# The Case for Enhanced Abstract Data Types

Praveen Seshadri
Miron Livny
Raghu Ramakrishnan

Technical Report #1319

July 1996

1319

# The Case for Enhanced Abstract Data Types *

**Praveen Seshadri** **Miron Livny** **Raghu Ramakrishnan**
Computer Sciences Department
U.Wisconsin, Madison WI
*praveen,miron,raghu@cs.wisc.edu*

**Abstract**

This paper is based on the thesis that the current notions of abstract data types in object-relational systems fall short of their potential in terms of usability, functionality and performance. We make the case that the ADT mechanisms in current O-R systems actually provide only limited support for complex data. Instead, we propose an enhanced notion of ADTs (E-ADTs ), by permitting each E-ADT to define a declarative language, query optimizer, execution engine, catalog management and storage management for data objects of that particular type. These extensions are in addition to the existing ADT functionality (which provides methods/functions over objects).

E-ADTs can be significantly easier to use, can provide greater functionality to the user, and can lend themselves to much more efficient implementations. Further, the E-ADT paradigm results in a plug-and-play system in which complex data types can be developed and can function independently of each other. The proposed architecture has been used in the $\mathcal{PREDATOR}$ database system to integrate relational data with sequence data in an extensible yet efficient manner. We use our experience from this implementation to describe the system-level interfaces needed, as well as possible migration paths that existing database systems can use to adopt the E-ADT paradigm.

## 1 Introduction

It has become the accepted practice to require that database systems allow for type extensibility, i.e., new data types can be added to the system without significant changes to any part of the existing code. Relational databases that support such type extensibility are called "object-relational" or OR for short (examples include Illustra [Ill94], Paradise [DKL+94], and Postgres [SRH90]). For the sake of concreteness, most of our arguments be restricted in this paper to OR databases, but we will extend them to OODBs as well. The move towards database type extensibility began in the early 80's, notably through work by Stonebraker on the Ingres database system [Sto86], leading to the the development of the Postgres system. A dozen years later, similar ideas have moved from research proposals to the forefront of today's object-relational (OR) technology. The current OR notion of Abstract Data Types (ADTs) permits users to define their own types using compositions of basic types. However, for important types of complex data (like images, time-series, matrixes, arrays, etc.), the ADT functionality is implemented by database system developers and is provided as a library (often called a "datablade" [Ill94]). We should therefore make the distinction between "user-defined ADTs" and "developer-defined datablade ADTs". The user-defined ADTs are usually relatively simple (for example, a user may define a "line" ADT as a struct containing two

*Praveen Seshadri was supported by IBM Research Grant 93-F153900-000 and an IBM Cooperative Fellowship. Miron Livny and Raghu Ramakrishnan were supported by NASA Research Grant NAGW-3921. Raghu Ramakrishnan was also supported by a Packard Foundation Fellowship in Science and Engineering, a Presidential Young Investigator Award with matching grants from DEC, Tandem and Xerox, and NSF grant IRI-9011563.

points), and do not require any database smarts. On the other hand, datablade ADTs usually deal with large data objects with complex and expensive operations. This paper concerns itself with the datablade ADTs.

Supporting new kinds of complex data is arguably one of the biggest challenges facing the database research community today. The datablade ADT paradigm has been promoted as the best approach to take in meeting this challenge [SM95]. To its credit, the ADT technology of O-R systems is a lot more efficient than the "blob" technology used by purely relational systems to support complex data (in which complex data objects are byte extents that are interpreted solely by the application). However, this paper finds several important deficiencies in the ADT mechanism, and proposes a greatly enhanced notion of ADTs that addresses these inadequacies. Specifically, the following claims are made:

- The ADT approach gives special status to relational data and the relational query language SQL, at the expense of other complex data types.
- The ADT approach makes it unwieldy and unintuitive to express queries over complex data.
- The ADT approach makes expressions over complex data extremely inefficient to execute.

Of the three arguments against the ADT approach, the first two are aesthetic (though still important), but the third is really the crucial argument. We claim (and have demonstrated) that there are improvements of orders of magnitude to be gained by enhancing the ADT approach in the manner that we propose. This is the E-ADT (standing for Enhanced Abstract Data Type) paradigm, which enhances the notion of an ADT in five different ways. Each ADT

1. Provides a query language.
2. Provides a query algebra and optimization techniques.
3. Provides an evaluation engine.
4. Maintains catalog information for its objects.
5. Performs storage management for its objects.

These enhancements are in addition to the existing ADT features, and do not all have to be provided by every E-ADT . We motivate the need for these enhancements with a simple example of a matrix data type. We then underscore it using the example of a sequence data type (as might be used for time-series data). This is not merely a proposal, however; the E-ADT paradigm has been used to design and implement the $\mathcal{PREDATOR}$ database system to support sequence data (in addition to relational data). We describe the low-level details of what the E-ADT paradigm entails, and we also provide performance measurements that demonstrate the improvements due to E-ADTs. We also discuss the interaction between E-ADTs and the possibilities for set-oriented optimizations across E-ADT boundaries. As a constructive and practical measure, we also describe possible paths for systems based on the ADT model to migrate to E-ADT technology without drastic changes to the underlying system. Finally, we discuss how the system can be extended to support several languages for the same data type, multiple heterogeneous systems all supporting similar types, and heterogeneous systems with different types.

## 2 Motivation

Assume that we have an O-R DBMS based on the ADT paradigm. This permits functions to be defined and registered with the database system. These functions can then be used within the WHERE or the SELECT clause of an SQL query. We wish to add a matrix ADT so that the DBMS may be used by mathematicians/scientists to store some experimental data. The functions provided with the matrix ADT are Multiply(M1, M2), Transpose(M1), and Clip(M1, Range). The first two functions have their obvious mathematical meanings, while the third function extracts a portion of the matrix as the result. Let us consider a single relation Exp of experimental data with a schema : { Station : char[30], Humidity : matrix }. The Humidity attribute is supposed to be a 100X50 matrix of readings where the dimensions correspond to Temperature and Pressure, and the Temperature varies from 1 to 100 degrees, while the Pressure varies from 500 mbar to 1500 mbar in steps of 20 mbar. We will use this example to study the following issues:

- How is the Humidity matrix defined?
- How is the matrix data stored?
- How are queries involving the matrices expressed?
- How are these queries evaluated?

## 2.1 Defining the Matrix

The first question to ask is: how exactly is the schema specified? While we have written "Humidity : matrix", obviously we have omitted the type of the entries, the dimensionality, the names Temperature and Pressure, the sizes of each dimension, and the step size in each dimension. How can one specify these in an ADT based system? Typically, there is a matrix "type constructor" which allows the user to define a specific sub-type of matrix to be used. In this case, the matrix type constructor would have to take all this information as arguments and generate a new specific type. The attribute declaration would look something like:

```
create type
  MyMatrix matrix(double,                          /*** the type of entries    ***/
                  2,                               /*** dimensions             ***/
                  {"Temperature", 1, 100, 1}, /*** name, start, end, step ***/
                  {"Pressure", 500, 1500, 20} /*** name, start, end, step ***/
                 )
Humidity : MyMatrix
```

Of course, while MyMatrix has exactly the same functions that can operate on it as any other matrix, it is entered as a new ADT into the table of types defined by the system. If we are to make this distinction between ADT constructors and actual ADTs, then it would be fair to say that almost all complex data belong to ADTs created using ADT constructors. This is true for images, time-series, matrixes, arrays, and even for something as simple as a string (a String[30] is constructed using the String type constructor and the length argument 30)! However, in all these cases, the ADT instances generated by the application of an ADT constructor do not support any functions over and above what is supported by the generic ADT (as in this case, MyMatrix has the same functions that all matrices do). Therefore, in this paper, when we talk about ADTs, we often refer to the generic ADT (like matrix) that provides the functionality, rather than the specific ADT (MyMatrix) that is used at the level of type checking.

