PROGRAMMING CONSTRUCTS FOR DATABASE SYSTEM IMPLEMENTATION IN EXODUS

by

Joel E. Richardson
Michael J. Carey

Computer Sciences Technical Report #680

January 1987

Programming Constructs for Database System Implementation in EXODUS

Joel E. Richardson
Michael J. Carey

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

(To appear in Proc. of SIGMOD '87)

This research was partially supported by the Defense Advanced Research Projects Agency under contract N00014-85-K-0788, by the National Science Foundation under grant DCR-8402818, by IBM through a Fellowship and a Faculty Development Award, and by a grant from the Microelectronics and Computer Technology Corporation (MCC).

Programming Constructs for Database System Implementation in EXODUS

Joel E. Richardson
Michael J. Carey

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

ABSTRACT

The goal of the EXODUS extensible DBMS project is to enable the rapid development of a wide spectrum of high-performance, application-specific database systems. EXODUS provides certain kernel facilities for use by all applications and a set of tools to aid the database implementor (DBI) in generating new database system software. Some of the DBI's work is supported by EXODUS tools which generate database components from a specification. However, components such as new abstract data types, access methods, and database operations must be explicitly coded by the DBI. This paper analyzes the major programming problems faced by the DBI, describing the collection of programming language constructs that EXODUS provides for simplifying the DBI's task. These constructs have been embedded in the E programming language, an extension of C++ designed specifically for implementing DBMS software.

1. INTRODUCTION

In the 1970's, the relational data model was the focus of much of the research in the database area. At this point, relational database technology is well understood, a large number of relational systems are available in the market place, and they support the majority of business applications relatively well. One of the foremost database problems of the 1980's is how to support classes of applications that are not well served by relational systems. For example, computer-aided design systems, scientific and statistical applications, image and voice applications, and large, data-intensive AI applications all place demands on database systems that exceed the capabilities of relational technology. Such application classes differ from business applications in a variety of ways, including their data modeling needs, the types of operations of interest, and the storage structures and access methods required for their operations to be efficient.

The EXODUS project at the University of Wisconsin [Care85, Care86b] is addressing the problems posed in these emerging applications by providing tools that will enable the rapid implementation of high-performance, application-specific database systems. EXODUS provides a set of kernel facilities for use across all applications, such as a versatile storage manager and a general-purpose type manager. In addition, EXODUS provides a set of tools to help the database implementor (DBI) to develop new database system software. The implementation of some DBMS components is supported by tools which actually generate the components from specifications; for example, tools are provided to generate a query optimizer from a rule-based description of a data model, its operators, and their implementations. Other components, such as new abstract data types (classes), access methods, and database operations, must be explicitly coded by the DBI due to their more widely-varying and highly algorithmic nature.[1] EXODUS attempts to simplify this aspect of the DBI's job by providing a set of high-leverage programming language constructs for the DBI to use in writing the code for these components.

This paper describes the constructs that we have chosen to include in the E programming language, an extension of C++[Stro86] designed specifically for implementing DBMS software. Section 2 describes the EXODUS project in more detail, outlines the major programming problems faced by the DBI, and then briefly discusses the relevant previous work. Sections 3 through 6 each cover one of the major problems facing the DBI, describing the problem and then demonstrating that a judicious choice of programming language constructs can solve the problem. Section 7 presents the "big picture", showing how the E programming language in some sense serves as the heart of the EXODUS approach to database system extensibility. Finally, Section 8 presents a summary and then discusses the research problems that we face in implementing the E language. We should point out that, while some of the constructs presented here are new ideas, a number of them are not. We draw heavily on ideas developed by the programming language research community in the past ten years. The contribution of this paper lies largely in its analysis

This research was partially supported by the Defense Advanced Research Projects Agency under contract N00014-85-K-0788, by the National Science Foundation under grant DCR-8402818, by IBM through a Fellowship and a Faculty Development Award, and by a grant from the Microelectronics and Computer Technology Corporation (MCC).

[1] Actually, we will provide a library of generally useful components (e.g., widely-applicable access methods such as B+ trees and some form of hashing), but the DBI will obviously have to implement those which are needed but not available in the library.

of how these constructs are useful for solving the rather unique problems that arise in the context of implementing and extending database systems.

2. THE EXODUS APPROACH

A number of database research projects have recently begun to address the problem of building database systems to accommodate a wide range of potential applications. Related projects include PROBE at CCA [Daya85, Mano86], POSTGRES at Berkeley [Ston86b, Ston86c], GENESIS at UT-Austin [Bato86], GEMSTONE at Servio Logic [Cope84, Maie86], and STARBURST at IBM Almaden [Schw86]. The EXODUS project is distinguished from all but GENESIS by virtue of being a "database generator" effort rather than an effort to build a single (although extensible) DBMS for use by all applications. The EXODUS and GENESIS efforts differ significantly in terms of the philosophies and technical details of their approaches to solving the DBMS software generation problem; GENESIS takes more of a "building block" plus "pluggable module" approach to the problem, whereas EXODUS has certain powerful fixed components plus a collection of tools for a DBI to use in building the desired system based around these components. We review the EXODUS approach in more detail in this section, after which we outline the resulting programming issues and relate them to previous work.

2.1. An Overview of EXODUS

Figure 1 presents a sketch of the architecture of an application-specific DBMS implemented using EXODUS. As shown in the figure, there are three classes of components: those that are fixed across all EXODUS applications, those implemented by the DBI using the programming constructs described in this paper, and those generated by an EXODUS tool from a DBI-written specification. Here we primarily concern ourselves with the first two component types. The overall architecture and the latter type of components are discussed more fully in [Care86b].

The fixed components of EXODUS are the storage object manager and the type manager. As detailed in [Care86a], the storage object manager provides *storage objects* for storing data and *files* for logically and physically grouping storage objects together. A storage object is an uninterpreted container of bytes which can be as small (e.g., a few bytes) or as large (e.g., hundreds of megabytes) as demanded by an application. Storage object operations include reading a byte range from within an object, writing (i.e., updating) a byte range, appending bytes to the end of an object, inserting a sequence of bytes at a specified point, and deleting a specified range of bytes. For reads, the specified bytes are buffered contiguously in the buffer pool (even if they are distributed over several pages on disk). The storage manager also supports an efficient scheme for versioning storage objects, and any sequence of actions over storage objects and files may be bracketed as a transaction.

In EXODUS, a database system is fully compiled; that is, the system code is written in E, user schema are compiled into E types, and user queries are compiled into E procedures. An interesting aspect of this design is that all EXODUS systems have a homogeneous view of types; the type system available to the DBI is also available at run time. While the front end may restrict or redefine the types available to the user, ultimately, all pieces are compiled by E.

