Persistence in the E Language: Issues and Implementation

by
Joel E. Richardson
Michael J. Carey

Computer Sciences Technical Report #791
September 1988

Persistence in the E Language: Issues and Implementation

Joel E. Richardson
Michael J. Carey

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

ABSTRACT

E is an extension of C++ providing, among other features, database types and persistent objects. In a language offering persistence, there are many important design and implementation issues which must be resolved. This paper discusses some of these issues, comparing the approach taken in the E programming language with other persistent systems. The basis of persistence in E is a new storage class for variables, and physical I/O is based on a load/store model of the long-term storage layer. In addition to discussing the issues and E's general approach, this paper also details the current implementation.

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 IRI-8657323, by IBM through a Fellowship, by DEC through its Initiatives for Excellence program, and by grants from the Microelectronics and Computer Technology Corporation (MCC) and GTE Laboratories.

1. INTRODUCTION

The EXODUS Project at the University of Wisconsin has been exploring a toolkit approach to building and extending database systems [Care85, Care86b]. The first component of EXODUS to be designed and built was the EXODUS Storage Manager [Care86a]. It provides basic management support for objects, files, and transactions. The E programming language [Rich87a] was originally conceived, in part, as a vehicle for conveniently programming against this persistent store. E is the language in which database system code is written; that is, the abstract data types (e.g. time), access methods (e.g. grid files), and operator methods (e.g. hash join) are all written in E. E is also the target language for schema and query compilation; user-defined schema are translated into E types, and user queries into E procedures. In this way, E reduces the impedance mismatch [Cope84] between the database system and the application. Finally, the EXODUS Optimizer Generator [Grae87a, Grae87b] allows the database implementor (DBI) to produce customized query optimizers, given rules describing the query algebra. The first demonstration of a database system built with the EXODUS tools was given at SIGMOD-88. In the three weeks prior to the conference, we built a relational system prototype (of course!) complete with indices and an optimizer. Most recently, the EXTRA data model and EXCESS query language have been designed [Care88], and this system is now being implemented with the EXODUS tools.

The design of E has evolved considerably from the early descriptions in [Care86b]. The original intent was to design a language for writing database system code; the resulting language [Rich87a, Rich88a, Rich88c] is an extension of C++ [Stro86] providing generic classes, iterators, and persistence. C++ provided a good starting point with its class structuring features and its expanding popularity as a systems programming language. Generic classes were added for their utility in defining database container types, such as sets and indices. Iterators were added as a useful programming construct in general, and as a mechanism for structuring database queries in particular. Persistence — the ability of a language object to survive from one program run to the next — was added because it is an essential attribute of database objects. In addition, by describing the database in terms of persistent variables, one may then manipulate the database in terms of natural expressions in the language. This paper describes the design and current implementation of persistence in E.

The remainder of the paper is organized as follows. Since E depends on the EXODUS Storage Manager to provide the basic persistent store, we begin Section 2 with a review of that interface. The remainder of the section introduces the E language by example, concentrating on those features related to persistence. Section 3 presents a discussion of some of the important issues in designing and implementing a persistent language. The approaches taken by several other systems are compared with that of E. Section 4 details the current prototype implementation of the E compiler. Finally, Section 5 concludes with a summary and a report on our current research. An appendix expands on the example given in Section 2, presenting other features of the E language.

2. REVIEW OF THE EXODUS STORAGE MANAGER AND E

2.1. The Storage Manager Interface

Of the components mentioned in the introduction, the EXODUS Storage Manager [Care86a] is most important to the implementation of persistence. The Storage Manager provides *storage objects*, which are uninterpreted byte sequences of virtually any size. Whether an object is several bytes or several gigabytes, clients of the Storage Manager see a uniform interface. Each object is named by its object ID (OID), which is its physical address. Two basic operations, *read* and *release*, provide access to objects. The read call specifies an OID, an offset, and a length. These parameters name a byte sequence within the object which the Storage Manager reads into the buffer pool. The Storage Manager then returns a *user descriptor* to the client. This descriptor contains a pointer to the data in the buffer pool, and the client accesses the data by dereferencing through this pointer. When finished, the client returns the user descriptor to the Storage Manager in a release call. The release call also includes a flag indicating whether the client wrote the data or not.[1]

It is important to understand that data requested in a read call is *pinned* in the buffer pool until the client releases it. Pinning is a two-way contract: the Storage Manager guarantees that it will not move the data (e.g. page it out) while it is pinned, and the client promises not to access anything outside the pinned range. In addition, the client promises to release (unpin) the data in a "timely" fashion, because pinned data effectively reduces the size of the buffer pool.[2] In the subsequent discussions, pin and unpin are used interchangeably with read and release.

An example of a client's interaction with the Storage Manager is illustrated in Figure 1. A range of bytes containing a `struct S` is embedded at `offset` within the Storage Manager object having the given `oid`. The read call pins that range of bytes in the buffer pool. On return, `ud` points to a user descriptor, whose first word contains a pointer to the pinned data. The statement after the read then multiplies the x field by 10. Finally, the data is

```
struct S { int x; float y; };
USERDESC * ud;

sm_read( oid, offset, sizeof(struct S), &ud );
((struct S *) *ud) -> x *= 10;
sm_release( ud, DIRTY );
```

Figure 1: Interacting with the EXODUS Storage Manager

[1]Many details have been glossed over. For example, there are a number of other parameters in the read call, and releasing something "dirty" is actually a different call than releasing something "clean". However, this simplified model is sufficient for this paper.

[2]Actually, the EXODUS Storage Manager provides *buffer groups*. A buffer group is a set of pages requested by a transaction and managed with a specified page replacement policy. The idea is to avoid interference in the paging characteristics of different transactions. Thus, if a transaction leaves data pinned, the performance degradation is largely to itself.

released (unpinned). While the EXODUS Storage Manager is a powerful utility, it is necessary for the programmer to learn numerous procedures and to execute many steps in order to perform even simple tasks[3]; this complexity is one of the primary motivations for the E language.

The other important abstraction provided by the Storage Manager is the *file*. A file is a set of EXODUS objects ordered by OID. Files are disjoint; every EXODUS object resides in one and only one file. Operations include those to create or destroy an object within a file and to scan a file, returning each object's OID. The current implementation of E uses Storage Manager files and objects to realize the persistent store, although the basic approach could be adapted to other storage systems. An interesting result of this work will be an evaluation of the suitability of the current Storage Manager design for supporting a persistent language and insight into ways the design might be improved.

2.2. Review of E

As noted, E is an extension of C++ [Stro86], which is itself an extension of C [Kern78]. The essential concept in C++ is the *class*. A class defines a type, and its definition includes both the physical representation of any *instance* of the class as well as the operations that may be performed on an instance.[4] In C++ parlance, the former are called data members, and the latter, member functions (a.k.a. methods). Member functions are always applied to a specific instance; within the function, any (unqualified) reference to a data member of the class is bound to that instance. This binding is realized through an implicit parameter, `this`, which is a pointer to the object on which the method was invoked. An unqualified reference to a member `x` of the class is equivalent to `this->x`.

The example in Figure 2 is a (nonsensical, but) complete C++ program which defines and uses a bounded stack of integers. Though not yet very interesting, it will serve as the basis for future examples. The physical representation of a stack comprises an integer array holding the stack elements and an integer index for the current top-of-stack. Because this representation is declared in the private section of the class definition, it is hidden from users of the class. Following the keyword `public` are declarations of the methods available to users of the class. The stack methods, of course, are those to push and pop elements, and to test if the stack is empty. The bodies of the stack routines are elaborated following the class declaration. Consider the member function `push`. Note that, as mentioned, the references to `stackTop` and `elems` are equivalent to the expressions `this->stackTop` and `this->elems`, respectively. If the stack instance is not already full, the `push` routine increments the top-

[3]There are 23 interface routines to the Storage Manager, although only a few may be needed for a given application. Not shown in the above example are the (necessary) steps of initializing the Storage Manager, mounting a volume, and allocating buffer space via the buffer group mechanism [Care86a].

[4]Unlike the abstraction mechanisms provided in CLU [Lisk77] or Smalltalk [Gold83], a C++ class does not necessarily hide the physical representation of instances. It is up to the designer of a class to declare explicitly which members (data and function) are private and which are public.

```
const STACK_MAX = 100;
class stack
{
    int             stackTop;
    int             elems[ STACK_MAX ];
public:
    stack();
    void push( int );
    int pop();
    int empty();
}; /* class stack */

stack::stack()
{
    stackTop = -1;
}

void stack::push( int val )
{
    if( stackTop < STACK_MAX-1 )
      elems[ ++ stackTop ] = val;
    else
      printf("stack::push: Sorry, stack is full.");
}

int stack::pop()
{
    if( stackTop >= 0 )
      return elems[ stackTop -- ];
    else
      printf("stack::pop: Sorry, stack is empty.");
}

int stack::empty()
{
    return (stackTop == -1);
}

stack S;

main()
{
    while( ! S.empty() )
      printf("popped %d", S.pop() );

    for(int i = 1; i <= 10; i++)
      S.push(i);
}
```

Figure 2: The Stack Example

of-stack index and places the new element at that location.

In addition to the usual stack operations, notice that there is also an operation named `stack`. In general, a method whose name is the same as its class is called a *constructor.* Constructors are intended to initialize instances of a class and are called automatically whenever an instance is created, e.g. by coming into scope. In this example, the constructor initializes the top-of-stack index to -1, indicating that the stack is empty.

Following the member function definitions, there is a declaration of a global stack instance, `S`. It is guaranteed that the stack constructor will be called for `S` by the time the main program runs; the mechanism to accomplish this initialization is described more fully in a later section. The main program itself does nothing particularly interesting in this example. It begins by popping all the elements, printing each value that it receives. Since the stack is initially empty, this loop does nothing. Then the integers 1 through 10 are pushed onto the stack, and the program exits. Clearly, running this program produces no output, nor will it in any future run.

Figure 3 shows (in boldface) the changes needed to convert the example of Figure 2 into an E program in which the stack is persistent. By changing the keyword `class` to `dbclass` and the type name `int` to `dbint`, and by giving `S` the storage class `persistent`, the effect of running the main program is altered as follows: The first time the program is run, no output is produced; persistent stacks also are initialized to empty. However the ten

```
const STACK_MAX = 100;
dbclass stack
{
    dbint        stackTop;
    dbint        elems[ STACK_MAX ];
public:

    /* as before... */

}; /* class stack */


/* all methods are as before... */


persistent stack S;

main()
{
    /* as before... */
}
```

Figure 3: Making the Stack Persist

integer values pushed at the end of the main program are preserved when the program exits. The next time the program is run, the first loop will pop these elements, and the user will see:

```
popped 10
popped 9
...
popped 1
```

Some explanatory notes are in order. First, the role of `db` is explained more fully in the next section, but briefly, it is an attribute of a type which allows objects of that type (optionally) to be persistent. Second, changing `int` to `dbint` is not strictly necessary, since the compiler can (and does) infer this change within the context of the `dbclass`. Third, in this very simple example, the class, the persistent object, and the main program are all declared in one module. This is usually not the case, as one typically declares persistent objects in separate modules, and then links these with the various main programs that use them. This is a very brief introduction to the E language. For more details, the reader is referred to the Appendix and to [Rich87a]. (We note that several aspects of the language have been improved; the full revised description will appear shortly in [Rich88a]. The example in the appendix demonstrates all the major features of E by expanding the stack class into one which is generic, unbounded, and uses an iterator.)

3. ISSUES IN A PERSISTENT LANGUAGE

Persistence is an attribute describing an object's lifetime. In conventional languages, all object lifetimes are bounded by the length of the program run. Access to long term storage is provided by extralingual constructs such as operating system files or an embedded database system interface. The shortcomings of this approach are well known [AtkM85a, Cope84, AtkM87]. In a language with persistence, data objects survive between program runs. Furthermore, the programmer can manipulate the objects with normal expression syntax, i.e. physical I/O is transparent to the programmer.

Within this very broad definition of persistence, there is considerable latitude for defining precisely what it means and how it is implemented. As might be expected, the issues are interrelated, and so a proposed solution for one problem usually affects the possible solutions for others. The following discussion presents a number of design considerations, and for each, outlines the solution adopted in E, comparing that solution with those taken in several other systems. (An earlier paper [Rich87b] outlined our thinking before we had actually implemented the first prototype of the system.) The major reference point for comparison is the PS-Algol language [AtkM83a] mainly because it is one of the few systems whose implementation has been extensively described in the literature. We defer until Section 4 a number of implementation issues specific to adding persistence to C++.

3.1. Persistent Handles / Persistent Name Spaces

An essential property of a language with persistence is that objects in the database may be manipulated using the same expression syntax as for volatile objects. In order to execute such an expression, however, there must first exist a binding between symbols in the program and objects in the persistent store. When such bindings are established, how they are specified, and to which program symbols they apply are all important questions in determining the nature of the "handles" a program has on the database.

In a language without persistence, long term store has traditionally been implemented by files. The persistent name space is simply the space of file names maintained by the operating system. In this case, the program's handle on the database is a character string representing a file name. A runtime call to a system routine (e.g. open) establishes the binding (e.g. a file descriptor) between the program and an actual file. The program can then access the persistent data via read and write calls. One obvious drawback of this approach is safety; since a file may be accessed independently, there is no guarantee that the file bound to one run of the program has any relationship to the one bound in another run, except that they have the same name.

PS-Algol [AtkM83a] was the first system to integrate persistence in a fully general (orthogonal) way. In this language, a program's persistent handle is, again, a character string naming an operating system file, and the actual binding is established with a runtime call.[5] This time, however, the file contains a persistent PS-Algol heap, and the open call binds the heap to a pointer in the program called a database *root*. The program is then free to dereference through the root pointer to access the rest of the database. The top level object in every database (i.e. what the root points to) is an associative index of pointers keyed on string names. By providing a character string argument to a lookup routine, the user gets back a pointer to the object associated with the string. The persistent name space thus comprises whatever strings the user has stored in the index. It should be noted that these strings are not recognized as variable names by the compiler, so there is no persistent binding between program symbols and database objects. A program can cause an object to persist at runtime by establishing an access path to the object from the root of an open database. When the program closes the database, all objects reachable from the root are written out to disk.[6]

Object-oriented database systems (OODBS) are closely related to persistent languages. In recent years, a number of such systems have appeared both in the literature and in the marketplace. GemStone [Cope84, Maie87a] is an OODBS based on Smalltalk [Gold83]. Its persistence mechanism has much in common with PS-Algol's, although it appears from [Maie87a] that *all* GemStone objects are implicitly persistent, not merely the "reachable" ones. A persistent name space consists of a dictionary of <name,value> associations, where the name is a string,

[5]More recent work on Napier, PS-Algol's successor, eliminates this reference to an external object, i.e. to a file [AtkM85a].

[6]This is an oversimplification. Actually, an object is written back only if necessary, i.e. if it was created or changed during the program run.

and the value is either an atomic value, e.g. the integer "10", or a reference to another object. The OPAL compiler expects a list of name spaces at invocation; the compiler binds identifiers by searching these name spaces in the order that they appear on the list. At the first occurrence of the desired name, the program variable is bound to the corresponding <name,value> object. Since users may organize dictionaries as they wish, there is considerable flexibility in this approach.

Vbase [Andr87, Onto87] is a new commercial product calling itself an "integrated object system." It seeks to blend an OODBS with the C programming language. The system presents to the programmer two languages and their respective compilers: the type definition language, TDL, in which the one specifies classes and operations, and the C superset, COP, in which one writes methods to implement the operations. Application programs are also written in COP. In order to bind persistent names within a program, both the TDL and COP compilers require a database file name as a command line argument. A Vbase database implements a global persistent name space in which type names and instance names are resolved. It also supports a module construct, however, so that names within a module do not necessarily conflict with names at the global level. In the current release, databases are self-contained and disjoint; a given database contains all of the types, methods, and instances needed for its applications, and there is no sharing between databases.

The O2 system [Banc88] is another recently-developed OODB. It is very similar to Vbase in that it attempts to integrate an object-oriented database system, O2, with a superset of the C language, CO2. (Actually, both Vbase and O2 intend to support a set of languages, but only the C extensions have been implemented.) Like Vbase, type definitions are written in one language, while methods are written in the C extension. O2 associates persistence with a type *extent*, that is, each O2 class implicitly owns a persistent collection of objects of the class.[7] There are primitives for applying operations to all objects in an extent, but there are no persistent handles on individual instances. Type names are the only persistent identifiers, and there is a single global space of type names.

In E, if a variable is declared to have the `persistent` storage class, that variable's name is a persistent handle. The existing scoping rules of C++ determine the organization of the name space. That is, a name *n* in one source module does not conflict with *n* declared in another module unless one attempts to link the two into a single program. The binding between the name and the persistent object is established at compile time, and it remains valid until the module is explicitly destroyed. There is no run time open call, and there are no references to external file names. Bindings between the source code and persistent objects are maintained implicitly by the programming environment, as described in a forthcoming paper [Rich88b]. In order to access the database, one compiles and runs an E program which manipulates persistent objects. These objects may be declared in the same module with

[7]The description of O2 in [Banc88] emphasizes that the current implementation is a "throwaway" prototype, so the details of persistent name spaces may change in future versions.

the program, or they may be declared as `extern` variables. In the latter case, one must link the program together with the object modules (.o files) containing the desired persistent variables.

An E program may also create persistent objects dynamically. There is a predefined dbclass, `file`, which has, among others, operations to create and destroy objects within a file instance. Unlike operating system files, `file` is actually a type in the language. One may declare variables of this type, and in particular, one may declare `persistent file` variables. An object created in a persistent file will exist as long as the file does unless the object is explicitly destroyed by a delete operation. Thus, in E, a persistent `file` is similar to a persistent heap in PS-Algol in that it provides the means for dynamically creating and destroying objects in the database. Note that the presence of the type `file` does not prevent E programs from also manipulating operating system files as might be needed, for example, in loading a database.

