Experiences in DBMS Implementation Using an Object-oriented Persistent Programming Language and a Database Toolkit

Eric N. Hanson[†] **Tina M. Harvey**[‡] **Mark A. Roth**[§]

Abstract

The EXODUS database toolkit, and in particular the E persistent programming language, have been used in two substantial database system implementation efforts by the authors (the Ariel database rule system and the Triton nested relation DBMS). Observed advantages of using a persistent programming language for database system implementation include ease of implementation of special-purpose persistent objects used by the DBMS such as catalogs, data indexes, rule indexes, and nested relational structures. Other advantages of using E (a persistent version of C++) that are independent of the persistence issue are the usefulness of object-oriented programming in developing large software systems, and the utility of the *Collection* abstraction in E. Observed disadvantages include (1) the inability to map the type system of the DBMS to the type system of the underlying programming language while still retaining good performance for ad-hoc queries, and (2) software engineering difficulties due to the distinction in E between database types and main-memory types.

[†]Eric Hanson is with the Artificial Intelligence Technology Office (WL/AAA-1), Air Force Wright Laboratory, Wright-Patterson AFB, OH 45433, and with Wright State University. His work was supported in part by the Air Force Office of Scientific Research under grant number AFOSR-89-0286.

[‡]Tina Harvey's work was done while with the Department of Electrical and Computer Engineering Air Force Institute of Technology. She is currently with the 7th Communications Group/DOWI, The Pentagon, Washington DC 20330.

[§]Mark Roth is with the Department of Electrical and Computer Engineering (AFIT/ENG), Air Force Institute of Technology, Wright-Patterson AFB, OH 45433.

89791-446-5/91/0010/0314

1 Introduction

It is well-known in the database community that implementing DBMS code is difficult and time-consuming. Recent research on persistent programming languages and other tools to support database implementation has given hope that the burden of implementing DBMS code could be substantially reduced. In an attempt to simplify the implementation of two different prototype database systems (the Ariel database rule system [13, 14] and the Triton nested relational database system [15, 27]) we have used the EXODUS database toolkit extensively [7]. In particular, we have made significant use of the E programming language of EXODUS [24], a version of C++ [32] extended with persistent objects. This paper reviews the advantages and disadvantages of using a database toolkit and a persistent programming language (E) that we observed while implementing non-trivial DBMS software.

The next section describes the EXODUS toolkit. Section 3 discusses the impact of persistence on our implementations, as well as issues related to the type systems of the DBMS and the underlying programming language. Section 4 discusses the impact of features of the language and toolkit unrelated to persistence, including the impact of object-oriented programming, collections, and the EXODUS optimizer generator [12]. Section 5 covers issues related to performance, Section 6 briefly reviews related research, and Section 7 summarizes and presents

conclusions. We now turn to the discussion of EXODUS.

2 Overview of EXODUS

EXODUS provides some powerful tools to help automate the generation of application-specific database systems, including a storage manager, the persistent programming language E, a rule-based query optimizer generator and a B+tree class generator. One possible architectural framework for using EXODUS to build a database system is shown in Figure 1.

The EXODUS storage manager is accessed via procedural calls which allow creation and destruction of database files containing sets of objects, and iteration through the contents of files. Objects can be inserted in and deleted from a file at any offset in the file, and explicit clustering of objects on disk can be specified. The storage manager provides procedures for transaction and version management.

The E programming language provided by the EXODUS toolkit is an extension of C++ with persistent objects. Persistence in E is implemented on top of the EXODUS storage manager. E extends C++ types and defines a corresponding **db** type (database type) for each C++ and user defined type. These db types are used to define objects in the database. There are four kinds of db types in E:

- fundamental db types – dbshort, dbint, dblong, dbfloat, dbdouble, dbchar, and dbvoid
- dbclass, dbstruct and dbunion (every sub component of a dbclass must be of a db type)
- pointer to a db type object
- arrays of db type objects

If the `persistent` keyword is used before the declaration of a db type, EXODUS will map the persistent db variables to a permanent storage location.

In E, a *collection* is an unordered set of objects. E also has a feature called *generator classes* which allows defining a generic template for a C++-style class. Customized classes can then be declared using the generator class name plus additional parameters for customization. Collections are supported in E using a built-in generator class called collection, which is invoked by *collection [T]* where T is any db type. A collection must be instantiated for a specific type before it can be used to declare collection objects. EXODUS provides a generator class for B+trees to allow straightforward creation of indexes for different data types. A typical way to create an indexed data set is to create a collection, and then create a B+tree as an index on the objects in that collection.

In E, *iterators* are controlled looping functions that are used to step through a sequence of values such as collections. An iterator is made up of an iterator function and an interate loop. The iterate loop consumes values that the iterator function produces. The iterator function yields values to the iterate loop.