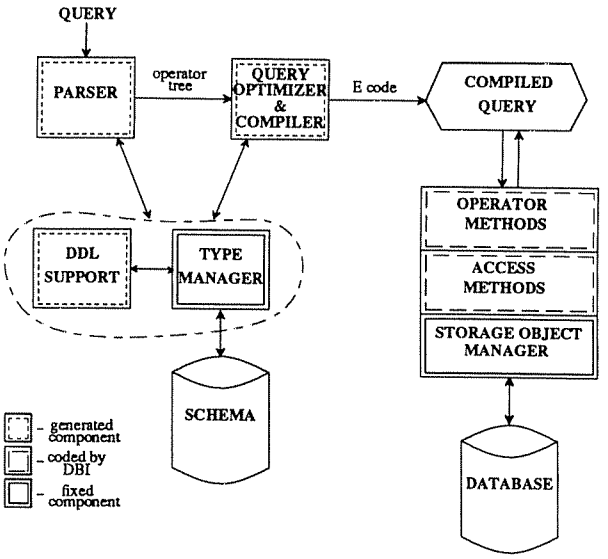

Figure 1: Architecture of a DBMS produced by EXODUS.

The EXODUS type manager[2] [Fran86] maintains information about the modules which make up the database system. In particular, it is responsible for maintaining a dependency graph between the modules, and for keeping the various pieces of the graph up to date. For example, dependencies exist between modules containing type definitions and modules which use those definitions, and between user files and procedures (compiled queries) which access them.

The components in Figure 1 that are implemented by the DBI are the access methods and operator methods. In addition, the DBI must write code for the operations associated with each class (abstract type) that he or she wishes to define. To clarify by using a familiar example, a DBI who wants to implement a relational DBMS for business applications via the EXODUS approach will have to write code for the desired access methods (e.g., B+ trees and dynamic hashing), for the operator methods (e.g., relation scan, indexed selection, nested loops join, merge join, etc.), and for various useful classes (e.g., date and money). A DBI implementing a database management system for an image application will have to implement an analogous set of routines, presumably including various spatial index structures, operations that manipulate collections of images, and an appropriate set of classes. To simplify the DBI's programming tasks, the constructs of the E programming language — the topic of this paper — are provided. We will have more to say about E very shortly.

Finally, the top of the EXODUS architecture consists of a set of components that are generated from DBI specifications. One such component is the query optimizer and compiler. The query optimizer takes an algebraic query tree and transforms it into a tree representing an access plan; the query compiler then translates this plan tree into an E program which is compiled and executed. To produce a new query optimizer, the DBI writes a rule-based description of the data model — its operators, how they can be legally reordered, the available operator implementations and their associated cost functions,

[2]Really, this is a generalized *make* facility; the name "type manager" was kept for historical reasons.

etc. — and this description is then fed to the EXODUS optimizer generator [Grae86]. Similarly, we plan to provide tools to automate the process of producing new DML/DDL components, which are the query parser and DDL support components of Figure 1. (This last idea is similar to the data model compiler notion discussed in [Mary86].) As a prototype for demonstration purposes, we are currently implementing from scratch a GEM [Zani83] front end extended with limited ADT facilities.

2.2. Problems Facing the DBI

In using a typical systems programming language such as C or C++ to write code for ADTs, access methods, and operator methods, implementors of database system software are faced with several problems that complicate the task. These problems can be traced to two sources that are fairly unique to the task of implementing a DBMS:

(1) Virtually all data of interest resides on secondary storage (for persistence and size reasons).

(2) A great deal of data type information is missing while the DBMS is being implemented.

The fact that the data to be manipulated by DBMS code resides on disk affects the DBI's task in several ways. First, disk storage is typeless, while from a schema standpoint the data to be manipulated has type (e.g., it may consist of records with named, typed fields). Thus, the DBI's code must explicitly map typed data onto untyped storage by manually handling offset, length, and type indicator information. Second, the DBI must explicitly move data between main memory and the disk, being sure to fix it and unfix it in the buffer pool at appropriate times, etc. Lastly, when writing code for the access method component of the system, the DBI may have to worry about concurrency control and recovery issues in order to make index operations serializable and recoverable without overly restricting concurrency. It has been estimated that as much as 70% of the code for a typical access method in a relational DBMS such as Commercial INGRES is due to secondary storage considerations such as these [Ston85].

What gives a DBMS its power and general purpose nature is the fact that users are able to create and destroy schema (i.e., types) throughout the life of the system; it also means that the DBI must program in a world of unknown types. For example, in writing index code, the DBI must write the code in such a way that it can work for multiple key types. Users may build indices over fields of any type for which indexing is meaningful; B+ trees may be built over any data type for which a total ordering exists. While the primitive types are known when the index code is being written, it should also work for types that may be defined later (as long as they meet the appropriate ordering constraints) [Ston86a]. Again, the DBI must resort to writing code that basically interprets low-level, type-specific information (including functions to operate on the type). Things are further complicated by the fact that, while some data types are fixed length or at least of bounded length, other types will be variable length in nature. This missing type information problem is even more serious when the DBI has to write code that implements operations at the data model level, such as the select, project, and join routines of a relational DBMS. At this level, the DBI can assume very little indeed about the data types of interest.

If the DBI must deal explicitly with all of these problems when implementing or extending software in a so-called "extensible DBMS", then the system is not as extensible as it really needs to be. We address these problems by providing the DBI with the E programming language, an extension of C++ that lessens or eliminates each of these problems in some way. For example, E extends the class[3] facility of C++, allowing classes to be parameterized and allowing class objects to live on disk. Of course, class definitions may be arbitrarily nested, and there is no restriction on the size of a class object. (Nested type definitions are not supported by ADT-INGRES [Ong84], and we are aware of no DBMS ADT facility that permits types to be arbitrarily large.) E also provides control flow constructs which enable the DBI to write operator methods in a manner allows them to be "plugged together" to form executable queries in an elegant yet efficient manner. Basically, the E language is an attempt to allow the DBI to concentrate on implementing the data model, while the translator handles such "nonessential" details as mapping types onto secondary storage, interacting with the buffer manager and transaction manager. To insure reasonable performance, E allows the DBI to provide declarative "hints" about such issues as storage layout and buffering policies. We describe the major E constructs and how they address these problems in the remaining sections of the paper.

2.3. Related Language Work

Languages to which E and its constructs are related fall into one of two categories: those whose design has directly influenced the design of E, and those whose domain (i.e., persistent storage) is similar. On the design side, there are three languages of primary relevance. As has already been noted, E is an extension of C++ [Stro86] and is, in fact, upward compatible. We found C++'s class mechanism to be an excellent starting point for the kind of constructs that we needed; also, C++ has a growing audience, and we didn't want to introduce yet another (completely new) language. Next, as we shall see, secondary storage access in E is reminiscent of the use of file windows in Pascal [Jens75]. Finally, several of the constructs that we identify as being useful to the DBI, as well as ideas regarding their implementation, have been borrowed from the CLU language [Lisk77] [Atki78]. For example, E has a class generator mechanism inspired by CLU type generators, and we plan to implement them in a CLU-like fashion to make them independently compilable. Finally, we should note that E is related to Trellis/Owl [Scha86] in that both borrow many features from CLU. Work by the Trellis/Owl group also led us to examine the use of iterators in a database environment [OBri86].

There are several classes of programming languages that address problems related to having disk-based data. Database programming languages are one such class, including languages that integrate relations into their design. Examples include Rigel [Rowe79], Pascal/R [Schm77], THESEUS [Shop79], and Plain [Wass79]. This class also includes other languages designed for writing database applications at the conceptual level, such as Galileo [Alba85]. Persistent programming languages form another relevant language class, including languages like PS-Algol [Atki84] and Napier [Atki85].

