# The EXODUS Extensible DBMS Project: An Overview

*Michael J. Carey, David J. DeWitt,*
*Goetz Graefe, David M. Haight,*
*Joel E. Richardson, Daniel T. Schuh,*
*Eugene J. Shekita, and Scott L. Vandenberg*

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

**ABSTRACT**

This paper presents an overview of EXODUS, an extensible database system project that is addressing data management problems posed by a variety of challenging new applications. The goal of the project is to facilitate the fast development of high-performance, application-specific database systems. EXODUS provides certain kernel facilities, including a versatile storage manager. In addition, it provides an architectural framework for building application-specific database systems; powerful tools to help automate the generation of such systems, including a rule-based query optimizer generator and a persistent programming language; and libraries of generic software components (e.g., access methods) that are likely to be useful for many application domains. We briefly describe each of the components of EXODUS in this paper, and we also describe a next-generation DBMS that we are now building using the EXODUS tools.

## 1. INTRODUCTION

Until fairly recently, research and development efforts in the database systems area have focused primarily on supporting traditional business applications. The design of database systems capable of supporting non-traditional application areas, such as computer-aided design and manufacturing, scientific and statistical applications, large-scale AI systems, and image/voice applications, has now emerged as an important research direction. Such new applications differ from conventional database applications and from each other in a number of important ways. First of all, their data modeling requirements vary widely. The kinds of entities and relationships relevant to a VLSI circuit design are quite different from those of a banking application. Second, each new application area has a different, specialized set of operations that must be efficiently supported by the database system. For example, it makes little sense to talk about doing joins between satellite images. Efficient support for such specialized operations also requires new types of storage structures and access methods. For applications like VLSI design, involving spatial objects, R-Trees [Gutt84] are a useful access method for data storage and manipulation; to manage image data efficiently, the database system needs to provide large arrays as a basic data type. Finally, a number of new application areas require support for multiple versions of their entities [Snod85, Daya86, Katz86].

A number of research projects are addressing the needs of new applications by developing approaches to making a database system *extensible* [DBE87]. These projects include EXODUS[1] at the University of Wisconsin [Care86a, Carey86c], PROBE at CCA [Daya86, Mano86], POSTGRES at UC Berkeley [Ston86b, Rowe87], STARBURST at IBM Almaden Research Center [Schw86, Lind87], and GENESIS at the University of Texas-Austin [Bato88a, Bato88b]. Although the goals of these projects are similar, and each uses some of the same mechanisms to provide extensibility, their overall approaches are quite different. For example, POSTGRES is a complete

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 two Fellowships, by DEC through its Incentives for Excellence program, and by donations from Apple Corporation, GTE Laboratories, the Microelectronics and Computer Technology Corporation (MCC), and Texas Instruments.

[1] EXODUS: A departure; in this case, from traditional approaches to database management. Also an **EX**tensible **O**bject-oriented **D**atabase **S**ystem.

database management system, with a query language (POSTQUEL), a predefined way of supporting complex objects (through the use of procedures as a data type), support for "active" databases via triggers and alerters, and inferencing. Extensibility is provided via new data types, new access methods, and a simplified recovery mechanism. A stated goal is to "make as few changes as possible to the relational model." The PROBE system, on the other hand, is an advanced DBMS with support for complex objects and operations on them, dimensional data (in both space and time dimensions), and a capability for limited recursive query processing. Unlike POSTGRES, PROBE provides a mechanism for directly representing complex objects; the PROBE query language is an extension of DAPLEX [Ship81]. STARBURST is 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. Like EXODUS, STARBURST uses a rule-based approach to query optimization to enable it to handle such extensions [Lohm88].

In contrast to these efforts, EXODUS and GENESIS are modular and modifiable systems, rather than being complete, end-user DBMSs for handling all new application areas. The GENESIS project is aimed at identifying primitive building blocks, together with facilities for describing how to combine building blocks, in order to allow a new DBMS to be automatically composed from a library of existing database components. The goal of the EXODUS project (which is in some sense a "database software engineering" project) is to provide a collection of kernel DBMS facilities together with software tools to enable the semi-automatic construction of an application-specific DBMS for a given new application area. Included in EXODUS are tools intended to simplify the development of new DBMS components (e.g., a new access method or a new query language operator).

In this paper we describe the EXODUS approach to achieving extensibility. Section 2 of the paper provides an overview of the various components of EXODUS. Section 3 describes the lowest level of the system, the Storage Manager. Section 4 discusses the EXODUS approach to handling two difficult tasks involved in extending a database system: implementing new access methods, and implementing new, application-specific database operations. EXODUS simplifies these tasks by providing a programming language called E, which extends C++ [Stro86] with facilities for persistent systems programming. Section 5 describes the rule-based approach to query optimization employed in EXODUS. Section 6 describes EXTRA and EXCESS, a data model and query language that we are now building using the aforementioned tools. Finally, Section 7 summarizes the paper and discusses the implementation status of the various components of the EXODUS project.

## 2. AN OVERVIEW OF THE EXODUS ARCHITECTURE

Since one of the principal goals of the EXODUS project is to provide extensibility without sacrificing performance, the design of EXODUS reflects a careful balance between what EXODUS provides for the *user*[2] and what the *user* must explicitly provide. Unlike POSTGRES, PROBE, or STARBURST, EXODUS is not intended to be a complete system with provisions for user-added extensions. Rather, it is intended more as a "toolkit" that can be easily adapted to satisfy the needs of new application areas. In this section we summarize our overall approach and briefly introduce each of the key components and tools of EXODUS.