The EXODUS optimizer generator takes as input (1) a set of operators, (2) a set of methods that implement the operators, (3) transformation rules that describe equivalence-preserving transformations of query trees, and (4) implementation rules that describe how to replace an operator with a specific method. Using these rules, a specific optimizer is generated for the particular application. Neither the Ariel nor Triton developers made use of the optimizer generator, so we are not able to comment extensively on it. The developers of Ariel made the decision to implement a custom optimizer rather than use the optimizer generator. One reason for this is that the original optimizer generator required use of C functions and structures, and we were committed to using object-oriented programming in C++. The latest version of the optimizer generator now also handles C++ and E objects, so we would no longer object to using the optimizer generator on these grounds. Another reason we decided against using the optimizer generator was the need to be able to optimize a set of commands in the action of an Ariel rule. We felt it might be difficult to implement special-purpose optimization

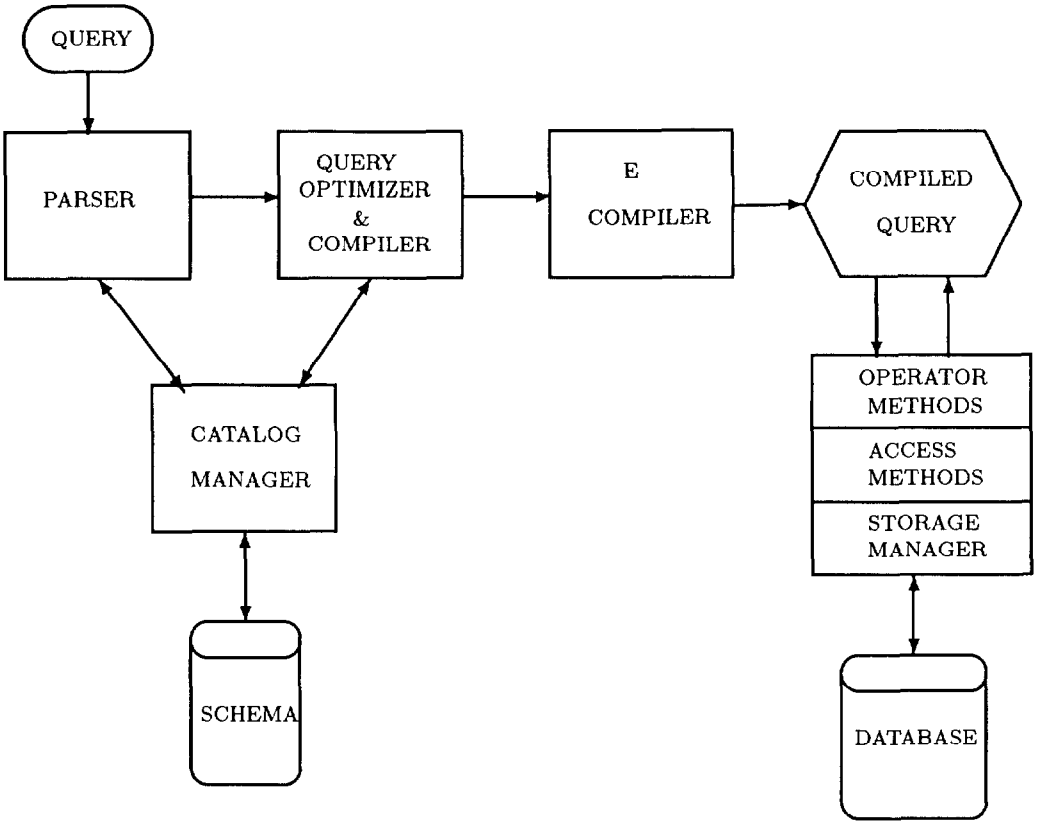

Figure 1: An architecture for a DBMS based on EXODUS

routines for rule actions using the optimizer generator. The Triton optimizer has not been developed, but the intent is to use the optimizer generator. A more thorough discussion of the merits of the optimizer generator awaits more experience using it.

In the next section we comment on the impact of persistence in the programming language based on our experiences with E.

3 The Impact of Persistence

The availability of persistent objects and collections in E has definitely proved worthwhile to use, significantly simplifying implementation of system catalogs, data indexes, rule indexes, and data storage structures. Having persistent objects in the programming language is a convenient interface to the storage manager that frees the programmer from most of the details of mapping data between disk and main-memory data structures. The primary disappointment with persistence in E is related to issues of interaction between the database type system and E language type system, which will be discussed later in Section 4. Also, our experiences reinforce the belief that persistence should be a property of data independent of type. This property was defined as *persistence data type orthogonality*, in the design of PS-algol [4], and is also sometimes called simply *persistence orthogonality*.

3.1 Catalogs

The catalogs in Ariel have been implemented using persistent E objects. The *Relation* catalog consists of a collection of objects of type Relation. The

Relation object has methods on it to set and get information about attributes, presence of indexes, statistics about relation size, number of unique values per attribute and so on. Instance variables of the Relation object include a list of *Attribute* objects to describe the attribute names, data types, and other information about each attribute. Using E provided high-performance access to the catalog data without the need to implement any code for mapping the catalog information into special internal data structures. Typically, relational database systems store catalog information in relations, and map data about a recently-accessed set of relations into a main-memory *catalog cache* (this is the approach used in POSTGRES [31]). Using persistent objects for the catalogs freed us from having to implement a catalog cache.

The only drawback to our approach was that it is not possible to use the query language to query our catalogs. Special-purpose commands have been provided to get information from the catalogs to make up for this, but these commands do not give access in as flexible a manner as a general-purpose query language. Similar advantages to using persistent data for the catalogs have been realized in the Triton project.

3.2 Indexes

Using persistent objects definitely simplifies the implementation of complex permanent storage structures such as data indexes and other special purpose indexes such as rule indexes. As with the catalogs, the primary simplification is that the programmer does not need to be concerned with mapping data between the disk and main-memory data structures. Hence, it becomes essentially no more difficult to implement a persistent index structure such as a B+-tree than it would be to implement a main-memory index structure, except that performance artifacts such as node size and clustering must be addressed more carefully in the persistent implementation.