[3]C++ uses the term "class" instead of "ADT".

The E language is significantly different from all of these languages because it is targeted specifically for implementing DBMS software, as opposed to writing applications on top of an existing DBMS. Database programming languages typically treat relations or entity sets as collections of objects, hiding the details of their storage structures; as far as the programmer is concerned, the underlying DBMS is a given. E, however, is intended for *implementing database systems*, and must therefore give the DBI a great deal of control over storage structures. And, whereas persistent programming languages strive to provide a uniform view of their storage and name spaces, hiding the distinction between persistent and non-persistent data, a DBI must be ever-aware of this distinction and of the placement of objects in files in order to write efficient code.

3. ABSTRACTION VIA CLASSES

One goal which EXODUS shares with several other projects is to allow for the definition of abstract data types (ADTs) in a database system [Alba85] [Daya85] [Ong84] [Osbo86] [Ston86a]. The C++ class mechanism is a good starting point since it provides much of the semantics that we need. It is assumed that the reader is familiar with such concepts as abstract data types, modules, and object-oriented programming, so we present only a brief introduction to C++ classes.

3.1. C++ Classes: A Short Review

Let us assume that the DBI needs a certain abstract type called BOX, such that each BOX object represents a rectangular region in the plane. Certain operations will be needed on boxes, such as a means of initializing a box and an operation returning its area. The internal representation of a box, however, is to remain hidden from users. Figure 2 shows a partial implementation of BOX as a C++ class. In general, a C++ class consists of a sequence of *member* definitions. Usually, classes have both data members (i.e. the representation) and function members (i.e. methods). The keyword **public** separates private from public members; only those member declarations following **public** are visible to users of the class. In Figure 2, a box is represented by two points[4] ul and lr, marking the upper-left and lower-right-hand corners, respectively, of the box. Because both data members precede **public**, they

```
class BOX {
  // representation == upper-left and lower-right corners
  struct { float x, y; } ul, lr;

public:

  BOX( float x1, float y1, float x2, float y2 ) {
       ul.x = x1;  ul.y = y1;
       lr.x = x2;  lr.y = y2;
  }

  float area( ) {
       return ((lr.x - ul.x) * (ul.y - lr.y));
  }

  ...
};
```

Figure 2: A Simple Class Definition

(and thus, the representation of BOX) are hidden.

Class BOX has two public member functions, BOX( ) and area( ). The real-valued function area( ) applied to a particular box computes the area of that box by multiplying the x-distance between the corner points times their y-distance. In general, one invokes an operation on a class object using dot notation: *object.operation(arguments)*. For example, assuming we have a BOX object named mybox, we might say:

```
if( mybox.area( ) > 10)
  // do something
else
  // do something else
```

By definition, a reference to the object is passed implicitly to the operation, and every class operation is written with this in mind. Thus, the above invocation passes to area( ) a reference to mybox; within the area( ) function, references to ul and lr are equivalent to mybox.ul and mybox.lr (for this invocation).

The member function named BOX is called a *constructor* for class BOX; in general, a member function whose name is the same as its class is a constructor for that class.[5] Constructors are used to ensure proper initialization of class objects. In this example, whenever a BOX object comes into scope or is allocated on the heap, the function BOX( ) is called on that object; since this function takes four real arguments, the compiler will complain if these values are not provided in the declaration of a box. For example:

```
BOX mybox(0, 1, 1, 0);  // unit square at origin
BOX my_other_box;  // error: no initializing arguments
```

3.2. Database Classes

One of the goals of E is to provide the DBI with the following model of typed secondary storage: instead of explicitly issuing calls to the buffer manager and explicitly casting structure onto a page of bytes, the DBI should simply be able to dereference a pointer to a named field of an object, while buffering and type casting is handled "under the covers." A similar idea is found in Pascal's file window mechanism [Jens75]; the programmer may create files of any type, and then read and write objects in the file through a private buffer (the file window). The window holds exactly one object, and each *get* or *put* operation moves a whole object. The two major differences from Pascal's approach are (1) in E, typed files are type-*safe*[6], and (2) E buffers only the portion of the object that is actually needed. Doing a reasonable job of transparent buffering is an interesting, non-trivial research problem that will be discussed further at the end of the paper. In this section, we introduce *dbclasses*, an extension of the basic class mechanism designed for secondary storage; the relationship of typed files to dbclasses is covered in Section 4.

While C++'s class mechanism provides much of the power needed to implement abstract types, certain implementation issues make it impractical to store regular class objects in a (persistent) database. The basic problem involves pointers

[4]Note that we could have implemented POINT as an ADT in its own right; then the representation of BOX would have been *POINT ul, lr;* .

[5]C++ also has *destructors* which are called when the object goes out of scope. Destructors are useful for classes that allocate storage, e.g. Linked_List.

[6]For example, it is possible in Pascal to create a file of arrays with one program, and then read it as a file of records with another.

and buffering: regular C pointers do not carry enough state to allow for the buffering of pieces of an object. Since class functions expect the entire object to be available in memory, we would be forced to buffer whole objects. Such a requirement would severely limit the maximum object size that the system could reasonably handle. One approach would be to reimplement all pointers as object descriptors of the kind needed for this "incremental buffering." However, not only would *all* pointer accesses (even to regular memory objects) be slowed somewhat, but E would lose its upward compatibility with C++. Another problem is that regular C pointers are simply too dangerous to allow them to point into the buffers, and the compiler would find it difficult (or impossible) to know when buffer space may be unfixed safely. We considered all these implications to be unacceptable, and therefore chose another approach.

Instead of trying to provide a completely uniform view of storage, E divides the world of classes into two kinds: those whose objects can live *only* in volatile memory and those whose objects may live *either* in memory or on disk. The former are simply normal C++ classes; the latter we call dbclasses, to suggest that these are the classes that make up the database. Our philosophy is that a DBI can (and should) know which types are used strictly in memory (for example, a hash table in a hash join algorithm), and which are used in conjunction with persistent data (for example, employees). Those types used only in memory should not suffer any performance penalties. On the other hand, we do not wish to divide the world so completely that dbclass objects can live *only* on disk; for this reason, E also allows dbclass objects to be declared locally and to be allocated on the heap.[7]

Syntactically, one declares a dbclass exactly like a regular class (except for the keyword **dbclass**) with the restriction that all data members of a dbclass must themselves be dbclass objects. In the BOX example, if we wish to store boxes in the database, we would first change the declaration to read:

dbclass BOX { ... };

One minor problem is that the data members of BOX are of type float, which is not a dbclass. Therefore, E provides a collection of built-in dbclasses which are duals of the fundamental types: dbshort, dbint, dblong, dbfloat, dbdouble, dbchar, and dbvoid[8]. All the normal arithmetic operations are defined for these types. More importantly, assignment and coercion are defined between these dbclasses and their duals and are implemented by copying values. Thus, we may reimplement BOX as follows[9]:

```
dbclass BOX {
  dbstruct { dbfloat x, y; } ul, lr;
public:
  // as before...
};
```

Finally, for reasons already stated, we cannot allow unrestricted pointer usage with dbclass objects. While it is legal to take the address of a dbclass object, to assign the address to a pointer, and to dereference a pointer, it is illegal in E to perform arithmetic on dbclass pointers. (This reduces the risk of inadvertently walking all over the buffer pool.) In order to prevent the (always erroneous) circumvention of these rules, it is illegal to convert a dbclass pointer into a normal pointer. In these respects, dbclass pointers in E look similar to pointers in Pascal, except that they may also point to local stack objects.