Returning to our example, the question that arises is : "Why should we have to create a new ADT for each specific kind of matrix, if they all support the same functions?". Indeed, since there are an infinite number of such matrix ADT instances, they have to be created as needed by users. Consider another matrix almost identical to MyMatrix with the dimensions called A and B instead of Temperature and Pressure. Is this really a different type? What if the step size is slightly different, or the limits of the dimensions are different? Any practical system needs to draw the line somewhere, and there may be further specializations that distinguish between two uses of the very same type instance.

A more intuitive way to think of this example instead is that the Humidity attribute is of type matrix, and there is some additional meta-information that specifies the structure, properties, etc., of the matrix. This is totally analogous to thinking of all tables in an RDBMS as relations, with meta-information for each relation specifying the schema and other properties. The term "parametric types" [CW85] has also been used in this context to describe a type which is parameterized by additional meta-information.

## 2.2 Storing the Matrix

There may be several different implementations of a matrix. For example, there is the issue of whether the matrix is stored in a row-major or a column-major format (in fact for multi-dimensional matrices, there are many more possibilities), or whether it is broken into tiles [SS94]. Further, different implementation techniques may be used depending on the size of the matrix. Depending on the sparseness of the matrix, compression may be used. How can one specify what storage implementation is to be used? The ADT solution is to add this information to the type constructor. Therefore, MyMatrix stored in column-major format is a different type from MyMatrix stored in row-major format. This definition of types is definitely not derived from programming language type theory, since this is merely a storage issue, not a logical issue. Once again, the abstraction of meta-information stored with the type is much more intuitive to address issues like storage formats.

## 2.3 Expressing and Evaluating Matrix Queries

Now let us consider how a query may be posed against this experiment database.

```
SELECT Clip(Transpose(E.Humidity), 700, 800, 30, 60)
FROM Exp E
WHERE E.station = ''Erewhon''
```

There are different algorithms that may be applied to implement the transpose function. Their relative merits depend on the size of the matrix, its sparseness, the size of main memory available and the row or column-major storage technique. In typical ADT-based DBMSs, code for the Transpose function is written assuming a generic matrix. The same code is executed for all matrices; therefore it must take into account the factors that affect the algorithmic choices. Some of these factors may be known from the MyMatrix type instance of the argument (in this case, the size and storage technique may be known). However, the sparseness of the matrix is often a property of the *data*, not of the type. Consequently, if the function is to make the right decision, it must apply its decision rules for each function application depending on the characteristics of its particular argument during that invocation. Typically, database functions ignore such information and apply the same algorithm whatever the argument. However, proponents of the ADT approach may argue that this is merely for reasons of convenience rather than because of any real flaw in the paradigm. It is even conceivable that the available memory is passed by the system to the function as a special argument and is used at run-time to decide which specific algorithm to apply.

If the Clip function is applied after the Transpose function, its arguments are the range 700-800 on the first dimension (Pressure) and 30-60 on the second dimension (Temperature). Note that the dimensions were switched as a result of the Transpose function. On the other hand, it is also possible to evaluate the Clip function *before* the Transpose function. That expression evaluation might look like: Transpose(Clip(E.Humidity, 30, 60, 700, 800)). There are cases where this may be more efficient.

Now consider the case where the expression involving the matrix is a little more complicated. Specifically, let the expression be:

```
Transpose(Multiply(A, B))
```

There are several different and equivalent ways of evaluating this. The most obvious way is to first multiply the two matrices and generate the intermediate result; then compute the transpose of the intermediate matrix product. If the functional expression is interpreted procedurally (which is more or less the current state of the art in database ADT technology), this is the specific evaluation strategy chosen too. However, it is obvious that there are several other possibilities: for example,

```
Multiply(Transpose(F.Humidity), Transpose(E.Humidity))
```

Also, it may not be necessary to materialize the entire intermediate result of each function. It may be possible to stream or pipeline the intermediate results into the computation of the next function. In general, for any reasonably complex expression, there can be a wide range of possible evaluation strategies, each with different costs depending on the statistics of the underlying data. Ideally, the system should decide automatically among these choices in a cost-based manner. In current OR database systems, there is no framework for these choices to be considered. Instead, the user's specification of the functional expression also serves as a procedural specification of the evaluation strategy.

## 2.4 Summary

To summarize the discussion of this section:

1. In order to model and manipulate matrixes, they need to be embedded inside relations and queried using a functional syntax within SQL. Since the top-level query language is SQL, there is no way to directly execute some expression involving matrices without embedding it inside an SQL expression. Further, the use of a functional syntax for all matrix operations is not always aesthetic or convenient. For example, expressing a complex arithmetic operation in a purely functional syntax is awkward. It would also be nice for the user to be able to use the index labels (Temperature and Pressure) meaningfully inside a matrix expression. These issues are basically at the level of query syntax; however, the next issue deals with query performance.

2. The functional expressions over matrixes are evaluated in a *procedural* fashion — in contrast to the traditional database technique of keeping queries as *declarative* as possible.

One way to characterize our contribution is as trying to make expressions involving ADT objects *more declarative*, and consequently more efficient. We would like to do this without sacrificing the goal of *extensibility* – i.e., it should be possible to add new data types without causing dramatic changes to the rest of the system. Figure 1 demonstrates the various possible options if one wants to support complex data in database systems. A query can involve several subexpressions, some of them dealing with relations, and some of them manipulating objects of various other data types. As we have just discussed, the current OR systems allow the relational portion of the query to be declarative, but the other parts of the query are procedural.

Another proposed approach is to construct a "mega-model" that can represent all the various kinds of complex data, and queries over such data. A system based on this model would require a single unified algebra that can represent all operations over the various types of data (one such proposal is the Aqua algebra [SLVZ95]). While in this scenario, the entire query expression can be declarative, the important question is whether such a system is extensible, and whether it is practical. If there are already several types in the system and a new type (like a