The amount of code saved by using a persistent language to implement indexes depends on a number of factors including the complexity of the index being implemented, the programming language features available independent of persistence, etc. Research prototypes using PS-algol have shown that code size can be reduced by a factor of 3 in some cases [18]. The Ariel implementors are using persistent objects to implement a fairly complex rule index [14]. We believe that this rule index would have been infeasible to implement without the aid of a persistent programming language.

3.3 Data Storage Structures and Language and Database Type System Issues

Implementation of data storage structures for relations and nested relations was made simpler than it would have been otherwise by the availability of persistent collections provided in E. There are two main approaches to building database storage structures using E, and using them to process database queries (these approaches also apply for other persistent programming languages that do all type checking at compile time and use conventional compiler and linker technology). The first approach, which we call the *compiled approach*, is to compile database type definitions and object (e.g., relation) creation commands into E code, and compile this code into object files using the E compiler. To compile a query, the system generates another file of E code, which is compiled and linked with the object files containing the compiled type definitions. The resulting executable is then run to process the query. This approach has the appealing property that database types are mapped directly into types in the underlying programming language. It can also provide fast query execution since queries are compiled directly into machine code. Unfortunately, it results in a severe performance problem for query compilation if a conventional compiler and linker are used by the persistent programming language, as they are in E (we will discuss performance figures later).

The second approach, which we call the *interpreted* approach, does not use the persistent pro-

gramming language at all for compiling types and queries. Instead, type definitions and queries are interpreted directly by the DBMS. Data is stored in persistent collections of generic storage objects (e.g., byte strings) for which one type (e.g., TupleCollection in Ariel) is defined when the database system itself is compiled. Implementing code to interpret the format of these generic objects stored in the persistent collections of data is left to the DBMS implementor. Execution of queries is done by compiling a query into an execution plan, which is then interpreted. A drawback of this approach is that interpreted queries will run slower than the compiled queries in the first approach, given that the same query plan is used in both approaches. However, response time for any query generated in text form and sent to the DBMS for execution is dramatically better in the second approach (a fraction of a second vs. many seconds). Such long response time for ad-hoc queries is not tolerable.

We chose to implement the interpreted approach in Ariel since we felt the response time of the compiled approach was unacceptable. Relations in Ariel are persistent collections of byte strings. Tuples are mapped onto these byte strings explicitly. A **TupleDescriptor** object describes how tuple fields are arranged in the byte strings in a collection representing a relation. An alternative to this approach would have been to map a relation definition directly into E language constructs, and then compile the resulting E code with the E compiler.

The Ariel type system allows separate commands for defining relation types and constructing instances of relations with those types, similar to the mechanism provided in the EXCESS query language [8]. The following is an example definition of a relation type and a relation in Ariel:

```
define relation type
   emp_type(name=c20, age=int,
            salary=float, dept_no= int)
create relation emp : emp_type
```

The E code that would be generated to represent the same information is:

```
dbclass emp_type
{
   dbchar name[20];
   dbint age;
   dbfloat salary;
   dbint dept_no;
};


dbclass emp_Collection :
 collection[emp_type];
persistent emp_Collection emp;
```

Compiling a source file containing the E code above into an object file takes 3 seconds on a Sun SPARCstation 1 computer. Compiling an E source file containing a trivial one-relation selection query and linking that file with the appropriate object file for the relation and the E library to create an executable file takes more than 15 seconds.

An alternative approach to implementing a persistent language to support DBMS development would be to start with a language supporting incremental compilation of both types and program code, such as Smalltalk [11] or Lisp and CLOS [17]. This would allow direct mapping of database types into language types, and fast compilation of both types and queries, making the "compiled" approach discussed earlier practical. However, this approach would bring with it the larger run-time overhead associated with Smalltalk or Lisp.

3.4 Type System Mapping Example

As another more sophisticated example of a translation from a database type to underlying E types, in Triton, a nested relation definition is mapped directly into a persistent collection of E objects, which in turn have fields which contain collections of objects. Figure 2 shows a nested relation that holds information on VHSIC Hardware Description Language (VHDL) systems[3]. Figure 3 gives the E code representation of the *Systems* relation.

This implementation illustrates the ease with which nested relation types can be mapped into E and also allows compilation of Triton queries

number	*name*	*comps*	*ports*				
		comp#	*name*	*mode*	*type*	*start_bit*	*stop_bit*
43191	COUNTER	15899	STRT	in	BIT	0	0
			STROBE	in	BIT	0	0
			CON	in	BIT_V	0	1
		30018	DATA_BUS	in	BIT_V	0	3
			CNT	out	BIT_V	0	3
14701	FULL_ADDER	41572	X	in	BIT	0	0
			Y	in	BIT	0	0
			CIN	in	BIT	0	0
		81909	Z	out	BIT	0	0
			COUT	out	BIT	0	0

Figure 2: The *Systems* Relation

into machine code. However, it suffers from the performance problem mentioned above for ad hoc queries and thus an interpreted implementation of the Triton type system and query processor is being considered for use by Triton application developers. On the other hand, Since the Triton system is targeted for use by embedded database applications (such as CAD or CASE tools), extended SQL commands [25] will be embedded in a host programming language and compiled as part of the application. Thus, ad hoc queries will not normally be performed.