### 2.1. The EXODUS Approach

Two basic mechanisms are employed in EXODUS to help achieve our extensibility and performance goals: First, where feasible, we furnish a generic solution that should be applicable to database systems for most any application area. As an example, EXODUS supplies at its lowest level a layer of software termed the Storage Manager which provides support for concurrent and recoverable operations on storage objects of any size. Our feeling is that this level provides sufficient capabilities such that user-added extensions will not be necessary. However, due to both generality and efficiency considerations, such a single, generic solution is not possible for every component of a database system.

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

In cases where a single, generic solution is inappropriate, EXODUS instead provides either a *generator* or a *library* to aid the user in constructing 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; this also makes it impossible for a single query optimizer to suffice for all applications. Instead, we provide a generator for producing query optimizers for algebraic query 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., moving selections before joins in a relational algebra query), 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 a C program.

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 loosely 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). Rather, this user of the EXODUS facilities is a "database engineer" or DBE; our goal has been to engineer EXODUS so that only a moderate amount of database expertise is needed in order for a DBE to architect a new system using the tools. Once EXODUS has been customized into an application-specific database system, the DBE's initial role is completed and the role of the database administrator begins. Thereafter, the DBE's role is to provide incremental improvements (if any), such as more efficient access methods or faster operator implementations.

### 2.2. EXODUS System Architecture

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 it clearer to describe the system from the viewpoint of an application-specific database system that was constructed using EXODUS. In doing so, we hope to make it clear which pieces of the system are provided without modification, which can be generated automatically, and which must be directly implemented by the DBE using the E programming language.

Figure 1 presents the general structure of an application-specific database management system implemented using EXODUS. The major facilities provided to aid the DBE in the task of generating such a system are as follows:

(1) The Storage Manager.

(2) The E programming language and its compiler.

(3) A library of type-independent Access and Operator Methods.

(4) A rule-based Query Optimizer Generator.

(5) Tools for constructing query language front-ends.

At the bottom level of the system is the Storage Manager. The basic abstraction at this level is the storage object, which is an untyped, uninterpreted, variable-length byte sequence of arbitrary size. The Storage Manager provides capabilities for reading and updating storage objects 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 support a variety of application-specific versioning schemes is provided. A more detailed description of the Storage Manager is presented in Section 3.

Although not shown in Figure 1, which really depicts the run-time structure of an EXODUS-based DBMS, the next major component is the E programming language and its compiler. E is the implementation language for all components of the system for which the DBE must provide code. E extends C++ by adding generic classes, iterators, and support for persistent object types to the C++ type facilities and control constructs. For the most part, references to persistent objects look just like references to other C++ objects; the DBE's index code can thus deal

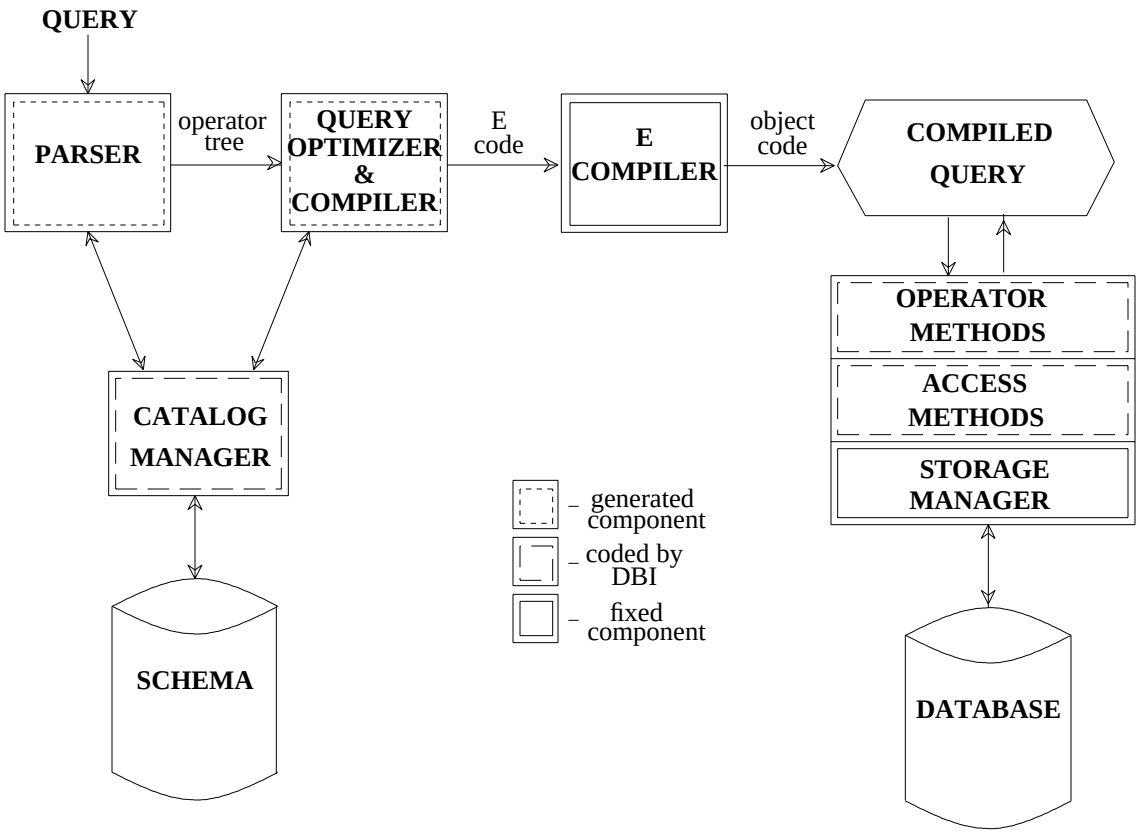


Figure 1: General EXODUS database system structure.