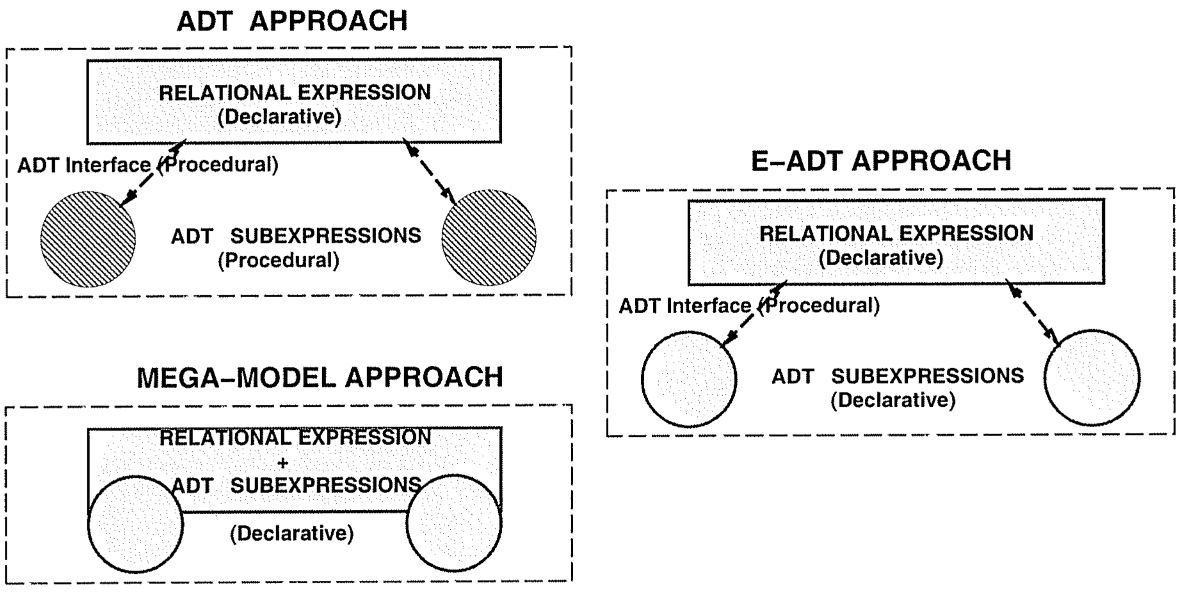

Figure 1: Alternatives Approaches for Complex Data Support

matrix) needs to be added, how can all the interactions between matrixes and the other types be specified, and how can this information be used? Obviously, we will be sacrificing the goal of clean extensibility by requiring that each data type fully understand its interactions with all other data types. Further, it is not clear that queries will be easy to pose, or practical to optimize and evaluate.

There are two other possibilities that the figure does not show. One is to support just the relational data and handle everything else in application code. This is the "blob" technology supported by relational database systems, and it basically addresses the problem of complex data by avoiding it altogether. This approach suffers from a lack of both efficiency and extensibility. The other approach is to try to "flatten" all complex data into tables which can then be manipulated using SQL. Since SQL is becoming a computationally complete language [ISO93], this approach has the potential to provide the desired functionality without dramatic changes to the relational database. Unfortunately, queries may be very awkward to pose, and extremely inefficient to evaluate. The ADT approach has gradually gained acceptance as being superior to these two earlier approaches.

Our proposal of Enhanced ADTs (E-ADTs) lies between the two extremes of the ADT approach and the "mega-model" approach. Each of the sections of the query corresponding to a specific data type is declarative. However, across sections of the query, the interaction is procedural. For example, in the example of matrices embedded inside relations, the expression involving the matrices (Transpose(Multiply(A,B))) is treated as a declarative matrix manipulation, but the interaction between this sub-expression and the rest of the SQL query is procedural. We will of course explain this in greater detail in the following sections, along with possible extensions to the basic ideas. At this stage, it is important to note that we retain the extensibility of the ADT approach, by giving up some of the declarativeness of the mega-model approach.

## 3 High-Level System Design

In this section, we describe the E-ADT paradigm in the context of the $\mathcal{PREDATOR}$ database system, which is based on E-ADTs. We first describe the overall data model design, and then its instantiation in the $\mathcal{PREDATOR}$ system architecture.

## 3.1 Data Model Design

The $\mathcal{PREDATOR}$ design enhances the ADT notion by supporting "Enhanced Abstract Data Types"(E-ADTs ). The enhancements improve the declarativeness and hence the performance of queries involving values of the E-ADT . This is a nested data model in which a complex object like a sequence can be a field within a relational tuple, and vice versa. Note that the type Relation is also modeled as an E-ADT . Notions of object-orientation, identity or inheritance are orthogonal to the issues addressed in this paper, and consequently we shall ignore them. Each E-ADT may provide support for one or more of the following:

- Query Language: An E-ADT can provide a query language with which expressions over values of that E-ADT can be specified (for example, the relation E-ADT may provide SQL as the query language, and the matrix E-ADT may provide its own language). Note that as a special case, the language provided can have a functional syntax just like OR systems.
- Query Operators and Optimization: If a declarative query language is specified, the E-ADT must provide optimization capabilities that will translate a language expression into a query evaluation plan in some evaluation algebra.
- Query Evaluation: If a declarative language is specified, the E-ADT must provide capabilities to execute the optimized plan.
- Catalog Management: Each E-ADT can provide catalog capabilities so that schema information can be stored and statistics maintained on values of that E-ADT . Further, certain values may be named, and a name mapping may be maintained.
- Storage Management: Each E-ADT can provide multiple physical implementations of values of its type. The particular implementation used for a specific value may be specified by the user when the value is created, or determined automatically by the system.

We argue in this paper that the E-ADT paradigm can be applied to provide database support for *any* complex collection type. At least in the context of sequence data (which is one specific kind of complex data), the merits of this approach over the ADT-method approach have been clearly demonstrated in a quantitative manner [SLR96]. The ability to name objects belonging to different E-ADTs allows *any* E-ADT to be the top-level type. Relations are modeled on par with other E-ADTs . Consequently, even if the entire relation E-ADT were not implemented or not compiled in, the database system would still be able to provide the functionality supported by the other E-ADTs . This allows users who are primarily interested in matrix data, for example, to directly query named matrices without having to embed the matrices inside relational tuples.

## 3.2 System Implementation

The design philosophy of E-ADTs is carried directly over into the system architecture. $\mathcal{PREDATOR}$ is a client-server database in which the server is a loosely-coupled system of E-ADTs . The high-level picture of the system is shown in Figure 2. The server is built on top of a layer of common database utilities that all E-ADTs can use. Code to handle arithmetic and boolean expressions, constant values and functions is part of this layer. The primary portion of the utility layer is the SHORE Storage Manager [CDF+93]. SHORE provides facilities for concurrency control, recovery and buffer management for large volumes of data. It also provides a threads package that interacts with the rest of the storage management layers; $\mathcal{PREDATOR}$ uses this package to build a multi-threaded server. Multiple clients can connect to the server and have their requests serviced.

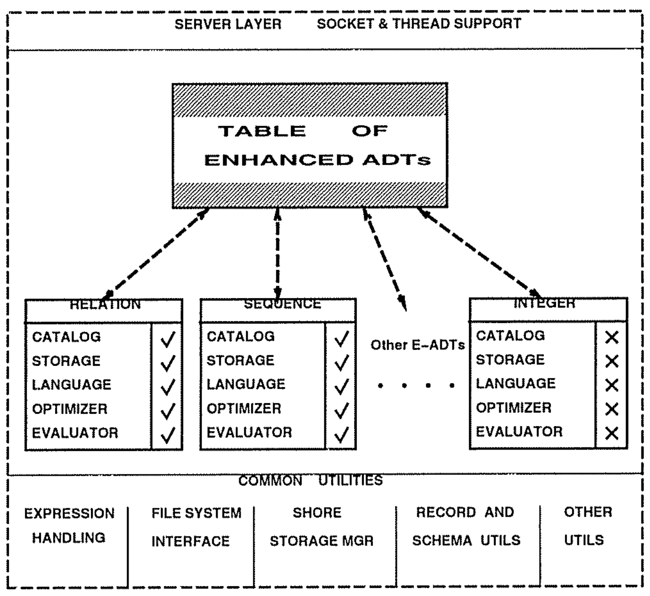

Figure 2: System Architecture

The core of the system is an extensible table in which E-ADTs are registered. Each E-ADT may support and provide code for a query language, an optimizer, a query evaluation engine, and storage and catalog management for data belonging to that type. As shown in the figure, some of the basic types like integers do not support any enhancements (although one could argue that a 'language' like integer arithmetic is itself a declarative language over the integers). The figure shows two E-ADTs that do support enhancements: sequences and relations. A matrix E-ADT could also be added to the system.