It is unfortunate that the difficulty in making use of the E compiler to compile database types and queries negates some of the advantages of using a persistent programming language. Essentially the only language feature necessary to support stored data using the "interpreted" approach to DBMS implementation is the persistent collection of byte strings. It would thus be about the same amount of work to implement stored relations using a direct interface to the storage manager (e.g., a C++ class called PersistentCollection with the same methods provided by E collections, including get_first, get_next, get_last and get_prev). This would not require extensions to the C++ compiler on the part of the EXODUS implementors. Moreover, it would have allowed use of a standard C++ programming environment including code inspectors, debuggers, etc., without modification. It would also be possible to build on the PersistentCollection class by creating subclasses. The E language does not allow subclasses to be derived from the built-in collection classes of E, although there appears to be nothing preventing this. Although object persistence in E was not especially useful for storing relations in Ariel, we still believe persistence is worthwhile in a database implementation language because of its usefulness for implementing indexes, catalogs, and any other data structures with object types that cannot change at run time.

Another difficulty we have faced implementing Ariel is a distinction in E between database classes defined using the **dbclass** notation, and normal classes. There is a slight overhead to accessing a **dbclass** object compared to a normal object since **dbclass** objects reside in the storage manager, and a pointer to a dbclass object is a 16-byte record, compared to a 4-byte word for a main-memory pointer. The designers of E wanted to give the implementor a choice whether or not to use classes or dbclasses to have more control over performance. Objects that needed to be persistent would be defined using dbclasses, and the rest of the objects would be defined using classes.

This design choice in E violates the principle of persistence orthogonality, which states that all data objects should be allowed the full range of

```
dbstruct port {
  dbchar name[12];
  dbchar mode[4];
  dbchar type[6];
  dbint  start_bit;
  dbint  stop_bit;
public:
  port (char *, char *, char *, int, int);
  char * get_name();
  void change_name (char *);
  char * get_mode();
  void change_mode (char *);
  char * get_type();
  void change_type (char *);
  int get_start_bit();
  void change_start_bit (int);
  int get_stop_bit();
  void change_stop_bit (int);
  void print (port *);
};
dbstruct comp {
  dbint  comp_num;
public:
  comp (int);
  int get_comp_num();
  void change_comp_num (int);
  void print (comp *);
};

dbstruct system {
  dbchar name[12];
  dbint  number;
  dbclass compRVA:collection[comp];
  compRVA comps;
  dbclass portRVA:collection[port];
  portRVA ports;
public:
  system (char *, int);
  char * get_name();
  void change_name (char *);
  int get_number();
  void change_number (int);
  void print (system *);
};

dbclass systemRVA:collection[system];
persistent systemRVA systems;
```

Figure 3: The E Code Representation of the *Systems* Relation

persistence. Our experience in implementing Ariel reveals that the lack of persistence orthogonality causes software engineering difficulties since a DBMS implementor does *not* know in advance all the types for which he or she would like to create persistent instances. For example, at first we did not intend to store query plan operator objects in the Ariel database, but now we have decided that it would be natural to store compiled queries as persistent plan objects. Accomplishing this will involve a significant modification to our code. In E it is not trivial to simply change all classes to db-classes since all subobjects of a db-object must also be db-objects, which can cause a single change to propagate through many objects. Also, some basic library routines such as string manipulation functions are not the same for db-objects as they are for main-memory objects. This mismatch can result in the need to extensively modify a class definition in order to make it into a dbclass, creating an inordinate workload on the programmer. This extra work inhibits the process of prototyping a complex software system.

In the design of persistent programming languages, we thus feel that it is very important to make no distinction between database types and main-memory types, even if it involves a small sacrifice in performance. It will be a challenge to the language implementors to make access to both kinds of objects as efficient as possible. Investigation of efficient ways to implement a persistent language in this manner is worthy of continued research. We are encouraged by recent developments in the area of transaction-based virtual memory storage systems, including work on Cricket [29], ObjectStore [21] and Bubba [6, 10], which potentially can provide access to persistent objects with no overhead beyond that needed for concurrency control and recovery. In these systems, once a persistent object is in memory, it can be accessed at the speed of a main-memory object.

In summary, the main area where we felt that the persistent programming language features of E were the most useful was in creating special-purpose persistent data structures, such as cata-

logs, data indexes, and rule indexes. If good ad-hoc query response time is required, the persistent features of the language have approximately the same utility for actually storing database data as a direct interface to a storage manager providing transaction support would have. Finally, it is best to make no distinction between database and main-memory types.

4 The Impact of Object-Oriented Programming

In this section we discuss the impact of the C++-derived features of E that support object-oriented programming, as well as E's extensions to C++ including generator classes and collections. The implementations of Ariel and Triton have derived substantial benefits from using the C++ object-oriented programming features of E. In Ariel, we have implemented a terminal monitor, lexer, parser, semantic analyzer, system catalogs, query optimizer and query executor, and system utilities in about 16000 lines of code written using E and the Unix compiler generation tools LEX and YACC [20, 16]. A system of similar, or slightly greater complexity is the terminal monitor, front-end, query executor, and utilities of the university INGRES system [30], which contain approximately 32000 lines of C code. It is hard to make a precise comparison, but it appears that a savings of somewhere between 25 to 50% in the amount of code written can be achieved using object-oriented programming in E (or C++) relative to using C to implement a DBMS. Moreover, object-oriented implementation has provided us with some reusable code which will facilitate extensions as Ariel grows.

Object-oriented programming features including classes, polymorphism, and inheritance are used throughout Ariel. Use of inheritance and polymorphism has been particularly beneficial in the design of the Ariel syntax tree structure generated by the parser, the internal representation of built-in data types, and the query plan operator tree representation. As an example, the class hierarchy for the query plan operators in Ariel is shown in Figure 4.

```
QueryPlanOp
   Scan
      RelationScan
      SequentialScan
      IndexScan
      StoreTemporary
   Join
      NestedLoopJoin
         NestedLoopJoinIndexInner
      SortMergeJoin
   Project
```

Figure 4: Class hierarchy for query plan operators in Ariel.