We have said that buffering of dbclass objects occurs under the covers, and that only the pieces actually needed will be read in. We have also said that doing a reasonable job of this remains as significant future research. Let us illustrate the problem with an example. Figure 3 shows a partial definition of an abstract type IMAGE, where each image is a 4K-by-4K

```
dbclass IMAGE {
  PIXEL     map [4096][4096];

public:

  void average( ) {
      int i, j;

      for( i = 1; i < 4095; i++ )
        for( j = 1; j < 4095; j++ )
            map[i,j].pix_avg(map[i,j-1], map[i-1,j],
                             map[i,j+1], map[i+1,j]);
  }

  ...
}; // IMAGE
```

Figure 3: A More Interesting Example

map of pixels, and where PIXEL is another dbclass. The IMAGE member function average( ) presents an interesting problem. For each (i,j) generated in the loop, this function touches pixel (i,j) plus its 4 neighboring pixels. Storing the image in either row- or column-major order is clearly suboptimal for this problem. We plan to examine ways of allowing the designer of a dbclass to specify *layout hints* for cases in which the performance loss would be significant. Such hints will be associated with the definition of the class. For this example, one might specify that images should be stored in rectangular blocks.

Another kind of hint concerns buffering. It is well known that the overall performance of a system can be improved significantly if one chooses a buffering policy appropriate for the particular access pattern [Ston81] [Khos84] [Chou85a]. Since different operations on a type are likely to have different access patterns, we plan to allow the DBI to associate buffering hints with dbclass methods. For the average( ) example, LRU with 3 buffer pages would be sufficient (assuming block layout), while an operation which touches each pixel once (and in order) might specify MRU with 1 page.

In the next section, we round out our discussion of persistent data, introducing files into the language, showing how dbclass objects are stored in files, and how files are made part of the persistent database.

4. FILES & PERSISTENCE IN E

Unlike the general-persistence and database programming languages mentioned in Section 2.3, E is intended for DBMS implementation. Thus, E takes an approach to persistence

[7] An interesting implication of this design is that, should the DBI use dbclasses exclusively, then storage once again appears uniform!

[8] C++ includes the type "void" (in part) for those functions that return no values. We include dbvoid mostly for completeness.

[9] C++ defines **struct** and **union** as classes; thus E also defines **dbstruct** and **dbunion** as dbclasses.

which we feel is more appropriate for implementing the lower levels of a database system.

4.1. File as Class

The concept of a file as a means of storing data on disk is universal. Unfortunately, the traditional difficulty with files is that, in general, they are operating system objects and are not part of the language. This means that files are untyped and do not obey scope rules or have names known to the compiler. One way of introducing files into a language is to make file a type constructor (like struct or array); while the declaration syntax would be relatively simple, one must then introduce a host of new language constructs to express operations over files. Fortunately, new syntax is not really needed.

In E, files are introduced as another built-in class where, in accordance with the underlying storage system, a file is an unordered collection of storage objects. A given file may contain only one type of object, and that type must be a dbclass. Operations on files are defined simply as member functions of class file, and access to objects in a file takes the form of operations on (dbclass) pointers. For example, there is an operation that returns a pointer to the first object in a file, one which creates an object in a file, etc.

To clarify this discussion, the function in Figure 4 loads a file with the integers 1..200, then deletes all the odd ones. In this example, f is a file object local to the function; that is, a physical file is created when we enter the function, and destroyed when we exit. (We shall have more to say about file lifetimes shortly.)

While the syntax[10] for declaring the file's type looks a little strange, the semantics are clear enough: f is a file (initially empty) containing a collection of objects of type dbint. The function *newobj* creates a new dbint object in file f and returns a pointer to it. Similarly, *getnext*, given the address of an object A, returns a pointer to the next object in the file; if A is the last object, then the operation returns the null pointer value 0. Finally, *deleteobj* destroys the object in the file having the given address.

```
void simple( ) {
   fileof[< dbint >]  f;
   dbint  *p;

   for( int i = 1; i <= 200; i++ ){
        p = f.newobj( );
        *p = i;
   }

   for( p = f.getfirst( ); p != 0; p = f.getnext( p ) )
        if( (*p) & 1 )
           f.deleteobj( p );
}
```

Figure 4: An Example of File Usage

Two other operations of interest are *setroot* and *getroot*, which are particularly useful in writing index code (or, in general, for dealing with files that have a user-defined internal structure). Every file may have zero or one distinguished (root) object; the identity of this object is maintained by the storage system. If p points to some object in file f, then *f.setroot(p)* makes that object the (one and only) root of the file. The call *f.getroot( )* returns the address of f's root object if there is one, and null (0) if not. A file initially has no root, and if *setroot* is called with a null address, then it will once again have no root.

4.2. File Lifetimes

The implementation of a file object has two parts: a physical storage system file and its representative data structure in memory. A physical file is created by issuing a CreateFile call to the storage manager which returns the file's id (fid). Thus, E compiles a declaration for a local file into a CreateFile call at scope entry. The fid returned is then saved in the local representative data structure. A corresponding DestroyFile is issued at scope exit. Handling **static** files is also easy; the create (destroy) call is simply issued at the beginning (end) of the program run.

Clearly, a database is of little use if files live no longer than the program run! E addresses the problem with a new C++ storage class specifier, **persistent**:

```
persistent fileof[< some_dbclass >] f;
```

This specifier is only applicable to file objects[11] and, in this example, means that f is associated with the same physical file for all runs of the program. Furthermore, the contents of f is preserved between runs of the program. We implement this storage class by having the E compiler issue the CreateFile call at *compile time*; the fid it receives is then compiled into the object code as a constant.

The lifetime of a **persistent** file is not infinite, however. In fact, a persistent file lives only as long as its associated object module, i.e. if the source code is recompiled, then the old file is deleted, and a new one takes its place. Some applications (e.g. destroy relation) do not want to redeclare a file, but merely wish it deleted. In this case, one may invoke the 'erm' utility which, given an E source file, removes the object module and any persistent files associated with it. For example, suppose the file employee.e contains the declaration of two emp relations:

```
#include "EMP_TYPE.h"
persistent fileof[< EMP_TYPE >] BOSTON_emps;
persistent fileof[< EMP_TYPE >] SEATTLE_emps;
```

Then the command

```
% erm employee.e
```

destroys the associated object module, employee.o, and the physical files representing BOSTON_emps and SEATTLE_emps.

5. WRITING GENERIC ACCESS METHODS

As described earlier, we expect the implementation of new access methods to be one of the more difficult tasks involved in implementing a DBMS for a new application class. One of the problems that we mentioned was that the DBI's code should be written so as to enable it to work with new key types that might be defined long after the index code is written. This problem worsens when we consider that index lookup routines typically return object references. For example, if we

[10] In fact, the class "fileof" is a particular example of a generator class (see Section 5).

