The Architecture of the EXODUS Extensible DBMS

Michael J. Carey
David J. DeWitt
Daniel Frank
Goetz Graefe
Joel E. Richardson
Eugene J. Shekita
M. Muralikrishna

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

ABSTRACT

With non-traditional application areas such as engineering design, image/voice data management, scientific/statistical applications, and artificial intelligence systems all clamoring for ways to store and efficiently process larger and larger volumes of data, it is clear that traditional database technology has been pushed to its limits. It also seems clear that no single database system will be capable of simultaneously meeting the functionality and performance requirements of such a diverse set of applications. In this paper we describe the initial design of EXODUS, an extensible database system that will facilitate the fast development of high-performance, application-specific database systems. EXODUS provides certain kernel facilities, including a versatile storage manager and a type manager. In addition, it provides an architectural framework for building application-specific database systems, tools to partially automate the generation of such systems, and libraries of software components (e.g., access methods) that are likely to be useful for many application domains.

This research was partially supported by the Defense Advanced Research Projects Agency under contract N00014-85-K-0788, by the Department of Energy under contract #DE-AC02-81ER10920, by the National Science Foundation under grants MCS82-01870 and DCR-8402818, and by an IBM Faculty Development Award.

1. INTRODUCTION

Until recently, research and development efforts in the database management systems area have focused on supporting traditional business applications. The design of database systems capable of supporting non-traditional application areas, including engineering applications for CAD/CAM and VLSI data, scientific and statistical applications, expert database systems, and image/voice applications, has emerged as an important new research direction. These new applications differ from conventional applications such as transaction processing and from each other in a number of important ways. First, each requires a different set of data modeling tools. The types of entities and relationships that must be described for a VLSI circuit design are quite different from those of a banking application. Second, each new application area has a specialized set of operations that must be efficiently supported by the database system. It makes little sense to talk about doing joins between satellite images. Efficient support for the specialized operations of each new application area is likely to require new types of storage structures and access methods as well. For example, R-Trees [Gutt84] are a useful access method for storing and manipulating VLSI data. For managing image data, the database system needs to support large multidimensional arrays as a basic data type; storing images as tuples in a relational database system is generally either impossible or terribly inefficient. Finally, a number of new application areas require support for multiple versions of entities [Daya85, Katz86].

Recently, a number of new database system research projects have been initiated to address the needs of this emerging class of applications: EXODUS[1] at the University of Wisconsin [Care85a, Care86], PROBE at CCA [Daya85, Mano86], POSTGRES [Ston86b, Ston86c] at Berkeley, GEMSTONE at Servio Logic Corporation [Cope84, Maie86], STARBURST at IBM Almaden Research Center [Schw86], and GENESIS [Bato86] at the University of Texas-Austin. Although the goals of these projects are similar, and each uses some of the same mechanisms to provide extensibility, the overall approach of each project is quite different. For example, POSTGRES will be a more "complete" database management system, with a query language (POSTQUEL), a predefined way of supporting complex objects (through the use of POSTQUEL and procedures as a data type), support for "active" databases via triggers and alerters, and inferencing. Extensibility will be provided via new data types, operators, access methods, and a simplified recovery mechanism. A stated goal is to "make as few changes as possible to the relational model". The objective of the PROBE project, on the other hand, is to develop an advanced DBMS with support for complex objects and operations on them, dimensional data (in both space and time

[1] EXODUS: A departure, in this case, from the ways of the past. Also an **EX**tensible **O**bject-oriented **D**atabase **S**ystem.

dimensions), and a capability for intelligent query processing. Unlike POSTGRES, PROBE will provide a mechanism for directly representing complex objects. Like EXODUS, PROBE will use a rule-based approach to query optimization so that the query optimizer may be extended to handle new database operators, new methods for existing operators, and new data types. An extended version of DAPLEX [Ship81] is to be used as the query language for PROBE. GEMSTONE, with its query language OPAL, is a complete object-oriented database system that encapsulates a variety of ideas from the areas of knowledge representation, object-oriented and non-procedural programming, set-theoretic data models, and temporal data modeling. STARBURST is an architecture for an extensible DBMS based on the relational data model, and its design is intended to allow knowledgeable programmers to add extensions "on the side" in the form of abstract data types, access methods, and external storage structures.

In contrast to these efforts, and like GENESIS, EXODUS is being designed as a modular (and modifiable) system rather than as a "complete" database system intended to handle all new application areas. In some sense, EXODUS is a software engineering project — the goal is to provide a collection of kernel DBMS facilities plus software tools to facilitate the semi-automatic generation of high-performance, application-specific DBMSs for new applications. In this paper we describe the overall architecture of EXODUS. Section 2 presents an overview of the components of EXODUS. Section 3 describes the lowest level of the system, the Storage Object Manager, summarizing material from [Care86]. Section 4 describes the EXODUS Type Manager, which provides (among other things) a general schema management facility that can be extended with application-specific abstract data types. Section 5 addresses a difficult task in extending a database system: the addition of new access methods. EXODUS simplifies this task by hiding most of the storage, concurrency control, and recovery issues from the access method implementor via a new programming language, E; E is an extension of C that includes support for persistent objects via the Storage Object Manager of EXODUS. Section 6 discusses how application-specific database operations are implemented in EXODUS, and Section 7 describes the rule-based approach to query optimization employed in EXODUS. Section 8 outlines some of the user interface issues that lie ahead, and Section 9 briefly summarizes the paper and discusses our implementation plans.

2. AN OVERVIEW OF THE EXODUS ARCHITECTURE

In this section we describe the architecture of the EXODUS database system. Since one of the principal goals of the EXODUS project is to construct an **extensible** yet high-performance database system, the design reflects a careful balance between what EXODUS provides the *user*[2] and what the *user* must explicitly provide. Unlike POSTGRES and PROBE, EXODUS is not intended to be a complete system with provisions for user-added extensions. Rather, it is intended more as a toolbox that can be easily adapted to satisfy the needs of new application areas. Two basic mechanisms are employed to help achieve this goal: where feasible, we furnish a generic solution that should be applicable to any application-specific database system. As an example, EXODUS supplies at its lowest level a layer of software termed the Storage Object Manager which provides support for concurrent and recoverable operations on arbitrary size storage objects. Our feeling is that this level provides sufficient capabilities such that user-added extensions will most probably be unnecessary. However, due to both generality and efficiency considerations, a single generic solution is not possible for every component of a database system.

In cases where one generic solution is inappropriate, EXODUS instead provides either a **generator** or a **library** to aid the user in generating the appropriate software. As an example, we expect EXODUS to be used for a wide variety of applications, each with a potentially different query language. As a result, it is not possible for EXODUS to furnish a single generic query language, and it is accordingly impossible for a single query optimizer to suffice for all applications. Instead, we provide a generator for producing query optimizers for algebraic languages. The EXODUS query optimizer generator takes as input a collection of rules regarding the operators of the query language, the transformations that can be legally applied to these operators (e.g., pushing selections before joins), and a description of the methods that can be used to execute each operator (including their costs and side effects); as output, it produces an optimizer for the application's query language in the form of C source code.

In a conventional database system environment it is customary to consider the roles of two different classes of individuals: the database administrator and the user. In EXODUS, a third type of individual is required to customize EXODUS into an application-specific database system. While we referred to this individual as a "user" in the preceding paragraphs, he or she is not a user in the normal sense (i.e., an end user, such as a bank teller or a cartographer). Internally, we refer to this "user" of the EXODUS facilities as a "database implementor" or DBI. While the Jim Grays of the world would clearly make outstanding DBIs, our goal is to engineer EXODUS so that only a moderate amount of expertise is required to architect a new system using its tools. Once EXODUS has been customized into an application-specific database system, the DBI's role is completed and the role of the database administrator begins.

[2] Our use of the word *user* will be more carefully explained in the following paragraphs.

We present an overview of the design of EXODUS in the remainder of this section. While EXODUS is a toolkit and not a complete DBMS, we find that it is clearer to describe the system from the viewpoint of an application-specific database system that was constructed using it. In doing so, we hope to make it clear which pieces of the system are provided without modification, which must be produced using one of the EXODUS generators, and which must be directly implemented by the DBI using the E programming language.

2.1. EXODUS System Architecture

Figure 1 presents the structure of an application-specific database management system implemented using EXODUS. The following tools are provided to aid the DBI in the task of generating such a system:

(1) The Storage Object Manager.
(2) The E programming language and its compiler for writing database system software.
(3) A generalized Type Manager for defining and maintaining schema information.
(4) A library of type independent access methods which can be used to associatively access storage objects.
(5) Lock manager and recovery protocol stubs to simplify the task of writing new access methods and other database operators.
(6) A rule-based query optimizer and compiler.
(7) Tools for constructing user front ends.

At the bottom level of the system is the Storage Object Manager. The basic abstraction at this level is the storage object, an untyped, uninterpreted variable-length byte sequence of arbitrary size. The Storage Object Manager provides capabilities for reading, writing, and updating storage objects (or pieces of them) without regard for their size. To further enhance the functionality provided by this level, buffer management, concurrency control, and recovery mechanisms for operations on shared storage objects are also provided. Finally, a versioning mechanism that can be used to implement a variety of application-specific versioning schemes is supported. A more detailed description of the Storage Object Manager and its capabilities is presented in Section 3.

