining	No	Yes	Partial	Yes
Cost-Based	No	No	No	Yes
Parallelism	No	Yes	Yes	No

Based on the categories of ADT optimizations listed in Section 2, Table 1 summarizes the comparison between Illustra, Paradise, *Predator*, and the Informix Universal server being developed.

Object-Oriented Databases: Early work on semantic data models incorporated domain semantics into relational query optimization. More recently, the OO-DBMS community has been exploring techniques to optimize queries involving complex objects. Much of the work in object-oriented query optimization has focused on issues like path expressions [CD92] and not on method expressions, although [CD92] recognizes that methods can be very expensive and merit further attention. While the OQL query language [Cat94] for OO databases does permit a method to have several implementations, it does not suggest a mechanism for choosing between these implementations. We direct readers to [GM88] for an excellent survey of work on query optimization for complex data types in OO-DBMSs. Most closely related to E-ADTs is the REVELATION project [GM88] which correctly identifies that the semantics of methods should be revealed to the query optimizer. There is a notion of a common object algebra, and every complex type expands (or "reveals") its methods into expressions in the common algebra. In contrast, E-ADTs can use individual algebras to represent query plans for their expressions, since there is no notion of a global query optimizer. In [AF95], the actual method code was analyzed to determine the "meaning" of the method, which was then used in query optimization. We believe that this approach is not viable; most complex data types will be developed in some imperative programming language like C++ or Java. Instead, we allow the E-ADT developer to explicitly specify the method semantics.

Heterogeneous and Distributed Databases: There have been many recent developments in the area of heterogeneous database systems [PGH96,LRO96]. As an example of the flavor of this work, in the Garlic system [CCH+95,TRS97], heterogeneous data repositories are integrated by using "wrappers" around each site. The wrappers export a common data model and provide query optimization capabilities at the level of operations (like scans) on collections of objects. There is a valid observation to be made that both wrappers and E-ADTs use encapsulation to achieve their goals. However, there some subtle differences: (a) E-ADTs do not need to export a common data model, (b) E-ADT optimizations operate at the level of individual data values, whereas wrappers typically operate at the level of collections. These differences arise because the problems being addressed are different. The E-ADT design of *Predator* is directed at optimizing access to complex data values, rather than physically distributed data. The major issue in heterogeneous databases is to run relational queries across possibly non-relational sources. Consequently, we would expect to wrap those sources with psuedo-relational functionality. However, we surmise that there are certain heterogeneous database problems that could benefit from the E-ADT approach.

Several distributed object broker architectures like CORBA and OLE have recently emerged. The capabilities of distributed objects are described using a common low-level interface. The OLE-DB standard [Bla96] supports the notion of a component database system with well-defined interfaces between different modular components. However, there is a distinction between exporting query capabilities, and exposing query optimization semantics.

Other Related Work: Rule-based query optimizers have received much attention recently [GM93,CZ96]. A common misconception is that a rule-based optimizer instantiated with the appropriate rules (in this case, with rules for E-ADT optimizations) provides all that E-ADTs do. This ignores several crucial aspects of E-ADTs — the maintenance of meta-information, the support for multiple algorithms for the same method, the ability to define multiple storage formats, etc. Rules provide one specific structuring mechanism for optimization semantics – in fact, *Predator* uses a rule engine to actually execute the optimizations for each E-ADT . An important distinction is that we promote local spheres of optimization for each E-ADT , with possibly different control strategies, different kinds of rules, etc. On the other hand, conventional rule-based optimizer proposals do not have such notions of optimization locality. The closest in spirit are the "region" architecture for query optimizers proposed in [MDZ93], the "module" architecture proposed in [SS90], and other research on extensible search strategies [RH87,LV91]. Our work differs in that it is focused on complex data type expressions, rather than relational expressions. In fact, we can make the interesting case that the E-ADT paradigm provides an excellent argument in favor of optimizer toolkits in general, and rule-based optimizer toolkits [GM93,KD98] in particular! Since the DBMS has several mini-optimizers, one in each E-ADT, a toolkit for optimizer generation is needed.