with index nodes as arrays of key-pointer pairs, for example. Where persistent objects are referenced, the E compiler is responsible for inserting calls to fix/unfix buffers, to read/write the appropriate portions of the underlying storage objects, and to handle other such low-level details. Thus, the DBE is freed from having to worry about the internal structure of persistent objects. In order to regain performance, E will also enable the DBE to provide guidance to the compiler in certain ways (e.g., by providing information to aid it in doing buffer management). E should not be confused with database programming languages such as Pascal/R [Schm77] or RIGEL [Rowe79], as these languages were intended to simplify the development of database applications code through a closer integration of database and programming language constructs. Similarly, despite its object-orientedness (stemming from C++), it should not be confused with object-oriented database languages such as OPAL [Cope84, Maie87] or COP [Andr87]. The objective of E is to simplify the development of *internal* systems software for a DBMS.

Layered above the Storage 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 such as B+ trees, Grid files [Niev84], and linear hashing [Litw80]. These access methods will be written using the generic class capability provided by the E language, as described in Section 4. This capability enables existing access methods to be used with DBE-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 DBE may wish to implement new types of access methods in the process of developing an application-specific database system. Since new access methods are written in E, the DBE is shielded from having

to map main memory data structures onto storage objects and from having to deal directly with other low-level details of secondary storage.

While the capabilities provided by the Storage Manager and much of the Access Methods Layer are general-purpose and are intended for use in each application-specific DBMS constructed using EXODUS, the third layer in the design, the Operator Methods Layer, contains a mix of mostly DBE-supplied code and relatively little 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. While EXODUS will provide a library of methods for the operators of a prototype DBMS that we are building with the EXODUS tools (see Section 6), we expect that a number of application-specific or data-model-specific operator methods will be needed. In general, the DBE will have 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. Operator methods are discussed further in Section 4.

The data model of a given application-oriented DBMS is defined by the DBE, with EXODUS providing what amounts to an internal data model via the type system of the E programming language. E's type system includes the basic C++ types (e.g., int, float, char) and type constructors (e.g., class, struct, union, array), plus it provides additional support for generic classes and typed files. These facilities provide sufficient power to implement the higher-level abstractions required by end-user data models, as discussed further in Section 4. The DBE is responsible for this implementation task, and also for implementing (in E) the associated catalog manager for storing user schema information. However, EXODUS does provide a tool, the Dependency Manager, which is designed to help keep track of schema-related dependency information. In particular, this tool is intended to maintain type-related dependencies that arise between types and other types, files and types, stored queries and types, etc., at the data model level. More information about the EXODUS dependency manager can be found in [Care87].

The execution of a query in EXODUS follows a set of transformations similar to that of a relational query in System R [Astr76]. The parser is responsible for transforming the query from its initial form into an initial tree of database operators. After parsing, the query is optimized, converted into an E program, and then compiled into an executable form. The output produced by the query optimizer 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 DBE 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. Section 5 describes the optimizer generator in greater detail.

Finally, the organization of the top level of a database system generated using EXODUS depends on whether the goal is to support some sort of interactive interface, a query facility embedded in a programming language, or an altogether different kind of interface. In the future we would like to provide tools to facilitate the creation of interactive interfaces. We are currently implementing one such interface for the object-oriented data model and query language described in Section 6. Through doing so, we hope to gain insight into the kind of tools that would be helpful at the interface level of the system (in addition to learning how effective the current tool set is).

## 3. THE STORAGE MANAGER

In this section we summarize the key features of the EXODUS Storage Manager. We begin by discussing the interface that the Storage 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. A more detailed discussion of these issues can be found in [Care86b].

### 3.1. The Storage Manager Interface

The Storage Manager provides a procedural interface. This interface includes procedures to create and destroy files and to open and close files for file scans. For scanning purposes, the Storage Manager provides a call to get the object ID of the next object within a file. It also provides procedures for creating and destroying storage objects within a file. For reading storage objects, the Storage 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 range is returned to the caller. Another call is provided to inform the system that these bytes are no longer needed, which "unpins" them in the buffer pool. For writing storage objects, a call is provided to ask the system to modify a subrange of the bytes that were read. 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 Manager provides begin, commit, and abort transaction calls; additional hooks are planned to aid in implementing concurrent and recoverable operations for new access methods efficiently.

In addition to the functionality outlined above, the Storage 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 object ID *X*"). The buffer manager accepts hints about the size and number of buffers to use and what replacement policy to employ; these hints are supported by allowing a *buffer group* to be specified with each object access, and having the buffer manager accept these hints on a per-buffer-group basis. Buffer management policies ranging from simple schemes like global LRU to complex schemes such as DBMIN [Chou85] are thus easily supported.