Methods on these object types include those for accessing result tuples, getting statistics on the expected cost of execution, and constructors and destructors. C++ *virtual* functions are used so that methods are inherited from above unless they are reimplemented in a subclass. Polymorphism proved useful – for example, every object in the class hierarchy shown responds to the *get_next* method. It is not necessary to know the type of the node to get the next tuple from it. A substantial number of instance variables and some methods are inherited by the subclasses of Scan and Join.

There is an inherent benefit from the organization enforced on the code by designing the code using C++ classes. Subjectively, the code seems easier to understand and modify than a C program accomplishing a similar task with which the authors are familiar (e.g., the front-end of university INGRES).

Another object-oriented feature of E is generator classes, a mechanism for creating parameterized types. For example, EXODUS provides a generator class for building B+-trees for different data types [35]. A simplified and shortened version of

```
dbclass BplusTree [
  // keys for entities stored in the tree
  dbstruct key_type {
    void print();   },

  // key comparison function
  int compare(const key_type &,
      const key_type &),
  // entities to be stored in the tree
  dbstruct entity_type {},
]{

  // Definitions of instance variables for
  // BplusTree
    ...

public:
  // Constructors, destructors, functions for
  // building an index, inserting and deleting
  // records etc.
    ...
};
```

Figure 5: Sketch of B+-tree class generator in E.

the definition of this generator class is shown in Figure 5.

Users of this class generator create a new class by specifying parameters for the items in the square brackets (key_type, compare, and entity_type). For example, this piece of code defines an instance of BplusTree for keys of type integer and entities of type Tuple (IntKey is a structure type containing an integer, and IntKeyCompare is a function that takes two IntKeys and compares them):

```
dbclass IntBtreeIndex :
  BplusTree[IntKey, IntKeyCompare, Tuple];
```

The ability to derive classes using a generator can be useful, significantly reducing the amount of code that needs to be written to implement closely related types. However, one difficulty with the E implementation of generator classes is that classes created with a generator class cannot be made subtypes of another type. In Ariel, this made implementing the IndexScan query plan operator more complex than necessary by not allowing use of polymorphism with types derived from BplusTree.

In object-oriented programming, a commonly used, powerful technique is to define a base class B and subclasses of B, say $b_1, b_2, \ldots, b_k$. Each of the subclasses responds to the same set of messages. Then, another class C can be implemented generically, storing one of B's subtypes in a variable of type B. Messages can be sent to the object contained in that variable, and the object will respond correctly, regardless of its type. This generic implementation, which makes use of polymorphism, can save a substantial amount of code in the implementation of C, by letting a single line of the form

```
object->message(parameters...)
```

replace a multi-line SWITCH statement with one CASE for each of B's subclasses b_1–b_k. We believe that an implementation of generator classes should provide a way to create a hierarchy of types, with virtual (inheritable, polymorphic) methods, so that the object-oriented programming technique described above can be used when working with classes derived from a generator class. The experimental parameterized class facility for C++ described in [33] appears to support the desired features, although it is not yet part of the C++ standard.

An alternative to using generator classes that allows object-oriented implementation style is to provide *base classes* from which sub-classes can be derived. This sub-classing approach to genericity does not require any extensions to an object-oriented programming language (no generator class facility is needed). For example, the BplusTree class in EXODUS could have been implemented as a standard E class with virtual functions. A guideline could have been written for deriving subclasses from BplusTree by re-implementing a very small amount of code in each subclass (e.g., the key-comparison function). The vast majority of the complex code for implementing BplusTree would

be inherited by the subclasses. This approach does not completely eliminate the need for a generator class facility (e.g., the key-comparison function would have to be re-implemented for each type), but it provides a workable alternative in many cases, and it does not interfere with object-oriented programming style.

The *collection* generator class available in E proved very useful in implementing data storage structures. The Triton system is built on the nested relational data model, which allows relation-valued attributes in relations. The nested relational data model is mapped very nicely using E collections. Nested relational attributes are represented by using collections of collections. The EXODUS storage manager automatically uses *near* hints to group collections and sub-collections together on disk to increase efficiency. Unfortunately, EXODUS only provides the capability for sequential scanning of collections, making access via a search key slow for large relations. The only way around this shortcoming is to build indexes on every frequently accessed or sufficiently large relation. One way we feel EXODUS could be improved to simplify the programmer's task would be to provide a library of additional types of collections including ordered and hashed collections. This would be somewhat simpler to use than a separate index mechanism.

5 Performance Issues

Obtaining good performance from a DBMS implemented with a persistent programming language is crucial, as it is in any DBMS implementation. We currently do not have a great deal of information on performance of our database implementations based on E – no extensive application benchmarks have been done. However, subjectively we feel that the speed at which individual persistent objects can be accessed using the E language, which has a built-in interface to the EXODUS storage system, is excellent. For example, access to Ariel catalog information stored in a persistent data structure made up of a hash table and linked lists is extremely fast. A performance study done on the EXODUS storage system shows that the overhead for accessing an E persistent object that is already in the buffer pool is about 47 MIPS RISC architecture machine cycles greater than the overhead to access a C++ main-memory object [28]. Our experience suggests that this level of performance is adequate for implementing system catalogs without the need for a special cache. Performance is also good for scanning persistent collections of objects or collections of collections as in the Triton system. The ability to map nested relations directly to nested collections of tuples in the EXODUS storage system allows us to directly benefit from the "nearness" of nested tuples to decrease object access time. A comparison of a relational and nested relation database design for a software engineering CASE tool, using the Triton system on a Sun 3 computer, showed code generation and compilation times in the range of 2 to 3 seconds for relational queries and 3 to 7 seconds for more complex nested relational queries. Query execution times were about 0.1 seconds for the relational queries and about 0.5 seconds for more complex nested relational queries. Given an equivalent *set* of relational queries and a single nested relational query, code generation and compilation times were 70 to 80% faster and query execution times were 10 to 75% faster for the nested relational query [15]. In summary, the speed of object access in E, or any similarly implemented persistent language, does not seem to be an impediment to implementing a DBMS using the language.