3.2. Orthogonality

One of the terms coined by PS-Algol is *orthogonal persistence* [AtkM83a], that is, the possibility that any object can be made to persist, independent of its type or the way it is used in the program. Orthogonality is convenient for the programmer, who may not know in advance that a certain type of object will need to persist or that a certain function may be passed persistent arguments. It is also convenient for the implementation of the runtime system, since it allows for a uniform treatment of all objects. Orthogonal persistence was one of the major contributions of PS-Algol.

In systems such as Vbase and O2, there is a very clear distinction between what may and may not persist. The type systems introduced into their C extension languages are very different from the the type system of C, and expressions to manipulate the persistent objects rely on an embedded "escape" syntax. In both cases, the embedded type system is object-oriented; that is, pointers are implicit in the implementation but are not directly available to the programmer, and operation invocation follows the message-passing paradigm. While persistence is not orthogonal in these systems, the separation between the underlying OODB syntax and the host language may ease the extension of persistence to other host languages.

In E, persistence is not strictly orthogonal to type, as it is in PS-Algol, although it comes much closer than Vbase or O2. E is intended as an implementation language for building database systems. In such systems, there are many data structures such as lock tables which are known *not* to be persistent and which experience a very high frequency of access. If a given expression is a reference to an object that may or may not be persistent, we have no choice — short of specialized hardware — but to include a runtime check to see if the object needs to be read. This extra checking would result in an unacceptable decrease in the performance of access to these critical nonpersistent structures.

Because we desired to leave the C++ subset of E unaffected, and yet enjoy all the benefits of its type system in a persistent environment, we decided to mirror the existing type constructs with counterparts having the *db* (database) attribute. Let us informally define a db type to be:

(1) one of the fundamental db types: dbshort, dbint, dblong, dbfloat, dbdouble, dbchar, and dbvoid.

(2) a dbclass, dbstruct, or dbunion. Such classes may have data members only of other db types, but there are no restrictions on the argument or return types of member functions, i.e. they may accept or return non-db type values.

(3) a pointer to a db type object.

(4) a vector of db type objects.

An object may be persistent *only* if it is of a db type. However, a db type object need not be persistent. Note that any type definable in C++ may be analogously defined as a db type. Furthermore, since persistence is orthogonal over all db types, one can, if desired, program exclusively in db types and achieve the effect of strict orthogonality.[8]

Because every expression in E is typed, the compiler can statically distinguish references to (possibly persistent) db type objects from references to (guaranteed nonpersistent) C++ objects. The latter case runs at no loss of performance. Note that since db type objects need not be persistent, then a db type reference must be checked at runtime, as discussed above. However, db types are provided with the expectation that the majority of accesses will be to persistent objects. Nonpersistent db type objects are supported for completeness and convenience. Given that most checks for persistence will succeed, and that such cases result in calling the EXODUS Storage Manager, the cost of the check itself is negligible.

3.3. Representation of Objects & Pointers

Another important consideration in the implementation of a persistent language concerns the implementation of objects on disk. How are their addresses represented? How are objects organized internally? Is there a format-conversion as objects are brought into memory?

S-Algol [Morr82], the starting point for PS-Algol, is a heap-based language in which "the data type pointer comprises a structure with any number of fields, and any data type in each field" [AtkM83a]. Any pointer may point

[8]An interesting "hack" is to define the following macros:

```
#define class      dbclass
#define struct     dbstruct
#define union      dbunion
#define int        dbint
// etc...
```

to any structure, that is, pointers are inherently untyped. This design has two important implications for PS-Algol. One is that performance must be affected since all pointer dereferencing is subject to run-time type checking. For example, in the expression[9] `p(x)`, the compiler cannot know in general that `p` will point to a structure with a field named `x` whose type matches that required in the current context. Therefore, a runtime check is included to validate the expression. The other important implication is that runtime type descriptors must be available in order to carry out this check. In a persistent extension, these descriptors must also be persistent. The efficient representation and processing of type descriptor information thus formed a significant part of the implementation effort of PS-Algol [Cock84, Brow85].

The heap-based nature of PS-Algol exacerbates the problem described above since we are required to do garbage collection. In a persistent extension, the garbage is on disk. Not only does this present obvious performance obstacles, but it also requires that there be enough information to *do* garbage collection. Because one may define arbitrary compositions of structures, pointers, and arrays, finding all the pointers in a reachability traversal is somewhat complicated. The solution adopted in PS-Algol is to make all objects self-describing to the extent that pointers may be located.

In contrast, E, being derived from C++, is a language in which the physical structure of objects is known (and specified) by the programmer. Heap space is allocated and freed under explicit programmer control, and there is no garbage collection. Furthermore, pointer is a type *constructor*, rather than a fundamental type, so all pointer dereferences are type checked at compile time. Because there is no garbage collection, and because type checking is static, there is no general need to maintain persistent type descriptors.[10] We will have more to say about garbage collection shortly.

Another way in which E differs from PS-Algol is in the representation of Lvalues (i.e. pointers). PS-Algol recognizes two kinds of pointers, both one word in length, and distinguished by their most significant bit (msb). A pointer with an msb of zero is called a Local Object Number (LON) and it contains the memory address of the object it references. If the msb is a one, the pointer is a Persistent IDentifier (PID) and contains the database address of the object. At runtime, PIDs are converted to LONs and back, as objects are moved in and out of memory (as described more fully in the next section).

Like PS-Algol, E recognizes two kinds of pointers. Unlike PS-Algol, they are distinguishable by type at compile time. Any pointer whose type is defined in the C++ subset of E is a normal C++ pointer, i.e. it is one word in length and contains a memory address. Any pointer whose type is a db type has a different format. The Lvalue of a

[9]PS-Algol uses parentheses to express structure access.

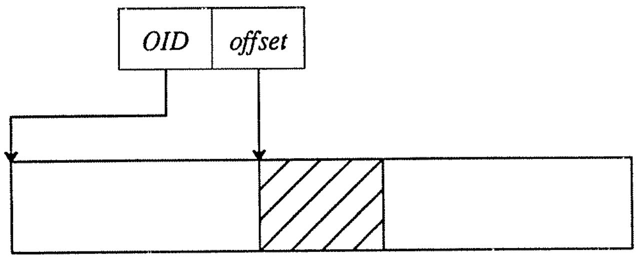
[10]This is not to say that persistent objects are unconnected to persistent type information, only that physical descriptors are not needed in E.

db type object comprises an EXODUS storage object id and an offset into the object, as shown in Figure 4. This <OID,offset> pair is called a _DBREF (which is also the name of the C structure type used to implement them). The offset is necessary because it is possible (and quite common) for a program to produce an address which is in the middle of a storage object. For example, one often processes an array by incrementing a pointer to each element in turn. Also, member functions of a dbclass, like their non-db counterparts, are passed a pointer, `this`, to the beginning of the class object; this object, however, may well be embedded as a member in some containing class object. This aspect of E pointers — really, a straightforward extension of current C++ semantics — is in contrast to PS-Algol, in which pointers are constrained to point only at "top-level" objects, i.e. to the start of an independently allocated unit of storage. As we shall see in the next section, this difference in pointer semantics has implications for the way in which physical I/O is realized.

One more detail concerns the representation of the address of a *non*persistent db type object. In E, "db" is an attribute of a type, while "persistent" is an attribute optionally given to objects of that type. One may also create db type objects on the stack or in the heap. If a function expects a db type pointer, that function should work whether the address is of a persistent or of a nonpersistent object. To handle such cases, we construct a _DBREF with a special OID indicating main memory, and we embed the address of the object in the offset field. Such _DBREFs are detected at runtime, and the embedded address is then used.

3.4. Physical I/O

Given that the persistent objects of a program reside on secondary storage, we must decide the mechanism by which these objects migrate in and out of main memory during a program run. We must also decide how external addresses are mapped to internal addresses that are usable by the program. The approach pioneered by PS-Algol is



EXODUS Storage Object

Figure 4: The Lvalue of a DB Type

to view the storage layer as a persistent virtual memory. A software[11] check of the address format precedes each pointer dereference. If the pointer is in PID format, an "object fault" is signaled. The object is read into memory, giving it a valid LON address, and the pointer which caused the fault is overwritten with that LON. (Maier refers to this pointer translation as "pointer swizzling" [Maie87b].) Once a pointer has been converted into LON format, it will no longer cause an object fault, although every reference is still subject to the runtime check. When an object X is written back to disk, any pointers in X which were converted to LONs must be restored to PID format. Furthermore, any pointers (e.g. in other objects) which contain X's LON must have X's PID restored. This description is obviously simplified, but the basic idea was used in several different implementations [AtkM83b, Cock84, Brow85]. It should be noted that this approach has much in common with LOOM [Kaeh83], which supports Smalltalk [Gold83]. The major difference is that LOOM does not attempt to support persistence or transactions [Kaeh86].

By contrast, the model of persistent storage used in the current implementation of E views the buffer pool and disk as the registers and main memory, respectively, in a load/store machine. That is, in order for persistent data to be manipulated by a program, it must first be loaded into the buffers (registers). When the program is finished using the data, it must release the buffer space (free up a register), and if the data has been written, it must inform the buffer manager that the data is dirty (store the register). Of course, this analogy is not literal. Hardware registers are fixed in width and in number, whereas one may pin an arbitrary range of bytes from an object, and one may pin simultaneously as many ranges as will fit in the available buffer space. These and other differences give rise to a number of interesting code generation problems that do not arise for conventional compilers. The prototype implementation described in this paper, however, makes a number of simplifying assumptions. For example, it is assumed that the buffer pool is large enough to hold any and all byte ranges that might need to be pinned simultaneously. Also, buffer space is held only long enough to evaluate the expression needing the data. No optimization techniques have yet been applied.

It is useful to consider some of the implications of the differences between the machine models adopted in PS-Algol and in E. In the software virtual memory approach, I/O is transparent to the compiler, and it is easy to generate code. The compiler simply issues instructions against the persistent address space, and the runtime system takes care of the rest. Because the virtual memory handles *object* faults, objects are always loaded in their entirety. By contrast, in a load/store model, I/O is scheduled by the compiler[12], and so code generation is more complicated. Because the persistent storage space is byte-addressable, the compiler pins only those pieces of an object actually needed, giving E programs the ability to work with very large objects.

[11]Hardware implementations are also possible and, in fact, some have been designed. See, for example, [Cock87].

[12]Note that the machine presented by the EXODUS Storage Manager also has a cache (the buffer pool). Physical I/O is actually scheduled by the buffer management component of the Storage Manager.

The implementations of PS-Algol and GemStone rely on "pointer swizzling" to help regain performance. That is, the cost of faulting on an object is amortized over the number of times the object is referenced. If a program tends to access the same object repeatedly through the same pointer, then the amortization may be significant. However, if the program tends to touch each object only once or twice, e.g. while traversing a graph or scanning a relation, then the gains will be lower. In addition, the task of freeing the memory space occupied by an object is complicated somewhat because there may be other objects in memory containing the LON (in PS-Algol terms) of that object, i.e. its swizzled pointer. Access patterns such as graph traversal may tend to fault with high frequency, clogging memory with objects whose pointers need de-swizzling when they are removed. In the implementation of PS-Algol, the reclamation of buffer space is deferred until memory is used up. At that point the garbage collector runs, and the (memory) heap is compacted, possibly forcing a subset of objects to disk [Coop88].

In E, there is no format conversion of objects as they are read into memory. Thus the space occupied by an object can be reclaimed relatively cheaply. On the other hand, because loads and stores are generated at compile time, we run the risk of generating poor code, e.g. by redundant fixing of data. The next version of E's code generator will employ optimization techniques to improve overall code quality. The differences between buffer allocation in the Storage Manager and register allocation in a hardware machine, however, make this a research problem rather than a straightforward application of known optimization techniques. It seems likely, though, that significant speedups can be achieved. The overall performance of optimized E programs relative to PS-Algol programs remains to be seen.

3.4.1.1. Garbage Collection

We have decided not to implement garbage collection in E. This decision was based on a number of factors. First, as already noted, in order to find all reachable objects, we would need to embed physical descriptors in objects. Their storage and maintenance is a nontrivial task. Second, garbage collecting the disk is an expensive operation [Butl87]. Third, and perhaps most convincingly, garbage is created in E only "by mistake." Unlike languages such as CLU [Lisk77, AtkR78] and PS-Algol, in which garbage is an implicit by-product of a program's normal execution, the loss of the "last" reference to an object in E is usually an error, and in any case, would be inappropriate style in a systems-level programming language. Moreover, if a reference is erroneously lost to a persistent object, then presumably that object is part of the database and should be restored. Even if a garbage collector were implemented, it is not at all clear how it would distinguish these lost fragments from true garbage, nor, were this possible, how addressability should be restored. Note that PS-Algol programs are also subject to this kind of error, and its garbage collector will simply sweep up the lost objects.

Having argued that garbage collection is inconsistent with a language like E, we must address the problem that garbage, if it is created on disk, will be persistent garbage. In E, as mentioned earlier, there are two ways to create objects that persist. One is to compile the declaration of a persistent variable. In that case, the variable name constitutes a persistent handle on the object. In the first implementation of E, there was no environment support; if the object module containing the handle was destroyed without using the special removal utility (erm), the object in the storage manager became garbage. In the environment described in [Rich88b], this kind of error is prevented; source and object modules are subject to controlled access. That is, if one destroys a module containing the handles of persistent objects, those persistent objects are destroyed first.

Persistent objects may also be created under program control, i.e. by invoking `new_obj` on a persistent file. Since one can scan the file to retrieve the addresses of all the objects it contains, those objects are never technically garbage; the address of every object allocated in a file by an E program can be recovered by an E program that scans the file. Unfortunately, the combination of C++ semantics (which allows type casting) and the implementation of `file` as a class means that it is still possible for an E program to lose its reference to a file. We briefly considered restricting the use of files to cases where this kind of error could not occur; we rejected this approach, however, because it would have seriously diminished the usefulness of E. As argued above, garbage collection is not the answer to this problem, either.

This discussion leads to a more general issue which, to date, has received seemingly little attention. The problem is simply stated: programs have bugs, and if such programs are run against the database, the resulting database state may be inconsistent. Note that the issue here is not whether we are able to abort a transaction (we can), but what to do when a transaction commits having done the wrong thing. For transactions against a relational database, the range of such errors is limited by what can be expressed in the query language. In many cases, the damage can be repaired by running a compensating transaction. But in a persistent language, the range of possible errors is much broader, including possible corruption of database structures. The erroneous creation of garbage is one example; leaving a dangling pointer is another. In such cases, a compensating program may not even be runnable.

One approach to this problem is to minimize the probability of such errors by establishing a methodology in which programs are thoroughly tested against either nonpersistent data or against a throwaway database. At some point the program is "certified" bug free and is released for use against the real database. While this is a good and useful policy in any case, nevertheless for a program of any complexity, the certification is probably a fiction. We are still faced with the problem of what to do when things go wrong. An interesting area for future research, therefore, is in the development of debugging strategies and tools for a persistent environment.

3.5. Schema Evolution

The problem of supporting schema evolution is important for any system which stores long term data. Until recently, however, very little work has been done in this area. Relational database systems, for example, typically support only a few kinds of schema change, e.g. creating a relation, or adding an attribute. PS-Algol does not address the issue at all, although for that system, the problem is not as severe; structures are self describing, and all field accesses are interpreted at runtime. Recent work in the area of object-oriented database systems has produced a few papers on the subject. For example, Skarra and Zdonik have proposed language constructs to filter objects between the database and the executing process [Skar86]. When a new version of an existing type is defined, the programmer (perhaps with automated guidance) defines filtering functions which allow existing programs to deal with objects of the new version of the type and to allow new programs to deal with objects of the existing version of the type. As pointed out in [Penn87], this approach incurs a cost every time an object is referenced.

An alternative to filtering is object conversion. In such an approach, one first defines a set of invariants which determines what it means for a schema to be "consistent." Then one defines the allowable schema change operations and, for each change, the updates that must be performed in order to preserve the invariants. This approach was pioneered by the Orion OODB [Bane87] at MCC, and is currently being pursued in GemStone [Penn87]. The major difference between the two is that the Orion scheme is lazy— changes are not propagated until the data is next accessed— while the GemStone scheme is eager— with changes being propagated immediately.

For a number of reasons, E does not attempt to support schema evolution in an automated fashion.[13] We do not wish to insert a filtering layer because, as will be described, E programs access data directly in the buffer pool. Filtering would place a procedure layer between the application and its data. We also reject an automatic update solution because, unlike a truly object-oriented system, the physical structure of objects is much more apparent in E. Defining the set of invariants, the possible schema changes, and the associated updates would appear to be a far more complicated process. For example, if a class contains data members of other classes, those members are physically embedded in the object. In addition, pointers are under the explicit control of the programmer. As a result, one may, for example, obtain and store persistent pointers into the middle of some object. Since many schema change operations would alter the size of a class object (e.g. dropping an attribute), an automated system would have to track down and change such pointers or leave them invalid. The former would be extremely difficult, if not impossible, and the latter is not a solution. Note that this problem does not occur for object-oriented systems such as Orion because pointers are completely hidden from the user and point only to "top level" objects. Finally (and perhaps, most importantly), schema evolution is not our research interest and appears still to be an open problem. In this instance, no solution is probably better than a partial one.

4. THE CURRENT IMPLEMENTATION OF E

In this section, we describe how db types and persistence have been implemented in the first version of the E compiler. We begin with a macro level description of the compiler's structure, showing how new functionality has

[13]This is not to say that evolving schema is impossible, only that the system will not propagate changes automatically.

been integrated with the phases of an existing C++ compiler. We then describe the phase involving db types and persistence, showing how declarations and expressions are processed.