### 3.2. Storage Objects and Operations

As described earlier, the *storage object* is the basic unit of data in the Storage 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 has the form (*volume #*, *page #*, *slot #*, *unique #*), with the *unique #* being used to make OID's unique over time (and thus usable as surrogates). The OID of a small object points to the object on disk; for a large 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 a large object are private rather than being shared with other objects (although pages are shared between versions of an object). When a small object grows to the point where it can no longer be accommodated on a single page, the Storage Manager automatically converts it into a large object, leaving its object header in place of the original small object. We considered the alternative of using purely 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 surrogate index.

Figure 2 shows an example of our large object data structure. 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).[3] 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.

[3] This data structure was inspired by the ordered relation index of [Ston83], but our update algorithms are quite different [Care86b].

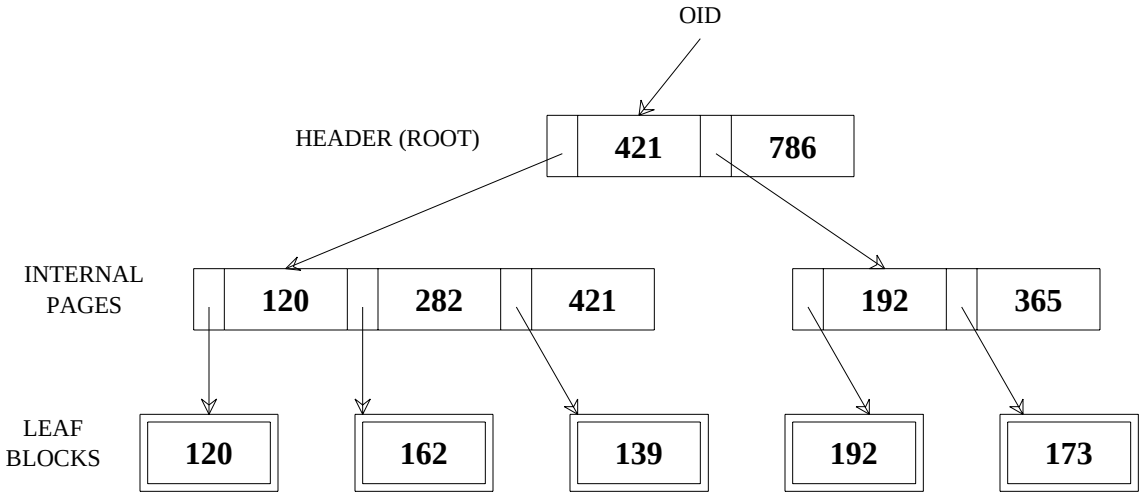


Figure 2: An example of a large storage object.

Searching is accomplished by computing overall offset information while descending the tree to the desired byte position. As described in [Care86b], 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) into a large storage object poses some unique problems compared to inserting or deleting a single record from a B+ tree or an ordered relation. Algorithms for these operations are described in detail in [Care86b] 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 (e.g., 80% for large dynamic objects, and very close to 100% for large static objects).

### 3.3. Versions of Storage Objects

The Storage Manager provides primitive support for versions of storage objects. Versions of objects are identified simply by OIDs. A storage object can have both *working* (current) versions and *frozen* (old) versions; this distinction is recorded in each version's object header. Working versions may be updated by transactions, while frozen versions are immutable. A working version of an object can be made into a frozen version, and new working versions can be derived from frozen versions as desired. It is also possible to delete a version of an object when that particular version is no longer of interest. The reason for providing this rather primitive level of version support is that different EXODUS applications may have widely different notions of how versions should be supported [Ston81, Dada84, 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 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,

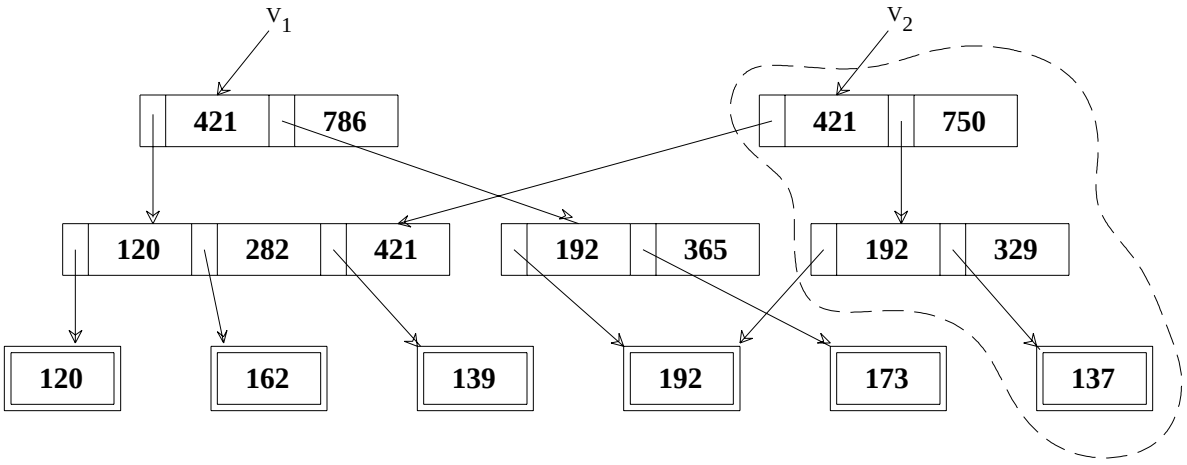


Figure 3: Two versions of a large storage object.