[11] Actually, one may also apply **persistent** to classes whose data members are (recursively) all files.

have an index on EMP name, then a lookup on "Smith" should return a reference to Smith's record (if it is found). In E, all references are typed, so the value returned by the lookup routine is of type (EMP *). Just as the DBI cannot anticipate the key type, the return type is also unknown when the code is written.

5.1. Generator Classes

An elegant solution to this problem is the notion of a *generator class*, introduced in CLU [Lisk77] as parameterized clusters. A generator is a class which is defined in terms of one or more type parameters and, as such, defines a family of related types. Both dbclasses and regular classes may be defined as generators, and a type parameter may specify either dbclass or class. To define a B+tree node, then, we would make it a dbclass generator (since these nodes are to be stored in a file), and the two type parameters would also be dbclasses. Thus, the heading of the definition for the class generator might look like:

```
dbclass BTnode[< dbclass { } keytype; dbclass { } enttype >]
{ ... };
```

The doubled bracket symbols "[<" and ">]" are new tokens in E and are required to disambiguate the grammar.

Within a generator class definition, code can be written (and compiled [Atki78]) without specific knowledge of what actual types might be supplied as arguments, although certain constraints may be declared (as we shall see in Section 7). Within the BTnode class, keytype and enttype are used just like any other type names. The advantage of this approach is that the code implementing B+ trees can be written and compiled only once, and yet the DBI does not have to pass around and manipulate low-level type information.

Given a generator class definition, one creates new classes by providing actual parameters in the context of a declaration; this process is called *instantiating* a class from the generator. In this example, since keytype is specified as dbclass, then one may instantiate BTnodes whose keys are, for example, dbint or BOX; however, it would not be possible to ask for a Btree with keys of type float. Similar comments apply to enttype. Instantiating a BTnode with a given pair of type parameters can be accomplished, as in CLU [Atki78], through a form of linking (whereby a type parameter leads to an "under the covers" implementation that is very similar to the sort of tricks that the DBI would otherwise have to code explicitly in a language like C). This leads to a fairly efficient implementation as there is only one copy of the "generic" code for all specific instances. Each new instantiated type adds only a small linkage table. (Again, see [Atki78] for more details.)

There are several more issues to address, however, before we can be completely satisfied with our solution to the BTnode problem. The first issue is that we may need different comparison routines for different key types. Consider building a B+ tree index over a file of BOXes. The usual comparison operators for keys in a B+ tree are "less" and "equal", but in what sense is one BOX less than another? In area? Circumference? Distance from the origin? Ideally, we would like to be able to choose different ordering criteria for different key types; even for different trees built over the same key type, we might like to be able to use different ordering functions [Ston86a].

The solution in E is that a generator class may be instantiated with function parameters as well as type parameters. These procedures effectively become part of the type's definition. It is important to note the distinction between this approach and that of passing the comparison operator as a formal parameter to each BTnode procedure: while the latter is certainly possible in E, some agent external to the type system (e.g., a catalog manager) would then have to keep track of the association between specific B+ tree instances and their comparison operators. If the type is actually *instantiated* with a specific comparison operator, then the type system can remember this association.

Figure 5 gives a portion of the definition of the BTnode class. The first few lines specify that BTnode is a generator class that takes two dbclass parameters and two procedure parameters. As we saw previously, enttype and keytype may be any dbclasses; the parameters "operator==" and "operator<" may be any two integer valued functions, provided that they both take two keytype values as parameters.[12] (Hopefully, the semantics of these functions is somehow related to the concepts of "equal" and "less", although of course the compiler cannot enforce this.) With this specification, we are able to build a B+ tree, for example, over BOX area, circumference, or whatever we wish by instantiating BTnode with BOX as the key_type and with the appropriate comparison routines, e.g.

```
BTnode[<
  BOX, BOX,
  int area_less(BOX, BOX),
  int area_eq(BOX, BOX)
>] a_node;
```

The representation of BTnode also illustrates the use of auxiliary type definitions. The type "kpp" (for key/pointer pair) defines a structure suitable for building secondary indices. Each key has a list of one or more associated pointers. A new generator class *varray* implements the list. A varray is a variable length array which can grow or shrink at arbitrary points via the insertion or deletion of array elements.[13] In order to handle both internal and leaf nodes, the pointer type in this example is a union of BTnode pointers and enttype pointers.

Like fileof and the fundamental dbclasses, varray is a built-in dbclass (i.e. one implemented by the compiler) whose operations appear as member functions of the class. These operations include inserting and deleting elements, querying the current array size, and of course, array indexing. For example, given a varray A, the operation *A.newslot(3, 10)* inserts 3 new slots in A such that the first new slot has index 10; other elements are "pushed down" to make room. An inverse operation deletes slots from the array. The call *A.length()* returns the current number of elements in A. We should point out here that the inclusion of varrays in E is not just an ad hoc device that is convenient for writing B+ tree code. Varrays provide a way for the DBI to utilize the full power and flexibility of the storage object abstraction provided by the underlying storage system [Care86a].

As in the case of large, relatively simple classes, there are certain physical details that the DBI may wish to influence for performance reasons. As mentioned before, one such issue is buffering, and the solution again is that the DBI may include

[12] In C++, one may overload operator symbols with user-defined functions. Thus, if v1 and v2 are both keytype values, then "v1 == v2" is equivalent to "operator==(v1, v2)".

[13] Equally descriptive terms might be "indexable list" or "sequence".

```
dbclass BTnode[<  // a Btree is a file of these
  dbclass { } keytype;
  dbclass { } enttype;
  int operator<(keytype, keytype);
  int operator==(keytype, keytype);
>] {
  // Each key has a varray of pointers
  typedef dbstruct {
    keytype   key;
    varray[<
      dbunion{ BTnode* child; enttype* entity; }
    >] ptrs;
  } kpp ;

  // The data part of each node:
  dbint height;  // leaf level == 0
  varray[< kpp >]  slots;  // the key/pointer pairs

  // Simple binary search of node
  status searchnode( keytype key, int* index ) {
        int min = 0;
        int max = slots.length() - 1;
        int mid;

        // NOTE: "<" and "==" are class parameters
        while( min <= max ){
          mid = (min + max) >> 1;
          if( slots[mid] < key )
                { min = mid + 1; }
          else if( slots[mid] == key )
                { *index = mid; return FOUND; }
          else
                { max = mid - 1; }
        } // while
        *index = mid;
        return NOT_FOUND;

  } // BTnode::searchnode

public:
  // Recursive tree search
  status search( keytype key, enttype** ptr ) { ... }

}; // dbclass BTnode
```

Figure 5: A Partial B+ Tree Example.

declarative, buffer-related hints. Also, since index nodes should map one-to-one with disk pages, certain system parameters are available to the DBI in the form of ".h" files. One such parameter, PAGESIZE, gives the maximum number of bytes available in a small (one page) storage object; the DBI can then write insert code, for example, such that node overflow can be detected. Another issue is related to index concurrency control and recovery — performance goals may necessitate the use of non-2PL locking (e.g., [Baye77]) and different recovery techniques than those normally employed by the storage manager. This is an issue that we plan to address in the future via hints [Care86b].