Although not shown in Figure 1, which depicts the runtime structure of an EXODUS-based DBMS, the next major component is the E programming language and compiler. E is the implementation language for all components of the system for which the DBI must provide code. E extends C by adding abstract data types and a notion of persistent object pointers to the language's type definition repertoire. For the most part, references to persistent objects look just like references to other C structures; the DBI's index code can thus deal with index nodes as arrays of key-pointer pairs, for example. Whenever persistent objects are referenced, the E translator is responsible for adding the appropriate calls to fix/unfix buffers, read/write the appropriate piece of the underlying storage object,

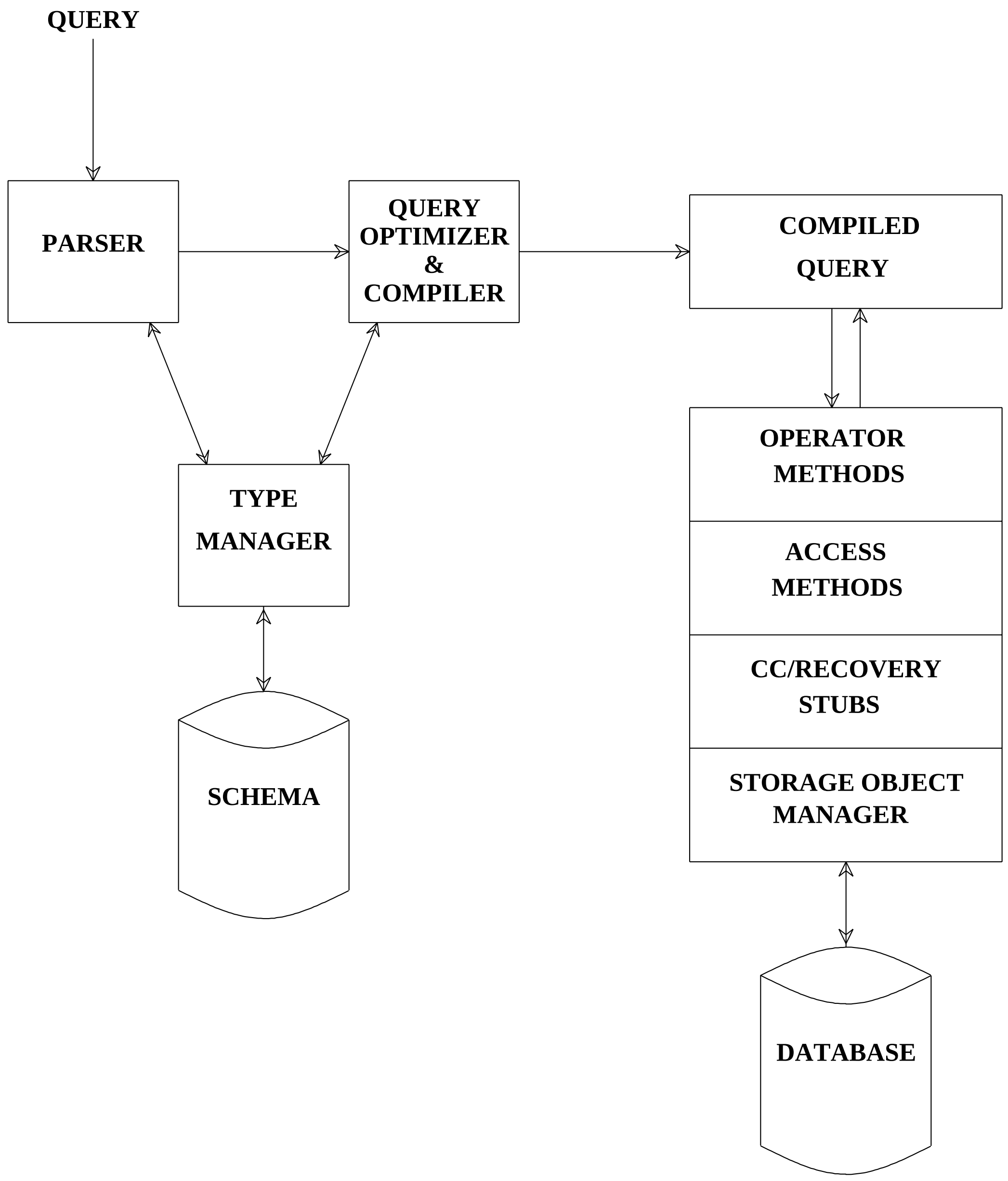

Figure 1: EXODUS System Architecture.

lock/unlock objects, log images and events, etc. Thus, the DBI is freed from having to worry about the internal structure of persistent objects. For buffering, concurrency control and recovery, the E language includes statements for associating locking, buffering, and recovery protocols with variables that reference persistent objects. Thus, the DBI is provided with a mechanism by which he or she can exercise control (declaratively) — insuring that the appropriate mechanisms are employed. E should not be confused with either database programming languages such as RIGEL [Rowe79], Pascal/R [Schm77], Theseus [Shop79], or PLAIN [Kers81], as these languages were intended to simplify the development of database applications code through a closer integration of database and programming language constructs, or with object-oriented query languages such as OPAL [Cope84, Maie86] — the objective of E is to simplify the development of **internal** systems software for a DBMS.

Layered above the Storage Object Manager is a collection of access methods that provide associative access to files of storage objects and further support for versioning (if desired). For access methods, EXODUS will provide a library of type-independent index structures including B+ trees, Grid files [Niev84], and linear hashing [Litw80]. These access methods will be implemented using the "type parameter" capability provided by the E language (as described in Section 5). This capability enables existing access methods to be used with DBI-defined abstract data types without modification — as long as the capabilities provided by the data type satisfy the requirements of the access methods. In addition, a DBI may wish to implement new types of access methods in the process of developing an application-specific database system. EXODUS provides two mechanisms to greatly simplify this task. First, since new access methods are written in E, the DBI is shielded from having to map main memory data structures onto storage objects and from having to write code to deal with locking, buffering, and recovery protocols. EXODUS also simplifies the task of handling concurrency control and recovery for new access methods using a form of layered transactions, as discussed in Section 5.

While the capabilities provided by the Storage Object Manager and Access Methods Layer are general purpose and are intended to be utilized in each application-specific DBMS constructed using EXODUS, the third layer in the design, the Operator Methods Layer, contains a mix of DBI-supplied code and EXODUS-supplied code. As implied by its name, this layer contains a collection of methods that can be combined with one another in order to operate on (typed) storage objects. EXODUS will provide a library of methods for a number of operators that operate on a single type of storage object (e.g., selection), but it will not provide application or data model specific methods. For example, it cannot provide methods for implementing the relational join operator or for examining an object containing satellite image data for the signature of a particular crop disease. In general, the DBI will need to implement one or more methods for each operator in the query language associated with the target application. E will again serve as the implementation language for this task.

At the center of the EXODUS architecture is the Type Manager. The EXODUS Type Manager is designed to provide schema support for a wide variety of application-specific database systems. The data modeling facilities provided by EXODUS are those of the type system of the E programming language, with the Type Manager acting as a repository (or "persistent symbol table") for E type definitions. The type system of E includes a set of primitive, built-in types (e.g., int, float, char), a set of type constructors (record, union, variant, fixed-length array, and insertable array or variable-length sequence), and an abstract data type (ADT) facility which allows the DBI to define new data types and operations. In addition, the Type Manager maintains the associations between EXODUS files and the E types of the objects that they contain; it also keeps track of type-related dependencies that arise between types and other types, files and types, stored queries and types, etc. In designing the type facilities of EXODUS, our goal was to provide a set of facilities that would allow the capabilities of the Storage Object Manager to be exploited, and that would allow the modeling needs of a wide range of applications to be handled with little or no loss of efficiency.[3] Section 4 presents a more detailed overview of the capabilities provided by the Type Manager, including a discussion of its dependency-maintenance role.

Execution of a query in EXODUS follows a set of transformations similar to that of a relational query in System R [Astr76]. After parsing, the query is optimized, and then compiled into an executable form. The parser is responsible for transforming the query from its initial form into an initial tree of database operators. During the parsing and optimization phases, the Type Manager is invoked to extract the necessary schema information. The executable form produced by the query compiler consists of a rearranged tree of operator methods (i.e., particular instances of each operator) to which query specific information such as selection predicates (e.g., name = "Mike" and salary > $200,000) will be passed as parameters. As mentioned earlier, EXODUS provides a generator for producing the optimization portion of the query compiler. To produce an optimizer for an application-specific database system, the DBI must supply a description of the operators of the target query language, a list of the methods that can used to implement each operator, a cost formula for each operator method, and a collection of transformation rules. The optimizer generator will transform these description files into C source code for an optimizer for the target query language. At query execution time, this optimizer behaves as we have just described, taking a query expressed as a tree of operators and transforming it into an optimized execution plan expressed as a tree of methods.

Finally, the organization of the top level of a database system generated using EXODUS will depend on whether the goal is to support some sort of interactive interface, an embedded query interface such as EQUEL [Allm76], or an altogether different form of interface. We plan to provide a generator to facilitate the creation of

[3] While we initially considered providing a generalized class hierarchy, such a facility can be efficiently supported on top of the type facilities that we provide. We also felt that different applications would be likely to want very different things from such a hierarchy.

interactive interfaces, and we are exploring the use of the Cornell Program Synthesizer Generator [Reps84] as a user interface generator for EXODUS[4]. This tool provides the facilities needed for implementing structured editors for a wide variety of programming languages, the goal of such editors being to help programmers formulate syntactically and semantically correct programs. Since the syntax and semantics of typical query languages are much simpler than that of most modern programming languages, it is clear that we will be able to apply the tool in this way; it remains to be seen whether or not it is really "too powerful" (i.e., overkill) for our needs. As for supporting queries that are embedded programs, two options exist. First, if the program simply contains calls to operator methods, bypassing the parser and optimizer, then a linker can be used to bind the program with the necessary methods (which can be viewed as a library of procedures). The second option, which will be a difficult task, is to provide a generalized tool to handle programs with embedded queries (ala EQUEL). It will be relatively easy to provide a generic preprocessor which will extract queries and replace them with calls to object modules produced by the parser, optimizer, and compiler; however, it is unclear how to make the underlying interface between the application program and database system independent of the (application-specific) data model. For example, in the relational model a tuple-at-a-time or portal [Ston84] interface is commonly used, whereas with the Codasyl data model, the database system and application program exchange currency indicators as well as record occurrences. These issues will be explored further in the future.