the original (frozen) version, $V_1$, and a newer (working) version, $V_2$. In this example, $V_2$ was created by deriving a working version from $V_1$ and subsequently deleting its last 36 bytes. Note that $V_2$ shares all pages of $V_1$ that are unchanged, and it has its own copies of each modified page; each page pointer has a bit associated with it (not shown in the figure) that distinguishes pointers to shared pages from pointers to unshared pages. Deriving a new version of a large storage object creates a new copy of the root of the object, with subsequent updates leading to the creation of copies of other nodes as needed. Since the number of internal pages in an actual large object is small relative to the number of data pages in the object (due to high fanout for internal nodes), the overhead for versioning large objects in this scheme is small — it is basically proportional to the difference between adjacent versions, and not to the overall size of the objects. In the case of small objects, versioning is accomplished by simply copying the entire object when creating a new version.

A working version such as $V_2$ may be updated via the insert, append, delete, and write operations provided for all storage objects, and the Storage Manager also supports the deletion of unwanted versions of objects, as noted above. When deleting a version of a large object, however, we must be careful — we must avoid discarding any of the object's pages that are shared (and thus needed) by other versions of the same object. An efficient version deletion algorithm that addresses this problem, providing a safe way to delete one version with respect to a set of other versions that are to be retained, is presented in [Care86b].[4]

### 3.4. Concurrency Control and Recovery

The Storage Manager provides concurrency control and recovery services for storage objects. Two-phase locking [Gray79] of storage objects and files is used for concurrency control. 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

[4] The notion of frozen/working versions was not present in [Care86b], but its version deletion algorithm is still applicable. The frozen/working version distinction was added in order to allow versions of a large object to be updated by a series of transactions without forcing each one to derive a new version of the object.

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 ensures that operations on large storage objects can be undone or redone as needed.

### 3.5. Buffer Management for Storage Objects

An objective of the EXODUS Storage Manager design is to minimize the amount of copying from buffer space that is required. A 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, but without requiring that large objects be small enough to fit entirely in the buffer pool. 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 are 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 descriptor is maintained for the current region of *X* in memory, 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 receives a pointer to the descriptor through which the buffer contents may be accessed.[5] Free space for the buffer pool is managed using standard dynamic storage allocation techniques, and buffer block allocation and replacement is guided by the Storage Manager's hint mechanism.

### 3.6. Files of Storage Objects

*Files* are collections of storage objects, and they are useful for grouping objects together for several purposes. First, the EXODUS Storage Manager provides a mechanism for sequencing through all of the objects in a file, so 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 files provide support for objects that need to be tightly clustered on disk. A file is identified by a file identifier (FID) that points to its root page. Like large storage objects, files are represented by an index structure similar to a B+ tree, but the key for the index is different — a file index uses *disk page number* as its key. Each leaf page of the file 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 index thus serves as a mechanism to gather the pages of a file together, and it also enables rapid scanning of all of the objects within a given file. Rapid scanning is a consequence of the fact that the file B+ tree is keyed on page number, meaning that a scan of the objects in a file will access them in *physical order*. Note that since all of the objects in a file are directly accessible via their OIDs, a file is *not* comparable to a surrogate index — secondary indices on the objects in a given file will contain OID entries, which point *directly* to the objects being indexed; this is important from a performance standpoint. Further discussion of file representation, operations, concurrency control, and recovery may be found in [Care86b].

## 4. METHOD IMPLEMENTATION SUPPORT

As described in Section 2, application-specific database systems include access methods and operator methods appropriate for their intended class of applications, and these will undoubtedly vary from one application area to another. For example, while B+ trees and hashing are usually sufficient as access methods for conventional business database systems, a database system for storing and manipulating spatial data is likely to need a spatial access method such as the KDB tree [Robi81], R tree [Gutt84], or Grid file [Niev84]. Unfortunately, such structures, being highly algorithmic in nature, require the DBE to implement them rather than simply specifying them in

[5] As is discussed in Section 4, the E language hides this structure from the DBE.

some high-level form. A complication is that a given index structure often needs to handle data of a variety of types (e.g., integers, reals, character strings, and various ADTs) as long as they satisfy the requirements for correct operation of the index structure. For instance, a B+ tree should work for all key types that provide an appropriate comparison operator [Ston86a]; this includes working for data types that are not defined by the DBE until after the index code has been completely written and debugged. Similar issues arise for operator methods, which must also be written in a general manner in order to handle new, unanticipated data types.

In adding a new access method to a DBMS, sources of complexity include (i) coding and verifying the new algorithms, (ii) mapping the new data structure onto the primitive objects provided by the storage system of the DBMS, (iii) making the access method code interact properly with the buffer manager, and (iv) ensuring that concurrency control and recovery are handled correctly. Although access method designers are mostly interested in item (i), this can comprise as little as 30% of the actual code that must be written; items (ii)-(iv) comprise the remaining 70% or so of the overall code needed to add an access method to a typical commercial DBMS [Ston85]. Items (i)-(iii) are all issues for operator methods as well, and again (i) is the issue that the DBE would presumably like to focus on. To improve this situation, EXODUS provides a programming language, E, for the DBE to use when implementing new methods. E is intended to shield the DBE from items (ii)-(iv), so the E compiler produces code to handle these details based on the DBE's index code (plus some declarative "hints").