6. ITERATORS FOR ABSTRACTION AND EXECUTION

A common notion in database system architecture is the *scan*. A scan is a control abstraction that is used to structure the execution of queries, providing a state-saving, record-at-a-time interface between the lower level storage system calls and the higher level operators. For example, a simple relational select operation may "open a scan" on the desired relation, examine the tuples one at a time, and pass on those tuples that qualify under the selection predicate. The same operation may also open a scan over an index if an index exists that can produce the tuples more efficiently. A common implementation of scans employs a table to record the states of all currently open scans. For example, the Wisconsin Storage System (WiSS) keeps track of open scans, the file or index associated with each scan, the current position of each scan, etc. [Chou85b]. The procedural interface to the scan routines includes calls to allocate and open a scan, to retrieve the next record, and to close the scan.

While the same implementation is certainly possible in E, we provide a control construct that the DBI is likely to find much more convenient and elegant, the *iterator*[14] [Lisk77]. The E iterator construct is another that we borrow from CLU, although we generalize its usage somewhat. Basically, an iterator is a function that saves its state between calls; each successive call resumes execution where the previous one left off. The intended usage of an iterator is as a control abstraction which produces a sequence of items while hiding the details of how the items are obtained. Clearly, the notion of a database style scan fits this model extremely well [OBri86]. More generally, there is a nice match between the notion of an iterator and the role of each operator method in the execution of a query; we will say more about this correspondence in the next section.

The syntax of an E iterator function is very close to that of a normal E function. The symbols "(<" and ">)" replace the usual parentheses, and the body may contain one or more **yield** statements. A **yield** is similar to a **return** in that it returns a value to the caller and allows the caller to continue execution. However, the iterator invocation is suspended at the point of the yield rather than being lost as with a normal **return**. When the caller next invokes the iterator, it continues from the statement following the most recent **yield**. An iterator terminates when it executes a normal **return**.

Clients may invoke an iterator only within the context of an **iterate** loop, which may be viewed as a generalization of the **for** loop; both have a control variable which is initialized when the loop is entered and "incremented" after each iteration, and both have some stopping condition (which may never be met). Often, an **iterate** loop is associated with only one iterator function. (We shall see a more general example.) When control enters the loop for the first time, the function is invoked, and it executes until it **yield**s a value. That value is assigned to the loop control variable, and the **iterate** continues execution. Each time control returns to the top of the loop, the iterator resumes and **yield**s another value. When the iterator executes a normal **return**, the **iterate** loop terminates. Figure 6 shows a very simple iterator function/iterate loop pair. The iterator yields the first N prime numbers.

The general approach to implementing scans via iterators should now be obvious. All the data kept as table entries in a WiSS-like scan implementation become either local variables or parameters of the iterator function. There is no longer a

[14]The term "iterator" really refers to two cooperating agents, the iterator function itself and the iterator client. We use the term interchangeably to mean the iterator function and the cooperating function/client pair; context should resolve the meaning.

need to allocate a scan explicitly, since this is implicit in the initialization of the iterator. For example, the iterator in Figure 7 shows how one could implement a sequential scan over a file of objects.[15] Each scan_class object is initialized with the file

```
main( ) {
  iterate( int x = primes(< 100 >) )
       printf("%d ", x);
}

int  primes(< int N >) {
  int count=0;
  int pr=2;

  while( count < N ){
       if( is_prime( pr ) ) // assume is_prime() exists
          { yield pr;  count++ }
       pr++;
  }
}
```

Figure 6: A Simple Iterator.

to be scanned as shown in the constructor. The actual scan occurs by invoking the next_tup(< >) iterator on the scan object. Note that this routine yields object references rather than the objects themselves. Also note that this example only shows one possible way of implementing a scan. An alternative would be to pass the file as a parameter directly to the iterator function; the scan_class would then need no data members.

We mentioned above that E generalizes the CLU usage of iterators; this is because CLU's **for...in** loop alone is not powerful enough for certain applications. For example, consider the problem of merging two lists of integers (or files of tuples, as in a merge join), where each list is produced by a different iterator. Using a **for...in** loop, a client can interact with multiple iterators only in a nested fashion, e.g.:

```
for i in iter_i( ) do
  for j in iter_j( ) do
       S;
  end;
end;
```

In this code, S is executed over the cross product of i's and j's. Unfortunately, it is not possible to treat iterators as streams using the **for...in** construct, which means that our list merging example cannot be solved using this construct. Essentially, the problem is that each instance of a CLU iterator loop may be associated with exactly one iterator. In E, therefore, an **iterate** loop may have any number of active iterator functions, and the programmer may "advance" each iterator independently via the **next** statement. If control reaches the top of the loop and all iterators have terminated, the loop terminates as well; if control reaches the bottom of the loop, and no **next** has been executed during that iteration, then **next** is invoked on *all* iterators. (Thus, an **iterate** loop having only one iterator and no **next** statements reduces to a **for...in** loop.)

[15]This example also suggests that one may use a class generator to effectively implement an iterator (or function) generator as found in CLU [Lisk77]. If a generator class has no data members (which is legal), then each instantiation simply produces a new set of member functions and iterators.

```
class scan_class[<
  dbclass { } T;
>] {

  fileof[< T >] *  fileptr;

public:

  // constructor: saves pointer to file
  scan_class( fileof[< T >] *  p ){ fileptr = p; }

  T * next_tup(< >) {
       T * objptr;

       objptr = fileptr->getfirst();
       while( objptr  != 0 ){
          yield objptr;
          objptr  =  fileptr->getnext( objptr );
       }
  } // next_tup
}; // scan_class
```

Figure 7: Sequential Scan Iterator.

Figure 8 shows an iterator that merges two streams of ordered integers. Assume that s1 and s2 in the figure are both integer-returning iterators (and it may be that s1 == s2). When control first enters the loop, each iterator function is initialized and yields its first element (if any). The programmer has explicit control over testing whether a stream is exhausted via the call **empty**( ). Thus, the iterate loop in Figure 8 tests to see if either stream is empty; if so, then it yields the element from the other stream and advances it. If both streams have elements remaining, the smaller is yielded and that stream advanced. When both streams are exhausted the loop terminates automatically.

```
int merge(< >) {
  int   choice;

  iterate( int val1 = s1(< >); int val2 = s2(< >) ) {
       if( empty( val1 ) )
          choice = 2;
       else if( empty( val2 ) )
          choice = 1;
       else if (val1 < val2)
          choice = 1;
       else
          choice = 2;

       switch( choice ) {
          case 1: yield val1 ; next val1 ; break;
          case 2: yield val2 ; next val2 ; break;
       }
  }
} /* merge */
```

Figure 8: A Generalized Iterator Example.

7. PIECING QUERIES TOGETHER — AN EXAMPLE

7.1. Operator Methods and Unknown Types

Let us finally consider the problem of implementing the types and procedures associated with a data model in EXODUS. As we pointed out early in this paper, there seems to be a rather fundamental problem facing the DBI in writing the operator methods. We have said that in EXODUS, user schema definitions are translated into E type definitions. But how can the DBI write operator methods when type information is completely unknown? For example, how can the DBI implement the relational project operator in E without knowing the names of the fields to be projected?