3. THE STORAGE OBJECT MANAGER

In this section we summarize the key features of the design of the EXODUS Storage Object Manager. We begin by discussing the interface that the Storage Object Manager provides to higher levels of the system, and then we describe how arbitrarily large storage objects are handled efficiently. We discuss the techniques employed for versioning, concurrency control, recovery, and buffer management for storage objects, and we close with a brief discussion about files of storage objects (known as file objects). A more detailed discussion of these issues can be found in [Care86].

[4] We have experimented with the idea of generating a QUEL-like interface using this tool.

3.1. The Storage Object Manager Interface

The Storage Object Manager provides a procedural interface. This interface includes procedures to create and destroy file objects and to open and close file objects for file scans. For scanning purposes, the Storage Object Manager provides a call to get the object ID of the next object within a file object. It also provides procedures for creating and destroying storage objects within a file. For reading storage objects, the Storage Object Manager provides a call to get a pointer to a range of bytes within a given storage object; the desired byte range is read into the buffers, and a pointer to the bytes there are returned to the caller. Another call is provided to inform EXODUS that these bytes are no longer needed, which "unpins" them in the buffer pool. For writing storage objects, a call is provided to tell EXODUS that a subrange of the bytes that were read have been modified (information that is needed for recovery to take place). For shrinking/growing storage objects, calls to insert bytes into and delete bytes from a specified offset in a storage object are provided, as is a call to append bytes to the end of an object. Finally, for transaction management, the Storage Object Manager provides begin, commit, and abort transaction calls; additional hooks are provided to aid the access methods layer in implementing concurrent and recoverable operations for new access methods efficiently (as discussed in Section 5).

In addition to the functionality outlined above, the Storage Object Manager is designed to accept a variety of performance-related hints. For example, the object creation routine mentioned above accepts hints about where to place a new object (i.e., "place the new object near the object with id *X*") and about how large the object is expected to be (on the average, if it varies); it is also possible to hint that an object should be alone on a disk page and the same size as the page (which will be useful for the access methods level). The buffer manager accepts hints about the size and number of buffers to use and what replacement policy to employ. These hints will be supported by allowing a *scan group* to be specified with each object access, and then having the buffer manager accept these hints on a per-scan-group basis, allowing easy support of buffer management policies like DBMIN [Chou85].

3.2. Storage Objects and Operations

As described earlier, the **storage object** is the basic unit of data in the Storage Object Manager. Storage objects can be either small or large, a distinction that is hidden from higher layers of EXODUS software. Small storage objects reside on a single disk page, whereas large storage objects occupy potentially many disk pages. In either case, the object identifier (OID) of a storage object is an address of the form (*page #*, *slot #*). The OID of a small storage object points to the object on disk; for a large storage object, the OID points to its *large object* *header*. A large object header can reside on a slotted page with other large object headers and small storage objects, and it contains pointers to other pages involved in the representation of the large object. Other pages in large storage objects are private rather than being shared with other objects (although pages are shared between versions of a storage object). When a small storage object grows to the point where it can no longer be accommodated on a single page, the Storage Object Manager will automatically convert it into a large storage object, leaving its object header in place of the original small object. We considered the alternative of using logical surrogates for OID's rather than physical addresses, as in other recent proposals [Cope84, Ston86b], but efficiency considerations led us to opt for a "physical surrogate" scheme — with logical surrogates, it would always be necessary to access objects via a dense surrogate index[5].

Figure 2 shows an example of our large object data structure; it was inspired by Stonebraker's ordered relation structure [Ston83], but there are a number of significant differences [Care86]. Conceptually, a large object is an uninterpreted byte sequence; physically, it is represented as a B+ tree like index on byte position within the object plus a collection of leaf blocks (with all data bytes residing in the leaves). The large object header contains a number of (*count*, *page #*) pairs, one for each child of the root. The count value associated with each child pointer gives the maximum byte number stored in the subtree rooted at that child, and the rightmost child pointer's count is therefore also the size of the object. Internal nodes are similar, being recursively defined as the root of another object contained within its parent node, so an absolute byte offset within a child translates to a relative offset within its parent node. The left child of the root in Figure 2 contains bytes 1-421, and the right child contains the rest of the object (bytes 422-786). The rightmost leaf node in the figure contains 173 bytes of data. Byte 100 within this leaf node is byte 192 + 100 = 292 within the right child of the root, and it is byte 421 + 292 = 713 within the object as a whole. Searching is accomplished by computing overall offset information while descending the tree to the desired byte position. As described in [Care86], object sizes up to 1 GB or so can be supported with only three tree levels (header and leaf levels included).

Associated with the large storage object data structure are algorithms to *search* for a range of bytes (and perhaps update them), to *insert* a sequence of bytes at a given point in the object, to *append* a sequence of bytes to the end of the object, and to *delete* a sequence of bytes from a given point in the object. The insert, append, and delete operations are novel because inserting or deleting an arbitrary number of bytes (as opposed to a single byte)

[5] This is true unless the objects are kept sorted on surrogate ID. In this case, a non-dense surrogate index can be used.

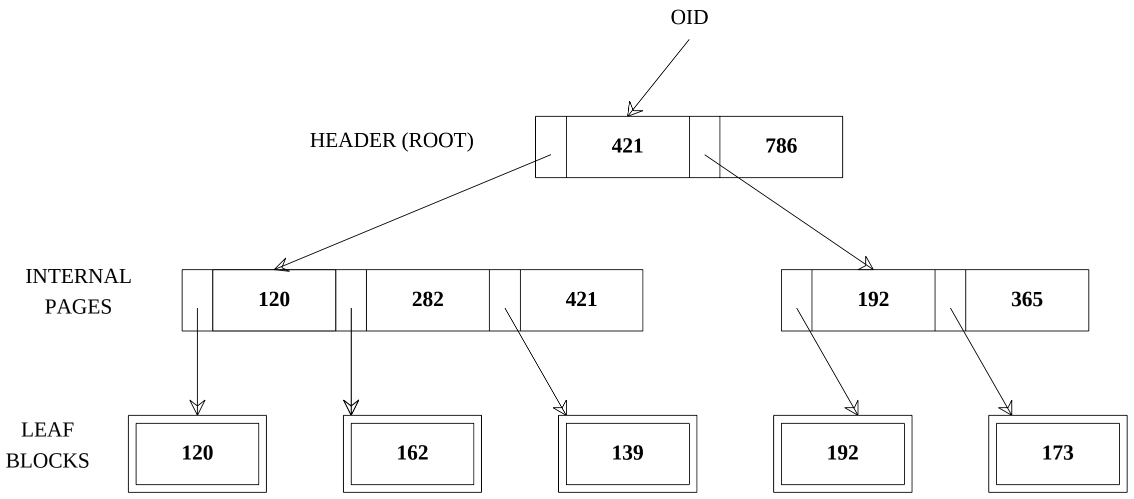

Figure 2: An example of a large storage object.

into a large storage object poses some unique problems compared to inserting or deleting a single record from an ordered relation. Algorithms for these operations are described in detail in [Care86] along with results from an experimental evaluation of their storage utilization and performance characteristics. The evaluation showed that the EXODUS storage object mechanism can provide operations on very large dynamic objects at relatively low cost, and at a reasonable level of storage utilization (typically 80% or higher).

3.3. Versions of Storage Objects

The Storage Object Manager provides primitive support for versions of storage objects. One version of each storage object is retained as the current version, and all of the preceding versions are simply marked (in their object headers) as being old versions. The reason for only providing a primitive level of version support is that different EXODUS applications may have widely different notions of how versions should be supported [Ston81, Dada84, Katz84, Bato85, Clif85, Klah85, Snod85, Katz86]. We do not omit version management altogether for efficiency reasons — it would be prohibitively expensive, both in terms of storage space and I/O cost, to maintain versions of large objects by maintaining entire copies of objects.