In the remainder of this section, we describe the E language and how its various features simplify the DBE's programming tasks. E was designed with the DBMS architecture of Section 2 in mind, so the access methods, operator methods, and utility functions of the DBMS are all intended to be written in E. In addition to these components, the DBMS includes the Storage Manager and the E compiler itself. At runtime, database schema definitions (e.g., "create relation" commands) and queries are first translated into E programs and then compiled. One result of this architecture is a system in which the "impedance mismatch" [Cope84] between type systems is reduced. Another is that the system is easy to extend. For example, the DBE may add a new data type by coding it as an E class. The E programming language is an upward-compatible extension of C++ [Stro86]; E's extensions include both new language features and a number of predefined classes. We present the major features of E briefly here, and refer to the reader to [Rich87, Rich89a, Rich89b] for additional details and examples of how E constructs apply to DBMS implementation problems.

### 4.1. Persistence in E

In order to provide support for persistence, shielding the DBE from having to interact directly with the low-level, typeless view of storage objects provided by the Storage Manager, the E language mirrors the existing C++ type system with constructors having the *db* (database) attribute. Informally, a db type is defined to be:

(1) A fundamental db type, including *dbshort, dbint, dblong, dbfloat, dbdouble, dbchar,* or *dbvoid.*

(2) A *dbclass, dbstruct,* or *dbunion.* Such classes may have data members (fields) only of other db types, but the argument and return types of member functions (methods) are not similarly restricted.

(3) A pointer to a db type object.

(4) An array of db type objects.

An object that is to be persistent is required to be of a db type. However, a db type object can be either persistent or non-persistent. Note that any type definable in C++ may be analogously defined as a db type. Furthermore, since persistence is orthogonal [Atki87] over db types, one could program exclusively in db types and achieve the effect of strict orthogonality if so desired.[6] Db types were introduced in E so that the compiler can always distinguish between objects that can only reside in memory and those that generally reside on disk (but may also reside in memory), as their underlying implementation is very different; this distinction is made so that critical, main-memory-only types can be implemented every bit as efficiently as normal C++ types.

[6] Note that one could even use macros to redefine the keywords *class* to mean *dbclass*, *struct* to mean *dbstruct*, *union* to mean *dbunion*, *int* to mean *dbint*, etc.

Given db types for describing the type structure of objects that may be persistent, E supports the declaration of actual persistent objects via a new storage class called *persistent*; this is in addition to the usual C++ storage classes (extern, automatic, static, and register). For example, to declare a persistent object named `emp` of a dbclass named `Employee` in an E program, one would simply write:

```
persistent Employee emp;
```

This declaration causes the name `emp` to become a handle for a persistent `Employee` object; the object itself will reside in the Storage Manager. Uses of the object then look exactly like they would if `emp` were an instance of a normal C++ class. Similarly, member functions of the `Employee` dbclass are coded just as if it were an equivalent non-db class. The E compiler is implemented as an E-to-C translator, and it translates references to portions of persistent objects into C code that calls the Storage Manager as needed to manipulate the contents of persistent objects. Persistent E objects are mapped one-to-one onto storage objects, and pointers to db type objects are actually OIDs; thus, in E the DBE retains control over the format of persistent objects without having to explicitly make calls to the Storage Manager.

### 4.2. Generic Classes for Unknown Types

As described earlier, many of the types involved in database programming are not known until well after the code needing those types is written by the DBE. Access method code does not know what types of keys it will contain, nor what types of entities it will index, and the code implementing a join algorithm does not know what types of entities it will be called upon to join.

To address this problem, E augments C++ with *generic* (or generator) classes, which are very similar to the parameterized clusters of CLU [Lisk77]. Such a class is parameterized in terms of one or more unknown types; within the class definition, these (formal) type names can be used freely as regular type names.[7] This mechanism allows one to define, for example, a class of the form `Stack[T]` where the specific class `T` of the stack elements is not known. The user of such a class can then *instantiate* the generic class by providing actual type parameters to the class. For example, one may define a stack class `IntStack` for handling integer data and then declare an integer stack `x` by saying:[8]

```
class IntStack: Stack[int];
IntStack x;
```

Similarly, the DBE can implement a B+ tree as a generic dbclass `BTree[KeyType, EntityType]` where both the key type and the type of entity being indexed are dbclass parameters. Later, when a user wishes to build an index over employees on social security number, the system will generate and compile a small E program that instantiates and uses the dbclass:

```
Emp_BTree: BTree[SSNo_type, Emp_type];
persistent Emp_BTree EmpSSNoIndex;
```

Such instantiations are dealt with efficiently via a linking process along the lines of that used in the implementation of CLU [Atki78].

Figures 4a and 4b give a partial example of a generic dbclass, `BTreeNode`, to illustrate the flavor of how such classes are described in E. This dbclass represents the node structure of a B+ tree index, and once defined it could be used in writing the generic B+ tree dbclass discussed above (as we will show shortly). The interface portion of the dbclass is shown in Figure 4a. The dbclass has two dbclass parameters, `KeyType` and `EntityType`, and the dbclass `KeyType` is required to have a `compare` member function. `KeyType` and `EntityType` are

---

[7] In addition to type parameters, generic classes and dbclasses can also have constant and function parameters.