[CS93] suggests that queries involving "foreign" relations can be optimized by specifying the semantics of the foreign relations through high-level rules. The rules are syntactic, and cost-based optimization is performed after an exhaustive application of the rules.

We should note that method transformations of the kind suggested in this paper are common in the functional programming community. The pipelining optimizations are similar to the use of lazy evaluation [Jon87]. While the early work on ADTs [Gut77] did emphasize the equational theory of the methods, this aspect was not carried into the use of ADTs in database systems. Our work corrects this oversight, and focuses on optimizations based on statistics and costs in a database environment.

Local vs Global Approach?: In contrast to our loosely-coupled approach of E-ADTs , others have taken a holistic approach. Instead of breaking a query into many components with local query optimization on each E-ADT expression, these approaches try to find a global solution. This requires that the entire query be modeled in an integrated framework. AQUA [SLVZ95] is an algebraic frameworks proposed for this purpose, while CPL/Kleisli [Won94] is a framework based on comprehensions as a query language and monadic operations. These are all frameworks for collection types (like sets, bags, lists, and arrays). However, many important complex data types including multi-media types do not fall in this category.

In theory, an integrated framework is bound to find at least as good an evaluation strategy as the E-ADT approach, if not a better one, because the space of possible evaluation plans is at least as large. A DBMS specially built for relations and images should be able to perform better than *Predator*. Another possibility is to implement all the data types in a high-level language which can be reasoned about. We believe that while this may be a promising idea, current database systems are implemented in languages like C or C++. Working within this limitation, we have localized the optimizations to the E-ADT boundaries, in order to preserve the modularity of the data types.

6 Future Work

The main focus of our ongoing work is in extending optimizations across E-ADT boundaries. As a simple example of this, if we wish to have rules that span multiple E-ADTs (for example, image and video, which are closely related), the modularity of the individual data types is compromised to a certain degree. *Predator* already has mechanisms which allow such "bridges" between types.

We are especially interested in interactions between relational query optimization and E-ADT query optimization. The blackbox ADT approach for executing expensive methods in SQL is to execute them once for each new combination of arguments. As an example using images, *Overlap(I1, I2)* and *Overlap(I1, I3)* would be executed separately. If it were possible to execute the overlap of *I1* with both *I2* and *I3* at the same time, we could exploit locality by reading *I1* only once from disk. As a generalization of this idea, any function can be executed with individual arguments, or can be called on a set of argument instantiations. The blackbox ADT approach cannot exploit this set-at-a-time strategy because the semantics of the ADT functions are not known to the DBMS. However, with E-ADTs , this is indeed feasible.

Since E-ADT expressions are expensive, where should they be placed in the SQL query evaluation plan? Expensive function placement has traditionally been studied purely in a tuple-at-a-time execution context [Hel95,CS96], with caching of function results (an exception to this is [CDY95]). The assumption has been that every ADT function has a fixed cost specified in the system catalogs. This assumption is not valid when ADT expressions are being optimized. Since these expressions may be able to provide more details on their evaluation plans (for example, the main-memory requirements), the SQL optimizer should be able to find better overall execution strategies (possible requiring set-at-a-time evaluation).

It is possible to improve the interaction between E-ADTs using a mechanism whereby each E-ADT specifies its query processing capability. For example, if an E-ADT specifies that it understands the notion of a boolean connective (AND or OR), the expression *f(X) AND g(X)* in the WHERE clause of an SQL query could be replaced by the expression *f_and_g(X)*. This is more efficient because X is only accessed once. In general, constraints and other such information should flow across the query optimization interfaces. It is an open problem to design such a mechanism and the appropriate E-ADT interfaces.