Versions of large storage objects are maintained by copying and updating the pages that differ from version to version. Figure 3 illustrates this by an example. The figure shows two versions of the large storage object of Figure 2, the original version, V_1, and a newer version, V_2. In this example, V_2 was created by deleting the last 36 bytes from V_1. Note that V_2 shares all nodes of V_1 that are unchanged, and it has its own copies of each modified

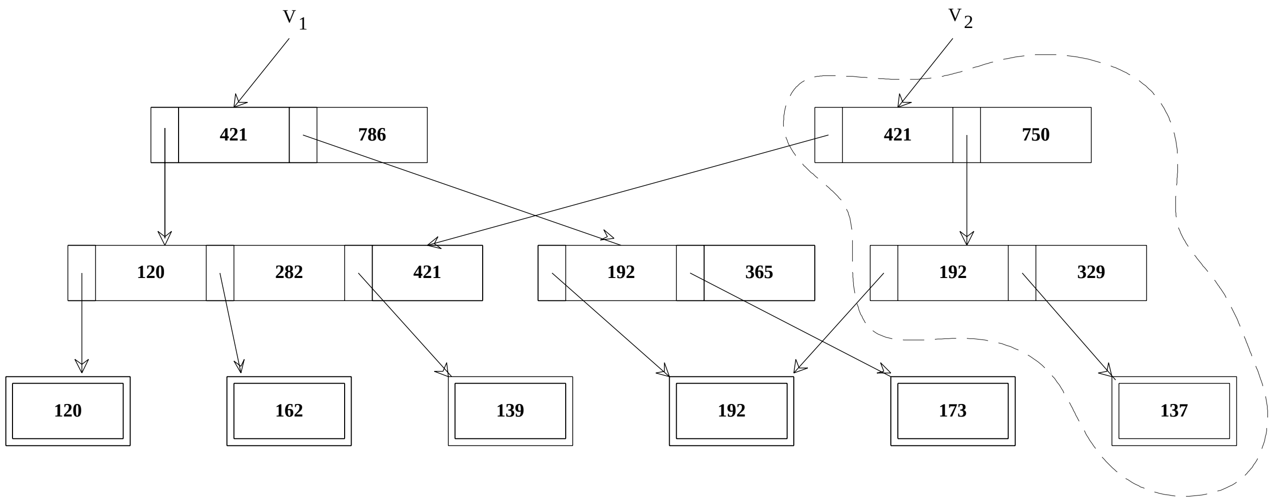

Figure 3: Two versions of a large storage object.

node. A new version of a large storage object will always contain a new copy of the path from the root to the new leaf (or leaves); it may also contain copies of other internal nodes if the change affects a large fraction of the object. Since the length of the path will usually be two or three, however, and the number of internal pages is small relative to the number of pages of actual data (due to high fanout for internal nodes), the overhead for versioning large objects in this scheme is small — for a given fixed tree height, it is basically proportional to the difference between adjacent versions, and not to the size of the objects.

Besides allowing for the creation of new versions of large storage objects, which is supported by allowing the insert, append, delete, and write (i.e., read and modify a byte range) operations to be invoked with versioning turned on, the Storage Object Manager also supports deletion of versions. This is necessary for efficiency as well as to maintain the clean abstraction. The problem is that when deleting a version of a large object, we must avoid discarding any of the object's pages that are shared (and thus needed) by other versions of the same object. [Care86] describes an efficient version deletion algorithm that addresses this problem, providing a way to delete one version with respect to a set of other versions that are to be retained.

3.4. Concurrency Control and Recovery

The Storage Object Manager provides concurrency control and recovery services for storage objects. Two-phase locking [Gray79] of byte ranges within storage objects is used for concurrency control, with a "lock entire object" option being provided for cases where object level locking will suffice. To ensure the integrity of the internal pages of large storage objects while insert, append, and delete operations are operating on them (e.g., changing their counts and pointers), non-two-phase B+ tree locking protocols [Baye77] are employed. For recovery, small storage objects are handled using before/after-image logging and in-place updating at the object level [Gray79]. Recovery for large storage objects is handled using a combination of shadowing and logging — updated internal pages and leaf blocks are shadowed up to the root level, with updates being installed atomically by overwriting the old object header with the new header [Verh78]. The name and parameters of the operation that caused the update are logged, and a log sequence number [Gray79] is maintained on each large object's root page; this is done to ensure that operations on large storage objects can be logically undone or redone as needed. A similar scheme is used for versioned objects, but the before-image of the updated large object header (or entire small object) is retained as an old version of the object.

3.5. Buffer Management for Storage Objects

An objective of the EXODUS Storage Object Manager design is to minimize the amount of copying from buffer space that is required. A second (related) objective is to allow sizable portions of large storage objects to be scanned directly in the buffer pool by higher levels of EXODUS software. To accommodate these needs, buffer space is allocated in variable-length *buffer blocks,* which are integral numbers of contiguous pages, rather than in single-page units. When an EXODUS client requests that a sequence of N bytes be read from an object X, the non-empty portions of the leaf blocks of X containing the desired byte range will be read into one contiguous buffer block by obtaining a buffer block of the appropriate size from the buffer space manager and then reading the pages into the buffer block in (strict) byte sequence order, placing the first data byte from a leaf page in the position immediately following the last data byte from the previous page. (Recall that leaf pages of large storage objects are usually not entirely full.) A scan descriptor will be maintained for the current region of X being scanned, including such information as the OID of X, a pointer to its buffer block, the length of the actual portion of the buffer block containing the bytes requested by the client, a pointer to the first such byte, and information about where the contents of the buffer block came from. The client will receive a pointer to the scan descriptor through which the buffer contents may be accessed[6]. Free space for the buffer pool will be managed using standard dynamic storage allocation techniques, and buffer block allocation and replacement will be guided by the Storage Object Manager's hint

[6] As is discussed in Section 5, the E language actually hides this structure from the DBI.

mechanism.

3.6. File Objects

File objects are collections of storage objects, and they are useful for grouping objects together for several purposes. First, the EXODUS Storage Object Manager provides a mechanism for sequencing through all of the objects in a file, so that related objects can be placed in a common file for sequential scanning purposes. Second, objects within a given file are placed on disk pages allocated to the file, so file objects provide support for objects that need to be co-located on disk. Like large storage objects, a file object is identified by an OID which points to its root (i.e., an object header); storage objects and file objects are distinguished by a header bit. Like large storage objects, file objects are represented by an index structure similar to a B+ tree, but the key for the index is different in this case — a file object index uses *disk page number* as its key. Each leaf page of the file object index contains a collection of page numbers for slotted pages contained in the file. (The pages themselves are managed separately using standard disk allocation techniques.) The file object index thus serves as a mechanism to gather the pages of a file together, but it also has several other nice properties — it facilitates the scanning of all of objects within a given file object *in physical order* for efficiency, and it allows fast deletion of an object with a given OID from a file object (since the OID includes a page number, which is the key for the file object index). Note that since all of the objects in a file are directly accessible via their OIDs, a file object is *not* comparable to a surrogate index — any indices on the objects within a given file will contain entries that point directly to the objects being indexed, a feature important for performance. Further discussion of file object representation, operations, concurrency control, and recovery may be found in [Care86].

4. TYPES AND THE TYPE MANAGER

4.1. EXODUS Type Definition Facilities

Files in EXODUS contain what we refer to as typed objects, which are storage objects as viewed by the E programming language through its type system. Files are constrained to contain objects of only one type; this is not as restrictive as it may sound, however, because E includes union and variant as type constructors. Thus, if Employee and Department records should be stored together on disk for efficiency reasons, for example, an Emp-Dept variant type could be defined to serve as the type of file for storing the records.

The EXODUS type system allows types for typed objects to be defined by combining base types via type constructors. Base types are types that are accessible only through the set of operations defined on them; they correspond to the abstract data types of ADT INGRES [Ston86a] or of conventional programming languages. The primitive EXODUS base types include int, float, char, and enumerations. In addition, EXODUS provides support for the addition of new base types (e.g., rectangle, complex number, or even a large base type such as image) and their operations via a flexible ADT facility (described further in Section 5). The definition of such ADTs is one of the tasks of the DBI. The type constructors provided by E include pointers, fixed length arrays, insertable arrays (i.e., variable length sequences), records, unions, and variants. Type constructors may be used in a nested fashion (e.g., an array of records is permissible). Whenever possible, such as for arrays of fixed length records, the E compiler will produce code which minimizes the amount of run time interpretation incurred in accessing an instance of a constructed type.

As a final note, observe that the presence of pointers (which are physically realized as object IDs) together with the other type constructors makes it possible to model more complex recursive types (i.e., typed objects within other typed objects) at a higher level of an application-specific DBMS. We expect this extensible type system to be sufficiently powerful to serve most applications satisfactorily.

4.2. The Role of the Type Manager

The Type Manager provides a facility for the storage of all persistent type information. In addition to E type definitions, it maintains information about all of the pieces (called *fragments*[7]) that go into making up a compiled query, as well as information about the relationship of these pieces to each other. It also keeps track of the correspondence of files to their types. In short, the Type Manager keeps track of type information and most everything else that is related to or dependent upon such information.

Since the fragments of a query include the types of the objects stored in the files that it references, there is a close relationship between what the Type Manager does and what traditional schema facilities do. The Type Manager does not, however, provide a complete schema facility for for end users, as it does not store information about things such as cardinalities or protection and security. Requirements for such non-type-related schema information are expected to vary from application to application, so maintaining this sort of information is left to each application-specific DBMS. The DBI is expected to maintain a set of catalogs for storing such information, presum-

[7] The type definitions themselves are also referred to as fragments, just like all other pieces of query-related information.

ably defined in terms of the end-user data model that the target system is designed to support.

4.2.1. Fragments

EXODUS is based on a *late binding model* of database architecture. That is, database systems built using EXODUS tools compile information on the database structure and operations into fragments that are stored by the Type Manager. Compiled query plans are the last fragments bound, and they are compiled and linked (sometime prior to the execution of the query) to fragments that were compiled earlier. This process requires that the Type Manager maintains dependency information between the fragments. As described in the next section, maintaining this ordering information is an important part of the Type Manager's job.

Files are treated like fragments, but they slightly different in that they do not exist as code. To the Type Manager, a file is a triple of the form <*fname, ftype, fid*>, where *fname* is the character string naming the file, *ftype* is the name of the type of the file, and *fid* is its file id as defined by the Storage Manager.

4.2.2. Dependencies

As indicated above, certain time ordering constraints must hold between all fragments constituting a complete query, including files. For example, a compiled and linked query plan must have been created more recently than the program text for any of the methods or ADT operations that it employs; otherwise, out-of-date code will have been used in its creation. In addition, we observe that a given type or set of operations is likely to have several representations: the E source code, perhaps an intermediate, parsed representation, a linkable object, and, for queries, an executable binary. Similar time ordering constraints must also hold between these representations.