With this high-level system design, the important question to ask is: how do interactions between the E-ADTs occur? To explain this, we will revert to our example involving matrices and relations. At the user level, we will describe how the two E-ADTs (relations and matrices) interact. We will also describe the lower level interfaces supported by each E-ADT that allow an extensible system to be built around the E-ADT concept.

## 4 E-ADT Interfaces

We stay with our example from Section 2.3 which we repeat here again.

```
SELECT Clip(Transpose(E.Humidity), 700, 800, 30, 60)
FROM Exp E
WHERE E.station = ''Erewhon''
```

In this case, the data is modeled as a relation, and the tuples of the relation have a field called Humidity which is a matrix. Another possible design is to have named matrices (for example, a name matrix called Erewhon). Staying with the original logical design, we now consider what it means to ask this query in the E-ADT framework of *PREDATOR*. We will also assume for simplicity that the query language for matrices is called MQL, and has a functional syntax exactly as is used in the query. We note that there have been other query languages proposed for multi-dimensional arrays ( [LMW96]), and any of these could be used instead. We now modify the syntax of the query, so that the matrix sub-expression is "marked" as belonging to the MQL query language.

```
SELECT MQL{"Clip(Transpose($1), 700, 800, 30, 60)", E.Humidity}
```

```
FROM Exp E
WHERE E.station = "Erewhon"
```

Note that the change is purely syntactic, and serves to help the user (and the system) identify the sub-expressions in the query that belong to each data type. For example, an equivalent syntax for the expression is shown below:

```
DEFINE FUNCTION Foo (matrix H) as MQL {Clip(Transpose(H), 700, 800, 30, 60)};

SELECT Foo(E.Humidity)
FROM Exp E
WHERE E.station = "Erewhon"
```

The specific syntax chosen is not important; what matters is that the system be able to identify the boundaries between sub-expressions belonging to different languages. We will now examine the various stages of query processing after the query has been specified. Note that each of the E-ADTs is specified independently of the others. We will also stick to the first syntax shown in the section for the embedded MQL expression.

**Parsing and Type Checking:** Each E-ADT provides a language parser by means of the following virtual method:

```
ErrorCode ExprParse(const char* ExprString,
                    const ArgEnv* ArgEnv,
                    BasicType& ReturnType,
                    GenericParse*& ParseStructure)
```

The ExprString is the textual string containing the query expression. The ArgEnv represents any values passed to the expression because it is embedded in a nested scope. In the example query, the nested MQL query receives the E.Humidity argument as part of its environment. The function returns the type of the expression, as well as a generic ParseStructure that will only be interpreted by the E-ADT that creates it (i.e., the matrix E-ADT ).

At the top-level, the query is an SQL query, and is therefore parsed by the SQL parser (which is the language provided by the relation E-ADT ). The SELECT clause of the query has an embedded expression in the MQL query language. The SQL parser treats the entire nested expression as an uninterpreted string, and the MQL parser is invoked on it immediately after the SQL query is parsed. The type checking for the query is straightforward; before the MQL expression is parsed, the type of the argument to it (E.humidity) is known. The return type of the MQL expression is used to complete the type checking of the SQL query.

**Query Optimization:** The optimization of an SQL query follows certain standard cost-based techniques in searching among a space of evaluation plans. Any sub-expressions need to be first optimized before the SQL query can be optimized. Optimizing the MQL subquery involves invoking the following virtual method defined on the matrix E-ADT .

```
ErrorCode ExprOptimize(const GenericParse* ParseStructure,
                       GenericPlan*& PlanStructure);
```

This method takes the parse structure generated as a result of parsing the matrix expression and returns a generic structure that represents an evaluation plan for that expression. Note that the PlanStructure is interpreted only by the matrix E-ADT , thereby conforming to the extensibility requirements. The only well-known method of the PlanStructure is a Cost() method that returns a cost function. As we discuss in more detail in Section 5, a common cost model is needed to integrate the optimizations of multi-E-ADT queries. In our example, the SQL optimizer uses the knowledge of the cost of the sub-expression to determine the best plan for evaluating the SQL query. For example, if the matrix subexpression is very expensive, it may be preferable to apply it as few times as possible.

We do not discuss the details of how exactly the matrix E-ADT can find a good plan to evaluate a query. There has been much recent work on database techniques to support and optimize manipulations of complex data types. We have extensively studied support for sequence data (which can be used to model time-series and event streams). Optimization techniques for operations in this domain are discussed in [SLR94], and we have shown that they can result in performance improvements of a couple of orders of magnitude [SLR96]. Recent work dealing with multi-dimensional arrays [LMW96] has suggested optimization techniques for array operations. There has also been work on optimizing manipulations of images, chemical molecules, genome sequences, lists, etc. Our contribution in this paper is a mechanism to combine these contributions together under a common systems framework.

**Query Evaluation:** The evaluation of the query is relatively straightforward. The SQL query plan starts to be executed by the relational query execution engine. A virtual method provided by each E-ADT performs this evaluation.

```
ErrorCode ExprEvaluate(GenericPlan* PlanStructure,
                       const ValueEnv* ArgEnv,
                       void *ReturnValue);
```

The optimized plan is interpreted by the evaluation engine. In the case of the SQL query evaluation plan, it does not need any arguments passed to it. However, while executing the portion of the query corresponding to the SELECT clause, it needs to invoke the matrix subexpression. As far as the SQL plan is concerned, this is some unknown subexpression which it does not know how to execute. However, it does know what the arguments to the expression are (E.Humidity), the type of the return value of the expression, and the E-ADT to whom the expression belongs (matrix). It also has a handle on the query evaluation plan for the matrix subexpression, though it cannot interpret it. Instead, it repeatedly invokes the ExprEvaluate() method of the matrix E-ADT , passing it the plan structure and the argument, and uses the return value to continue with its computation.

Therefore, the query is broken into components that correspond to sub-expressions in each query language. The sub-expressions may be nested within each other. Query evaluation begins at the highest level of the query. The interfaces across the different subexpressions are procedural, and are evaluated like function invocations. However, within each subexpression, the evaluation is declarative, based on a cost-based optimization. We should note that our proposed solution is "loosely-coupled" in the sense that there is little interaction between the various E-ADTs in the system. The primary interaction arises due to the shared table of types, the common cost model, and the use of the interface methods for handling nested expressions. There are also some other methods that contribute to the "virtual" interface of E-ADTs . These are listed in detail in the appendix. Mainly, these methods provide a certain degree of catalog management, so that nested complex objects may be modeled. For instance, while defining the experiment relation Exp, the embedded matrix field Humidity needs to be defined. This happens via the invocation of a method on the matrix E-ADT with arguments specifying the name of the new field (Exp.Humidity), and the textual string with which the user specifies the schema of the Humidity matrix.

## 5 Limitations, Enhancements, and Migration Paths

In this section, we provide a discussion of the basic ideas presented in the previous section. We also show possible extensions of these ideas, and a migration path that existing systems could take to incorporate different degrees of this functionality.