Transaction throughput is another performance issue. An important question is whether a DBMS implemented with a persistent language can achieve high transaction rates (e.g., greater than 100 transactions per second). Currently, we have no data on transaction rates using E since a multi-user version of EXODUS is not yet available. However, a DBMS implemented using a persistent programming language will clearly be limited to a transaction rate no greater than that which can be supported by the storage system underlying the language. We see no fundamental reason why such

a storage system cannot be performance-tuned to provide high transaction throughput, using techniques similar to those used in other DBMS implementations such as splitting the log tail, group commit, etc. [22]. Thus, in the long run, the native transaction rate of the persistent language's storage system should not hinder DBMS implementors using the language.

Variables related to throughput which the DBMS implementor can control include the CPU utilization per transaction, and contention for system-wide shared resources such as catalogs and indexes. The majority of CPU cycles utilized by the DBMS will probably be outside the storage system of the persistent language, and it is the DBMS implementor's responsibility to keep it to a minimum to achieve high transaction rates. As in any DBMS implementation, when using a persistent programming language, care must be taken to avoid creating concurrency control bottlenecks around *hot-spots* such as a tuple-count field in the system catalogs and other meta-data. If handled improperly, hot spot bottlenecks can drastically reduce concurrency and hence transaction throughput. For example, having each transaction set a write-lock on tuple count and hold it until the end of the transaction will severly limit throughput. This is exactly what will happen if the tuple count is treated as ordinary data by a concurrency control system based on two-phase locking.

In most DBMS implementations, hot-spots such as tuple-count are handled as special cases. In the case of tuple-count, updates to it are normally not logged, and write locks are held only while physically updating the tuple count, not until the end of the transaction. Given a persistent programming language such as E, it would be difficult or impossible to implement special-case treatment of hot-spots in a DBMS based on the language if the hot-spot data was implemented using persistent language objects. We feel that persistent programming language implementors should give more attention to this issue, perhaps providing an interface to their storage systems designed to handle hot-spots in a way which will allow high transaction rates to be achieved.[¶] If they don't, then DBMS implementors using the persistent language who want to achieve high transaction throughput will have to resort to ad-hoc approaches to storing hot-spot data such as using data files directly to by-pass the persistent programming language.

6 Review of Related Research

In this section we compare and contrast EXODUS to three other extensible systems, GENESIS, DASDBS, and POSTGRES, and discuss the relationship of E with four other persistent programming languages, O++, Vbase, O_2, and Object Design's ObjectStore that are all based on C or C++. Then we discuss other efforts to implement database systems using database toolkits or persistent programming languages.

6.1 Database Toolkits and Extensible Databases

GENESIS [5], like EXODUS, provides a modular approach to extensibility. This approach is supported by providing a library of modules with completely compatible interfaces. GENESIS provides a data definition language to define the schema of relations, as well as a data manipulation language that provides access to the basic objects in the database (which are records, files, and links).

The lowest layer of GENESIS is the file management system, JUPITER. Like the EXODUS storage manager, JUPITER provides buffer and recovery management; unlike EXODUS, JUPITER is extensible in that different buffer and recovery management schemes can be supported by replacing the appropriate module in JUPITER with a new one. JUPITER supports both single-keyed and multi-keyed file structures, such as indexing, B^+-trees, heap structures, and multi-key hash structures.

The Darmstadt Database System (DASDBS) [26] supports extensibility through the use

[¶]The need to support special protocols for handling meta-data was briefly mentioned in [23].

of a kernel storage component that allows flexible, application-specific front ends. The DASDBS kernel provides access (such as reading, insertion, and deletion) to sets of complex objects as opposed to a one-record-at-a-time interface by fetching or storing lists of pages via a variable size buffer. Thus, a single scan of a complex object retrieves all of the values of its sub-objects, which limits the number of disk accesses. This is very similar to the way the EXODUS storage manager works. The kernel provides operations to read, insert, and delete an object. Like the EXODUS storage manager, the DASDBS kernel provides concurrency control capabilities. Instead of using tuple indices, the kernel appends a virtual address attribute to each tuple which can be used in the application layers to build access paths (e.g., B^+-index trees) and provides direct access to the tuple. To enhance performance, the DASDBS kernel attempts to group pages representing a complex object together on disk.

POSTGRES [31] supports extensibility by allowing users to define new data types, operators, built-in functions, and access methods. Like EXODUS, built-in types support both scalar type fields and variable length records. However, unlike EXODUS, POSTGRES supports two interesting built-in types, which are POSTQUEL and procedure types. POSTQUEL types are data manipulation commands, while procedure types are programming language procedures with embedded data manipulation commands. POSTGRES provides these two types to allow users to represent and manipulate complex objects.

6.2 Persistent Programming Languages

Database programming languages are unique in that they should not only support strong typing of objects, but must allow the specification of persistent objects that can last beyond the programs that created them. These two objectives can either be met by providing a single language that does both (as does the E programming language of EXODUS), or providing a separate data definition language and data manipulation language.