7 Conclusion

The E-ADT paradigm is a novel yet simple approach to database systems design. Every data type can share the semantics of its methods with the DBMS. This allows several types of complex data to be efficiently supported within a single general-purpose DBMS. This paper makes the case that the next-generation of object-relational database systems should be based on E-ADTs. The type system of *Predator* is based on E-ADTs, and initial performance results provide empirical evidence of an order of magnitude increase in performance.

Acknowledgements. The basic E-ADT concept was developed along with Miron Livny and Raghu Ramakrishnan, who also co-authored the initial paper on this topic. Several students at Cornell have implemented parts of *Predator*. We appreciate the help of the Shore group at U.Wisconsin for the storage manager. Mike Olson at Informix and Fred Schneider commented exhaustively on a large part of this material. Praveen Seshadri was supported by an NSF Career Award and a Microsoft research grant.

References

[AF95] Karl Aberer and Gisela Fischer. Semantic query optimization for methods in object oriented database systems. In *Proceedings of the Eleventh IEEE Conference on Data Engineering, Taipei, Taiwan*, pages 70–79, 1995.

[Bla96] Jose Blakeley. Data Access for the Masses through OLE-DB. In *Proceedings of ACM SIGMOD '96 International Conference on Management of Data, Montreal, Canada*, pages 161–172, 1996.

[Cat94] R.G.G. Cattell. *The Object Database Standard:ODMB-93*. Morgan-Kaufman, 1994.

[CCH+95] M. Carey, W. Cody, L. Haas, W. Niblack, M. Arya, R. Fagin, M. Flickner, D. Lee, D. Petkovic, P. Schwarz, J. Thomas, M. Tork-Roth, J. Williams, and E. Wimmers. Querying Multimedia Data From Multiple Repositories By Content: The Garlic Project. *Proceedings of the IFIP Working Conference on Visual Database Systems, Lausanne, Switzerland*, March 1995.

[CD92] S. Cluet and C. Delobel. A General Framework for the Optimization of Object-Oriented Queries. In *Proceedings of ACM SIGMOD '92 International Conference on Management of Data, San Diego, CA*, pages 383–392, 1992.

[CDF+94] M.J. Carey, D.J. DeWitt, M.J. Franklin, N.E. Hall, M. McAuliffe, J.F. Naughton, D.T. Schuh, M.H. Solomon, C.K. Tan, O. Tsatalos, S. White, and M.J. Zwilling. Shoring up persistent objects. In *Proceedings of ACM SIGMOD '94 International Conference on Management of Data, Minneapolis, MN*, pages 526–541, 1994.

[CDY95] Surajit Chaudhuri, Umeshwar Dayal, and Tak Yan. Join Queries with External Text Sources: Execution and Optimization Techniques. In *Proceedings of ACM SIGMOD '95 International Conference on Management of Data, San Jose, CA*, pages 410–422, 1995.

[CS93] Surajit Chaudhuri and Kyuseok Shim. Query Optimization in the Presence of Foreign Functions. In *Proceedings of the Nineteenth International Conference on Very Large Databases (VLDB), Dublin, Ireland*, pages 526–541, 1993.

[CS96] Surajit Chaudhuri and Kyuseok Shim. Optimization of Queries with User-Defined Predicates. In *Proceedings of the Twenty Second International Conference on Very Large Databases (VLDB), Bombay, India*, pages 87–98, September 1996.

[CZ96] Mitch Cherniak and Stanley Zdonik. Rule Languages and Internal Algebras for Rule-Based Optimizers. In *Proceedings of ACM SIGMOD '96 International Conference on Management of Data, Montreal, Canada*, 1996.

[DKL+94] D.J. DeWitt, N. Kabra, J. Luo, J.M. Patel, and J. Yu. Client-Server Paradise. In *Proceedings of the Twentieth International Conference on Very Large Databases (VLDB), Santiago, Chile*, September 1994.

[GM88] G. Graefe and D. Maier. *Query Optimization in Object-Oriented Database Systems: The REVELATION Project*, volume 334 of *Advances in Object-Oriented Database Systems, Lecture Notes in Computer Science*, chapter . Springer-Verlag, 1988. Editor: K.R. Dittrich.