4.1. Organization of the Compiler

The current E compiler is an extended version of the AT&T C++ compiler. We chose this as our starting point for several reasons, the main one being that we had access to the source code. We did not want to start from scratch because reimplementing the C++ subset of E was not our interest and yet would require a significant amount of time and effort. At the time we began work on E, there were only a few C++ compilers available; the AT&T compiler, in addition to being available, also had the distinct advantage of being quite stable.

The AT&T C++ compiler (version 1.2.1) consists of a large shell script (CC) which spawns a number of processes, as illustrated in Figure 5. The original C++ source code is first processed by the standard C preprocessor, cpp. The result is then translated by the C++ front end, cfront, into C source code. This code is then compiled by the C compiler, cc, into binary form. Although not shown in the figure, cc itself comprises a series of processes: cpp (again!); ccom, which translates C into assembler code; as, the assembler; and ld, the link editor.

If the result of the last step is an executable program, the compiler then performs an additional series of steps (enclosed in dashed box in Figure 5). A C++ program may declare an object of a class with a constructor. If the object is declared in the global scope, then the constructor for that object must be executed before the main program runs. The extra steps in the dashed box are part of the mechanism that implements this feature.

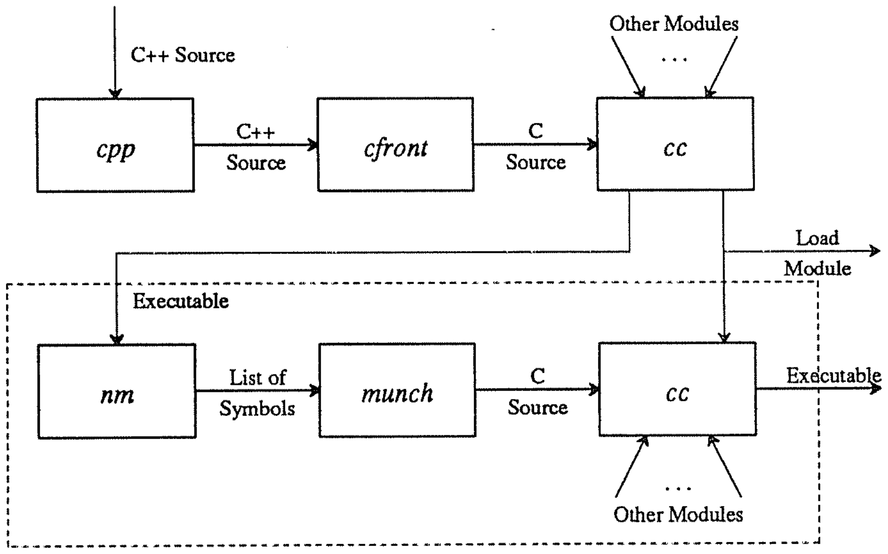


Figure 5: The Process Structure of CC

The majority of the work in implementing E has involved extending the source code for cfront into an E-to-C translator called, appropriately, efront. The internal organization of efront is shown in Figure 6. Of the five phases shown, all but "db simplify" were part of the original C++ compiler. The parsing phase builds an abstract syntax tree of the source text. The parser consumes one external declaration at a time, e.g. one function definition or one global variable declaration. Only a few new keywords were added to this phase in order to handle constructs related to persistence. In addition, when this phase adds a type node to the syntax tree, where that node represents a fundamental db type (e.g. dbint) or a dbclass, a special flag is set in that node. In all other respects, it is like any other type node. This fact is important for the second phase, which handles most of the type checking. Slight changes were needed here to prevent, for example, assigning a db address to a non-db type pointer. Most other type checking is handled normally, and allows, for example, assigning a dbint to an int, or adding an int and a dbfloat. The simplification phase transforms C++ constructs into equivalent C constructs. This phase has been extended to also handle E generators and iterators. The new fourth phase, and the one which is the concern of this section of the paper, transforms constructs related to db types and persistence. Finally, the print phase walks the resulting C syntax tree, producing C source code.

The input to the db-simplification phase is thus a C syntax tree in which certain nodes are decorated. Any expression or object of a db type will point to a type node in which the "is db" flag has been set. Any object declared persistent will also have this storage class recorded as part of its symbol table entry. The code generation of this phase will therefore involve looking for decorated nodes in the syntax tree and applying tree transformations.

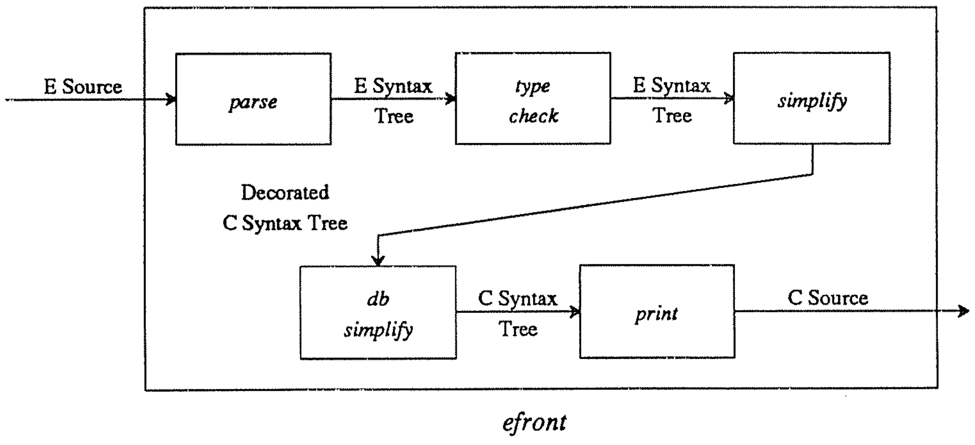


Figure 6: Compilation Phases in Efront

4.2. Transformations

This section details the transformations that are applied by the db simplification phase. It is the responsibility of this phase to locate the types, data declarations, and expressions that involve db types, and to transform them into equivalent C constructs. In particular, it is necessary to add the appropriate calls to the EXODUS Storage Manager so that persistent data is accessible to the program.

4.2.1. Types

E provides a full complement of fundamental types having the "db" attribute. These types are the duals of the fundamental C++ types: dbshort, dbint, dblong, dbfloat, dbdouble, dbchar, and dbvoid. For the purposes of assignment, arithmetic expressions, and parameter passing, such types are equivalent to their non-db counterparts. A node in the abstract syntax tree that represents a fundamental db type is not changed by db-simplification. Such a node is simply printed by the print phase as its non-db dual.[14] For example, the declaration

```
dbint x;
```

becomes

```
int   x;
```

A node representing a function type is processed by recursively transforming the function's return and argument types. If the function also has a function body, i.e. if it is a definition, then we apply the transformations described in Section 4.2.3. For now, we may simply observe that the following (pointless) function

```
dbvoid  fcn( dbfloat x,  dbint y )
{
        x = y;
}
```

is transformed[15] into:

```
char  fcn( x, y )
float x;
int   y;
{
      x = y;
}
```

A node representing a dbclass (or dbstruct or dbunion) is processed by recursively transforming the dbclass's data and function members. Although we do not show an example, it should be noted that, by the time db-simplify sees them, classes related to generators (i.e. generators and classes instantiated from generators) have already been transformed into an equivalent set of non-generator classes.

[14]The types "void" and "dbvoid" are printed as "char".

[15]For those more familiar with the AT&T cfront, names are still changed according to the usual rules. For the examples in this paper, such details are omitted where possible to avoid confusion.

So far, the transformation of types is not very interesting. The one important translation occurs when a type node represents a pointer to some db type. Recall from the discussion in Section 3.3 that address of a db type object is represented by an <OID,offset> pair. In the C translation of every E program is a series of typedefs culminating in the following:

```
struct _DBREF {
      OID          oid;
      int          offset;
};
```

Any type node representing a db pointer is transformed into `struct _DBREF`. So, for example, the declaration:

```
dbstruct  tree_node {
      tree_node * left;
      tree_node * right;
      dbint     data;
};
```

is printed as:

```
struct  tree_node {
      struct _DBREF left;
      struct _DBREF right;
      int           data;
};
```

For a member function of a dbclass, `C`, the type of `this` is likewise transformed, since `this` has type `C*`.

4.2.2. Data Declarations

When the source code contains the declaration of some db type object, the correct transformation depends on the scope of the object and its storage class. Obviously, the most interesting case is the declaration of a persistent object. In E, an object which is declared persistent establishes a binding with a physical object at compile time. The name of the persistent variable is then the programmer's "handle" on the persistent object. The object is accessible to any program in which the variable's name is visible.

The general approach is illustrated in Figure 7. When the db-simplification phase sees the declaration of the (db) integer, x, it asks the EXODUS storage manager to create a 4-byte object. The OID of the new object is then introduced into the output in the form of a _DBREF structure with an initializing expression. This _DBREF variable is called the *companion* of x, and its initializer assigns the OID returned by the storage manager with an offset of 0.[16] Note that only the companion declaration appears in the C output.

Since all EXODUS storage objects must reside in some file, the compiler first asks the storage manager to create one file; then all persistent objects declared in the source module are created within this file. Thus,

[16]The interpretation of the numbers composing the OID is not important for this discussion.

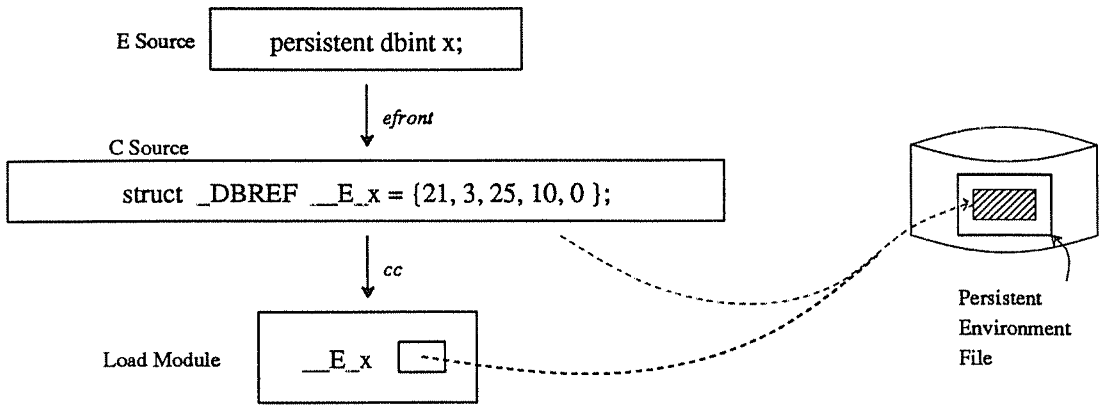


Figure 7: Compiling A Persistent Object Declaration

compiling an E source module which contains the declaration of one or more persistent objects yields both its C translation and a file in the storage manager containing the persistent objects. This file is called the *persistent environment* file, or simply, the .pe file, for the module.

If a db type object is declared external, db-simplification transforms this declaration into an external reference to the object's companion. For example,

```
        extern  dbint  x;
```

becomes

```
        extern struct _DBREF __E_x;
```

This allows functions in one module to access persistent objects declared in another via the usual C++ external reference mechanism. Conversely, it implies that if a module declares a *non*persistent db type object in the global scope, then a companion must be generated for that object as well. As usual, the companion must be initialized with the address of this object, which in this case, is in main memory. Such addresses use a special OID indicating "in memory", and the actual address of the object is embedded in the offset. Thus, if the x mentioned above is simply declared in the global scope as:

```
        dbint  x;
```

then the translation is:

```
        int  x;
        struct _DBREF __E_x = { 0, 0, 0, -1, (int) &x };
```

The example of Figure 7 showed a persistent object declared in the global scope. Persistent objects may also be declared locally in a block. For example,

```
int  counter()
{
      persistent dbint x;
      return x++;
}
```

In this case, although the object is persistent, its name is visible only within the block. Again, the object is created at compile time, and a companion is introduced into the local scope. Here, the companion is given the storage class `static` so that it need not be reinitialized every time the block is entered. The declaration in the above function becomes:

```
int  counter()
{
      static struct _DBREF __E_x = { ... };
      /* return ... */
}
```

For nonpersistent db type objects declared local to a block, we do not need to do anything special. A local dbint, x, in the E source simply becomes a local int, x, in the C translation. This is because most expressions which use x, e.g. addition, *know* that x is not persistent; a companion is not needed. In the case of an expression which takes the address of x, e.g. passing x by reference, a temporary companion is constructed, as described in the following section.

To be consistent with C++ semantics, a persistent object declared without an initializer will be initialized to all zero bytes. Thus, the above counter function will return 0 the first time it is called. A persistent object may also be declared with an initializer, as in Figure 8. In this example, the user has declared a persistent array of dbfloats, specifying the first 3 elements. In such cases, the compiler interprets the expressions (which must evaluate to constants) and sends the binary image of the object to the storage manager.

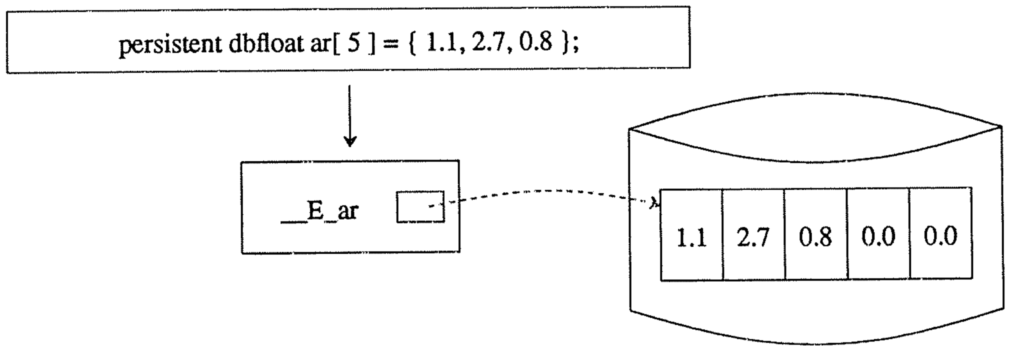


Figure 8: An Initialized Persistent Object

A more complicated initialization problem arises when the program declares a persistent object of some dbclass with a constructor. We will return to this problem shortly.

4.2.3. Expressions and Statements

Processing the declaration of persistent objects provides one part of the picture. Another part is the translation of expressions involving (possibly persistent) db type objects into regular C expressions which manipulate the objects. This section explains these steps in some detail. The set of expressions illustrated here is representative rather than exhaustive, and nonessential details have been omitted.

The current implementation employs a simple code generation technique. In a recursive descent of the expression tree, we look for nodes (subexpressions) whose type is marked "db." Such nodes are locally transformed into C expressions. If the node represents the *address* (Lval) of a db type object, then the translated expression will be of type _DBREF. If the node represents the *value* (Rval) of a db type object, then the translated expression will produce that value at runtime.

One of the parameters to the recursive call specifies whether the Rval or the Lval of the expression is desired. For example, the simplest expression is the name of a variable, say, `x`. Suppose `x` is persistent and appears in the expression `(x + y)`. Then we must transform the subexpression `x` into one which reads `x`, references it in the buffer pool, and (eventually) releases it. However, if `x` is instead part of the expression `(&x)`, the correct action is simply to substitute the companion, `__E_x`, since all that is wanted here is the address of `x`.

The other parameter to the transformation procedure specifies whether the data will be written by the containing expression. Continuing with the previous example, in the expression, `x + y`, `x` is used but not changed. When we generate the read call for `x`, we must also generate the corresponding release call, which in this case should specify that `x` is clean. However, in the assignment, `x = 1`, `x` will definitely be written; the release call must specify that `x` is dirty.

In general, the release calls are not inserted into the local transformation, but are placed on a list kept by the compiler. At some point, often at the statement level, these calls are inserted. Release calls usually cannot be inserted locally because the result would be incorrect code. Consider again the expression `x + y` where `x` is a persistent dbint. In order to form the sum, the expression `x` must be transformed into an expression which pins `x` in the buffer pool, then produces `x`'s value. The translated code looks something like[17] the following:

```
        (<read call>, <deref expr>)
```

If the release call were inserted right after the read call, then the dereference would be illegal because the data has

[17]In C, an expression, *e*, may comprise two expressions, *e1* and *e2*, separated by a comma: *e ::= e1 , e2*. The expressions are evaluated left to right, and the type and value of *e* are those of *e2* [Kern78].

been unpinned. If the call were inserted after the dereference, then the comma expression would no longer produce the value of `x` as its result.

A third possibility is to introduce a local temporary and copy the bytes out of the buffer pool.

```
(<read call>, _tmp = <deref expr>, <release call>, _tmp )
```

The release call can be inserted as shown, with the temporary finally producing the correct value. This approach works, and in fact is necessary in several places. In general, however, the E code generator tries to take advantage of an important performance feature of the EXODUS storage manager: the resulting C program accesses the data directly in the buffer pool when possible, avoiding the additional copy operation.

Because the current code generator is an initial prototype implementation, the code it produces is not yet optimized. Local transformations usually result in more storage manager calls than are necessary, and, in fact, it is possible to pin data in the buffer pool redundantly. For example, in the assignment, `x = x + y`, `x` is simultaneously pinned for both occurrences. While the Storage Manager allows such redundant pinning of data (since it supports sharing in general), clearly the performance of E programs will be significantly improved by eliminating such redundancies. These and other optimizations are the subject of current study and are not discussed further here.

4.2.3.1. Names

A simple name expression forms the basis of the code generator's recursive descent. Figure 9 shows the transformation of a name node. If the name is of a non-db type object, no action is taken.