The answer is actually quite simple. In the case of project, the only part of the procedure that cannot be written by the DBI is a series of assignment statements that move data from source tuple attributes to target tuple attributes. The DBI can thus write project( ) such that it expects a function parameter which will be produced by the query compiler. This function takes two typed objects as parameters and contains exactly the sequence of assignment statements needed to accomplish the projection. Similarly, the DBI could write a select operator method that expects a boolean valued function; the query compiler can then produce and pass to select( ) an E function that evaluates the desired selection predicate. Depending on how we choose to implement operator methods, these function parameters may be either part of the instantiation list for a module or simply formal function parameters.

Before giving an example, let us consider the larger question of how a user's query gets compiled and executed. As described in Section 2, a query is first translated into a tree of algebraic operators which is optimized to produce a plan tree. This tree represents an arbitrary composition of operator methods and file scans. Two questions still remaining are:

(1) How can the DBI write the operators such that they are composable?

(2) How does the query tree become an executable query?

To understand the issues surrounding question (1), consider that project might receive its source tuples from a file scan in one query, while in another it might be receiving the results of a join. In general, then, we are interested in writing operators such that they can be easily composed in a pipelined manner. One possible approach is to write all operator methods as iterator generators.[16] In addition to type parameters, we include *iterator parameters* to serve as tuple sources. Thus if project( ) is to receive its tuples from a file or index scan, we instantiate it with the appropriate scan iterator; if it is to be the next stage in a pipeline following a join, we instantiate it instead with the join iterator. Figure 9 shows how the DBI could write project,[17] illustrating each of the concepts in the preceding discussion.

Each project_class will be instantiated with the types of its source and destination tuples, and with a source iterator class. As was mentioned in section 5, it is possible for a generator class to specify constraints on any actual types used to instantiate the class; these constraints are expressed as declarations within the parameter class list. In this example, the class parameter src_iter_type specifies that any actual class provided as an argument must have a (public) iterator member named "next_tup" which takes no arguments and returns src_type pointers. In general, one may specify any number of such constraints. Furthermore, having named the constraints, the generator class is then free to invoke those members.

```
class project_class[<
        dbclass { } src_type;
        dbclass { } dest_type;
        class { src_type* next_tup(< >); } src_iter_type ;
          // means: src_iter_type must have a method, next_tup,
          // taking no arguments and returning src_type pointers
>] {

  // Defines type proj_type, which is a pointer to a function
  // returning nothing, and taking pointers to source and
  // destination type objects
  typedef void (*proj_type)(src_type*, dest_type*);

  // data members
  src_iter_type* srcIter;
  proj_type projFcn;

public:

  // the constructor
  void project_class( src_iter_type* isrc, proj_type p ) {
        srcIter = isrc;
        projFcn = p;
  }

  // This iterator implements the project filter.
  dest_type* next_tup(< >) {
        dest_type* p2 = new dest_type;

        iterate( src_type *p = srcIter->next_tup(< >) ) {
          projFcn( p, p2 );
          yield p2;
        }
        destroy p2;
  }
}; // project_class
```

Figure 9: Project Iterator Example.

Question (2) is now fairly easy to answer. The query compiler can produce an executable query through a rather straightforward translation of the query tree into (in general) some type definitions and E procedures. As an example, let us consider a simple project query, assuming that the type EMP has been defined as follows:

```
dbstruct EMP {
  dbchar name[50];
  dbint  age;
  dbfloat sal;
};
```

Assume also that "emp" is an existing file of EMP objects, and that the user types in the following QUEL query:

```
range of e is EMP;
retrieve into proj_emp ( sal = e.sal, age = e.age );
```

[16] I.e., as generator classes whose major purpose is to provide an iterator.

[17] We ignore duplicate elimination for simplicity.

```
extern fileof[< EMP >] emp;

// The class def of the result type.
dbstruct PROJ_EMP{
   dbint   sal;
   dbchar  name[50];
};

// The result relation
persistent fileof[< PROJ_EMP >] proj_emp;

// The projection function.
void proj( EMP* e,  PROJ_EMP* p ) {
   p->sal = e->sal;
   p->name = e-> name;
}

// Implements the query.
void main( ) {

   typedef scan_class[< EMP >] EMPscan;
   EMPscan  scanObj( &emp );

   proj_class[< EMP, PROJ_EMP, EMPscan >]
        projObj( &scanObj, &proj );

   // Execute the query by iterating over the results
   // of the project. Spool tuples into file proj_emp.
   iterate( PROJ_EMP * p = projObj->next_tup(< >) ) {
        // Create a new (uninitialized) object in then file.
        // Then copy the projected tuple.
        PROJ_EMP *p2  =  proj_emp.newobj( );
        *p2  =  *p;
   }
}
```

Figure 10: Example of Query Compiler Output.

The query optimizer will ultimately produce a plan for this query in the form of a tree of operator methods [Grae86]. This particular tree will have the EMP relation at the bottom, a sequential file scan at the next level, a project above that, and the main program at the top, spooling the projected results into the proj_emp relation. Figure 10 gives a set of type and procedure definitions which collectively implement this plan.

Main( ) first declares a scan object, initialized to scan file emp. It then declares a project object, initialized with the emp scan object and the routine which projects the attributes. The query then iterates over the results of the project, copying each result tuple into the proj_emp relation. Certain details have been glossed over for clarity here. For example, procedure and type names generated by the query compiler probably will have no mnemonic qualities. Also, we should note that the types and procedures will generally be split up into a number of ".h" and ".c" files in order to reduce the granularity of compilation and dependencies.

8. SUMMARY AND FUTURE WORK

In this paper, we have described the programming problems faced by an EXODUS database implementor, introducing the programming constructs provided to simplify the DBI's task. These constructs are provided in the form of the E programming language, an extension of C++ that aids the DBI in dealing with data on secondary storage and with missing type information. The major constructs include a typed file facility with a notion of persistence, the **dbclass** construct permitting the easy definition of nested and/or very large ADTs, generator classes for dealing with missing type information, and iterators for writing operator methods.

Needless to say, while the design of E is basically complete at this point, a significant research and implementation effort still lies ahead. It was said that E will accept hints from the DBI to help it in its job of "under the covers" buffering and in the physical layout of objects; while we have some ideas of useful hints, much research is needed to really solidify the notion. A second issue (related to buffering) concerns the frequency with which E emits calls to the storage manager. In traversing a large matrix, for example, the simplest approach is to issue a read request for each element touched; this would obviously waste many calls, since many elements will be brought in at once if they reside together on disk. We plan to approach this as an optimization problem since it has certain aspects in common with conventional problems such as common subexpression elimination and code hoisting. Finally, we will need to address the problems of concurrency control and recovery. The storage system already provides a transaction facility, and by default, E will provide 2-phase, object-level concurrency control with write-ahead logging. However, more specialized protocols can increase overall performance, particularly where index structures and/or very large objects are concerned.

ACKNOWLEDGEMENTS

We wish to acknowledge all of the members of the EXODUS project for their many helpful discussions and ideas, including David DeWitt, Goetz Graefe, Eugene Shekita, and Daniel Frank. We would also like to thank Larry Rowe and Marvin Solomon for their valuable insights and helpful criticisms.