## 5.1 Limitations

We now present some possible objections to the E-ADT scheme, and try to address some of the concerns raised. The first concern is that adding a new data type has become a complicated process, because so much new functionality (parsing, optimization, evaluation, catalogs and implementations) have to be added. We should note however, that all of these features are purely optional. Although the system supports E-ADTs , it still involves strictly no additional work to add the traditional style of ADTs (with methods on them, but little else). We would argue though, that for data types representing large complex data with complex manipulations (like matrices and time-series), the improvements in performance (shown to be orders of magnitude in the case of time-series) will be well worth the additional few thousand lines of code. Further, we have been studying ways of moving as much functionality as possible into the Utilities layer of the system, thereby making it easier to specify each E-ADT . As a very simple example, one could stick to the functional syntax for E-ADT expressions, and simply specify rules that identify the functional equivalences. These rules might be applied by an rule-driven optimizer provided as part of the Utilities layer. Also, heuristics may be used within any specific sub-expression, so that the implementation of an "optimizer" may be relatively easy.

The next question to ask is whether the mega-model approach (express the entire query in some unified algebraic model and then optimize it) would find better evaluation plans than the E-ADT approach. At first glance, it would seem that the answer is obvious: placing the query in a unified 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 options is strictly greater. Of course, one loses the advantages of extensibility, but one might buy performance benefits. On closer examination however, it becomes evident that simply because there is a larger space of plans does not mean that the space is searched, and that all the plans in it will be considered. For practical reasons, all database systems limit the possible plans considered, so that even within a totally relational query, there are many plans that are not ever considered. In light of this observation, using a mega-model algebra does not necessarily mean that a better plan will be found in practice. However, there is a legitimate concern that there may be some inter-E-ADT optimizations that might be important. We now describe some such optimizations that could be implemented as extensions to the basic E-ADT mechanism.

## 5.2 Extensions

Consider an SQL query which has an embedded expression which is also in SQL. It might be possible to recognize that it is the very same E-ADT on both sides of the expression interface, and merge the two expressions. The resulting merged expression might result in more efficient performance (possibly using techniques like those suggested in [PHH92]). Extrapolating from this, if there are two similar E-ADTs like relations and sequences involved in a query, it might be possible to propagate information from one E-ADT expression to the other. For example, a selection condition might be pushed across the interface.

The main source of procedural behavior in the E-ADT paradigm is at the interfaces between subexpressions. The evaluation mechanism across the E-ADT interface is *tuple-at-a-time*, i.e., the subexpression is evaluated once for each combination of arguments. Tuple-at-a-time execution is typically a source of inefficiency in database systems. However, this problem is identical to similar problems in nested SQL queries and heterogeneous queries. If each expression also provides a "set-at-a-time" plan (i.e., given a set of arguments, instead of a single one, evaluate the expression for each of the arguments), then this provides a set-oriented interface across E-ADTs . Recent work [SHR+96] has described how cost-based optimization can be used by a database system to determine how best to utilize set-oriented plans. The same techniques can be applied to E-ADT queries as well.

Another possible source of inefficiency could be the occurrence of the same E-ADT subexpression (or portion of it) in multiple places within a query. For example, within an SQL query, the very same matrix subexpression could occur within both the WHERE clause and the SELECT clause. This is a standard problem with nested expressions, and is not really caused by the E-ADT paradigm. The very same problem would arise with the functional expressions of the standard ADT approach. Possible solutions to the problem are based on recognizing common subexpressions in different parts of the query. In fact, the E-ADT approach may well be better suited to handling this, since the E-ADT optimizer could maintain its state across multiple invocations on different subexpressions in the query. In this way, it can detect common subexpressions in the query.

## 5.3 Migration Paths

Our E-ADT proposal requires reasonably extensive changes to existing database systems. Most current relational database systems are slowly incorporating portions of the ADT technology used by object-relational systems. We will assume a relational database system that has the notion of an ADT and has mechanisms to allow functions to be extensibly defined. In order for such a system to adopt some subset of the features of the E-ADT paradigm, we now suggest various possible migration paths.

### 5.3.1 Minimal Extension

The minimal extensions suggested here are typically already performed to some extent in object-relational systems.

- As a first step, each ADT may be allowed to have multiple implementations. The choice of implementation may be specified by the user or decided automatically by code provided as part of the ADT. This functionality is already present in the Illustra database system [Ill94]. Of course, there may be some ADTs for which a single implementation may suffice. On the other hand, for the matrix ADT, there may be a row-major and a column-major implementation, in addition to different implementations for sparse and dense matrices.
- The next step is to use the knowledge of the implementation of the data type inside each function evaluation. For example, the Transpose function can first check if its argument is in column-major or row-major format and then invoke different code for each case. Note that this form of optimization is performed at run-time for each independent invocation of the function.

### 5.3.2 Exploiting Heuristics

The next set of enhancements try to exploit heuristics for optimization. For instance, let us assume that the database system is associated with a rule-based engine that can apply heuristic transformation rules. Each ADT could provide rules that specify heuristic transformations that should be applied to expressions involving functions of that ADT. For example, there could be a rule for the matrix ADT that says :

```
Clip(Transpose(X), A, B, C, D) ==> Transpose(Clip(X, C, D, A, B))
```

Note that this rule is a *heuristic* and assumes that it is better to apply the Clip function early (this is analogous to the selection pushdown heuristic used in relational query optimization). The rule is applied at "compile time", which is a significant change.

There are a couple of important limitations to this extension. While the example we used could be specified as a relatively simple rule, the typical kinds of rules used in query optimization are much more complex. For instance, even a straightforward rule like pushing a selection through a relational join is difficult to express in this high-level manner. Various properties of the selection condition and the join condition need to be checked. In fact, most practical implementations of rule-based relational query optimizers use procedural code to implement each optimization rule. It is reasonable to expect that optimizations of expressions involving other kinds of complex data will also require rules of similar complexity. In fact, our experience with optimizing sequence queries bears out this expectation [SLR94].

The other limitation arises due to the fact that these rules are applied as heuristics, and not in a cost-based manner. In order for the rules to be treated as equivalences which can then be applied in either direction, the system needs to be able to supply cost estimates of the queries that result from applying the rules. Estimating the cost requires a cost model and cost formulae for the various functions. This leads us to the next possible extension.

### 5.3.3 Applying Cost-Based Transformations

Let us eliminate the two limitations of heuristic rules by (a) allowing the rules to be specified using complex procedural code, and (b) allowing the rules to be applied in conjunction with cost metrics.

We observe right away that since the rules require procedural code, they need to be written by a "database implementor" rather than by a naive user. This is keeping with the model of implementing important data types as vendor-supplied "datablade" libraries. In order for the rules to be applied in a cost-based manner, cost formulas for each of the functions need to be specified. While the common cost model could be based on CPU and I/O usage (just as in relational query optimization), we are faced with the following problem: to what extent does the cost formula depend on properties of the arguments to the function? Note that at least one of the arguments will be a value of the complex data type (for example, a matrix). The cost formula has to determine *at compile time* what the cost of executing the function will be at run-time. This is obviously a difficult problem without any knowledge of the properties of the data value. The equivalent problem in relational query optimization might be to find the optimal join order without any information about the implementations of and statistics on the relations being joined!

What if the optimization rules were applied at run-time using the known properties of the run-time arguments? While this would indeed be feasible, the issue becomes one of performance. Note that each expression involving the composition of ADT functions may be invoked several times within the same query. It is typically not desirable to require that each specific expression invocation be independently optimized at run-time.

Obviously, therefore, this extension has minimal benefit unless the system can provide *compile time* statistics about the values that will be passed as arguments. Using our example of Section 2, statistics need to be maintained on the implementation and sparseness of the Humidity matrices. If this information is made available to the optimization rules, they can be applied in a cost-based manner. Further, instead of performing run-time decisions about the specific implementations used for each function, compile-time decisions can be made. However, this added enhancement of providing compile time statistics requires a relatively major change to the database system.