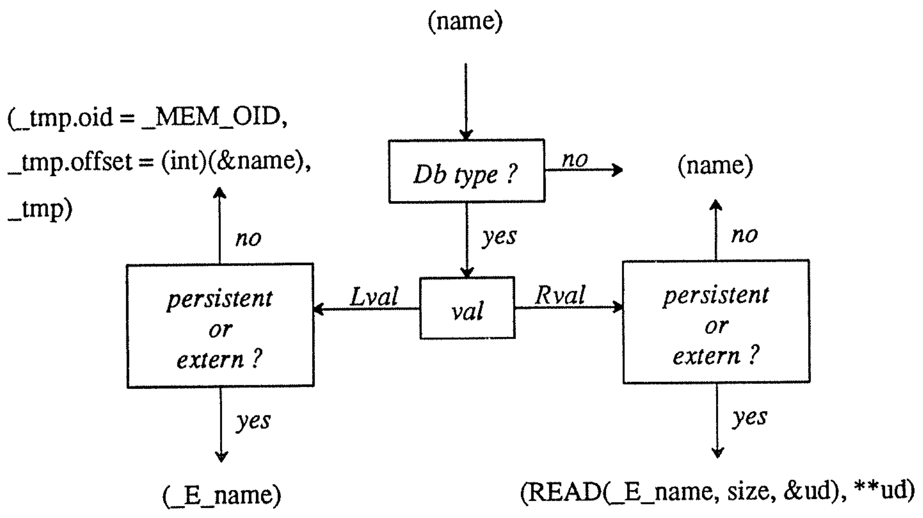


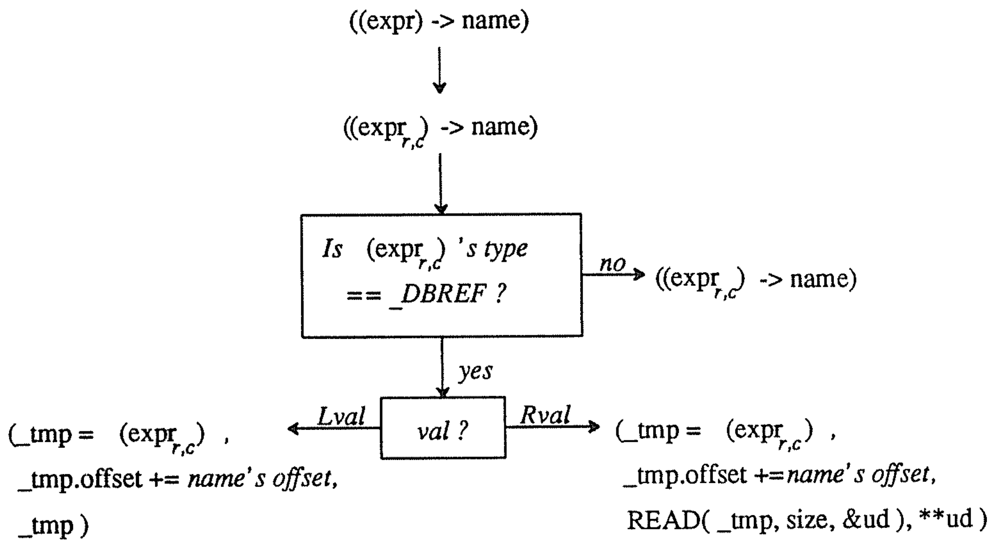
Figure 9: Transforming a Name Expression

As outlined in the previous section, if the node refers to a persistent or external object and the Lval is desired, then the name of the object's companion is substituted. The result is an expression yielding the address of the object. If the Rval is wanted, we substitute a call to the storage manager read routine, using the companion to obtain the OID and offset, and using the object's type to obtain the length. We also generate a release call (not shown here) whose mode (either dirty or clean) is determined by the second parameter to this invocation.

If the node refers to a nonpersistent db type object and the Rval is needed, no action is taken. Note that this meshes with the treatment of a nonpersistent object's declaration. That is, if we declare and use a nonpersistent dbint in a local scope, the translated code simply declares and uses an integer. Finally, the Lval of a nonpersistent, db type object is generated when needed by introducing a temporary _DBREF variable into the local scope and transforming the expression into one which first initializes the _DBREF and then produces its value as the result.[18]

4.2.3.2. Dereferencing

The operators dot (.), arrow (->), and star (*) are simple to handle. Consider the pointer dereferencing operator, arrow, whose translation is shown in Figure 10. The expression on the left of the arrow results in a pointer to a structure, and the name on the right specifies a field in the structure. Translation of this node first transforms the expression on the left. Since we need the value of the pointer (not its address), we request the pointer's Rval, and



((expr) -> name)
((expr$_{r,c}$) -> name)
Is (expr$_{r,c}$) 's type == _DBREF ?
no
((expr$_{r,c}$) -> name)
yes
val ?
Lval
Rval
(_tmp = (expr$_{r,c}$) , _tmp.offset += name's offset, _tmp)
(_tmp = (expr$_{r,c}$) , _tmp.offset += name's offset, READ(_tmp, size, &ud), **ud)

Figure 10: Transforming an Arrow Expression

[18]The alternative is to initialize, at the beginning of a block, a distinguished companion for each nonpersistent db type object declared in that block. This approach was deemed too expensive in general.

since the expression only reads the pointer value, the release call (if generated) should specify "clean." In the figure, the expression resulting from this recursive descent is denoted by appending subscripts: $expr_{r,c}$ is the expression resulting from the transformation requesting "Rval" and "clean." If the result of transforming the pointer expression is not an expression of type _DBREF, then the pointer was not a db pointer, and no further action is taken. Otherwise, if the Lval of the current expression is required, as in `&(p->x)`, then we produce a new _DBREF in which the offset of the pointer expression is incremented by the offset of the field `x` in the structure. If the current expression's Rval is required, we first produce its Lval, and then use this _DBREF in a read call to the Storage Manager. The size of `x`'s type determines the length parameter for the call. The other dereferencing operators, dot and star, are handled similarly.

One small optimization has been implemented for Lvalued expressions and is worth noting here. The illustration in Figure 10 shows that the result of the expression on the left side of the arrow is copied into a temporary _DBREF variable. In fact, if the result of the pointer expression is already held in a temporary, that variable is simply "promoted" to hold the Lvalue of the current expression, possibly with its offset incremented by the appropriate amount. In terms of the illustration, this step eliminates the copy: `_tmp = ` $(expr_{r,c})$.

4.2.3.3. Arithmetic Expressions

Processing arithmetic (and other) binary operators is particularly easy. As illustrated in Figure 11, we simply transform both operand expressions, specifying the Rval and "clean" in both cases. The only slight complication arises in handling pointer arithmetic. If, for example, after transforming the operands of plus (+), we discover that one of them resulted in a _DBREF value, then the expression represents arithmetic on a db type pointer. In a

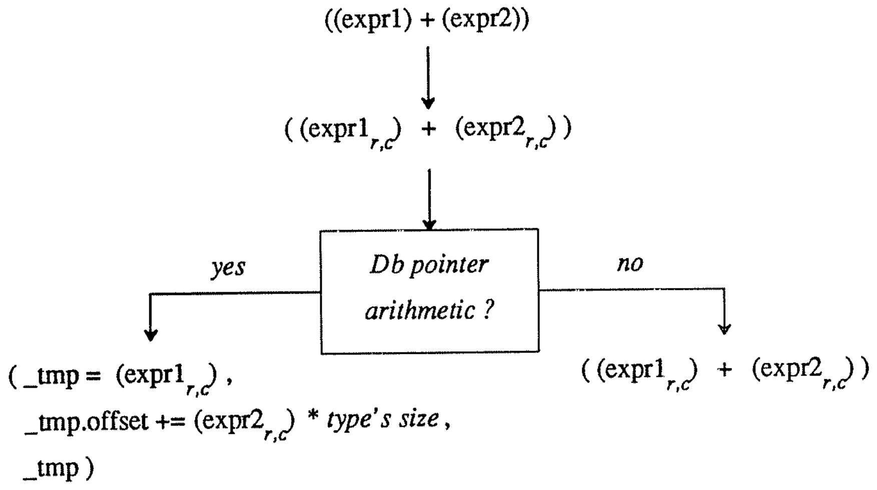

((exprl) + (expr2))
((exprl_{r,c}) + (expr2_{r,c}))
Db pointer arithmetic ?
yes
no
(_tmp = (exprl_{r,c}),
_tmp.offset += (expr2_{r,c}) * type's size,
_tmp)
((exprl_{r,c}) + (expr2_{r,c}))

Figure 11: Transforming an Arithmetic Expression

$$\begin{tikzcd}((\mathrm{expr1}) = (\mathrm{expr2})) \arrow[d] \\ ((\mathrm{expr1}_{r,d}) = (\mathrm{expr2}_{r,c}))\end{tikzcd}$$

Figure 12: Transforming an Assignment Expression

manner similar to the last section, we produce a new _DBREF expression in which the offset is incremented. This time, in accordance with C++ semantics, the increment is the product of the right operand times the size of the referenced type. Since pointers cannot be added, the operands in this example cannot both produce _DBREFs. However, two pointers *may* be subtracted, and the result is their integer difference divided by the size of the referenced type (as described in [Kern78]). If those pointers are db pointers, the difference is formed by subtracting the offsets in the _DBREFs.

4.2.3.4. Assignment

As a final example, assignments are trivially handled as shown in Figure 12. To transform the left hand side of the assignment, note that we could request its Lval, and if this produces a _DBREF, we could then insert a read call to pin the destination of the assignment. We can achieve the same effect more simply, however, by requesting the Rval of the left hand side; the proper read and dereference are then added automatically at a lower level. As mentioned previously, here we must specify a dirty release for the left hand side, and a clean release for the right.

4.2.3.5. Statements

It was mentioned previously that the release call corresponding to a given read is usually not inserted into the local expression transformation. However, there are certain situations, particularly in transforming statements, where this insertion is necessary. Consider the return statement, and suppose a function `f` simply returns the value of `x`, a global persistent integer.

```
int f( )
{
     return x;
}
```

Clearly, we must read `x` in order to return its value. It is in releasing `x` that we have a small problem. Obviously, we cannot release something before it is read. Without added complexity, we also cannot release it after the return, i.e. in the caller. Therefore, we have chosen to insert the release call within the return expression, and this requires the introduction of a temporary. The return statement thus looks something like:

```
        return (read(x, &ud), _tmp = **ud, release(ud), _tmp);
```

Other statements containing expressions are treated similarly. For example, an `if` statement translates to:

```
        if( _tmp = <transformed expr>, <release calls>, _tmp )
        { ... }
```

We should emphasize again that such temporaries are introduced only if necessary, i.e. if the expression pins data. Expressions involving only non-db types never suffer this overhead.

4.2.4. E Runtime Environment

Programs generated by E contain calls to the EXODUS Storage Manager. In our first implementation, E programs are actually linked with a copy of the Storage Manager. This is similar to PS-Algol, in which each program includes a copy of the CPOMS storage level [Brow85]. Efront itself also contains a copy of the Storage Manager, since it may need to create persistent objects at compile time. Any program that uses the Storage Manager must first call its initialization routines, specifying which database volume(s) must be mounted. Currently, E programs use only one volume, and its Unix file name is obtained from the environment variable, EVOLUME. Other parameters describing the runtime context are likewise given default values in the first implementation. For example, most Storage Manager calls also require a transaction id and a buffer group specifier (see footnotes 1 and 2). For now, programs run as a single transaction using a single buffer group.

A number of improvements will be made to this runtime environment in the near future. The Storage Manager, along with other support routines, will run as a server.[19] E programs will be dynamically loaded into the server, rather than each containing its own copy of the Storage Manager. The association between E programs and database volumes will be implicit because the programs themselves will be stored in the server. The EVOLUME environment variable, currently the "weak link", will no longer be necessary. The programmer will control transactions via a `transaction` block in the E language, and will be able to associate different buffer groups with different parts of the code. We are currently investigating means by which the latter may be specified.

4.2.5. Supporting Other C++ Features

So far, we have described the general scheme that we have used for implementing persistence in E. We described our solutions to the problems of creating persistent objects, binding objects to program symbols, and generating code for expressions. This section discusses some further problems that are specific to extending the semantics of C++ to the world of persistent objects. Part of the solution depends on the new environment in which E programs will be compiled and run. For this paper, we will describe certain of its features as needed.

[19]In fact, a prototype of this server ran in the SIGMOD-88 EXODUS demonstration.

4.2.5.1. Constructors & Destructors

Earlier, we outlined how the compiler processes the declaration of a persistent object for which an initializer has been specified. A more interesting problem arises when a persistent object is declared of a dbclass with a constructor. Consider again the stack class definition in Figures 2 and 3. The stack constructor initializes the top-of-stack index to -1, indicating that the stack is empty. By definition, a constructor is called when the object is created. Suppose, as in Figure 3, a program declares a persistent stack. Since the compiler creates the persistent object, it would appear that the compiler must also invoke the constructor. But how is this to be accomplished? The compiler, that is, efront, has only the abstract syntax tree for the program. Should we write an interpreter? In general, a constructor may call other functions arbitrarily; what if those functions are externally defined? In fact, unlike this example, it may well be that the constructor itself has not yet been defined but only declared.

In the first implementation of E, the problem is handled as follows. Observe that is it not strictly necessary to call the constructor at compile time. Rather, it is sufficient to ensure that the object is initialized before any program actually uses it, and that it is initialized only once. A very slight extension to an existing cfront mechanism provides a simple implementation satisfying these conditions. Before describing how constructors are called on persistent objects, then, we first review the mechanism used in cfront for initializing static objects.

Consider for the moment the non-db stack implementation in Figure 2, and suppose that the stack class and the stack `S` are defined in a module *m* separate from the main program. (The term "module" is used to emphasize that C++ programs are usually composed of separately compiled pieces.) Any program which includes *m* as one of its components must be sure to initialize `S` before the main program begins. In the general case, a given program comprises a set of modules, M, each of which contains a set, X_m, of objects needing initialization. The approach adopted in the AT&T C++ compiler involves first generating, as part of the C translation, an initializer function, f_m, for each module in M. (Obviously, modules for which X_m is empty do not need an initializer. We do not consider this case further.) The initializer function simply calls the appropriate constructor for each object in X_m:

```
f_m {
        constructor( x_1, args_1 );
        ...
        constructor( x_|X|, args_|X| );
}
```

In the case of the nonpersistent stack example, the initializer for module *m* would look something like[20]:

```
void _STI_m() {
        _stack_ctor( &S );
}
```

The first action of every C++ main program is to call the initializer function for every module in M. We shall omit

[20]The actual name of the function is the name of the source file prepended with "_STI", for STatic Initializer.

the details of how these functions are bound to the calls made by the main program. The compilation steps enclosed in the dashed box in Figure 5 accomplish this binding.

Now, an E source module m contains, in addition to X_m, a set P_m of persistent objects, each of which is of a dbclass with a constructor. To implement the desired semantics, efront amends the initialization function, f_m, as follows:

```
f_m {
        persistent BOOL init = TRUE;
        if ( init ) {
                init = FALSE;
                constructor ( p_1, args_1 );
                ...
                constructor ( p_|P|, args_|P| );
        }
        constructor ( x_1, args_1 );
        ...
        constructor ( x_|X|, args_|X| );
}
```

When f_m is called for the first time, the persistent flag has the value TRUE. The flag is then cleared and constructors are invoked on the persistent objects in m. If the same program is run again, or even if another program containing m is run, these constructors will not be called again because the flag is itself persistent and shared by all programs that include m.

Once the basic E persistence mechanism was working, this solution was trivial to implement. Like most first solutions, though, it has several drawbacks. First, testing the persistent flag requires a read call to the storage manager. For a program consisting of n modules, this implies a startup cost of as many as n disk reads.[21] Furthermore, a given module containing such a persistent flag will contribute one storage manager call to the startup cost of every run of every program in which it is ever used. Ideally, we would like to pay the initialization cost once, and thereafter incur no extra overhead.

Another shortcoming of this solution is that it does not extend to destructors. A destructor is the inverse of a constructor. Whenever an object of a class is destroyed, e.g. by going out of scope, the class's destructor (if it exists) is called first. A named persistent object is destroyed by deleting the module containing its persistent handle (companion). If that object is of a dbclass with a destructor, we should, to be consistent, call that destructor. The solution for constructors cannot be applied here because, while a flag can identify the *first* time the module is used, it cannot signal the *last*. Instead, there is a utility program, *erm*, which is used to destroy modules containing persistent handles. The problem with the current *erm* utility is that it requires the user to specify explicitly all the object

[21]However, given that a flag resides on the same page as other persistent objects in the module, at least some of these pages would presumably have been requested shortly anyway.

code modules required for calling the destructors.

The above problems derive from a common source. While the first implementation of E maintains persistent *objects*, it does not maintain persistent *types*. That is, the current system does not maintain the association between a persistent object and its type, e.g. the code implementing its methods. The next implementation of E will operate in the context of the new environment to which we have alluded, and it will enforce the following rule: *The lifetime of every persistent object of type T must be subsumed by the lifetime of an implementation of T.* When a persistent object, `x`, of type `T` is declared, the environment will allow the compiler to identify which implementation of `T` is being used, and to bind `x` to that implementation. Note that this rule does not specify "the same" implementation of `T` over the lifetime of `x`. (We wish to allow a careful user to modify an implementation without necessarily invalidating existing objects.)

The above rule implies that the new environment will impose somewhat more structure on the way programs are built. Currently, for example, a source module includes (via `#include`) the header files containing the definitions of needed classes. Calls to member functions of such classes are typically left unbound until the application module and the class implementation modules are linked together. Under the new environment, source modules will *use* rather than include one another. Use is a semantically richer form of include. In particular, if several source modules all use a given source module, *m*, they *share* that module. Under `#include`, each would receive an independent, inline copy of *m*.

Returning to the problem of constructors, destructors, and persistent objects, the new environment will enable the compiler to verify that the declaration of a persistent object uses an implementation of the object's type (and not simply a declaration). The result is that the constructor for a persistent object will be called when the object is created, eliminating the need to test a persistent flag at startup time. Similarly, when we destroy a module containing persistent objects, we can then locate and call destructors automatically. The details will be covered in [Rich88b].