REFERENCES

[Alba85] Albano, A., Cardelli, L., and Orsini, R., "Galileo: A Strongly-Typed, Interactive Conceptual Language," *ACM Trans. on Database Systems*, 10(2), June 1985.

[Atki78] Atkinson, R., Liskov, B., and Scheifler, R., "Aspects of Implementing CLU," *ACM National Conf. Proc.*, 1978.

[Atki84] Atkinson, M., et. al., "Persistent Object Management System," *Software Practice and Experience*, vol. 14, 1984.

[Atki85] Atkinson, M., and Morrison, M., "Types, Bindings and Parameters in a Persistent Environment," *Proc. of the Appin Workshop on Persistence and Data Types*, Glasgow, August 1985.

[Bato86] Batory, D., et. al., "GENESIS: A Reconfigurable Database Management System," Technical Report, #TR-86-07, University of Texas, Austin, 1986.

[Baye77] Bayer, R., and Schkonlick, M., "Concurrency of Operations on B-Trees," *Acta Informatica*, vol. 9, 1977.

[Care85] Carey, M., and DeWitt, D., "Extensible Database Systems," *Proc. of the Islamorada Workshop on*

Large Scale Knowledge Base and Reasoning Systems, Feb. 1986.

[Care86a] Carey, M., DeWitt, D., Richardson, J., and Shekita, E., "Object and File Management in the EXODUS Extensible Database System," *Proc. of the 12th VLDB Conf.*, Kyoto, Japan, Aug. 1986.

[Care86b] Carey, M., DeWitt, D., Frank, D., Graefe, G., Richardson, J., Shekita, E., and Muralikrishna, M., "The Architecture of the EXODUS Extensible DBMS," *Proc. of the Int'l Workshop on Object-Oriented Database Systems*, Pacific Grove, CA, Sept. 1986.

[Chou85a] Chou, H., and DeWitt, D., "An Evaluation of Buffer Management Strategies for Relational Databases," *Proc. of the 11th VLDB Conf.*, Stockholm, Sweden, Aug. 1985.

[Chou85b] Chou, H., DeWitt, D., Katz, R., and Klug, R., "Design and Implementation of the Wisconsin Storage System," *Software Practice and Experience*, 15(10), Oct. 1985.

[Cope84] Copeland, G., and Maier, D., "Making Smalltalk a Database System," *Proc. ACM-SIGMOD Int'l Conf. on Management of Data*, Boston, MA, 1984.

[Daya85] Dayal, U., and Smith, J., "PROBE: A Knowledge-Oriented Database Management System," *Proc. of the Islamorada Workshop on Large Scale Knowledge Base and Reasoning Systems*, Feb. 1986.

[Fran86] Frank, D., "The Functionality of the EXODUS Type Manager," EXODUS Working Document, University of Wisconsin, Madison.

[Grae86] Graefe, G., and DeWitt, D., "The EXODUS Optimizer Generator," to appear, *Proc. ACM-SIGMOD Int'l Conf. on Management of Data*, San Francisco, 1987.

[Jens75] Jensen, K., and Wirth, N., *Pascal: User Manual and Report*, Springer-Verlag, New York, 1975.

[Kern78] Kernighan, B., and Ritchie, D., *The C Programming Language*, Prentice-Hall, 1978.

[Khos84] Khoshafian, S., Bates, D., and DeWitt, D., "Efficient Support of Statistical Operations," *IEEE Trans. on Software Eng.*, vol. SE-11, no. 10, Oct. 1985.

[Lisk77] Liskov, B., Snyder, A., Atkinson, R., and Schaffert, C., "Abstraction Mechanisms in CLU," *Comm. ACM*, 20(8), Aug. 1977.

[Maie86] Maier, D., et. al., "Development of an Object-Oriented DBMS," *Proc. of the Conf. on Object-Oriented Programming Systems, Languages and Applications*, Portland, Oregon, Sept. 1986.

[Mano86] Manola, F., and Dayal, U., "PDM: An Object-Oriented Data Model," *Proc. Int'l. Workshop on Object-Oriented Database Systems*," Asilomar, California, Sept. 1986.

[Mary86] Maryanski, F., et. al., "The Data Model Compiler: A Tool for Generating Object-Oriented Database Systems," *Proc. Int'l. Workshop on Object-Oriented Database Systems*," Asilomar, California, Sept. 1986.

[OBri86] O'Brien, P., Bullis, B., and Schaffert, C., "Persistent and Shared Objects in Trellis/Owl," *Proc. of the Int'l Workshop on Object-Oriented Database Systems*, Pacific Grove, CA, Sept. 1986.

[Ong84] Ong, J., Fogg, D., and Stonebraker, M., "Implementation of Data Abstraction in the Relational Database System INGRES," *SIGMOD Record* 14(1), March 1984.

[Osbo86] Osborn, Sylvia L., and Heaven, T. E., "The Design of a Relational Database System with Abstract Data Types for Domains," *ACM Transactions on Database Systems*, 11(3), Sept., 1986.

[Rich86] Richardson, J., "The E Reference Manual," EXODUS Working Document, University of Wisconsin, Madison.

[Rowe79] Rowe, L., and Schoens, K., "Data Abstraction, Views, and Updates in RIGEL," *Proc. of the ACM-SIGMOD Int'l Conf. on Management of Data*, 1979.

[Scha86] Schaffert, C., et. al., "An Introduction to Trellis/Owl," *Proc. of the Conf. on Object-Oriented Programming Systems, Languages and Applications*, Portland, Oregon, Sept. 1986.

[Schm77] Schmidt, J.W., "Some High Level Language Constructs for Data of Type Relation," *ACM Trans. on Database Sys.*, 2(3), 1977.

[Schw86] Schwartz, P. et. al., "Extensibility in the Starburst Database System," *Proc. Int'l. Workshop on Object-Oriented Database Systems*," Asilomar, California, Sept. 1986.

[Shop79] Shopiro, J.E., "THESEUS — A Programming Language for Relational Databases," *ACM Trans. on Database Sys.*, 4(4), 1979.

[Ston81] Stonebraker, M., "Operating System Support for Database Management," *Communications of the ACM*, 24(7), July, 1981.

[Ston85] Stonebraker, M., personal communication.

[Ston86a] Stonebraker, M., "Inclusion of New Types in Relational Database Systems," *Proc. Second Int'l Conf. on Database Engineering*, Los Angeles, California, February 1986.

[Ston86b] Stonebraker, M. and Rowe, L., "The Design of POSTGRES," *Proc. of the ACM-SIGMOD Int'l Conf. on Management of Data*, Washington D.C., 1986.

[Ston86c] Stonebraker, M., "Object Management in POSTGRES Using Procedures," *Proc. Int'l. Workshop on Object-Oriented Database Systems*, Asilomar, California, Sept. 1986.

[Stro86] Stroustrup, B., *The C++ Programming Language*, Addison-Wesley, Reading, 1986.

[Wass79] Wasserman, A. "The Data Management Facilities of PLAIN," *Proc. of the ACM-SIGMOD Int'l Conf. on Management of Data*, 1979.

[Zani83] Zaniolo, Carlo, "The Database Language GEM," *Proc. of the ACM-SIGMOD Int'l Conf. on Management of Data*, 1983.