This is not unlike the problem of determining whether or not the constituent parts of a large software system are all up to date, and in fact the functionality of the Type Manager shold not be unfamiliar to users of the Unix™ **make** facility [Feld79]. However, unlike **make**, which only examines dependencies and timestamps when it is started up, the Type Manager maintains a graph of inter-fragment dependencies at all times; this graph may change as fragments and dependencies are added to and subtracted from the database.

The Type Manager also plays a role in maintaining data abstraction that distinguishes it from **make**. In particular, a type used by a query plan is likely to in turn use other types to constitute its internal representation. The first type is not, strictly speaking, *dependent* upon the linkable object code of these constituent types; that is, while it must be eventually be linked with with their code, it is not necessary that their object code be up to date, or even compiled, until link time. We call fragments of this sort *companions*; **make** has no facilities for specifying and using companions. The Type Manager, however, requires such a facility, as otherwise it would be unable to provide a complete list of objects constituting a query, which is necessary when a query is to be linked.

4.2.3. Rules and Actions

The Type Manager maintains the correct time ordering of fragments via two mechanisms: *rules* and *actions*. The set of fragments constitutes the nodes of an acyclic directed graph; rules generate the arcs of this graph. When a fragment is found to be older than those fragments upon which it depends (with the dependencies being determined from the rules), a search is made for an appropriate action that can be performed to bring the fragment up to date. Both rules and actions are defined using a syntax based on regular expressions so as to allow a wide range of default fragment dependencies to be specifed with a minimum of actual rule text. Disambiguating heuristics exist to deal with possible action conflicts (such as when two regular expressions match the same fragment name).

5. ACCESS METHODS IN EXODUS

Application-specific database systems will undoubtedly vary from one another in the access methods that they employ. For example, while B+ trees and an index type based on some form of dynamic hashing are usually sufficient for conventional business database systems (e.g., a relational DBMS), a database system for storing and manipulating spatial data is likely to need a spatial index structure such as the KDB tree [Robi81], R tree [Gutt84], or Grid file [Niev84] structures. We plan to provide a library of available access methods in EXODUS, but we expect this library to grow — new, specialized index structures will undoubtedly continue to be developed as emerging database applications seek higher and higher performance. A complication is that a given index structure is expected to be able to handle data of a variety of types (e.g., integers, reals, character strings, and even newly-defined types) as long as the data type meets the prerequisites for correct operation of the index structure (e.g., a B+ tree requires the existence of a total ordering operator for its key type) [Ston86a]; this includes operating on data types that are not defined by the DBI until after the index code has been completely written and debugged.

As described in Section 2, access methods reside on top of the Storage Object Manager of EXODUS in the architecture of application-specific database systems. In addition, type information missing at compile time must be somehow provided to the access method code at run time. One of the goals of the EXODUS project is to simplify the task of adding new access methods to a new or existing application-specific database system. The major sources of complexity in adding a new access method seem to be (i) programming (and verifying) the access method algorithms, (ii) mapping the access method data structure onto the primitive objects provided by the storage system, (iii) making the access method code interact properly with the buffer manager, and (iv) ensuring that concurrency control and recovery are handled correctly and efficiently. Although the access method designer is probably only interested in item (i), this can comprise as little as 30% of the actual code that he or she must write in order to add an access method to a typical commercial DBMS, with items (ii)-(iv) comprising the remaining 70% [Ston85]. To improve this situation — dramatically, we hope — EXODUS provides a programming language for the DBI to use when implementing new access methods (and other operations). This language, E, effectively shields the DBI from items (ii)-(iv) — the E translator produces code to handle these details based on the DBI's index code plus a few declarative "hints".

In the remainder of this section, we outline the way in which new access methods are added to EXODUS, including how the programming constructs chosen for E simplify the writing of access method code, and how buffering, concurrency control, and recovery issues are handled "under the covers" in a nearly transparent fashion. We should note that many important details of the design of E are necessarily omitted from the following discussion; it is intended only to give the reader the general introduction to the ideas. For more detail, see [Rich86].

5.1. The E Implementation Language

The E language is a derivative of C [Kern78] with the addition of a set of programming constructs carefully chosen to provide high leverage to the DBI. A number of these constructs were inspired by developments in programming languages over the last 10 years, most notably, from CLU [Lisk77] and Pascal [Jens75]. Its major features include the ability to bind a pointer variable to an object in a file, and to declare abstract data types in the spirit of CLU clusters. Program structure is fully modular, with separate compilation possible for all modules (including parameterized modules, which have a certain amount of missing type information). In addition there are several new type constructors and control abstractions.

By providing these facilities, E allows the DBI to define and then to manipulate the internal structure of storage objects for an access method (e.g., a B+ tree node) in a more natural way than by making direct calls to the Storage Object Manager and explicitly coding structure overlays and offset computations. In particular, E allows the DBI to ignore the fact that storage objects can be arbitrarily large; the E translator will insert appropriate calls to get the storage object byte ranges needed by the DBI. The output of the E language translator is C source code with EXODUS-specific constructs replaced by collections of appropriate lower-level C constructs, additional routine parameters, and calls to the Storage Object Manager. In other words, the Storage Object Manager is effectively the E translator’s "target machine", and the resulting C source code will be linked with the Storage Object Manager.

In addition to these facilities, the E language provides the DBI with declarative access to the Storage Object Manager’s hint facilities (which were described in Section 3). Associated with each file-bound pointer variable in an E source program is a scan descriptor as described in Section 3; such a variable inherits the hints associated with its type definition. In the absence of hints, E will provide reasonable default assumptions, but hints make it possible for a knowledgeable DBI to tune the performance of his or her code by recommending the appropriate lock protocol, buffer replacement strategy, storage object size, etc., to be associated with a persistent object (scan) variable. E will also provide a hint mechanism that will permit the DBI to influence the way that E types are laid out on secondary storage (providing the dual of a buffering hint, in a sense).

5.2. Writing Access Methods in E

To demonstrate the usefulness of the E implementation language and to further explain its features, let us consider how the DBI might go about implementing B+ trees. We define B+ tree as an abstract data type, that is, as a type whose operations are available to users of the type, but whose internal representation is hidden. In this case, the operations probably include *create_index, destroy_index, lookup, insert,* and *delete*. To describe the internal structure of a B+ tree node, the DBI would define a C-like structure to represent its contents. Within an ADT module, there may be many typedefs; the one which has the same name as the ADT is the representation type. The ADT module *BTree* defined in Figure 4 is an example of such a definition; the variant typedef BTree is the representation type for the ADT.

Note that the module has several parameters representing the "unknowns" at the time the DBI is writing the code. These include the type of the key over which the index is built, the ordering operators on those keys, and the type of entity being indexed; within the module, these parameters are used freely as type and procedure names. The implementation of parameterized types such as BTree is such that the unknown quantities are compiled into extra, hidden parameters to the routines that form the ADT’s interface.

```
adt BTree[ entity_type, key_type, equal, less ]
type entity_type;
type key_type has int equal( ), int less( );
{
   typedef enum { INTERNAL, LEAF } NodeType;

   typedef struct {   key_type  value;    BTree  *child;   } key_ptr_pair;

   typedef struct {   key_type  value;    entity_type  *ptr[ ? ];   } key_ptr_list;

   typedef variant {  NodeType   nodetype;
            INTERNAL: {  int  height;   key_ptr_pair  data[ ? ];  };
            LEAF:          {  key_ptr_list  data[ ? ];  };
   } BTree # obj(PAGESIZE);  lock(HIERARCHICAL);  buffer(LIFO, 3) #;

   /* figure out which pointer to follow */

   private int search_node( node, key )
   BTree   *node;
   key_type    key;
   {
            /* simple binary search over node */
            int            min, max, middle;

            min = 0;
            max = lengthof( node->data ) - 1;
            while( min <= max ) {
               middle = (min + max)/2;
               if( equal(key, node->data[middle].value) )  return( middle );
               else if( less(key, node->data[middle].value) )  max = middle - 1;
               else  min = middle + 1;
            }

            return( -1 );
   } /* search_node */

   /* find first entity with specified key */

   public entity_type *BT_lookup( key )
   key_type key;
   {
            ...
   } /* BT_lookup */

   ...

} /* BTree */
```

Figure 4: A Partial B+ Tree Example.

BTree's definition also hints[8] to the E translator that new B+ tree nodes should be one page in size, that

[8] The hint facilities are currently being designed, so the hint syntax used in our example is preliminary; our intent is simply to convey the

hierarchical locking should be used in B+ tree operations, and that a LIFO buffer management policy with three buffer blocks should be used for the scan. (Buffers and locks are allocated on a per scan basis.)

Given these definitions, the DBI can proceed to access and manipulate storage objects as though they were standard in-memory structures. Figure 4 also gives an example code fragment from a B+ tree search routine. In this routine, the parameter *node* is a pointer to a BTree. Each time the E translator encounters a statement in which *node* is dereferenced, it will translate this reference into a sequence of several C statements — at runtime this sequence will check to see if the appropriate bytes of the object are already in the buffer pool by inspecting *node*'s scan descriptor, calling the Storage Object Manager to read the desired bytes (and perhaps subsequent bytes) into the buffer pool if not; then the actual reference will take place in memory. Since *key_type* is unknown when this code is compiled, the E translator will compile code such that the needed information (e.g. its size) is passed in under the covers; the resulting offset calculations which index into the key-pointer-pair arrays will make use of these parameters.