### 5.3.4 Maintaining Compile Time Statistics

Note that in our example, there need to be some common statistics maintained for all the Humidity matrices. For example, that all of them are stored in row-major fashion, are dense, and have specific ranges along each dimension. Of course, the schema of the matrices (columns are Temperature and Pressure, and the matrix entries are of type double) also is maintained. On the other hand, if it is not possible to maintain common statistics (for instance, some of the matrices are stored row-major, while others are column-major), then this inhibits the ability to perform compile-time optimization.

At the level of the system code, there is one major change that is necessitated by the desire to maintain compile time meta-information on the various objects. At an intuitive level, the type name Matrix is no longer sufficient information to uniquely define the format of the data. For reasons of type checking, the schema of the matrix must also be specified. Further, for optimization purposes, the statistics must also be specified. In an extensible system, this implies that *at every stage of the code where the type of a field or expression is stored or passed as an argument, the meta-information must also accompany the type information.* This involves a major code change, over and above the work involved in collecting statistics on the various complex data types. Note that not only must the statistics be collected, they must also be maintained, thereby necessitating a catalog structure for each new data type.

Further, in order to be totally extensible, the meta-information on a specific data type like matrix must only be interpreted by routines provided as part of that type! This is reasonable, because it is only the optimization techniques for matrix expressions that will need to use the matrix meta-information.

If this major change is performed, then cost-based optimization can be performed using transformation rules. There are still some limitations though. One problem is that there may be a large space of possible equivalent queries, and there is really no guarantee that good alternatives are tried. In fact, rule-based optimization is still a topic of ongoing research for relational queries, so it is not clear that this is necessarily a practical solution. Another limitation is that the same optimization mechanism is being imposed on all expressions involving all data types. However, it might be desirable to optimize sequence expressions more exhaustively than the matrix expressions within the same query. Such control mechanisms for rule-based optimizers may be difficult to specify. Finally, rules can be used to specify a certain category of "equivalence" optimizations. However, something simple like streaming intermediate results between two functions, does not fall into this category. In the expression Clip(Transpose(X), A, B, C, D), the transposed matrix X does not need to be materialized in full. On the other hand, the result of the Clip needs to be materialized if it is to be stored as a field within a relational tuple. How does one exploit such optimizations?

The basic feature that is lacking is the recognition of distinct subexpressions that are composed totally of operations of one data type. If such sub-expressions can be recognized, then the entire subexpression can be treated declaratively (allowing for streaming of intermediate results). Other optimizations like common sub-expression recognition can also be performed. This is the subject of the next enhancement.

### 5.3.5 Expression Boundaries

The goal of this enhancement is to recognize that some complex functional expression is really a declarative expression involving a specific data type. Note that recognizing the expression boundaries has benefits independent of the maintenance of compile-time statistics, or indeed the specification of transformation rules. For example, the ability to stream intermediate results can result in a big improvement by itself. One simple way for existing systems to mimic this is to provide a special "EVALUATE" function along with each data type. The EVALUATE function will take a string argument (which represents the complex expression), and other arguments which will serve as parameters to the expression. The entire declarative semantics of the expressions can now be encapsulated within the EVALUATE function. If relations were implemented as just another data type, then the EVALUATE function would take an SQL string as an argument. This would allow relational expressions to be nested within any other kinds of expressions.

This use of the function mechanism provides a pathway to treat expressions over data types as declarative queries. With this as a basic enhancement, it is not very difficult to now permit any suitable language syntax for the string argument that denotes the query expression. For example, when implementing support for sequences, we found it convenient to use the Sequin query language for the sequence query subexpressions [SLR96].

#### 5.3.6 Multiple Top-Level Types

So far, we have still assumed that SQL is the top-level query language, and there are various subexpressions within an SQL query that manipulate objects of other complex data types. Since catalogs are already being maintained for each complex data type (see the previous section), it is not much additional work to allow these catalogs to maintain named objects. For example, this allows named sequences or named matrices to exist along with named relations. Therefore, a query involving named matrices can be expressed in the matrix query language without having to use SQL. Further, an SQL expression can be embedded within a query in some other language.

In order to migrate to this totally uniform model, relations need to be treated as merely another data type (like matrices, sequences and even integers). This is another major change to the system code, because the assumption of the special status of relations is usually implicitly used in many aspects of the system.

## 6 Implementation Issues

In this section, we will mention some of the interesting implementation issues that arose while building the $\mathcal{PREDATOR}$ database system based on the E-ADT paradigm. Several issues have already been discussed in the previous section, including the need to augment compile-time type information with meta-information, and the need to treat relations as just another data type.

### 6.1 Reentrant Components

The various components of the query processing system of each E-ADT (mainly the parsing, optimization and evaluation functions) need to be reentrant. This is because in a complex object model, a specific instance of a data type may be defined in terms of another instance of the same type. For example, a relation might contain a nested relation, or it may contain a nested sequence of relations. Queries over such data might be expressed in SQL, with embedded expressions in any other query language, including SQL again. Consequently, the SQL parser needs to be reentrant, and the reentrancy requirement holds for the query optimization and evaluation components as well.

### 6.2 Ubiquitous Identifiers

Since the interaction between the different types is so loosely-coupled, every E-ADT needs to maintain identifiers on various kinds of objects. At the very least, if the data values are large, then identifiers for them are used instead to store within records. This leads to the need to perform persistent reference counting for these identifiers. Identifiers are also needed for parse structures and query plans. For example, when a subexpression in the matrix query language is planned from within a SQL query, a handle (i.e. identifier) to a query plan is returned. This handle is stored within the SQL evaluation plan. When the SQL query is executed, it invokes the subexpression by asking the matrix E-ADT to evaluate the plan identified by the stored handle. In our current implementation, query parsing, optimization and evaluation happen in one pass, and so the data structures like parse tree and evaluation plan are maintained in memory. The addresses of these structures serve as their identifiers. However, in a full-fledged implementation, we will also have to consider the case where queries are pre-optimized, and this will require that query plans also have persistent identifiers.

### 6.3 Common Utilities and Plug-In Types

One important aspect of the extensible system design is that it should be possible to "plug-in" a new E-ADT or take out an existing E-ADT without adversely affecting the other data types in the system. Indeed, this is true in the $\mathcal{PREDATOR}$ system for all the data types[1]. Therefore, if the relational E-ADT is not compiled into the system, it can still support sequences, and vice-versa.

It is also important that as far as possible, code should be re-used between the different E-ADTs . The role of the Utilities layer of the system is to provide many useful abstractions that can be used by E-ADT builders to quickly and efficiently supply the interfaces that they need. The SHORE storage manager is part of the Utilities layer and provides persistent storage that all E-ADTs can use. Similarly, there are utility functions to handle boolean expressions, manipulate algebra trees, create records, and perform arithmetic. A rule-based optimizer engine could also be supplied as part of this layer. Ideally, most of the code complexity of the system would reside in the utilities layer, while most of the "intelligence" of the system would reside in the E-ADT methods. We still do not have a concrete sense of the extent to which such a separation can be made. However, this is an area that we are continuing to understand better as we develop the system.