4.2.5.2. Virtual Functions

The C++ mechanism which supports "true" object-oriented behavior — the late binding of code to a method invocation — is the *virtual* function. If a member function of a class is declared virtual, then the runtime calling sequence involves indirection through a dispatch table. In the AT&T C++ compiler, there is one such table for each class having virtual functions, and every object of the class contains a pointer to that table. Thus, the dispatch table is a kind of runtime type descriptor, and the embedded pointer plays the role of a type tag for each object. The amount of type information known to the compiler allows for a very fast implementation: A virtual function call adds at most one pointer dereference and one indexing operation to the cost of a normal procedure call.

When virtual functions are combined with persistence, the above implementation no longer suffices. Clearly, we cannot store the memory address of the dispatch table because that address is valid only for one program run. One approach is to make the dispatch table a persistent object. Then the addresses embedded in objects will be valid persistent addresses. This is the solution adopted in Vbase, for example [Andr88]. For E, however, this approach only pushes the problem back one step. The virtual functions themselves will be located at different addresses in different programs, and so the persistent dispatch table must be filled in when it is loaded. Furthermore, it leaves unresolved the addresses of other functions that may be called by the virtual functions. Since the actual dispatch table used in a given program is, in effect, specific to that program, there seems to be little benefit in making it persist.

For these reasons, we have implemented a different solution. For every dbclass C having virtual functions, the compiler generates a unique integer type tag, and every instance of C contains this tag. The dispatch tables are still main memory objects, and in addition, we introduce a global hash table (also a main memory object) for mapping type tags to dispatch table addresses. This table is initialized at program startup[22]; for each dbclass in the program having virtual functions, we enter its type tag and dispatch table address. The existing static initializer mechanism described in the previous section is used to initialize the hash table. Then, to call a virtual function at runtime, we hash on the type tag in the object to obtain the dispatch table address and proceed as before from there.

A problem that arises in this implementation is the management of type tags. Specifically, we must be able to distinguish the first use of a type from subsequent uses; in the former case, we must generate a new type tag, and in the latter, we must reuse the existing type tag. Obviously, name space management is a related issue, since types that happen to have the same name are not necessarily the same type. Once again, the current solution provides an initial implementation that will be improved in the next version. The compiler keeps a persistent table associating type names (character strings) with tags. Before generating a new tag for a type T, the compiler searches the table. If T's name is found, the associated tag is used. Otherwise, a persistent counter is incremented, generating a new tag, and a new entry is made in the table. Obviously, this solution disallows having two dbclasses with the same name (in different programs, of course), where both classes have virtual functions. The new environment for E will eliminate this minor restriction.

5. SUMMARY

5.1. Review

This paper has presented the design of the first implementation of the E compiler. We briefly reviewed E's place in the context of the EXODUS project and its client relationship with the EXODUS Storage Manager. We

[22]Unlike the (initial) constructor solution described above, this startup cost is negligible.

showed by example how one declares and manipulates persistent objects in E, emphasizing the ease with which one can convert a nonpersistent application into a persistent one.

Next we discussed at some length a number of the important issues that arise in the design and implementation of a persistent language. The topics discussed included the organization of persistent name spaces, the representation of persistent objects and their addresses, and the management of physical I/O at runtime. For each issue covered, we presented the approach taken in E, contrasting our solution with those of other systems.

The remainder of the paper then described in detail the compiler's current implementation. We began by showing at a macro level how we integrated E language extensions into an existing C++ compiler. We then described how the E compiler processes persistent object declarations, paying particular attention to how those objects are created and initialized. Next we described the current prototype code generator which converts expressions involving persistent objects into equivalent expressions involving address calculations and calls to the Storage Manager. Finally, we described our current solutions to the special problems of handling constructors, destructors, and virtual functions in the persistent context.

5.2. Relationship to Other Work

Like other persistent languages and object-oriented database systems, E reduces the impedance mismatch between the programming language and the persistent store (or database system). However, E is distinguished from other work in several ways. First, E is a direct extension adding persistence to an existing systems level programming language, C++. All of the concepts and semantics of the base language have been preserved in E. The addition of db classes, for example, is really a minor syntactic extension. All of the concepts associated with C++ classes, e.g. constructors and virtual functions, are supported by E dbclasses. Also, the pointer manipulation facilities familiar to C and C++ programmers are preserved in the persistent world. A db pointer may be cast, incremented, etc. This aspect of E's design contrasts sharply with Vbase and O2. Both of these systems embed within C an interface to an object-oriented database system. In Vbase, types are defined in one language (TDL) and methods are defined in another (COP). In O2, the CO2 language contains special type constructors and embedded "escapes" to the distinct message passing layer of the language. In both cases, persistent types and objects are much different from their counterparts in the host language.

Another distinguishing feature of E is in its approach to physical I/O. By viewing the storage manager as a load/store machine, E departs from the familiar virtual memory model used in PS-Algol and elsewhere. By scheduling loads and stores (pins and unpins), the E compiler realizes a number potential benefits. First, while object faulting implies loading the whole object, E pins only the portion of the object accessed by the program. When objects are small, the two methods are similar, but when objects are large, E should make significantly better use of available buffer space. This economy should be especially important in a multiuser environment. Secondly, the compile time scheduling of loads and stores provides the opportunity to apply optimizations. We expect to realize significant improvements in the performance of E programs in this way.

5.3. Status & Current Research

In addition to being a learning vehicle, the current compiler provides a demonstration of feasibility of the general approach to persistence taken in E. The db-simplification phase (and a number of supporting routines) were designed and added to the compiler during the Spring of 1988. The E compiler runs on a VAXstation III under Unix 4.3, and was an integral part of a demonstration given at SIGMOD-88. At this point, the full cross product of dbclasses and persistence works correctly. Other features of E not covered in this paper, i.e. generators and iterators, also compile correctly. A test suite developed by a student over the summer has greatly aided in debugging the compiler (and in building our confidence in it).

The design of a new environment to support E program management is nearing completion. This environment will address several issues which are not currently well-handled. These issues include a more integrated runtime environment, and the protection and control of source code. It will also allow a much more efficient implementation of constructors for persistent objects, as well as an equally efficient implementation of destructors.

The major research direction related to E is now in the optimization of E code. These optimizations will be applied to a number of areas. We seek to reduce drastically the the number of calls to the Storage Manager, both locally within an expression and globally within a procedure. The optimizations we seek are related to existing techniques such as the elimination of unnecessary loads and stores, global register allocation, and loop optimizations. However, because the "machine" for which E is targeted (i.e. the EXODUS Storage Manager) is quite different from a typical hardware machine (e.g. one with a fixed number of word-sized registers), optimizing E code is a research problem rather than a straightforward application of known techniques. For example, one potential new optimization is called "coalescing"; if several different pieces of the same object are pinned within the same or nearby regions of code, those separate requests could be combined into one request which pins a single spanning range of bytes. Whether coalescing is worthwhile depends both on the distance between the two ranges within the object and on the distance between the uses of those ranges within the program. Another important area for optimization will be processing arrays in block-at-a-time fashion, rather than one element at a time. We are looking at techniques used in vectorizing compilers (e.g. [Padu80]) for inspiration. After an appropriate set of optimizations have been chosen and implemented, we plan to investigate the performance of E code relative both to existing DBMSs and to systems that rely on object faulting.

ACKNOWLEDGEMENTS

While this paper has concentrated on the implementation of persistence in E, we should note that the other half of the implementation effort — generator classes and iterators — was the work of Dan Schuh. For that, and for his fearless journeys into the original source code, he has our deep gratitude. We would also like to thank the other members of the EXODUS project for their continued support.

REFERENCES

[Andr87] Andrews, T., and Harris, C., "Combining Language and Database Advances in an Object-Oriented Development Environment," *Proc. of the Conf. on Object-Oriented Programming Systems, Languages and Applications*, Orlando, FL., October, 1987.

[Andr88] Andrews, T., private communication, June, 1988.

[AtkM83a] Atkinson, M.P., et. al., "An Approach to Persistent Programming," *Computer Journal*, 26(4), 1983.

[AtkM83b] Atkinson, M., et. al., "Algorithms for a Persistent Heap," *Software — Practice and Experience*, Vol. 13, 1983.

[AtkM85a] Atkinson, M., and Buneman, O. Peter, "Database Programming Language Design," University of Glasgow, *Persistent Programming*, R.P.#17, 1985.

[AtkM85b] Atkinson, M., and Morrison, R., "Types, Bindings and Parameters in a Persistent Environment," *Proc. of the Appin Workshop on Persistence and Data Types*, Glasgow, Scotland, August 1985.

[AtkM85c] Atkinson, M., and Morrison, R., "Procedures as Persistent Data Objects," *ACM Trans. on Programming Languages and Systems*, 7(4), October, 1985.

[AtkM87] Atkinson, M., and Buneman, O.P., "Types and Persistence in Database Programming Languages," *ACM Computing Surveys*, 19(2), June, 1987.

[AtkR78] Atkinson, R., Liskov, B., and Scheifler, R., "Aspects of Implementing CLU," *ACM National Conf. Proc.*, 1978.

[Banc88] Bancilhon, F., et. al., "The Design and Implementation of O2 an Object-Oriented Database System," Rapport Technique *Altair* 20-88, April, 1988.

[Bane87] Banerjee, J., Kim, W., Kim, H-J., and Korth, H., "Semantics and Implementation of Schema Evolution in Object-Oriented Databases," *Proc. ACM-SIGMOD Int'l Conf. on Management of Data*, San Francisco, CA, 1987.

[Brow85] Brown, A.L., and Cockshott, P., "The CPOMS Persistent Object Management System," Persistent Programming Research Report #13, 1985.

[Butl87] Butler, M., "Storage Reclamation in Object-Oriented Database Systems," *Proc. ACM-SIGMOD Int'l Conf. on Management of Data*, San Francisco, CA, 1987.

[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.

[Care88] Carey, M., DeWitt, D., and Vandenberg, S., "A Data Model and Query Language for EXODUS," *Proc. ACM-SIGMOD Int'l Conf. on Management of Data*, Chicago, IL, 1988.

[Cock84] Cockshott, W.P, et. al., "Persistent Object Management System," *Software—Practice and Experience*, 14(1), 1984.

[Cock87] Cockshott, W.P., "Stable Virtual Memory," *Proc. of the Workshop on Persistent Object Systems: Their Design, Implementation, and Use,* Appin, Scotland, 1987.

[Coop88] Cooper, R., personal communication, August, 1988.

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

[Grae87a] Graefe, G., and DeWitt, D., "The EXODUS Optimizer Generator," *Proc. ACM-SIGMOD Int'l Conf. on Management of Data*, San Francisco, CA, 1987.

[Grae87b] Graefe, G., "Rule-Based Query Optimization in Extensible Database Systems," Ph.D. Thesis, University of Wisconsin, Madison, WI, August, 1987.

[Gold83] Goldberg, A., and Robson, D., *Smalltalk-80: The Language and Its Implementation,* Addison-Wesley, 1983.

[Kaeh83] Kaehler, T., and Krasner, G., "LOOM - Large Object Oriented Memory," in *Smalltalk-80: Bits of History, Words of Advice*, Addison-Wesley, 1983.

[Kaeh86] Kaehler, T., "Virtual Memory on a Narrow Machine for an Object-Oriented Language," *Proc. of the Conf. on Object-Oriented Programming Systems, Languages and Applications*, Portland, OR, 1986.

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

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

[Maie87] Maier, D., and Stein, J., "Development and Implementation of an Object-Oriented DBMS," in *Research Directions in Object-Oriented Programming*,, B. Shriver and P. Wegner, Eds., MIT Press, 1987.

[Maie87b] Maier, D., personal communication, November, 1987.

[Morr82] Morrison, R., "Toward Simpler Programming Languages," *IUCC Bulletin*, 4(3), October, 1982. Quoted in [AtkM83].

[Onto87] *Vbase Technical Notes*, Ontologic Corporation, 1987.

[Padu80] Padua, D., Kuck, D., and Lawrie, D., "High-Speed Multiprocessors and Compilation Techniques," *IEEE Transactions on Computers*, C-29(9), September, 1980.

[Penn87] Penny, D.J., and Stein, J., "Class Modification in the GemStone Object-Oriented DBMS," *Proc. of the Conf. on Object-Oriented Programming Systems, Languages and Applications*, Orlando, FL, October, 1987.

[Rich87a] Richardson, J., and Carey, M., "Programming Constructs for Database System Implementation in EXODUS," *Proc. ACM-SIGMOD Int'l Conf. on Management of Data*, San Francisco, CA, 1987.

[Rich87b] Richardson, J., Carey, M., DeWitt, D., and Schuh, D., "Persistence in EXODUS," *Proc. of the Workshop on Persistent Object Systems: Their Design, Implementation, and Use*, Appin, Scotland, 1987.

[Rich88a] Richardson, J., and Carey, M., "The Design of the E Programming Language," in preparation.

[Rich88b] Richardson, J., Schuh, D., and Carey, M., "Environment Support for E," in preparation.

[Rich88c] Richardson, J., "The E Reference Manual," EXODUS Working Document, University of Wisconsin, Madison, 1988.

[Skar86] Skarra, A., and Zdonik, S., "The Management of Changing Types in an Object-Oriented Database," *Proc. of the Conf. on Object-Oriented Programming Systems, Languages and Applications*, Portland, OR, 1986.

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

APPENDIX: AN EXPANDED EXAMPLE

This appendix gives a flavor of the rest of the E language. The stack class definition in Figure 13 expands on the simple example presented in the paper, illustrating three other important features of E: generic classes, iterators, and file variables. A generic class is defined in terms of type parameters, and a specific class is instantiated from the generic one by providing actual types for the parameters. (Formal parameters of classes may also be functions or simple constants, although we do not show such cases here.) The main advantage to a generic class is that the code needed to implement the class is written only once, and instantiated classes then share the generic code. An iterator is a control construct comprising two cooperating agents: an iterator function and an iterate loop. The function and the loop play the roles of producer and consumer, respectively, in processing a sequence of values. Both generic classes and iterators were inspired by CLU [Lisk77]. Finally, file variables, which were mentioned in the paper, provide the E programmer with bulk storage and with dynamic creation of persistent objects.

Let us walk through this example, examining each of the new features. Class formal parameters are specified within square brackets immediately following the class name. The stack class is defined in terms of one formal parameter, `eltype`, which is the (unknown) type of the elements to be placed on the stack. As in CLU, one may specify constraints on types used as actual parameters; in this case, any instantiating type must be a dbclass having a print function of no arguments. This function will be used by the stack print routine to print each of the elements in the stack.

The stack is to be implemented as a linked list. Within the body of the class specification, we first define an auxiliary type, `stkNode`, which is a structure containing a data field of the element type and a pointer to the next stack node.[23] Following the definition of `stkNode` are the data members of `stack`. Each stack instance consists of a pointer to the top stack element and a file containing all the stack nodes. Here we have used the auxiliary type to define the type of the pointer, `top`, and the predefined type `file` to define the type of `stkFile`.

The list of stack member functions follows the `public` keyword. The stack constructor initializes the top of stack pointer to null, indicating an empty stack; the method `empty` simply tests for a null pointer. The `push` routine takes a reference to an `eltype` object, creates a new node, places it at the head of the linked list, and copies in the data value. The node is created within `stkFile` by invoking the file method, `new_obj`. This method takes a size parameter, creates an object of that size in the file, and returns a pointer to the new object.[24]

[23]Note that E deviates from C++ semantics here. C++ allows nested class definitions but this is merely a syntactic device; the nested class is exported to the scope of the containing class [Stro86]. However, in order to support formal type parameters, E must limit their scope; `eltype` would be meaningless outside the scope of `stack`. The need to define auxiliary types, such as `stkNode`, is another reason for supporting nested class scoping.

[24]We note that the file class used in this implementation creates essentially untyped objects; casting is then used, as in `push`, to manipulate them. We are now adding to E a generic form of file, `fileof[T]`, which contains typed objects, i.e of type `T` or its subtypes.

```
/***********************************************************/
/* Here's the generic class def. */

#include <file.h>
dbclass stack
[
    /* type parameter: any dbclass with a print routine */
    dbclass eltype {
    public:
          void print();
    }
]
{
    /* auxiliary type of stack node */
    dbstruct stack_node {
          stack_node * nxt;
          eltype       data;
    };

    /* class data members */
    stack_node *  top;        // points to top element
    file          stkFile;    // holds all the stack elements

    /* class member functions */
public:
                  stack();          // constructor
    int           empty();          // is stack empty?
    void          push(eltype&);    // push an element
    eltype        pop();            // pop an element
    void          print();          // print all elements

private:
    eltype * iterator elements();   // yield pointer to each element
};

/***********************************************************/
/* Here are the stack class methods. */

stack::stack()
{
    top = NULL;
}

int stack::empty()
{
    return (top == NULL);
}

void stack::push(eltype& item)
{
    stack_node* p;

    /* create a new object in the file and put it at head of list */
```

```
        p = (stack_node*) stkFile.new_obj(sizeof(stack_node));
        p->nxt = top;
        top = p;

        /* assign the data value */
        p->data = item;
    }

    eltype stack::pop()
    {
        stack_node*    p;
        eltype         ret;

        if(top != NULL){
                /* save current top. Pop. Then delete (old) top. */
                p = top;
                top = top->nxt;
                ret = p->data;
                stkFile.del_obj(p);
                return ret;
        }
        else
                printf("stack::pop : Stack is empty.");
    }

    iterator eltype * stack::elements()
    {
        /* walk down list, yielding address of each data field */
        for(stack_node* p = top; p != NULL; p = p->nxt)
                yield &(p->data);
    }

    void stack::print()
    {
        /* invoke iterator. call eltype::print() for each element */
        iterate( eltype * p = this->elements() )
                p->print();
    }
```

Figure 13: A Generic, Unbounded Stack Example

The pop routine removes the top stack element by calling the file method, `del_obj`; the data value in the old top node is returned to the caller. The file `del_obj` method takes a pointer to an object in the file and destroys that object. Finally, in order to illustrate the definition and use of an iterator, the stack implementation includes an iterator function, `elements`, which is used by the `print` method. The iterator simply walks down the linked list, yielding a pointer to the data field of each node in turn. The iterate loop within `print` picks up each such pointer and then invokes the print routine (of `eltype`) on each referenced object. Note that since the iterator is declared `private`, it may be used only by other `stack` methods (such as `print`).

```
/* a class defining a memo type */
dbclass memo
{
    /* some appropriate representation */

public:
            memo( char* );      // constructor
    void    print();            // a print routine
};

/* instantiate a type to hold stacks of memos */
dbclass memoStack : stack[ memo ] ;

/* declare a persistent memo stack */
persistent memoStack S;

main(int argc, char** argv)
{
    for( int i = 1; i < argc; i++ )
    {
      memo  m( argv[i] );
      S.push( m );
    }
    S.print();
}
```

Figure 14: Using the Stack Class

A generic class is not in itself usable by an application; it must first be instantiated to a specific type. Figure 14 shows a simple program that keeps a stack of personal memoranda. We only show a skeletal outline of the dbclass `memo`. It has some representation, a constructor, and a print function. The constructor takes a character string argument forming the body of the memo. Next, we instantiate the type `memoStack` from the generic stack class and declare a persistent object, `S`, of this type. Since the memo print function has the type signature specified in the formal parameter section of `stack`, this instantiation is legal. Finally, the (simplistic) main program builds a memo, `m`, out of each argument on its command line and pushes `m` onto `S`. It then prints all of the memos on the stack.