E provides other operations on pointers into files as well. For example, the call *new(n)* creates a new object in the file to which *n* is bound and sets *n* to point at it; *free(n)* disposes of the object to which *n* is pointing. Other calls will also be provided, including calls to create and destroy files. The E translator will recognize these calls and replace them with appropriate lower-level Storage Object Manager calls.

5.3. Transparent Transaction Management

We have described how the E language simplifies the DBI's job by allowing access method structures and algorithms to be expressed in a natural way, and we have indicated how the E translator adds Storage Object Manager calls when producing C code. In this section we describe how concurrency control and recovery fit into the picture; the problem is complicated by the fact that access methods often require non-two-phase locking protocols and have specialized recovery requirements for performance reasons [Ston86a]. These functions will be handled by the E translator through a combination of *layered transactions* and *protocol stubs*.[9]

flavor of the hint facilities that E will provide.

[9]Currently, we are focusing our efforts on the difficult problem of automatically buffering pieces of large objects; our thoughts on how to deal with concurrency control and recovery are still in a preliminary state.

5.3.1. Layered Transactions

Transaction management for access methods in EXODUS is loosely based on the layered transaction model proposed by Weikum [Weik84]. Weikum's model is based on the notion of architectural layers of a database system, with each layer presenting a set of objects and associated operations to its client layers. Each operation is a "mini-transaction" (or nested transaction) in its own right, and thus a transaction in a client layer can be realized as a series of mini-transactions in one or more of its servant layers. Concurrency control is enforced using two-phase locking on objects within a given layer of the system. Objects in a servant layer are locked on behalf of the transaction in its client layer, and these locks are held until the client transaction completes. Recovery is layered in a similar manner. As a mini-transaction executes, it writes level-specific recovery information to the log; when it completes, its log information is removed and replaced by a simpler client-level representation of the entire operation. To undo the effects of an incomplete layered transaction at a given level, the effects of a number of completed mini-transactions plus one in-progress mini-transaction must be undone; we must first undo the incomplete mini-transaction (recursively, in general) using its log information, then run the inverse of each completed mini-transaction. Weikum proposes what amounts to a per-transaction stack-based log for recovery.

While we draw much inspiration from Weikum, our access method transaction management facilities differ in some respects. First, EXODUS is not strictly hierarchical in nature, instead being a collection of interacting modules. (This does not invalidate the notion of a layered transaction, however.) Also, we provide more general locking than the strict two-phase model in Weikum's proposal, allowing locks set by a servant to be either explicitly released or passed to its client. This is particularly important for access methods, as two-phase locking (even within a single index operation) is often considered to be unacceptable [Baye77]. Lock passing is also needed to prevent phantoms when a new key-pointer pair is inserted into an index — unless the client retains a lock on the index leaf page, other transactions may run into consistency problems due to incorrect existence information. Lastly, efficient log management is essential to overall performance, and we view Weikum's per-transaction stack-based log as too unwieldy. Instead, we employ standard circular log management techniques, ignoring entries for completed mini-transactions during recovery processing.

Returning to our discussion of access methods, note that the access methods layer presents objects (e.g., indices) and operations (e.g., insert, delete, search, etc.) to its clients. If a client transaction executes a series of inserts, its effects can be undone via a series of corresponding deletes. The access methods layer, in turn, is a client of the Storage Object Manager, which presents storage objects and such operations as create object, insert bytes, append bytes, etc.

5.3.2. Protocol Stubs

Layered transactions will simplify the task of writing access methods because calls to other layers can be viewed as primitive, atomic operations. However, this is just a transaction model; the task of actually implementing the model still remains. For example, someone must still write the code to handle B+ tree locking and recovery, and getting this correct can be quite difficult. EXODUS will provide a collection of protocol stubs, managed by the E compiler, to shield the DBI from the details of this problem as much as possible. Briefly, a protocol stub is an abstraction of a particular locking protocol, implemented as a collection of code fragments (which the E translator inserts at appropriate points during the compilation of an E program) plus related data structures. The code fragments consist of locking/logging calls to the EXODUS transaction manager (a component of the Storage Object Manager). The data structures describe information on lock modes (and their compatibility) which is passed to and used by the lock manager. We currently expect that the generation of new protocol stubs will be a complicated task, and that stubs will be considered by the average DBI as being a non-extensible part of the basic EXODUS system. The *use* of existing stubs will be easy, on the other hand, and EXODUS will provide a collection of stubs for two-phase locking and for the hierarchical locking (or lock chaining) protocol of [Baye77]. A DBI writing a new access method will only need to (1) select the desired protocol at compile time via E's declarative hints, as mentioned earlier; (2) bracket each access method operation with begin and end transaction keywords in E; and then maybe (3) include one or two stub-specific routine calls in the access method operation code (an example of which is given below).

As a concrete example, we briefly sketch a protocol suitable for concurrent and recoverable access to most sorts of hierarchical index structures. The basic idea is to use the B+ tree lock-chaining protocol of [Baye77] for concurrency control, and to use shadowing for operation-atomicity [Verh78]. Consider a B+ tree insert operation: Using lock chaining as we descend the tree, we can release locks on all ancestors of a safe node once we have locked that node. To realize this protocol, the hierarchical locking protocol stub will implicitly set locks on nodes as they are referenced, keeping track of the path of locked nodes. When the DBI's code determines that a "safe" node has been reached, it can call a lock stub routine called *top-of-scope* to announce that previously accessed nodes (excluding the current one) are no longer in the tree scope of interest to the insert operation. The appropriate lock release operations can then be transparently handled by the lock stub routine. As for recovery, the insert operation will cause node splitting to occur up to the last *top-of-scope*. If changed nodes below this level are automatically shadowed, then the insert can be atomically installed at end-of-operation by overwriting the *top-of-scope* node after its descendent pages have been safely written to disk [Care85b]. (While the Storage Object Manager does not directly support shadow-based recovery, the E translator can generate C code which uses the versioning mechanism of the Storage Object Manager to accomplish this task.)

5.4. Other Uses of E

We have described briefly how the E language provides the DBI with a facility for writing access method operations without worrying about such issues as the size of objects, making calls to the Storage Object Manager, or access method specific concerns of concurrency control and recovery. E is actually more than just an implementation language for access methods. The E compiler is really at the heart of an EXODUS system. For example, if some application needs a specialized fundamental type then the new type and its operations are written in E by the DBI. Since such operations may need to deal with arbitrarily large portions of objects — for example, the DBI might wish to add an ADT called "matrix" and then provide a matrix multiplication operator — the DBI’s job will be significantly simpler if he or she can write the desired code without regard for the size of the underlying storage objects. Finally, the operators which implement the application’s data model are also written in E.

6. OPERATOR METHODS

The Operator Methods Layer contains the E procedures used to implement the operators provided to the user of the database system. For each operator, one or more procedures (or methods) may exist. For example, in a relational DBMS this layer might contain both nested-loops and sort-merge implementations of the relational join operation. In general, the operators associated with a data model are *schema independent*. That is, the operators (and their corresponding implementations) are defined independently of any conceptual schema information — the join operator, for example, will join any two relations as long as the corresponding join attributes are compatible with one another (even if the result happens to be semantically meaningless).

There are two strategies for implementing such generic operators. First, the procedures implementing the operators could request the necessary schema information at run-time from the Type Manager. The second strategy is to have the query optimizer and compiler compile the necessary schema information into code fragments that the compiled query can pass to the operator method at run-time. For example, in the case of the join, the optimizer would produce four code fragments: two to extract the source relation join attributes (with one procedure for each source relation), one to compare the two join attributes, and one to compose a result tuple from the two source tuples. Again, the solution for this layer is the use of parameterized modules; it is the types of the tuples being joined (or projected, or...) which are unknown in this case.

Instead of providing generic (and, hence, semantics-free) operators to the database users, a number of researchers [Webe78, Rowe79, Derr86, Lyng86] have proposed to provide only "schema dependent" operations to the users. For example, in a database of employees and departments, the type of operations supported would be of the form hire-employee, change-job, etc. When the hire-employee operation is invoked, the necessary base entities are updated in such a fashion as to insure that the database remains consistent. Given the capabilities of EXODUS, implementing this style of operators is quite obviously feasible. The DBI could implement the operators directly using the functionality provided by the Access Methods and Storage Object Manager. Alternatively, they could be implemented using more generic operators. It appears that the database administrator of an IRIS database [Derr86, Lyng86] is expected to implement the schema-specific operators using an underlying database system that is basically relational in nature.

As is the case for access methods, we anticipate providing some level of operator support via a library of operator methods. For example, most data models are likely to want methods for performing associative accesses (i.e. selection) and for scanning through all of the objects contained in a particular file object.

7. RULE BASED QUERY OPTIMIZATION AND COMPILATION

Given the unforeseeably wide variety of data models we hope to support with EXODUS, each with its own operators (and corresponding methods), EXODUS includes an optimizer *generator* that produces an application-specific query optimizer from an input specification. The generated optimizer repeatedly applies algebraic transformations to a query and selects access paths for each operation in the transformed query. This transformational approach is outlined by Ullman for relational DBMSs [Ullm82], and it has been used in the Microbe database project [Nyug82] with rules coded as Pascal procedures. We initially considered using a rule-based AI language to implement a general-purpose optimizer, and then to augment it with data model specific rules. Prolog [Warr77, Cloc81], OPS5 [Forg81], and LOOPS [Bobr83] seemed like interesting candidates, as each provides a built-in "inference engine" or search mechanism. However, this convenience also limits their use, as their search algorithms are rather fixed and hard to augment with search heuristics (which are very important for query optimization).