[8] We chose this syntax over `Stack[int] x` to maintain compatibility with the existing C++ class derivation syntax.

```
enum status { FOUND, NOT_FOUND };


dbclass BTreeNode
[
    dbclass KeyType{                        // type of keys in tree
        int  compare(KeyType);              // (compare function needed)
    },
    dbclass EntityType{ }                   // type of indexed entities
]

{

public:

    // type for key-pointer pairs
    dbstruct KPpair {
      KeyType  keyVal;
      dbunion {
          EntityType*  entityPtr;           // used in leaf nodes
          BTreeNode*   childPtr;            // used in interior nodes
      } ptr;
    };

    // internal structure of a B+ tree node, consisting of the
    // node height (which is 0 for leaf nodes), the number of
    // keys currently in the node, and an array of key/pointer
    // pairs of the appropriate size
    dbint   height;
    dbint   nKeys;
    KPpair  kppArray[(PAGESIZE - 2*sizeof(int)) / sizeof(KPpair)];

    // binary search function for a single node
    status searchNode(KeyType key, int& index);

    // etc.  ...  (other node member functions)  ...

}; // dbclass BTreeNode
```

Figure 4a: The generic dbclass BTreeNode.

the key and entity types for the node, and the `compare` member function compares two keys[9] and returns -1, 0, or 1 depending on their relative order. The structure of a B+ tree node is described as having a height, a count of the keys in the node, and an array of key-pointer pairs. The size of the array is determined by the constant `PAGESIZE`, which is the maximum object size (in bytes) that will fit on one Storage Manager disk page, together with the size of

[9] It compares the key to which the function is applied with the key passed as an explicit function argument, as illustrated in Figure 4b.

```
// binary search of B+ tree node

status BTreeNode::searchNode(KeyType key, int& index) {

    int min = 0;
    int max = nKeys - 1;
    int mid;
    int cmpVal;

    while (min <= max) {

      mid = (min + max) / 2;
      cmpVal = kppArray[mid].keyVal.compare(key);

      if (cmpVal < 0)
          { min = mid + 1; }
      else if (cmpVal == 0)
          { index = mid; return FOUND; }
      else
          { max = mid - 1; }

    } // while

    return NOT_FOUND;

} // BTreeNode::searchNode
```

Figure 4b: Code for BTreeNode's searchNode member function.

a key-pointer pair.[10] The `searchNode` member function in Figure 4b shows that the code for potentially persistent (i.e., dbclass) objects looks just like normal C++ code.

### 4.3. Fileof[T] for Persistent Collections

In addition to providing support for persistent objects, and providing generic dbclasses for programming in the face of missing type information, E also provides support for handling large, scannable *collections* of persistent objects of similar type. E's provision here is a built-in generic dbclass, `fileof[T]`, where `T` must be a dbclass. As an example, if `Department` is a dbclass, a useful new dbclass can be created via the declaration:

```
dbclass DeptFile: fileof[Department];
```

Objects of type `DeptFile` can now be used to hold entire sets of `Department` objects (including objects of any subclass of `Department`). The operational interface of the generic `fileof` class allows the user to bind typed pointers to objects in a file, to create and destroy objects in a file, etc. A file can also be viewed as a "persistent heap" in a sense, as the `new` statement for dynamically allocating persistent objects requires the specification of a

[10] This is not particularly elegant, but it is necessary for performance.

file in which to create the object.

As an example, the following function returns the sum of the budgets of the departments in a file of `Department` objects. The file is passed by reference, and we assume that `Department` objects have a public data member called `budget`:

```
float TotalBudget(DeptFile& depts)
{
    Department* d;
    float sum = 0.0;
    for(d = depts.get_first(); d != NULL; d = depts.get_next(d)) {
        sum += d->budget;
    }
    return sum;
}
```

While this example is extremely simple, it illustrates how easy it can be to scan the contents of a file of objects. No typecasting is needed to use the pointer `d`, and no buffer calls are necessary. Each instance of the `fileof` dbclass is implemented as a Storage Manager file and represented by its associated FID; the DBE is shielded from file-related Storage Manager calls by the `fileof` dbclass interface. Finally, for cases where the restriction of storing only instances of a type `T` and its subtypes in a file is too limiting, an untyped, byte-oriented `file` dbclass is also provided.

### 4.4. Iterators for Scans and Query Processing

A typical approach for structuring a database system is to include a layer which provides *scans* over objects in the database. A scan is a control abstraction which provides a state-saving interface to the "memoryless" storage systems calls. Such an interface is needed for the record-at-a-time processing done in higher layers. A typical implementation of scans will allocate a data structure, called a scan descriptor, to maintain all needed state between calls to the storage system; it is then up to the user to pass the descriptor with every call.

The control abstraction of a scan is provided in EXODUS via the notion of an *iterator* [Lisk77]. An iterator is a coroutine-like function that saves its data and control states between calls; each time the iterator produces (*yields*) a new value, it is suspended until it is resumed by the client. Thus, no matter how complicated the iterator may be, the client only sees a steady stream of values being produced. The client can invoke an iterator using a new kind of control statement, the *iterate* loop of E (which generalizes the *for ... in* loop of CLU).