[GM93] G. Graefe and W. J. McKenna. The Volcano optimizer generator: Extensibility and efficient search. In *Proceedings of the Ninth IEEE Conference on Data Engineering, Taipei, Taiwan*, 1993.

[Gut77] J. Guttag. Abstract Data Types and the Development of Data Structures. *Communications of the ACM*, June 1977.

[Hel95] Joseph M. Hellerstein. *Optimization and Execution Techniques for Queries With Expensive Methods.* PhD thesis, University of Wisconsin, August 1995.

[Ill94] Illustra Information Technologies, Inc, 1111 Broadway, Suite 2000, Oakland, CA 94607. *Illustra User's Guide*, June 1994.

[Jon87] S.L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[KD98] Navin Kabra and David J. DeWitt. Opt++: An Object Oriented Design for Extensible Database Query Optimization. Submitted for publication to VLDB Journal, 1998.

[LRO96] A.Y. Levy, A. Rajaraman, and J. Ordille. Querying Heterogeneous Information Sources using SOurce Descriptions. In *Proceedings of the Twenty Second International Conference on Very Large Databases (VLDB), Bombay, India*, 1996.

[LV91] R.S.G. Lanzelotte and P. Valduriez. Extending the Search Strategy in a Query Optimizer. In *Proceedings of the Seventeenth International Conference on Very Large Databases*, pages 363–373, 1991.

[LZ74] B. Liskov and S. Zilles. Programming with Abstract Data Types. In *SIGPLAN Notices*, April 1974.

[MDZ93] Gail Mitchell, Umeshwar Dayal, and Stanley Zdonik. Control of an Extensible Query Optimizer: A Planning-Based Approach. In *Proceedings of the Nineteenth International Conference on Very Large Databases (VLDB), Dublin, Ireland*, pages 517–528, 1993.

[Ols96] Mike Olson, 1996. Personal Communication.

[PGH96] Y. Papakonstantinou, A. Gupta, and L. Haas. Capabilities-based Query Rewriting in Mediator Systems. In *Proceedings of the Parallel and Distributed Information Systems (PDIS) Conference*, pages 170–181, 1996.

[RH87] A. Rosenthal and P. Helman. Understanding and Extending Transformation-Based Optimizers. *Database Engineering*, 9(4):44–51, December 1987.

[Ses97] Praveen Seshadri. Predator: Design and Implementation, 1997. Documentation accompnaying code release.

[SLVZ95] Bharati Subramaniam, Theodore Leung, Scott Vandenberg, and Stanley Zdonik. The AQUA Approach to Querying Lists and Trees in Object-Oriented Databases. In *Proceedings of the Eleventh IEEE Conference on Data Engineering, Taipei, Taiwan*, March 1995.

[SRG83] M. Stonebraker, B. Rubenstein, and A. Guttman. Application of Abstract Data Types and Abstract Indices to CAD Data Bases. In *Proceedings of the Engineering Applications Stream of Database Week*, San Jose, CA, May 1983.

[SRH90] Michael Stonebraker, Lawrence Rowe, and Michael Hirohama. The Implementation of POSTGRES. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):125–142, March 1990.

[SS90] Edward Sciore and John Sieg. A Modular Query Optimizer Generator. In *Proceedings of the Sixth IEEE Conference on Data Engineering*, pages 146–153, 1990.

[Sto86] Michael Stonebraker. Inclusion of New Types in Relational Data Base Systems. In *Proceedings of the Second IEEE Conference on Data Engineering*, pages 262–269, 1986.

[TRS97] M. Tork-Roth and P. Schwarz. Don't scrap it, wrap it! A Wrapper Architecture for Legacy Data Sources. In *Proceedings of the Twenty Third International Conference on Very Large Databases (VLDB), Athens, Greece*, pages 266–275, August 1997.

[Won94] Limsoon Wong. *Querying Nested Collections*. PhD thesis, U.Pennsylvania, 1994.