Based on this limitation, and also on further considerations such as call compatibility with other EXODUS components and optimizer execution speed, we decided instead to provide an optimizer generator [Grae86] which produces an optimization procedure in the programming language C [Kern78].

The generated optimization procedure takes a query as its input, producing an access plan as its output. A query in this context is a tree-like expression with logical operators as internal nodes (e.g., a join in a relational DBMS) and sets of objects (e.g., relations) as leaves. We do not regard it as part of the optimizer's task to produce an initial algebraic query tree from a non-procedural expression; this will be done by the user interface and parser. An access plan is a tree with operator methods as internal nodes (e.g., a hash join method) and with files or indices as leaves. Once an access plan is obtained, it will then be transformed into an iterative program using techniques due to Freytag [Frey85, Frey86].

There are four key elements which must be given to the optimizer generator (in a description file) in order for it to generate an optimizer: (1) the operators, (2) the methods, (3) the transformation rules, and (4) the implementation rules. Operators and their methods are characterized by their name and arity. Transformation rules specify legal (equivalence-preserving) transformations of query trees, and consist of two expressions and an optional condition. The expressions contain place holders for lower parts of the query which will not be affected by the transformation, and the condition is a C code fragment which is inserted into the optimizer at the appropriate place. Finally, an implementation rule consists of a method, an expression that the method implements, and an optional condition. As an example, here is an excerpt from the description file for a relational DBMS:

```
%operator 2 join
%method 2 hash-join merge-join
join (R, S) <-> join (S, R);
join (R, S) by hash-join (R, S);
```

Both the operator and method declarations specify the number of inputs. The symbol "<->" denotes equivalence, and "by" is a keyword for implementation rules. If merge-join is only useful for joining sorted relations, then a rule for merge-join would have to include a condition to test whether each input relation is sorted.

In addition to this declarative description of the data model, the optimizer requires the DBI to supply a collection of procedures. First, for each method, a cost function must be supplied that calculates the method's cost given the characteristics of the method's input. The cost of an access plan is defined as the sum of the costs of the methods involved. Second, a property function is needed for each operator and each method. Operator property functions determine logical properties of intermediate results, such as their cardinalities and record widths. Method property functions determine physical properties (ie. side effects), such as sort order in the example above.

The generated optimization procedure operates by maintaining two principal data structures, MESH and OPEN. MESH is a directed acyclic graph containing all the alternative operator trees and access plans that have been explored so far. A rather complex pointer structure is employed to ensure that equal subexpressions are stored and optimized only once, and also that accesses and transformations can be performed quickly. OPEN is a priority queue containing the set of applicable transformations; these are ordered by the cost decrease which would be expected from applying the transformations.

MESH is initialized to contain a tree with the same structure as the original query. The method with the lowest cost estimate is selected for each node using the implementation rules, and then possible transformations are determined and inserted into OPEN using the transformation rules. The optimizer then repeats the following transformation cycle until OPEN is empty: The most promising transformation is selected from OPEN and applied to MESH. For all nodes generated by the transformation, the optimizer tries to find an equal node in MESH to avoid optimizing the same expression twice. (Two nodes are equal if they have the same operator, the same argument, and the same inputs.) If an equal node is found, it is used to replace the new node. The remaining new nodes are matched against the transformation rules and analyzed, and methods with lowest cost estimates are selected.

This algorithm has several parameters which serve to improve its efficiency. First, the *promise* of each transformation is calculated as the product of the top node's total cost and the *expected cost factor* associated with the transformation rule. A matching transformation with a low expected cost factor will be applied first. Expected cost factors provide an easy way to ensure that restrictive operators are moved down in the tree as quickly as possible; it is a general heuristic that the cost is lower if constructive operators such as join and transitive closure have less input data. Second, while it seems to be wasted effort to perform an equivalence transformation if it does not yield a cheaper solution, sometimes such a transformation is necessary as an intermediate step to an even less expensive access plan. Such transformations represent hill climbing, and we limit their application through the use of a *hill climbing factor*. Third, when a transformation results in a lower cost, the parent nodes of the old expression must be reanalyzed to propagate cost advantages. It appears to be a difficult problem to select values for each of these parameters which will guarantee both optimal access plans and good optimizer performance. Thus, it is would be nice if they could be determined and adjusted automatically. Our current prototype initializes all expected cost factors to 1, the neutral value, and then adjusts them using sliding geometric averages. This has turned out to be very effective in our preliminary experiments. We are currently experimenting with the hill climbing and reanalyzing factors to determine the best method of adjustment.

8. EXODUS USER INTERFACES

As discussed in Section 2, a database system must provide facilities for both ad hoc and embedded queries. While tuple-at-a-time and portal [Ston84] interfaces look appropriate for record-oriented database systems, we have only just begun thinking about how to provide a more general technique for handling embedded queries in programs. Certainly, given the goals of the EXODUS project, we will need to develop data model independent techniques to interface programs to application specific database systems, but this may prove to be quite difficult. For example, it is hard to envision a generic interface tool that could satisfactorily interface a VLSI layout tool to a VLSI database system; in such an environment, it may be that the only sensible approach is to treat the application program and its procedures as operators in the database system, thus enabling the program to directly access typed objects in the buffer pool. Alternatively, it may be possible to provide a library of interface tools: portals for browsing sets of objects, graphical interfaces for other applications, etc. We intend to explore alternative solutions to this problem in the future.

For ad hoc query interfaces, tools based on attribute grammars appear promising. Unlike the grammars used by generators like YACC, which can be used for little besides parsing the syntax of an input query, grammars which allow complex sets of attributes and attribution functions may capture the semantics of a query, incorporating knowledge of schema information to guide query construction, detect errors, and generate appropriate structures for transmission to the optimizer. To test these ideas we are constructing a QUEL interface using the Cornell Program Synthesizer Generator [Reps84]. The Generator takes a formal input specification, producing as its output an interactive, syntax- and semantics-driven editor similar in flavor to Emacs. For a query language, the editor will guide the user step-by-step in creating properly formed queries and will transform this calculus representation of the query into a syntax tree in the operator language recognized by the optimizer. During the process of producing this syntax tree, the editor will be responsible for translating from a calculus representation to an initial algebraic representation of the query. The editor will call on the Type Manager to provide access to schema information, as schema information determines a large part of the underlying semantics of the query. Since the concrete syntax of the query language, its abstract syntax, and the translation between the abstract syntax and the database operator language are all generated automatically from a formal specification, it should be a straightforward process to change or enhance the language recognized by the user interface.

9. SUMMARY AND CURRENT STATUS

In this paper we described the design of EXODUS, an extensible database system intended to simplify the development of high-performance, application-specific database systems. As we explained, the EXODUS model of the world includes three classes of database experts — ourselves, the designers and implementors of EXODUS; the database implementors, or DBIs, who are responsible for using EXODUS to produce various application-specific DBMSs; and the database administrators, or DBAs, who are the managers of the systems produced by the DBIs. In addition, of course, there must be users of application-specific DBMSs, namely the engineers, scientists, office workers, computer-aided designers, and other groups that the resulting systems will support. The focus of this paper has been the overall architecture of EXODUS and the tools available to aid the DBI in his or her task.

As we described, EXODUS includes two components that require little or no change from application to application — the Storage Object Manager, a flexible storage manager that provides concurrent and recoverable access to storage objects of arbitrary size, and the Type Manager, a repository for type information and such related information as file types, dependencies of types and code on other types, etc. In addition, EXODUS provides libraries of database system components that are likely to be widely applicable, including components for access methods, version management, and simple operations. The corresponding system layers are constructed by the DBI through a combination of borrowing components from the libraries and writing new components. To make writing new components as painless as possible, EXODUS provides the E database implementation language to largely shield the DBI from the details of internal object formats, buffer management, concurrency control, and recovery protocols. E is also the vehicle provided for defining new ADTs, which makes it easy for the DBI to write operations on ADTs even when they are very large (e.g., an image ADT). At the upper level of the system, EXODUS provides a generator that produces a query optimizer and compiler from a description of the available operations and methods, and tools for generating application-specific front-end software are also planned.

The initial design of EXODUS is now basically complete, including all of the components that have been described here, and implementation of several of the components has begun. Some preliminary prototyping work was done in order to validate the Storage Object Manager's algorithms for operating on large storage objects [Care86], and over half of the Storage Object Manager has been implemented since that time. A first implementation of the rule-based query optimizer generator is basically complete, and it has been used to generate most of a full relational query optimizer. The Type Manager and the E programming language translator implementation efforts are getting underway now, and we hope to have initial implementations of most of the key components of EXODUS by the middle of 1987. Soon thereafter we expect to bring a relational DBMS up on top of EXODUS as a test of our tools, and we will then begin looking for more challenging applications with which to test the flexibility of our approach.

REFERENCES

[Ait86] Ait-Kaci, H. and R. Nasr, "Logic and Inheritance," *Proceedings of the 1986 POPL Conference*, St. Petersburg, FA, January 1986.

[Allm76] Allman, E., Held, G. and M. Stonebraker, "Embedding a Data Manipulation Language in a General Purpose Programming Language," *Proceedings of the 1976 SIGPLAN-SIGMOD Conference on Data Abstraction*, Salt Lake City, Utah, March 1976.

[Astr76] Astrahan, M., et. al., "System R: Relational Approach to Database Management", *ACM Transactions on Data Systems* 1, 2, June 1976.

[Bato85] Batory, D., and W. Kim, *Support for Versions of VLSI CAD Objects*, M.C.C. Working Paper, March 1985.