## 7 From Structural Heterogeneity to System Heterogeneity

So far, we have used the E-ADT paradigm to explore how a database system can support complex data in an extensible and yet efficient manner. Since we allow an E-ADT to provide a language, optimizer, evaluation engine, catalogs and storage management, it is reasonable to characterize each E-ADT as a mini-system by itself. If we can put many mini-systems together in a loosely coupled manner, how difficult is it to extend this further to support system heterogeneity? In other words, can this paradigm be extended to support heterogeneous database systems. This section tries to address this issue at the logical level, though we have not yet performed any actual implementations of these ideas.

### 7.1 Types Versus Systems

The first point to note is that while it was convenient to introduce the concept of E-ADTs as declarative data types, there is really no compelling reason to tie each declarative language to a single data type. Each language is really supported by a *database engine*, though all the database engines operate within the same address space on the same system. If the database system is enhanced with a shared table of database engines (in addition to the table of E-ADTs ), then there is no basic change in the paradigm proposed. Except that each engine can now support a language manipulating more than one type. More importantly, there may be several engines providing languages for the very same E-ADT !

At this stage, we have not discussed how the data management (catalog and storage management) is modified. One option is for each engine to perform its own data management, but this implies that engine A cannot access the data managed by engine B. One could possibly go a further step and allow each engine to be registered with all the E-ADTs whose data it manages. Therefore, as long as the table of E-ADTs is common to all the engines, it may even be possible for engine A to process the data stored by engine B. So far however, we are assuming that all these engines are part of the same system, sharing the type table, cost model and utilities layer.

### 7.2 Shared Components

To move to a heterogeneous system environment, the shared components need to be examined. All the systems will have to make sure that their type systems agree (at least on the interacting types). Since arguments may be passed from one system to another, there must be a common data format for values of these types. Some common interchange format for data (like XDR) will need to be used. Transaction issues become much more complicated, and if any of the systems being integrated are existing commercial systems, it may not be possible

[1]The sequence E-ADT depends on the existence of the integer E-ADT , but beyond this small exception, there is complete independence among the types.

to modify them to introduce some form of two-phase commit. Consequently, strict transaction semantics may not be possible (we note that this is a difficult problem anyway in heterogeneous systems [Moh95]). Finding a common cost model across multiple database systems is a difficult research area [DKS92], and some approximate cost model must be used. Despite these problems, it should be possible to connect an external database system into the $\mathcal{PREDATOR}$ system without any large-scale modifications to the system code.

### 7.3 Our Contribution

We note that we have not addressed many of the important heterogeneous database system issues like schema and data integration, and query optimization. It can be argued that these are the really difficult issues in heterogeneous database research. However, what our approach does offer is a lower level mechanism for integrating multiple systems. The user of a heterogeneous database might be presented with a single unified abstraction and database schema against which to pose queries. Within the system, the queries may be mapped into subexpressions belonging to the various component systems and expressed in the corresponding languages.

We observe that this approach contrasts with the popular approach of providing heterogeneity by writing "wrappers" [CGMH$^+$94, CCH$^+$95]. In the wrapper-based approach to integrating diverse systems, each system provides a common abstraction of its data, usually using an object-oriented model. This common abstraction is provided by implementing a wrapper around each system. The wrapper understands expressions in a unified query language and can translate these expressions into the language provided by the underlying system. Since all the systems being integrated provide a common wrapper interface, expressing queries across the systems is not difficult. In contrast, the approach suggested by E-ADTs is that systems do not necessarily have to provide uniform wrappers. This intelligence can be provided at the system that interacts with the users. Most certainly, we have not explored this area sufficiently, and we only mention it here because there seems to be a strong connection between heterogeneous types supported by E-ADTs and heterogeneous databases collected into a unified system.

## 8 Related Work

There is much research work related to the E-ADT paradigm. The issues regarding support for ADTs in database systems were first explored in [Sto86]. There has been extensive work on nested data models (especially nested relational models [Hul87]), and there is even a commercial database system, UniData [Uni93], based on such a model. Object-oriented systems like O2 [BDK92] also support a nested model with composite objects. The recently proposed OQL query language [Cat94] for OO databases allows collection types to be nested, and permits nested queries over them. The idea of enhancing ADTs with query language and query processing capabilities seems to be unique to $\mathcal{PREDATOR}$. The loosely-coupled architecture with multiple top-level collection types with different query languages also appears to be novel. The notion of breaking a query into regions, each of which is optimized separately is similar in flavor to the "blackboard" architecture for relational query optimizers proposed in [MDZ93]; however, that work was not directed at supporting operations on complex data types.

As an interesting counterpoint to the approach taken here, there have been efforts that try to find a holistic solution to the problem of querying new data types. Instead of breaking a query into many parts with local query optimization, these approaches try to find a global integrated solution. Obviously, this requires that the entire query be modeled somehow in an integrated framework. AQUA [SLVZ95] is an algebraic framework that has been proposed for this purpose, while CPL/Kleisli [Won94] is a framework based on monoid theory. The primary difficulty is that the integrated solution must be capable of modeling all possible queries and query processing strategies for all possible data types. There is ongoing research on both AQUA and Kleisli to address this and other issues.

## 9 Conclusions

We have proposed a novel database system paradigm that allows several types of complex data to be efficiently supported within a single general-purpose DBMS. The $\mathcal{PREDATOR}$ database system has been built to demonstrate this E-ADT paradigm, and we have discussed several implementation issues. The case is made that current database systems should adopt the E-ADT design concepts, and migration paths have been suggested to help them move in this direction.

## Acknowledgements

Mike Zwilling and Nancy Hall helped greatly with details of SHORE. Illustra Information Technologies, Inc. give us a free version of their database and time-series datablade. Kurt Brown, Mike Carey, David DeWitt, Joe Hellerstein, Navin Kabra, Jignesh Patel, Kristin Tufte, and Scott Vandenberg provided useful discussions on the subject of E-ADTs .

## References

[BDK92] F. Bancilhon, C. Delobel, and P. Kanellakis. *Building an Object-Oriented Database System: The Story of O2*. Morgan-Kaufmann, 1992.

[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.

[CDF+93] 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, 1993.

[CGMH+94] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. The TSIMMIS Project: Integration of Heterogeneous Information Sources. In *"Proceedings of the IPSJ Conference, Tokyo, Japan"*, October 1994.

[CW85] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[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.

[DKS92] W. Du, R. Krishnamurthy, and M.C. Shan. Query Optimization in Heterogeneous Database Management Systems. In *Proceedings of the Eighteenth International Conference on Very Large Databases (VLDB), Vancouver, Canada*, pages 277–291, 1992.

[Hul87] Richard Hull. A Survey of Theoretic Research on Typed Complex Database Objects. In J. Paradeans, editor, *Databases*, pages 193–256. Academic Press, London, 1987.

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

[ISO93] ISO-ANSI. *ISO-ANSI Working Draft: Database Language SQL2 and SQL3*, 1993. X3H2; ISO/IEC JTC1/SC21/WG3.

[LMW96] Leonid Libkin, Rona Machlin, and Limsoon Wong. A Query Language for Multidimensional Arrays: Design, Implementation, and Optimization Techniques. In *Proceedings of ACM SIGMOD '96 International Conference on Management of Data, Montreal, Canada*, 1996.

[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.

[Moh95] C. Mohan. An Overview of the Exotica Research Project on Workflow Management Systems. In *Proceedings of the 6th International Workshop on High Performance Transaction Systems, Asilomar*, September 1995.

[PHH92] Hamid Pirahesh, Joseph Hellerstein, and Waqar Hasan. Extensible/Rule Based Query Rewrite Optimization in Starburst. In *Proceedings of ACM SIGMOD '92 International Conference on Management of Data, San Diego, CA*, 1992.