15. Appendix 1: The Cost Functions

Query 1 (Ancestor.bf)

1.1 Naive evaluation	$D\sum_{i=1}^{h}(h-i+1).a(i) + E.gsum(E,h'-1).$
1.2 Semi-Naive Evaluation	$D\sum_{i=1}^{h}a(i) + E.gsum(E,h'-1).$
1.3 QSQ, Iterative	$E.gsum(E,h'-1) + F.\sum_{i=1}^{h'}(h'-i+1).i.E^{i-1}$
1.4 QSQ, Recursive	$(F+E).gsum(E,h'-1) + D\sum_{i=1}^{h'}E^{i}.gsum(E,h'-i)$
1.5 Henschen-Naqvi	$(F+E).gsum(E,h'-1)$
1.6 Prolog	$gsum(F,h') + E.gsum(E,h'-1) + \sum_{i=1}^{h'}(F^{i}).gsum(F,h'-i)$
1.7 APEX	$(F+E).gsum(E,h'-1) + D\sum_{i=1}^{h'}E^{i}.gsum(E,h'-i)$
1.8 Kifer-Lozinskii	$D\sum_{i=1}^{h}a(i)+E.gsum(E,h'-1)$
1.9 Magic Sets	$(F+E).gsum(E,h'-1) + D\sum_{i=1}^{h'}E^{i}.gsum(E,h'-i)$
1.10 Counting	$(F+E).gsum(E,h'-1)$

Query 2 (Ancestor.fb)

2.1 Naive evaluation	$D\sum_{i=1}^{h}(h-i+1).a(i) + (1/E).gsum(1/E,h-h'-1)$
2.2 Semi-Naive Evaluation	$D\sum_{i=1}^{h}a(i) + (1/E).gsum(1/E,h-h'-1)$
2.3 QSQ, Iterative	$(1/E).gsum(1/E,h-h'-1) + D.\sum_{i=1}^{h-h'}(h-h'-i+1).i.(1/E)^{i-1}$
2.4 QSQ, Recursive	$1 + (1/E).gsum(1/E,h-h'-1) + F.\sum_{i=1}^{h-h'}(1/E)^{i}.gsum(1/E,h-h'-i)$
2.5 Henschen-Naqvi	$(D+1/E).gsum(1/E,h-h'-1)$
2.6 Prolog	$(1/E).gsum(1/E,h-h'-1) + \sum_{i=1}^{h}n(i).gsum(F,h-i)$
2.7 APEX	$(1/E)^{(h-h')}.(E.gsum(E,h-1)+D\sum_{i=1}^{h}E^{i}.gsum(E,h-i))$

2.8 Kifer-Lozinskii	$(D+1/E).\mathrm{gsum}(1/E,\ h-h'-1)$
2.9 Magic Sets	$1 + (1/E).\mathrm{gsum}(1/E,h-h'-1) + F.\sum_{i=1}^{h-h'} (1/E)^i.\mathrm{gsum}(1/E,h-h'-i)$
2.10 Counting	$(D+1/E).\mathrm{gsum}(1/E,h-h'-1)$

Query 3 (Ancestor.bf, Non-Linear Version)

3.1 Naive evaluation	$E.\mathrm{gsum}(E,h'-1) + D\sum_{i=1}^{h}(\log(h/i)+1).(i-1).a(i)$
3.2 Semi-Naive Evaluation	$E.\mathrm{gsum}(E,h'-1) + D\sum_{i=1}^{h}(i-1).a(i)$
3.3 QSQ, Iterative	$E.\mathrm{gsum}(E,h'-1) + F.\sum_{i=1}^{h'} (h'-i+1).i.E^{i-1}$
3.4 QSQ, Recursive	$F+E.\mathrm{gsum}(E,h'-1)+D\sum_{i=2}^{h'} (i-1).E^i$
3.5 Henschen-Naqvi	Does not apply.
3.6 Prolog	Does not terminate.
3.7 APEX	$E.\mathrm{gsum}(E,h'-1) + (1/E)^{h-h'}.(D\sum_{i=1}^{h}(i-1).E^i.\mathrm{gsum}(E,h-i))$ $+ E^{h'}.(F\sum_{i=1}^{h}(i-1).(1/E)^i.\mathrm{gsum}(1/E,h-i))$
3.8 Kifer-Lozinskii	$E.\mathrm{gsum}(E,h'-1) + D\sum_{i=1}^{h}(i-1).a(i)$
3.9 Magic Sets	$E.\mathrm{gsum}(E,h'-1) + D\sum_{i=1}^{h}(i-1).a(i)$
3.10 Counting	Does not apply.

Query 4 (Same Generation.bf)

In the following expressions, $h'_{up.down} = \min(h'_{up},h'_{down})$, and $h_{up.down} = \min(h_{up},h_{down})$.

4.1 Naive evaluation

$$A_{flat} + T_{up2.flat1}.E_{flat}.T_{flat2.down1}.D_{down}.\sum_{i=1}^{h_{up.down}} (h_{up.down}-i+1).a_{up}(i).E_{down}^{i} +$$

$$T_{up2.flat1}.E_{flat}.T_{flat2.down1}.F_{down}.\sum_{i=1}^{h'_{up.down}} (E_{up}.E_{down})^{i}$$

4.2 Semi-Naive Evaluation

$$A_{flat} + T_{up2.flat1}.E_{flat}.T_{flat2.down1}.D_{down}.\sum_{i=1}^{h_{up.down}} a_{up}(i).E_{down}^{i} +$$

$$T_{up2.flat1}.E_{flat}.T_{flat2.down1}.F_{down}.\sum_{i=1}^{h'_{up.down}} (E_{up}.E_{down})^i$$

4.3 QSQ, Iterative

$$(h'_{up.down}+1).F_{flat}+$$
$$T_{up2.flat1}.F_{flat}\sum_{i=1}^{h'_{up.down}} (h'_{up.down}-i+1).E_{up}{}^i +$$
$$E_{flat}.T_{up2.flat1}.T_{flat2.down1}.F_{down}.\sum_{i=1}^{h'_{up.down}} (h'_{up.down}-i+1).E^i_{up}.\mathrm{gsum}(E_{down},i-1) +$$
$$T_{up2.flat1}.E_{flat}.T_{flat2.down1}.F_{down}.\sum_{i=1}^{h'_{up.down}} (E_{up}.E_{down})^i$$

4.4 QSQ, Recursive

$$F_{up}.\mathrm{gsum}(E_{up},h'_{up}-1) + E_{up}.\mathrm{gsum}(E_{up},h'_{up}-1).T_{up2.flat1}.F_{flat} +$$
$$T_{up2.flat1}.E_{flat}.T_{flat2.down1}.D_{down}.\sum_{i=1}^{h'_{up.down}} E^i_{up}.\mathrm{gsum}(E_{up},h'_{up}-i).E^i_{down} +$$
$$T_{up2.flat1}.E_{flat}.T_{flat2.down1}.F_{down}.\sum_{i=1}^{h'_{up.down}} (E_{up}.E_{down})^i$$

4.5 Henschen-Naqvi

$$F_{up}.\mathrm{gsum}(E_{up},h'_{up}-1) +$$
$$\sum_{i=1}^{h'_{up.down}} (E^i_{up}.T_{up2.flat1}.F_{flat} + T_{up2.flat1}.E_{flat}.T_{flat2.down1}.F_{down}.E^i_{up}.\mathrm{gsum}(E_{down},i-1)) +$$
$$T_{up2.flat1}.E_{flat}.T_{flat2.down1}.F_{down}.\sum_{i=1}^{h'_{up.down}} (E_{up}.E_{down})^i$$

4.6 Prolog

$$\mathrm{gsum}(F_{up},h'_{up}-1) + F_{up}.\mathrm{gsum}(F_{up},h'_{up}-1).T_{up2.flat1}.F_{flat} +$$
$$T_{up2.flat1}.F_{flat}.T_{flat2.down1}.\sum_{i=1}^{h'_{up.down}} F^i_{up}.\mathrm{gsum}(F_{up},h'_{up}-i).F^i_{down} +$$
$$T_{up2.flat1}.E_{flat}.T_{flat2.down1}.F_{down}.\sum_{i=1}^{h'_{up.down}} (E_{up}.E_{down})^i$$

4.7 APEX

$$F_{up}.\mathrm{gsum}(E_{up},h'_{up}-1) + E_{up}.\mathrm{gsum}(E_{up},h'_{up}-1).T_{up2.flat1}.F_{flat} +$$
$$T_{up2.flat1}.E_{flat}.T_{flat2.down1}.D_{down}.\sum_{i=1}^{h'_{up.down}} E^i_{up}.\mathrm{gsum}(E_{up},h'_{up}-i).E^i_{down} +$$
$$T_{up2.flat1}.E_{flat}.T_{flat2.down1}.F_{down}.\sum_{i=1}^{h'_{up.down}} (E_{up}.E_{down})^i$$

4.8 Kifer-Lozinskii

$$A_{flat} + T_{up2.flat1}.E_{flat}.T_{flat2.down1}.D_{down}.(\sum_{i=1}^{h_{up.down}} (a_{up}(i).E^i_{down}) +$$
$$T_{up2.flat1}.E_{flat}.T_{flat2.down1}.F_{down}.\sum_{i=1}^{h'_{up.down}} (E_{up}.E_{down})^i$$

4.9 Magic Sets

$$F_{up}.\mathrm{gsum}(E_{up},h'_{up}-1) + E_{up}.\mathrm{gsum}(E_{up},h'_{up}-1).T_{up\,2.flat\,1}.F_{flat} +$$

$$T_{up\,2.flat\,1}.E_{flat}.T_{flat\,2.down\,1}.D_{down}.\sum_{i=1}^{h'_{up.down}} E_{up}^{i}.\mathrm{gsum}(E_{up},h'_{up}-i).E_{down}^{i} +$$

$$T_{up\,2.flat\,1}.E_{flat}.T_{flat\,2.down\,1}.F_{down}.\sum_{i=1}^{h'_{up.down}} (E_{up}.E_{down})^{i}$$

4.10 Counting

$$F_{up}.\mathrm{gsum}(E_{up},h'_{up}-1) +$$

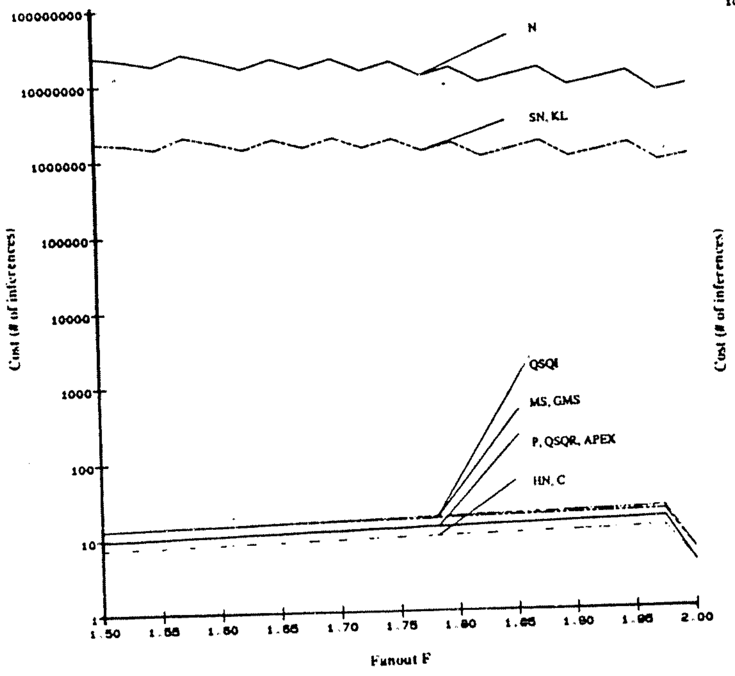
$$T_{up\,2.flat\,1}.F_{flat}(1+E_{up}\mathrm{gsum}(Eu,h'_{up}-1)) +$$

$$\sum_{i=1}^{h'_{up.down}} T_{up\,2.flat\,1}.E_{flat}.T_{flat\,2.down\,1}.D_{down}.(E_{up}.E_{down})^{i} +$$

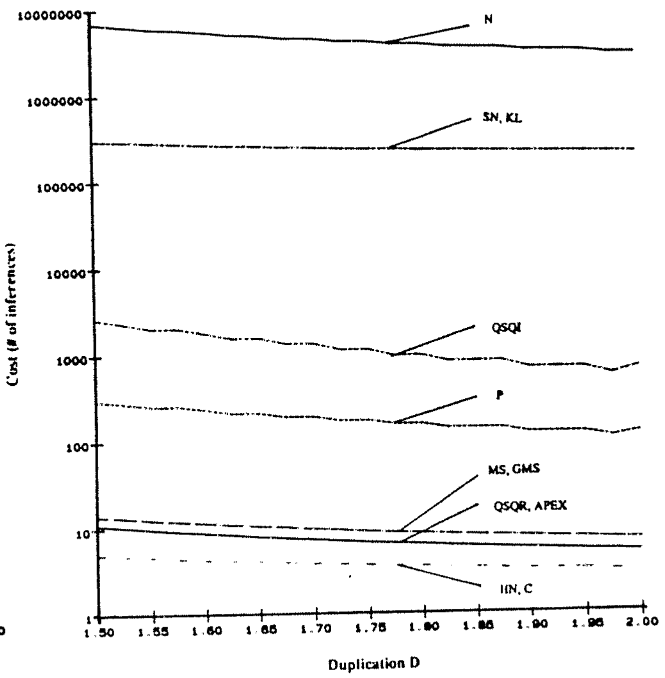
$$T_{up\,2.flat\,1}.E_{flat}.T_{flat\,2.down\,1}.F_{down}.\sum_{i=1}^{h'_{up.down}} (E_{up}.E_{down})^{i}$$