[Bato86] Batory, D., Barnett, J., Garza, J., Smith, K., Tsukuda, K., Twichell, C., and T. Wise, "GENESIS: A Reconfigurable Database Management System," Technical Report, TR-86-07, Department of Computer Sciences, University of Texas at Austin, March 1986.

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

[Bobr83] Bobrow, D.G. and M. Stefik, "The LOOPS Manual," in LOOPS Release Notes, XEROX, Palo Alto, CA., 1983.

[Care85a] Carey, M. and D. DeWitt, "Extensible Database Systems", *Proceedings of the Islamorada Workshop on Large Scale Knowledge Base and Reasoning Systems*, February 1985.

[Care85b] Carey, M., DeWitt, D., and M. Stonebraker, personal communication, July 1985.

[Care86] Carey, M. J., DeWitt, D. J., Richardson, J. E., and E. Shekita, "Object and File Management in the EXODUS Extensible Database System," *Proceedings of the 1986 VLDB Conference*, Kyoto, Japan, August 1986.

[Chou85] Chou, H-T., and D. DeWitt, "An Evaluation of Buffer Management Strategies for Relational Database Systems", *Proceedings of the 1985 VLDB Conference*, Stockholm, Sweden, August 1985.

[Clif85] Clifford, J., and A. Tansel, "On An Algebra for Historical Relational Databases: Two Views", *Proceedings of the 1985 SIGMOD Conference*, Austin, Texas, May 1985.

[Cloc81] Clocksin, W. and C. Mellish, *Programming in Prolog*, Springer-Verlag, New York, 1981.

[Cope84] Copeland, G. and D. Maier, "Making Smalltalk a Database System", *Proceedings of the 1984 SIGMOD Conference*, Boston, MA, May 1984.

[Dada84] Dadam, P., V. Lum, and H-D. Werner, "Integration of Time Versions into a Relational Database System", *Proceedings of the 1984 VLDB Conference*, Singapore, August 1984.

[Daya85] Dayal, U. and J. Smith, "PROBE: A Knowledge-Oriented Database Management System", *Proceedings of the Islamorada Workshop on Large Scale Knowledge Base and Reasoning Systems*, February 1985.

[Derr86] Derrett, N., Fishman, D., Kent, W., Lyngaek, P., and T. Ryan, "An Object-Oriented Approach to Data Management," *Proceedings of the 1986 COMPCON Conference*, San Francisco, CA., February 1986.

[Feld79] Feldman, S., "Make — A Program for Maintaining Computer Programs," *Software — Practice and Experience*, vol. 9, 1979.

[Forg81] Forgy, C.L. "OPS5 Reference Manual," Computer Science Technical Report 135, Carnegie-Mellon University, 1981.

[Frey85] Freytag, C.F. "Translating Relational Queries into Iterative Programs," Ph.D. Thesis, Harvard University, September 1985.

[Frey86] Freytag, C.F. and N. Goodman, "Translating Relational Queries into Iterative Programs Using a Program Transformation Approach," *Proceedings of the 1986 ACM SIGMOD Conference*, May 1986.

[Grae86] Graefe, G. and D. DeWitt, "The EXODUS Optimizer Generator," submitted for publication, December, 1986.

[Gray79] Gray, J., "Notes On Database Operating Systems", in *Operating Systems: An Advanced Course*, R. Bayer, R. Graham, and G. Seegmuller, eds., Springer-Verlag, 1979.

[Gutt84] Guttman, T., "R-Trees: A Dynamic Index Structure for Spatial Searching", *Proceedings of the 1984 SIGMOD Conference*, Boston, MA, May 1984.

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

[Katz84] Katz, R. and T. Lehman, "Database Support for Versions and Alternatives of Large Design Files", *IEEE Transactions on Software Engineering* SE-10, 2, March 1984.

[Katz86] Katz, R., E. Chang, and R. Bhateja, "Version Modeling Concepts for Computer-Aided Design Databases", *Proceedings of the 1986 SIGMOD Conference*, Washington, DC, May 1986.

[Kern78] Kernighan, B.W. and D.N. Ritchie, *The C Programming Language,* Prentice-Hall, Englewood Cliffs, N.J., 1978.

[Kers81] Kersten, M. L. and A. I. Wasserman, "The Architecture of the PLAIN Data Base Handler," *Software — Practice and Experience,* V 11, 1981, pp. 175- 186.

[Klah85] Klahold, P., G. Schlageter, R. Unland, and W. Wilkes, "A Transaction Model Supporting Complex Applications in Integrated Information Systems", *Proceedings of the 1985 SIGMOD Conference*, Austin, TX, May 1985.

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

[Litw80] Litwin, W., "Linear Hashing: A New Tool for File and Table Addressing," *Proceedings of the 1980 VLDB Conference*, Montreal, Canada, October 1980.

[Lyng86] Lyngbaek, P. and W. Kent, "A Data Modeling Methodology for the Design and Implementation of Information Systems", *Proceedings of the International Workshop on Object-Oriented Database Systems*, Pacific Grove, CA, September 1986.

[Nguy82] Nguyen, G.T., Ferrat, L., and H. Galy, "A High-Level User Interface for a Local Network Database System," *Proceedings of the IEEE Infocom,* pp. 96-105, 1982.

[Maie86] Maier, D., Stein, J., Otis, A., and A. Purdy, "Development of an Object Oriented DBMS," *Proceedings of OOPSLA '86*, Portland, OR, September 1986.

[Mano86] Manola, F., and Dayal, U., "PDM: An Object-Oriented Data Model," *Proceedings of the International Workshop on Object-Oriented Database Systems*, Pacific Grove, CA, September 1986.

[Niev84] Nievergelt, J., H. Hintenberger, H., and Sevcik, K.C., "The Grid File: An Adaptable, Symmetric Multikey File Structure," *ACM Transactions on Database Systems,* Vol. 9, No. 1, March 1984.

[Reps84] Reps, T. and T. Teitelbaum, "The Synthesizer Generator," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments,* Pittsburgh, Penn., Apr. 23-25, 1984. Appeared as joint issue: *SIGPLAN Notices* (ACM) *19,* 5, May 1984, and *Soft. Eng. Notes* (ACM) *9,* 3, May 1984, 42-48.

[Rich86] Richardson, J., and Carey, M., "Programming Constructs for Database System Implementation in EXODUS", submitted for publication.

[Robi81] Robinson, J.T., "The k-d-B-tree: A Search Structure for Large Multidimentional Dynamic Indexes," *Proceedings of the 1981 SIGMOD Conference,* June, 1981.

[Rowe79] Rowe, L. and K. Schoens, "Data Abstraction, Views, and Updates in RIGEL, *Proceedings of the 1979 SIGMOD Conference*, Boston, MA., 1979.

[Schm77] Schmidt, J., "Some High Level Constructs for Data of Type Relations," *ACM Transactions on Database Systems,* 2, 3, September 1977.

[Schw86] Schwarz, P., et al, "Extensibility in the Starburst Database System," *Proceedings of the International Workshop on Object-Oriented Database Systems,* Pacific Grove, CA, September 1986.

[Ship81] Shipman, D., "The Functional Data Model and the Data Language DAPLEX", *ACM Transactions on Database Systems* 6, 1, March 1981.

[Shop79] Shopiro, J., "Theseus — A Programming Language for Relational Databases," *ACM Transactions on Database Systems* 4, 4, December 1979.

[Snod85] Snodgrass, R., and I. Ahn, "A Taxonomy of Time in Databases", *Proceedings of the 1985 SIGMOD Conference*, Austin, TX, May 1985.

[Ston81] Stonebraker, M., "Hypothetical Data Bases as Views", *Proceedings of the 1981 SIGMOD Conference*, Boston, MA, May 1981.

[Ston83] Stonebraker, M., H. Stettner, N. Lynn, J. Kalash, and A. Guttman, "Document Processing in a Relational Database System", *ACM Transactions on Office Information Systems* 1, 2, April 1983.

[Ston84] Stonebraker, M., and L. Rowe, "Database Portals - A New Application Program Interface," *Proceedings of the 1984 VLDB Conference,* Singapore, August 1984.

[Ston85] Stonebraker, M., personal communication, July 1985.

[Ston86a] Stonebraker, M., "Inclusion of New Types in Relational Data Base Systems," *Proceedings of the 2nd Data Engineering Conference,* Los Angeles, CA., February, 1986.

[Ston86b] Stonebraker, M., and L. Rowe, "The Design of POSTGRES", *Proceedings of the 1986 SIGMOD Conference*, Washington, DC, May 1986.

[Ston86c] Stonebraker, M., "Object Management in POSTGRES Using Procedures," *Proceedings of the International Workshop on Object-Oriented Database Systems,* Pacific Grove, CA, September 1986.

[Ullm82] Ullman, J.D., *Principles of Database Systems,* Computer Science Press, Rockville, MD., 1982.

[Verh78] Verhofstad, J., "Recovery Techniques for Database Systems", *ACM Computing Surveys* 10, 2, June 1978.

[Warr77] Warren, D.H., Pereira, L.M., and F. Pereira, "PROLOG — The Language and its Implementation Compared With Lisp," *Proceedings of ACM SIGART-SIGPLAN Symp. on AI and Pro- gramming Languages,* 1977.

[Webe78] Weber, H. "A Software Engineering View of Database Systems," *Proceedings of the 1978 VLDB Conference,* pp. 36-51, 1978.

[Weik84] Weikum, G., and H-J. Schek, "Architectural Issues of Transaction Management in Multi-Layered Systems," *Proceedings of the 1984 VLDB Conference*, Singapore, August 1984.