[SHR+96] P. Seshadri, J. Hellerstein, R. Ramakrishnan, H. Pirahesh, T.Y.C. Leung, D. Srivastava, S. Sudarshan, and P. Stuckey. Cost-Based Optimization for Magic: Algebra and Implementation. In *Proceedings of ACM SIGMOD '96 International Conference on Management of Data, Montreal, Canada*, 1996.

[SLR94] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. Sequence Query Processing. In *Proceedings of ACM SIGMOD '94 International Conference on Management of Data, Minneapolis, MN*, May 1994.

[SLR96] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. The Design and Implementation of a Sequence Database System. In *Proceedings of the Twenty Second International Conference on Very Large Databases (VLDB), Bombay, India*, September 1996.

[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.

[SM95] Michael Stonebraker and Dorothy Moore. *Object-Relational DBMSs: The Next Great Wave.* ISBN 1-55860-397-2. Morgan Kaufmann, 1995.

[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.

[SS94] Sunita Sarawagi and Michael Stonebraker. Efficient Organization of Large Multidimensional Arrays. In *Proceedings of the Tenth IEEE Conference on Data Engineering*, 1994.

[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.

[Uni93] UniData Inc., Denver, CO. *UniSQL User's Guide (Release 2.1)*, 1993.

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

## A C++ Interface for E-ADTs

In this section, we present the C++ interface for E-ADTs .

```
class EADT {
 public:
   // This serves as an index into the global table of ADTs!!
  virtual BasicType TypeId() const;
   // This is the name of the type (used while parsing schemas)
  virtual char* TypeName() const;

   // Maximum "size" of any object of this type, given the specific
   // meta-information. This size is the maximum number of bytes needed
   // to store it in a memory location. This is used to allocate
   // memory locations to hold the results of an expression returning a
   // value of this E-ADT. If the objects are really large, they will not
   // be stored directly. Instead, they will be stored separately, and an
   // ID for them will be stored in the specified memory location.
 // The object size in this case is the size of the id. This decision
 // (id or not) is made by the EADT.
virtual int MaxObjectSize(const void* MetaInfo=NULL) const;
 // size of a particular object of this type: when stored in a tuple
virtual int TypeSize(const char* Object, const void* MetaInfo=NULL) const

 // Extract an object of this Type from ObjectRef into the
 // preallocated result object. Increments the ref count, if only
 // a handle is copied. Returns size of resultant object.
virtual int TypeCopy(const char* ObjectRef, char *ResultObject,
                     const void* MetaInfo=NULL) const;
 // Makes a copy of the ObjectRef and allocates memory for the new copy.
virtual char* TypeCreate(const char* ObjectRef,
                         const void* MetaInfo=NULL) const;
 // Makes a copy that can be stored (in memory representations need to
 // be converted to disk representations).
virtual int TypeStoreCopy(const char* ObjectRef, char *ResultObject,
                          const void* MetaInfo=NULL) const;
 // Destroy an object of the E-ADT.
 // This could involve decrementing ref counts.
virtual void TypeDestroy(char *Object, const void* MetaInfo=NULL) const;
 // Should a type object be stored in its record field, or should it be
 // put at the end of the record with an in-record pointer to it instead?
virtual Boolean StoreInField() const;

// Check for equality between two objects of this E-ADT
virtual Boolean  Equals(const char* Obj1, const char* Obj2,
                        const void* MetaInfo1, const void* MetaInfo2) const;

 //
 //  Textual I/O for the E-ADT
 //
 // Make a readable text string version of the object.
virtual ErrCode WriteText(ostream& OutStream, const char* Object,
                          const void* MetaInfo=NULL) const;
 // Read a text version of the object into memory allocated for it.
virtual ErrCode ReadText(istream& InStream, char*& ObjectRef,
                         const void* MetaInfo=NULL) const;

 //
 // Catalog maintenance for objects of this E-ADT. Note that the
 // meta-information is interpreted solely by this E-ADT.
 //
 // Does the EADT maintain catalogs or not.
virtual Boolean MaintainsCatalogs() const;
 // Find meta-information for a specific named object
virtual Boolean GetMetaInfo(const char* ObjName, void* &MetaInfo) const;
 // Copy the meta-information.
virtual ErrCode CopyMetaInfo(void* MetaInfo, void*& Copy) const;
 // Ask the E-ADT to reclaim the meta-information.
virtual ErrCode ReclaimMetaInfo(void* MetaInfo) const;
 // Associate Name in the catalogs with the meta-information
 // provided in textual form.
virtual ErrCode AddMetaInfo(const char* Name, const XxxBool Named,
                            const char* MetaInfoString);

 //
 //  Support for functions/methods on E_ADTS. Three E-ADT
 //  methods are provided that act as ``meta-functions''
 //
 // Function that provides type checking for allowed methods
 // of the ADT. During type checking, the return parameter is
 // ReturnType. To allow for variable methods (methods with the
 // same name but either with different type signatures), the actual
 // expression list must also be passed. For such cases, it is
 // usually sufficient in PL to pass the list of argument _types_.
 // However, in our case, we must also support function invocations
 // in which the first argument is a constant expression string, and
 // the function type signature depends on what is in the string.
 // Therefore, the actual list of constant argument expressions needs to
 // be passed in. A generic parse structure is returned, and this can be
 // queried for both the return type and the meta-information.
virtual ErrCode FuncTypeCheck(const char* FuncName,
                              ValueExprList *ArgList,
                              int NumArgs,
                              BasicType& ReturnType,
                              GenericParse*& FuncParse) const;

 // Function that optimizes method execution. This takes the
 // generic parse structure and returns a generic plan.
 // Therefore, it is only this E-ADT that actually interprets the
 // generic parse structure.
virtual ErrCode FuncOptimize(const GenericParse* FuncParse,
                             GenericPlan*& FuncPlan) const;

 // Function that executes all allowed methods for this ADT
 // If this function is applied to some object, the object is
 // passed as the first argument.
virtual ErrCode FuncExec(const GenericPlan *FuncPlan,
                         ValueEnv* ValueEnv,
                         void *ReturnValue) const;

 //
 //  Support for cast functions from one E-ADT to another
 //
 // Perform type checking for type cast.
virtual XxxBool CastCheck(const BasicType CastType, const void* MetaInfo) const;
 // Execute type cast.
virtual XxxErrCode CastExec(BasicType CastType, const void* MetaInfo,
    char* OldObject, char* NewObject) const;

 //
 // Support for a declarative language interface
 //
 // This is the name of the language supported (example, ''SQL'')
virtual char* LanguageName() const;
 // Function that parses parameterized expressions in the ADT language.
 // It returns a data structure that contains a parsed representation
 // of the ADT expression. It also specifies the return type of the
 // expression.
virtual ErrCode ExprParse(const char* ExprString,
                          const ArgEnv* ArgEnv,
                          BasicType& ReturnType,
                          GenericParse*& ParseStructure);
 // Function that optimizes a parsed expression in the ADT language.
 // It returns a data structure that contains the optimized execution
 // plan (or plan identifier) for the ADT expression.
virtual ErrCode ExprOptimize(GenericParse* ParseStructure,
                             GenericPlan*& PlanStructure);

 // Function that evaluates an optimized plan.
virtual ErrCode ExprEvaluate(GenericPlan* PlanStructure,
                             const ValueEnv* ArgEnv,
                             void *ReturnValue);
};
```