APPENDIX

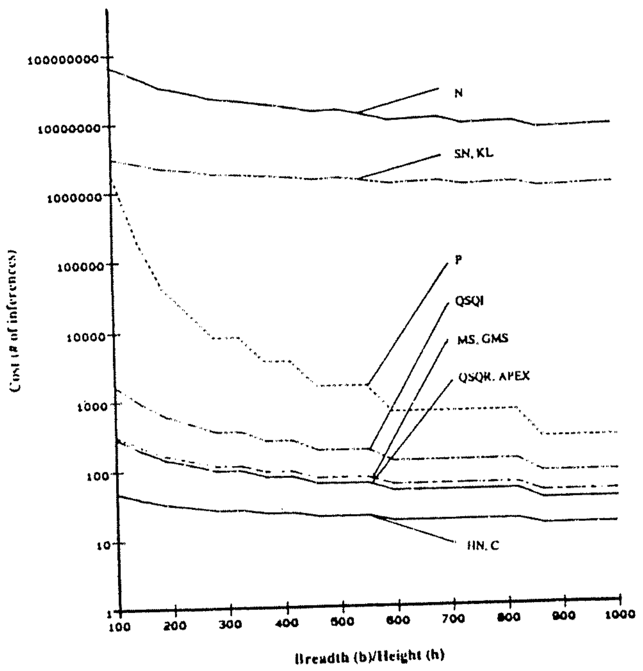
Query 1, Tree data



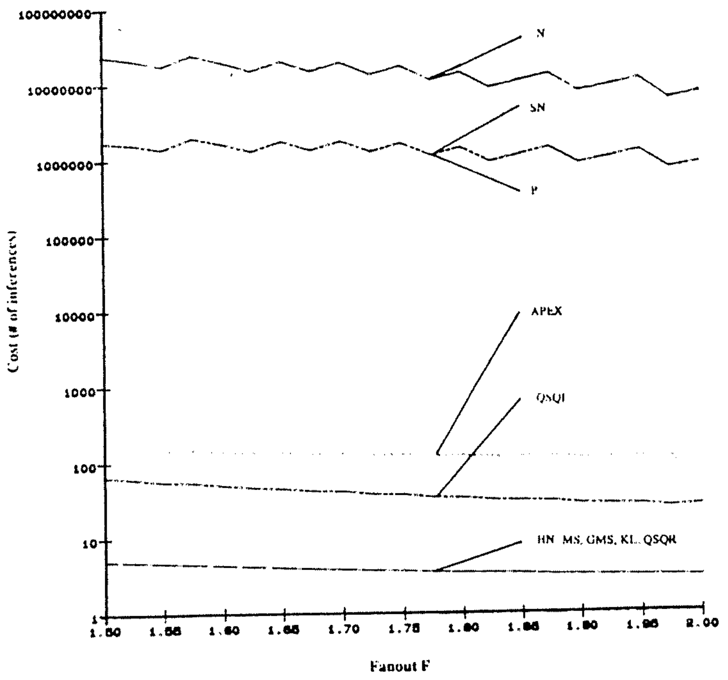
Query 1, Inverted tree data



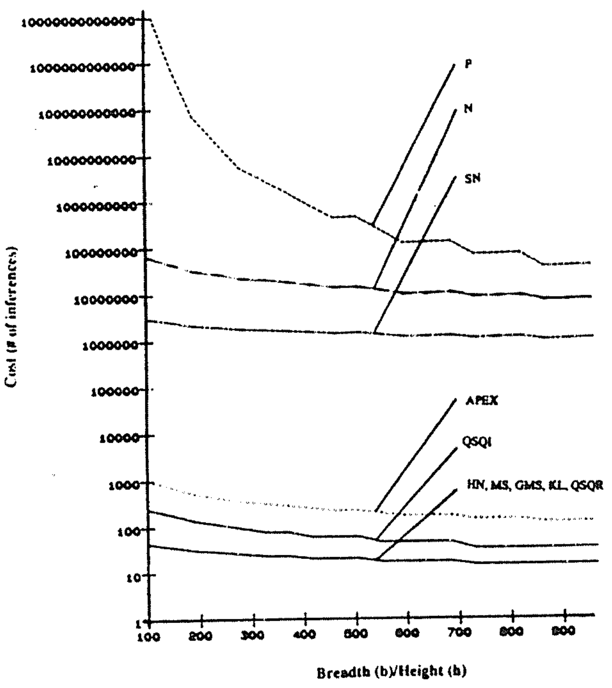
Query 1, Cylinder data



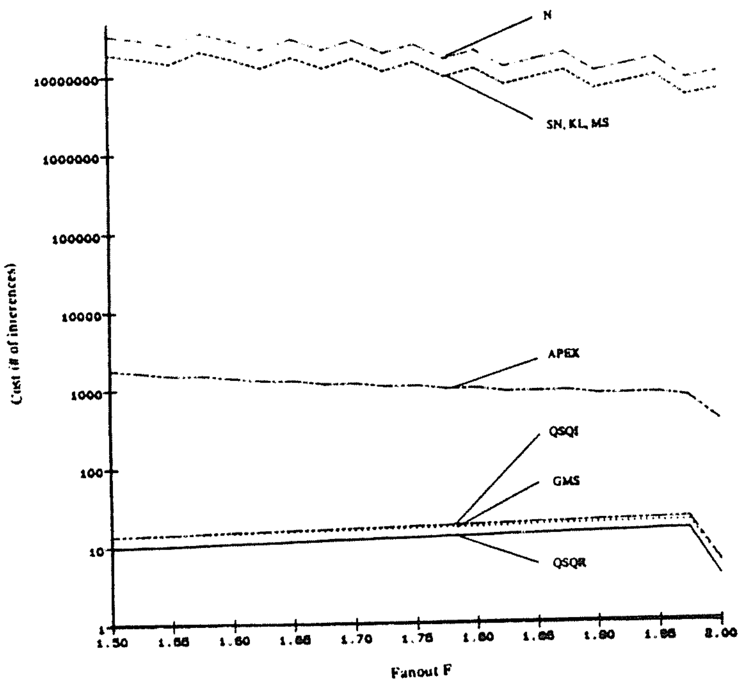
Query 2, Tree data



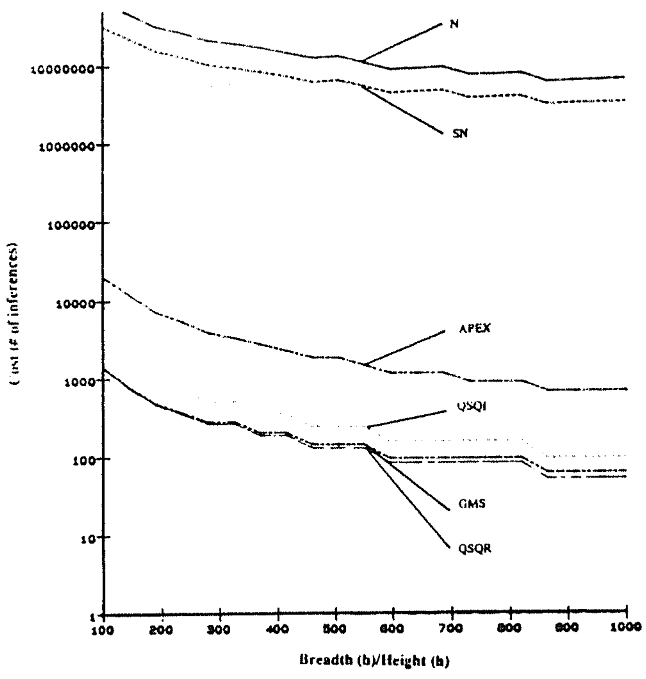
Query 2, Cylinder data



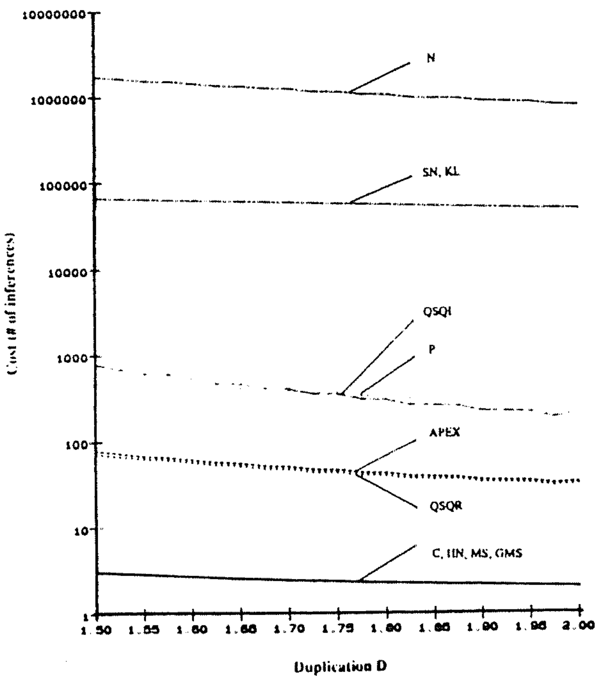
Query 3, Tree data



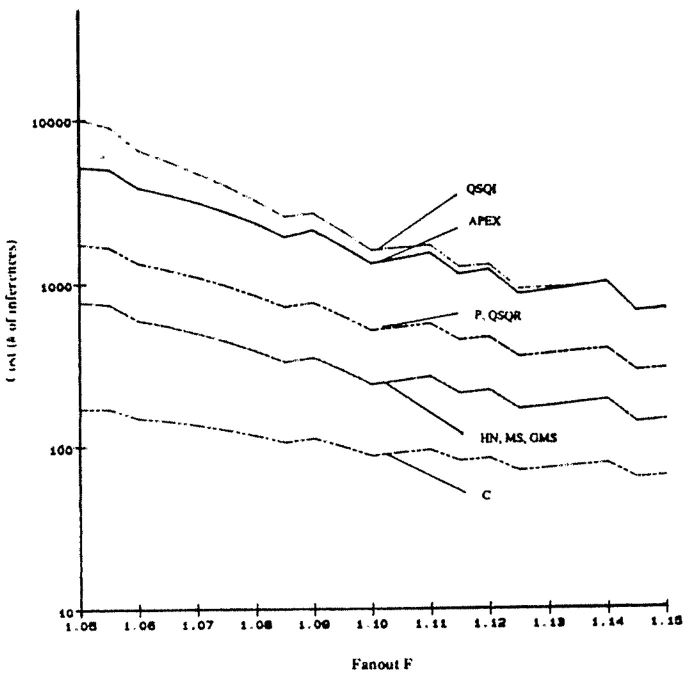
Query 3, Cylinder data



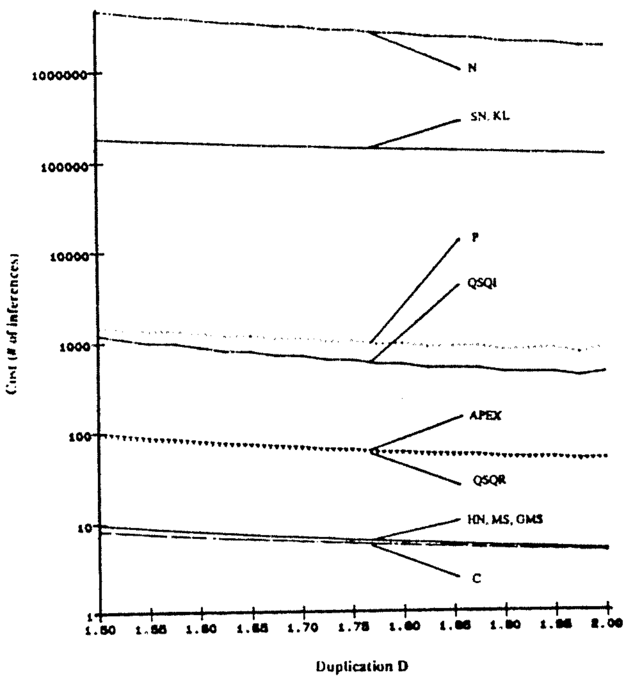
Query 4, Inverted tree data with Join Selectivity = 1%



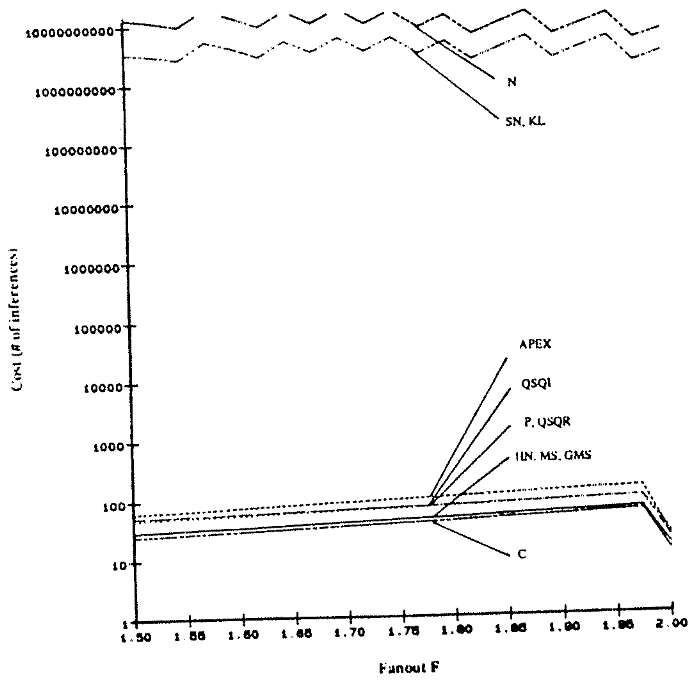
Query 4, Tree data with Join Selectivity = 100%



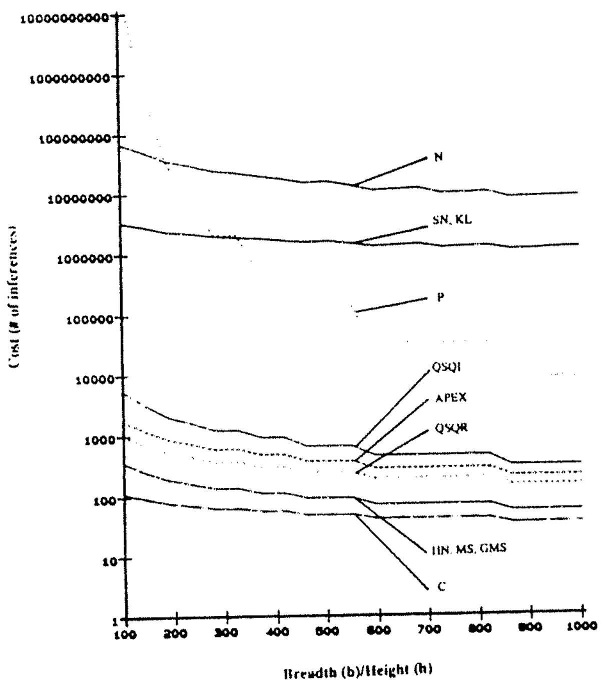
Query 4, Inverted tree data with Join Selectivity = 100%



Query 4, Tree data with Join Selectivity = 100%



Query 4, Cylinder data with Join Selectivity = 100%



Epilogue

Recursive queries have received a great deal of attention recently, and the preceding survey does not touch upon several interesting developments. In this epilogue, we provide some pointers to further work in this area. In order to be as comprehensive as possible, we have chosen to present an annotated bibliography, but even this is far from complete. However, this should provide a starting point for the interested reader. †

Projects

Two of the major research projects in this area have been carried out at MCC in Austin and at Stanford University. An overview of these projects is provided in the following references.

K. Morris, J.F. Naughton. Y. Saraiya, J.D. Ullman and A. Van Gelder [1987], "YAWN! (Yet Another Window on NAIL!)," *Bull. Data Engineering, Vol. 10, No. 4, Dec. 1987.*

D. Chimenti, T. O'Hare, R. Krishnamurthy, S. Naqvi, S. Tsur, C. West and C. Zaniolo [1987], "An Overview of the LDL System," *Bull. Data Engineering, Vol. 10, No. 4, Dec. 1987.*

Some of the other influential efforts include projects at ECRC in Munich, the ESPRIT projects, Honeywell-Bull, INRIA, SUNY-StonyBrook, the University of Maryland and the University of Melbourne. The work at ECRC, ESPRIT, and the Univ. of Melbourne is surveyed in the following references.

H. Gallaire and J.-M. Nicolas [1987], "Logic Approach to Knowledge and Data Bases at ECRC," *Bull. Data Engineering, Vol. 10, No. 4, Dec. 1987.*

D. Sacca, M. Dispinzeri, A. Mecchia, C. Pizzuti, C. Del Gracco and P. Naggar [1987], "The Advanced Database Environment of the KIWI System," *Bull. Data Engineering, Vol. 10, No. 4, Dec. 1987.*

K. Ramamohanarao, J. Shepherd, I. Balbin, G. Port, L. Naish, J. Thom, J. Zobel and P. Dart [1987], "The NU-Prolog Deductive Database System," *Bull. Data Engineering, Vol. 10, No. 4, Dec. 1987.*

While we are not aware of any overviews of the other projects, pointers may be found in the following references: Rohmer et al. [1986] (Honeywell-Bull), Gardarin and de Maindreville [1986] (INRIA) and Kifer and Lozinskii [1988] (StonyBrook). Pointers to the work at Maryland may be found in the following reference, which also provides a historical survey of the field.

J. Minker [1987], "Perspectives in Deductive Databases," *Proc. PODS 87, San Diego.*

Projects at CCA and Bell Labs have focussed on transitive closure and related path problems. The following references provide further pointers.

U. Dayal, A. Buchmann, D. Goldhirsch, S. Heiler, F. Manola, J. Orenstein and A. Rosenthal [1986], "PROBE- a Research Project in Knowledge-Oriented Database Systems: Preliminary Analysis," *Technical Report, CCA-85-03, July 1985.*

R. Agrawal and H.V. Jagadish [1987], "Direct Algorithms for Computing the Transitive Closure of Database Relations," *Proc. Conf. on Very Large Data Bases 87.*

Query Evaluation

We present brief descriptions of some recent work. Ullman [1985] presented the "capture rules" framework for planning the evaluation of a recursive query. An alternative framework, with a comparison, is proposed in Krishnamurthy et al. [1988]. In essence, it is proposed that all strategies be implemented by rewriting (e.g., using Generalized Magic Sets) followed by bottom-up evaluation. Testing capturability is now similar to testing for safety, and a general testing algorithm is presented. Aly and Ozsoyoglu [1987] presents a Petri-net based model for describing logic queries and the flow of control in algorithms for evaluating them.

Kifer and Lozinskii [1986b, 1988] extend the original "static" filtering algorithm to deal with general recursion and rules containing function symbols. They also consider the issue of safety. The method in the

† The references for some of the papers discussed in this epilogue appear in the list of references for the survey paper.

most general form is comparable to Generalized Magic Sets. Nejdl [1987] presents a recursive algorithm related to QSQ. Vieille [1988] extends the QSQ method and discusses its relationship to the generalized versions of Magic Sets and Counting. Grahne et al. [1987] consider efficient evaluation of a simple class of programs ("binary chain programs") and also consider the use of their algorithm to deal with more general programs. Sippu and Soisalon-Soininen [1988a] proposes an algorithm similar to the Generalized Magic Sets algorithm, but such that the rules defining the "magic sets" are simplified and separated from the other rules. The trade-off is that the magic sets so computed may be less restrictive. Gardarin [1987] presents a functional-style algorithm extending the results in Gardarin and de Maindreville [1986]. Sacca and Zaniolo [1987] addresses the issue of how to adapt the Counting method to the Magic Sets method when it is discovered that the former is not applicable. Ramakrishnan [1988] extends the Generalized Magic Sets to deal with arbitrary programs by treating rules in which some head variables do not appear in the body (even in the rewritten program). Such rules give rise to partial data structures, and allow the utilization of partially bound arguments and of data structures such as difference lists. Haddad and Naughton [1988] proposes an efficient way to adapt Counting in the presence of cyclic data. Marchetti-Spaccamela et al. [1987] presents a worst-case analysis of three algorithms (Counting, Magic Sets, and a method similar to Henschen-Naqvi), and suggests a modification to deal with Counting in the presence of cyclic data. Much of this work may be viewed as attempting to "push" selections through recursive rules. In Beeri et al. [1987], this was formalized as transforming a binary chain program with a selection query into an equivalent monadic program (i.e., all derived predicates have exactly one argument). This problem was shown to be undecidable (since it is equivalent to testing whether a context free grammar defines a regular language).