O++ [1] is implemented as an extension of C++ with persistent objects, and is thus closely related to E. In addition, O++ also provides additional language statements for defining queries. The main difference between O++ and E is that in O++ there is no distinction between database classes and in-memory classes, but there is a distinction between database pointers and in-memory pointers (there is also a third pointer type called a *dual* pointer that can point to a persistent of volatile object). The O++ approach to persistence is essentially the dual of the E approach. Neither O++ nor E completely separates the issue of persistence from the definition of types.

Vbase [2] and O_2 [19] are database systems that support a separate data definition language and data manipulation language. In both systems, the data manipulation language is based on an extension of C. In Vbase, the data definition language, called TDL, allows strong typing and inheritance. All objects are persistent until they are explicitly deleted, which is good in that persistent and volatile object interaction is not an issue. However, explicit deletion of objects can be tedious.

The Data Definition Language of O_2 is also strongly typed and supports inheritance. Persistent objects are declared from a persistent super object called *tuple*. All objects of type tuple or declared from a subtype of tuple are persistent, and sets of tuple objects can be identified. Methods for types are specified when the type is declared, and types are inherited down the type hierarchy unless they are redefined for a specific subtype. Methods are first order functions and are implemented in C.

The ObjectStore system [21] treats persistent data and persistent data access the same way as conventional virtual memory access. "During ObjectStore application sessions, referenced persistent data is dynamically mapped into the workstation's virtual address space." If persistent data is called for and it not in memory, a "memory fault" occurs and the missing data is retrieved from the database. ObjectStore also supports data caching, concurrency control and restart/recovery. The programmer can create persistent data via several methods:

a variant of the C++ **new** operator which also allows clustering hints, use of a **persistent** keyword, or use of a library call. Any C or C++ type can be made persistent; in addition, ObjectStore includes a collection class, and the Set, Bag, and List subclass of collection, and iterator functions over these classes.

6.3 Use of Database Toolkits and Persistent Languages

Relatively little has been published on experiences using database toolkits to implement a DBMS. Cooper *et al.* [9] discuss three systems implemented using PS-algol [4], a persistent version of Algol with the property of persistence orthogonality. One of the systems covered used PS-algol to implement a DBMS based on an extended functional data model (EFDM) [18]. The benefits of using PS-algol cited in the EFDM implementation were (1) automatic movement of persistent data to/from memory, (2) reduction in misuse of data due to strong typing, (3) usefulness of a universal pointer type, (4) fast access to persistent language objects. Our findings corroborate theirs, particularly (1) and (4) above.

7 Conclusions

The EXODUS system has proven to be a powerful tool for implementing a database system, although it is by no means an antidote for the all the complexities of DBMS implementation. At a minimum, DBMS designers still have to specify a data model, query language parser, catalogs, index and data storage structures, a query optimization strategy (with or without using the optimizer generator), and a query execution strategy.

Using a persistent programming language to implement a DBMS has proven very useful for implementing special-purpose persistent structures such as catalogs, data indexes, and rule indexes, and somewhat less useful for storing the data itself. The problem with using persistent collections in E to store data is due to the fact that one must resort to using persistent collections of generic objects (byte strings) to hold data in order to get adequate response time for ad hoc queries. In systems where ad hoc query capability is not necessary (as in Triton), or where all persistent types can be specified at compile time (e.g., in a computer-aided design database) this is not a major problem. A difficulty we experienced with the E implementation of persistence is the lack of persistence orthogonality in E, which led to software engineering problems in the implementation of Ariel. We assert that it is impossible for the designer of complex software system to know at the outset what data types will need to be persistent. Research on virtual-memory based storage systems (e.g., Cricket) may eliminate the incentive to distinguish between database and main-memory types. We highly encourage this and other research on ways to improve the speed of storage systems for persistent languages.

Language features of E independent of persistence, especially object-oriented programming capability, clearly helped simplify our systems. Ariel shows a significant reduction in code size relative to parts of university INGRES with comparable complexity. E generator classes were useful, but the inability to use polymorphism and inheritance with generated classes is a problem. Generator class facilities in an object-oriented language need to allow use of object-oriented style with generated classes. We were not able to adequately evaluate the usefulness of the optimizer generator. A useful evaluation of the optimizer generator would be to implement an optimizer with the generator and also code the optimizer by hand, and compare the resulting optimizers.

In terms of performance, we are pleased with the speed of access to persistent objects in E. Performance seems adequate for catalogs, indexes, and data storage structures. Any improvements in speed of persistent object access would, however, be welcome. The speed of the underlying storage system does not appear to stand in the way of achieving high transaction throughput. However, we are concerned about having the persistent language storage system handle meta-data such as

catalogs and indexes. Since the storage system will use a standard two-phase locking, write-ahead log strategy for all data, it almost certainly will cause a transaction throughput bottleneck around the system catalogs. Database toolkit designers need to provide some sort of support for meta-data to avoid the creation of a transaction bottleneck.

Using EXODUS has been a worthwhile experience for us. We encourage continued research on ways to improve database toolkits and persistent programming languages so that the job of DBMS implementors who follow in our footsteps might be simpler.

References

[1] R. Agrawal and N. H. Gehani. Rationale for the design of persistence and query processing facilities in the database programming language, O++. In Richard Hull, Ron Morrison, and David Stemple, editors, *Proceedings of the Second International Workshop on Database Programming Languages*, pages 25–40, Glenede Beach, Oregon, June 1989.