Sagiv [1987] introduced the notion of "uniform equivalence" of programs and showed that it was decidable to test whether two programs were uniformly equivalent. He also considered the deletion of rules and literals under uniform equivalence. The problem of pushing projections through recursive rules was studied in Ramakrishnan et al. [1988]. The problem was shown to be undecidable, using the result in Beeri et al. [1987]. Several optimization algorithms for dealing with projections were presented, including a sufficient condition for deleting rules under "uniform query equivalence". Naughton [1987] introduces a class of programs which permit efficient evaluation strategies. This class of programs generalizes transitive closure in a natural way. Han and Henschen [1987] discusses related strategies (as part of a more general study of how to reduce redundant computation).

Zhang and Yu [1987] presents an algorithm to obtain a linear recursive rule which is equivalent to a doubly recursive rule, when the latter satisfies certain conditions. Ceri et al. [1986], Ceri and Tanca [1987], and Ioannidis and Wong [1988] consider algebraic formalisms for representing and manipulating recursive queries. Ioannidis and Wong [1987] considers the use of simulated annealing to deal with large access plan spaces in recursive query evaluation. Whang and Navathe [1987] proposes an Extended Disjunctive Normal Form for recursive queries and considers evaluation strategies based on translation into this form, which allows recognition of common subexpressions in the evaluation of multiple rules. Raschid and Su [1986] presents an evaluation strategy based on evaluating a series of non-recursive expressions (generated by repeatedly expanding recursive rules) in parallel. Jagadish et al. [1987] show that linearly recursive rules can be translated into relational algebra augmented with transitive closure, but, in general, this involves taking cross-products of relations.

H. Aly and Z.M. Ozsoyoglu [1987], "Non-deterministic Modelling of Logical Queries in Deductive Databases," *Proc. ACM-SIGMOD Conference, 1987.*

C. Beeri, P. Kanellakis, F. Bancilhon and R. Ramakrishnan [1987], "Bounds on the Propagation of Selection into Logic Programs," *Proc. 6th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, 1987.*

S. Ceri, G. Gottlob and L. Lavazza [1986], "Translation and Optimization of Logic Queries: The Algebraic Approach," *Proc. 12th Inter. Conf. on Very Large Data Bases, 1986.*

S. Ceri and L. Tanca [1987], "Optimization of Systems of Algebraic Equations for Evaluating Datalog Queries," *Proc. 13th Inter. Conf. on Very Large Data Bases, 1987.*

L. Raschid and S.Y.W. Su [1986], "A Parallel Processing Strategy for Evaluating Recursive Queries," *Proc. 12th Inter. Conf. on Very Large Data Bases, 1986.*

G. Gardarin [1987], "Magic Functions: A Technique to Optimize Extended Datalog Recursive Programs," *Proc. 13th Inter. Conf. on Very Large Data Bases, 1987.*

G. Grahne, S. Sippu and E. Soisalon-Soininen [1987], "Efficient Evaluation for a Subset of Recursive Queries," *Proc. 6th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, 1987.*

R.W. Haddad and J.F. Naughton [1988], "Counting Methods for Cyclic Relations," *Proc. 7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, 1988.*

J. Han and L. Henschen [1987], "Handling Redundancy in the Processing of Recursive Queries," *Proc. ACM-SIGMOD Conference, 1987.*

Y.E. Ioannidis and E. Wong [1987], "Query Optimization by Simulated Annealing," *Proc. ACM-SIGMOD Conference, 1987.*

H.V. Jagadish, R. Agrawal and L. Ness [1987], "A Study of Transitive Closure as a Recursion Mechanism," *Proc. ACM-SIGMOD Conference, 1987.*

A. Marchetti-Spaccamela, A. Pelaggi and D. Sacca [1987], "Worst-case Complexity Analysis of Methods for Logic Query Implementation," *Proc. 6th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, 1987.*

J.F. Naughton [1986], "Redundancy in Function-Free Recursive Rules," *Proc. 3rd IEEE Symposium on Logic Programming, 1986.*

J.F. Naughton [1987], "One-Sided Recursions," *Proc. 6th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, 1987.*

W. Nejdl [1987], "Recursive Strategies for Answering Recursive Queries - The RQA/FQI Strategy," *Proc. 13th Inter. Conf. on Very Large Data Bases, 1987.*

R. Ramakrishnan, C. Beeri and R. Krishnamurthy [1988], "Optimizing Existential Datalog Programs," *Proc. 7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, 1988.*

Y. Sagiv [1988], "Optimizing Datalog Programs," *Proc. 6th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, 1987.*

S. Sippu and E. Soisalon-Soininten [1988a], "An Optimization Strategy for Recursive Queries in Logic Databases," *Proc. 4th Data Engineering Conf., 1988.*

K.-Y. Whang and S. Navathe [1987], "An Extended Disjunctive Normal Form Approach for Optimizing Recursive Logic Queries in Loosely Coupled Environments," *Proc. 13th Inter. Conf. on Very Large Data Bases, 1987.*

W. Zhang and C.T. Yu [1987], "A Necessary Condition for a Doubly Recursive Rule to be Equivalent to a Linear Recursive Rule," *Proc. ACM-SIGMOD Conference, 1987.*

Transitive Closure

There has been considerable effort on optimizing the evaluation of transitive closure queries. Most of the work has concentrated on evaluating the entire transitive closure of a relation. Much of the effort has been to reduce the I/O cost when the relations are sufficiently large that they must reside mostly on disk. Agrawal and Jagadish [1987] discusses the implementation of well-known algorithms (Warshall, Warren) based on matrix-multiplication with careful blocking to reduce I/O. (Lu et al. [1987] also consider these algorithms, but their implementation did not utilize blocking.) Agrawal et al. [1987] extends this work to deal with path computations, such as the shortest paths between pairs of nodes. Their work (in particular, the formalism), as well as that of Ioannidis and Ramakrishnan [1988] is influenced by the work of Carre [1979]. Ioannidis and Ramakrishnan [1988] presents an algorithm based on depth-first traversal. The algorithm can be viewed as a refinement of the Seminaive and Schmitz algorithms, and can deal with path computations and selections. The Schmitz algorithm, also based on depth-first traversal, is unable to deal with path computations. Ioannidis [1986] and Valduriez and Boral [1986] present an algorithm called the "Logarithmic" or "Smart" algorithm that is an iterative algorithm like Seminaive, but converges in fewer iterations. Ioannidis derived this algorithm based on an algebraic formulation of queries, which is further developed in Ioannidis and Wong [1986, 1988]. Lu [1987] presents a refinement of Seminaive based on using hash-joins, dynamically reducing the size of intermediate relations, and aggressively processing in-memory tuples. Sippu and Soisalon-Soininen [1988b] considers a generalization of transitive closure, based on a generalization of the composition (join) operation from binary to n-ary relations. The references below also contain pointers to related work such as transforming general queries into transitive closure computations, and pushing selections in the special case of transitive closure.

R. Agrawal, S. Dar and H.V. Jagadish [1987], "Transitive Closure Algorithms Revisited: The Case for Path Computations," *Manuscript, 1987.*

B. Carre [1979], "Graphs and Networks," *Clarendon Press, Oxford, England, 1979.*

Y.E. Ioannidis [1986], "On the Computation of the Transitive Closure of Relational Operators," *Proc. 12th Inter. Conf. on Very Large Data Bases, 1986.*

Y.E. Ioannidis and R. Ramakrishnan [1988], "Efficient Transitive Closure Algorithms," *Manuscript, 1988.*

Y.E. Ioannidis and E. Wong [1986], "An Algebraic Approach to Recursive Inference," *Proc. of the 1st Inter. Conf. on Expert Database Systems, 1986.*

Y.E. Ioannidis and E. Wong [1988], "Transforming Nonlinear Recursion into Linear Recursion," *To appear in Proc. 2nd Inter. Conf. on Expert Database Systems, 1988.*

H. Lu [1987], "New Strategies for Computing the Transitive Closure of a Database Relation," *Proc. 13th Inter. Conf. on Very Large Data Bases, 1987.*

H. Lu, K. Mikkilineni, and J.P. Richardson [1987], "Design and Evaluation of Algorithms to Compute the Transitive Closure of a Database Relation," *Proc. of the 3rd Inter. Data Engineering Conf., 1987.*

L. Schmitz [1983], "An Improved Transitive Closure Algorithm," *Computing, Vol. 30, 1983.*

S. Sippu and E. Soisalon-Soininen [1988b], "A Generalized Transitive Closure for Relational Queries," *Proc. 7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, 1988.*

Negation and Sets

The database approach to negation has differed significantly from the logic programming approach. The notion of "stratification" was proposed in Apt, Blair and Walker [1988], Naqvi [1986] and Van Gelder [1988]. The idea, intuitively, is that if predicate p is defined in terms of $\neg q$, then q must not depend on p. (Thus, we can think of the predicates in a program as being partioned into *strata* - p and q would be in different strata.) Przymusinski [1988] extends this notion by considering ground instances of rules ("local stratification"). Lifshitz [1988] considers the relationship between stratification and circumscription. Apt and Pugin [1987] considers the maintenance of materialized views based on stratified programs. Beeri et al. [1987], Balbin et al. [1987a] and Balbin et al. [1987b] consider the evaluation of stratified programs using Generalized Magic Sets. Beeri et al. [1987] also presents a description of sets and negation in the LDL language. Shmueli et al. [1988] considers the use of rewriting to implement set terms. Kuper [1987] presents another proposal for incorporating sets in a logic-based language, and Kuper [1988] considers the expressive power of these (and other) proposals. A more general overview of the expressive power of query languages is provided in Chandra [1988]. Kolaitis and Papadimitriou [1988] and Abiteboul and Vianu [1988] propose another semantics for negation, based on "inflationary fixpoints" (essentially, a fact once deduced is never discarded). Yet another proposal for dealing with negation, which properly includes local stratification, is presented in Van Gelder et al. [1988]. Imielinski and Naqvi [1988] suggest that there may be no one natural semantics for negation, and instead propose a rule algebra as a way to specify a limited amount of control, whereby a number of distinct semantics for negation may be realized by the user.

S. Abiteboul and V. Vianu [1988], "Procedural and Declarative Database Update Languages, *Proc. 7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, 1988.*

K.R. Apt, H. Blair and A. Walker [1988], "Towards a Theory of Declarative Knowledge," *In "Foundations of Deductive Databases and Logic Programming," Ed. J. Minker, Morgan Kaufmann, 1988.*

K.R. Apt and J.-M. Pugin [1987]. "Maintenance of Stratified Databases Viewed as a Belief Revision System," *Proc. 6th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, 1987.*

I. Balbin, G.S. Port and K. Ramamohanarao [1987a], "Magic Set Computation of Stratified Databases," *Technical Report 87/3, University of Melbourne, 1987.*

I. Balbin, K. Meenakshi and K. Ramamohanarao [1987b], "An Efficient Labelling Algorithm for Magic Set Computation on Stratified Databases," *Technical Report 88/1, University of Melbourne, 1988.*

C. Beeri, S. Naqvi, R. Ramakrishnan, O. Shmueli and S. Tsur [1987], "Sets and Negation in a Logic Database Language (LDL1)," *Proc. 6th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, 1987.*

A. Chandra [1988], "Theory of Database Queries," *Proc. 7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, 1988, invited paper.*

T. Imielinski and S. Naqvi [1988], "Explicit Control of Logic Programs Through Rule Algebra," *Proc. 7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, 1988.*

P.G. Kolaitis and C.H. Papadimitriou [1988], "Why not Negation by Fixpoint?," *Proc. 7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, 1988.*

G. Kuper [1987], "Logic Programming with Sets," *Proc. 6th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, 1987.*

G. Kuper [1988], "On the Expressive Power of Logic Programming Languages with Sets," *Proc. 7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, 1988.*

V. Lifschitz [1988], "On the Declarative Semantics of Logic Programs with Negation," *In "Foundations of Deductive Databases and Logic Programming," Ed. J. Minker, Morgan Kaufmann, 1988.*

S. Naqvi [1986], "A Logic for Negation in Database Systems," *In preprints of the Workshop on the Foundations of Deductive Databases and Logic Programming, Washington, 1986.*

T.C. Przymusinski [1988], "On the Declarative Semantics of Deductive Databases and Logic Programs," *In "Foundations of Deductive Databases and Logic Programming," Ed. J. Minker, Morgan Kaufmann, 1988.*

A. Van Gelder [1988], "Negation as Failure Using Tight Derivations for Logic Programs," *In "Foundations of Deductive Databases and Logic Programming," Ed. J. Minker, Morgan Kaufmann, 1988.*

A. Van Gelder, K. Ross and J.S. Schlipf [1988], "Unfounded Sets and Well-founded Semantics for General Logic Programs," *Proc. 7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, 1988.*

Bounded Recursion

A set of Horn clauses is called *bounded* if it is equivalent to a finite number of nonrecursive ones. A recursive Horn clause is *uniformly bounded* if it is uniformly equivalent to a finite number of nonrecursive ones. One can easily show that a recursive Horn clause is uniformly bounded if and only if its combination with any nonrecursive Horn clause is bounded. Thus, uniform boundedness implies boundedness, but the converse is not true.

Minker and Nicolas originally gave a sufficient condition for characterizing uniformly bounded recursion in a class of recursive Horn clauses (Minker and Nicolas [1983]). Representing a Horn clause by a tableau, and under various restrictions on the form of the tableau, Sagiv gives necessary and sufficient conditions for a set of Horn clauses to be uniformly bounded (Sagiv [1985]). Similar results have also been presented in Cosmadakis and Kanellakis [1986]. Necessary and sufficient conditions for a single linear recursive Horn clause to be uniformly bounded within an assortment of restricted classes of Horn clauses have been presented in Ioannidis [1986] and Naughton [1986]. Naughton has also dealt with the problem of (nonuniform) boundedness within the same classes [Naughton 86]. Recently this assortment of classes has been unified into a more abstract (super)class of recursive Horn clauses, where the characterization of Ioannidis and Naughton holds (Naughton and Sagiv [1987]). For general programs, both boundedness and uniform boundedness have been proven undecidable (Gaifman et al. [1987], Vardi [1987]). Finally, it has been shown that boundedness is decidable but NP-complete for programs which contain a single linear recursive Horn clause (Kanellakis [1986] and Vardi [1988]). The boundedness problem is decidable but EXPTIME-hard for monadic problems, and PSPACE-complete for linear monadic programs (Cosmadakis et al. [1988]).

J. Minker and J.-M. Nicolas [1983], ''On Recursive Axioms in Deductive Databases,'' *Information Systems, Vol. 8, No. 1, 1983.*

Y.E. Ioannidis [1986], ''A Time Bound on the Materialization of Some Recursively Defined Views,'' *Algorithmica, Vol. 1, No. 4, October 86.*

S.S. Cosmadakis and P.C. Kanellakis [1986], ''Parallel Evaluation of Recursive Rule Queries,'' *Proc. of the 5th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems, 1986.*

P.C. Kanellakis [1988], ''Logic Programming and Parallel Complexity,'' *In ''Foundations of Deductive Databases and Logic Programming,'' Ed. J. Minker, Morgan Kaufmann, 1988.*

J.F. Naughton [1986], ''Data Independent Recursion in Deductive Databases,'' *Proc. of the 5th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems, 1986.*

Y. Sagiv [1985], ''On Computing Restricted Projections of Representative Instances,'' *Proc. of the 4th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems, 1985.*

J.F. Naughton and Y. Sagiv [1987], ''A Decidable Class of Bounded Recursions,'' *Proc. of the 6th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems, 1987.*

M.Y. Vardi [1988] ''Decidability and Undecidability Results for Boundedness of Linear Recursive Queries,'' *Proc. of the 7th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems, 1988.*

H. Gaifman, H. Mairson, Y. Sagiv, and M.Y. Vardi [1987], ''Undecidable Optimization Problems for Database Logic Programs,'' *Proc. 2nd Inter. Symposium on Logic in Computer Science, 1987.*

S.S. Cosmadakis, H. Gaifman, P.C. Kanellakis and M.Y. Vardi [1988], ''Decidable Optimization Problems for Database Logic Programs,'' *Manuscript.*

Conclusions

We have attempted to provide a roadmap for the adventurous reader. The objective of the epilogue has been to complement the survey, and work which has been mentioned in the survey has been omitted (even work which has not been adequately discussed, e.g., the recent work on safety). Exigencies of space and time have also precluded mention of work in other important areas, e.g., updates, intelligent query answering, the expressive power of logic-based query languages and connections to logic programming. A good starting point in these areas is the collection of papers in Minker [1988] and the proceedings of PODS 88.

The area of recursive query processing has progressed very rapidly in the last few years, and we now have a good insight into both the theoretical and practical aspects of the field. The area is extremely vigorous, as evidenced by the large numbers of papers in this area appearing in recent database and logic programming conferences. We observe however, that a critical note has recently been raised: Do we really need more general forms of recursion than transitive closure in a database query language? (This question is raised, for example, in Laguna Beach [1988], and partly underlies the decision of the CCA, Bell Labs and other projects to focus on transitive closure.) This is an important question, and will not be completely settled until the community has had the opportunity to use and experiment with some of the more expressive recursive query languages such as LDL. While research in this area will continue to be productive, and is interesting from a theoretical point of view and for its relevance to the field of logic programming, the importance of general recursive query processing for database applications remains to be established. In fact, settling this question is, in our opinion, an important research objective.

Laguna Beach Participants [1988], ''Future Directions in DBMS Research,'' *Manuscript.*

Acknowledgements

Yannis Ioannidis helped to compile the section on bounded recursion. Ioannidis and Jeff Naughton also provided comments on a draft version of the epilogue.