[2] Timothy Andrews. The Vbase object database environment. In Alfonso F. Cardenas and Dennis McLeod, editors, *Research Foundations in Object-Oriented and Semantic Database Systems*, pages 221–240. Prentice-Hall, Englewood Cliffs, NJ, 1990.

[3] James R. Armstrong. *Chip-Level Modeling with VHDL.* Prentice-Hall, Englewood Cliffs, NJ, 1989.

[4] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, P. W. Cockshott, and R. Morrison. An approach to persistent programming. *The Computer Journal*, 26(4), 1983. (reprinted in [34]).

[5] D. S. Batory, J. R. Barnett, J. F. Garza, K.P. Smith, K. Tsukuda, B. C. Twichell, and T. E. Wise. GENESIS: An extensible database management system. In Stanley B. Zdonik and David Maier, editors, *Readings in Object-Oriented Database Systems*, pages 500–518. Morgan Kaufmann, San Mateo, CA, 1990.

[6] H. Boral. Prototyping Bubba, a higly parrallel database system. *IEEE Transactions on Data and Knowledge Engineering*, 2(1), May 1990.

[7] M. Carey, D. DeWitt, D. Frank, G. Graefe, J. Richardson, E. Shekita, and M. Muralikrishna. The architecture of the EXODUS extensible DBMS. In *Procedings of the International Workshop on Object-Oriented Database Systems*, September 1986.

[8] M. J. Carey, D. J. DeWitt, and Scott L. Vandenberg. A data model and query language for EXODUS. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, June 1988.

[9] R. L. Cooper, M. P. Atkinson, A. Dearle, and D. Abderrahmane. Constructing database systems in a persistent environment. In *Proceedings of the 13th VLDB Conference*, 1987.

[10] G. Copeland. Uniform object management. In *Proceedings of the Intl. Conf. on Extending Database Technology*, March 1990.

[11] Adele Goldberg and David Robson. *Smalltalk-80: The Language.* Addison Wesley, 1989.

[12] G. Graefe and D. J. DeWitt. The EXODUS optimizer generator. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, May 1987.

[13] Eric N. Hanson. An initial report on the design of Ariel: a DBMS with an integrated production rule system. *SIGMOD Record*, 18(3), September 1989.

[14] Eric N. Hanson, Moez Chaabouni, Chang-ho Kim, and Yu-wang Wang. A predicate matching algorithm for database rule systems. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, May 1990.

[15] Capt Tina M. Harvey. Access and operator methods for the Triton nested relational database system. Master's thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, December 1990.

[16] S. C. Johnson. YACC – yet another compiler compiler. Technical Report CSTR-32, Bell Laboratories, Murray Hill, NJ, 1975.

[17] Sonya E. Keene. *Object-Oriented Programming in Common Lisp.* Addision-Wesley, 1989.

[18] K. G. Kulkarni and M. P. Atkinson. Implementing an extended functional data model using PS-algol. *Software Practice and Experience*, 17(3):171–185, Marche 1987.

[19] Christophe Lécluse, Philippe Richard, and Fernando Velez. O_2, an object-oriented data model. In Stanley B. Zdonik and David Maier, editors, *Readings in Object-Oriented Database Systems*, pages 227–241. Morgan Kaufmann, San Mateo, CA, 1990.

[20] M. E. Lesk. LEX – a lexical analyzer generator. Technical Report CSTR-39, Bell Laboratories, Murray Hill, NJ, 1975.

[21] Object Design, Inc. ObjectStore technical overview, release 1.0, August 1990.

[22] Andreas Reuters, editor. *Proceedings of the 2nd International Workshop on High Performance Transaction Systems.* Springer Verlag, 1987.

[23] Joel E. Richardson and Michael J. Carey. Programming constructs for database system implementation in EXODUS. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, May 1987.

[24] Joel E. Richardson, Michael J. Carey, and Daniel T. Schuh. The design of the E programming language. Technical report, University of Wisconsin, 1989.

[25] Mark A. Roth, Henry F. Korth, and Don S. Batory. SQL/NF: A query language for ¬1NF relational databases. *Information Systems*, 12(1):99–114, 1987.

[26] Hans-Jeorg Schek et al. The DASDBS project: Objectives, experiences, and future prospects. *IEEE Transactions on Knowledge and Data Engineering*, 2(7):25–43, March 1990.

[27] Capt Craig W. Schnepf. SQL/NF translator for the Triton nested relational database system. Master's thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, December 1990.

[28] Dan Schuh, Michael Carey, and David Dewitt. Persistence in E revisited – implementation experiences. In *Proceedings of the 1990 Persistent Object Systems Workshop*, Fall 1990.

[29] Eugene Shekita and Michael Zwilling. Cricket: A mapped, persistent object store. Technical report, University of Wisconsin, Fall 1990.

[30] M. Stonebraker, E. Wong, P. Kreps, and G. Held. The design and implementation of INGRES. *ACM Transactions on Database Systems*, 1976.

[31] Michael Stonebraker, Lawrence Rowe, and Michael Hirohama. The implementation of POSTGRES. *IEEE Transactions on Knowledge and Data Engineering*, 2(7):125–142, March 1990.

[32] Bjarne Stroustrup. *The C++ Programming Language.* Addision Wesley, 1986.

[33] Bjarne Stroustrup. Parameterized types for C++. In *Proceedings of the Usenix C++ Conference*, 1988.

[34] Stanley B. Zdonik and David Maier, editors. *Readings in Object-Oriented Databases.* Morgan Kaufmann, 1990.

[35] Michael Zwilling. B+-tree external documentation, 1989. EXODUS Project Documentation.