The general idea for implementing scans should now be clear. For example, to implement a scan over B+ trees, one can write an iterator function for the `BTree` class that takes a lower bound and an upper bound as arguments. The scan will begin by searching down to the leaf level of the tree for the lower bound, keeping a stack of node pointers along the way. It will then walk the tree, yielding object references one at a time, until reaching the upper bound; alternatively, if leaves were linked together, it could walk through the sequence set. The iterator will then terminate. Figure 5 shows what the interface definition for such a generic `BTree` class might look like, including a constructor function to initialize a newly created B+ tree index, a destructor function that is invoked to clean up when a B+ tree is destroyed, member functions to insert and delete index entries, and the `scan` iterator that we just described.

Iterators can also be used to piece executable queries together from an access plan tree. If one views a query as a pipeline of processing filters, then each processing stage can be implemented as an iterator which is a client of one or more iterators (upstream in the pipe) and yields result tuples to the next stage (downstream in the pipe). Execution of the query pipeline will be demand-driven in nature. For example, the DBE for a relational DBMS would write various operator methods for the select, project, and join operations as iterators in this fashion. Given the access plan tree that results from optimizing a user query, it is not difficult to produce E code that implements the pipeline by plugging together instances of these iterators. This approach to forming queries is further described in [Rich87], and it was also the basis for a relational DBMS prototype that we developed using the EXODUS tools for a demonstration at SIGMOD-88. The idea is illustrated by the following excerpt from our relational DBMS

prototype. This code is an iterator member function from a (generic) class that provides a method for the equi-join operator:

```
iterator DstType* index_join::next_tuple()
{
    DstType   rslt;
    AttrType  joinVal;

    iterate(SrcType1* outer = outerQuery->next_tuple()) {
        extract(outer, &joinVal);
        iterate(SrcType2* inner = innerIndex->scan(joinVal, joinVal)) {
            concatenate(outer, inner, &rslt);
            yield(&rslt);
        }
    }
}
```

This code implements the `next_tuple` iterator for computing a join via an index-based algorithm. `SrcType1`, `SrcType2`, and `DstType` are the outer, inner, and result tuple types (respectively), and `AttrType` is the type of the join attribute. The join method iterates over a stream of outer relation tuples using the iterator `next_tuple` provided by the subquery (`outerQuery`) feeding the join, and for each tuple it extracts its join attribute value using the function `extract`. It then scans the inner relation via a B+ tree index on the join attribute

---

```
dbclass BTree [
    dbclass KeyType{                        // type of keys in tree
        int  compare(KeyType);              // (compare function needed)
    },
    dbclass EntityType{ }                   // type of indexed entities
]
{
    // instantiate types used for B+ tree index
    dbclass Node: BTreeNode[KeyType, EntityType];
    dbclass NodeFile: fileof[Node];

    // represent B+ tree as file of nodes plus root pointer
    NodeFile      tree;
    Node*         root;

public:
    BTree();      // constructor function
    ~BTree();     // destructor function
    EntityType* insert(KeyType, EntityType*);
    EntityType* delete(KeyType, EntityType*);
    iterator EntityType* scan(KeyType, KeyType);
}; // dbclass BTree
```

Figure 5: Interface for the generic dbclass BTree.

---

(`innerIndex`), using the B+ tree's `scan` iterator to find matching inner tuples. Each time it finds a matching tuple it calls the `concatenate` function to concatenate the outer and inner tuples, forming a result tuple, and yields the result tuple to the next operator method in the query stream. (The `extract` and `concatenate` functions and the `outerQuery` and `innerIndex` pointers are initialized based on arguments passed to the `index_join` class constructor, as shown in a more complete example in [Rich87].)

### 4.5. Method Performance Issues

Since E is the language used by the DBE to implement key portions of the code for a DBMS, performance is clearly an important issue. One performance issue related to E is how frequently the code produced by the E compiler issues calls to the Storage Manager. We are currently working on an optimization pass for the E compiler that will perform transformations to reduce this frequency. For example, if a number of fields of an object are referenced, the compiler should generate a single call to retrieve the relevant portion of the object all at once (as opposed to a series of individual calls). Or, if an object is a large array (e.g., an image), it may be useful/necessary to transform a loop that processes all of the elements in the array into a nested loop that processes a block of array elements at a time. We also plan to add "hint" facilities to E in order to allow the DBE to guide the E compiler in making performance-related decisions (e.g., by specifying buffer group sizes and replacement policies for critical operations).

A second performance issue, relevant especially to access method code, is that of specialized locking and recovery schemes (e.g., B+ tree locking protocols [Baye77]). While the two-phase locking and log-based recovery mechanisms employed by the Storage Manager will ensure the correct and recoverable operation of E programs, these mechanisms are likely to prove too restrictive for a truly high-performance DBMS. Our long-term goal is to add transaction control facilities to E in order to permit clever DBEs to implement index-specific concurrency control and recovery algorithms when they are needed.

### 4.6. Modeling End-User Schemas

As discussed briefly in Section 2, the type system of the E language can in some sense be viewed as the internal type system of EXODUS: To support a target end-user data model, the DBE must thus develop mappings from the data model's type system to E's type system; the collection of available primitive internal EXODUS types thus includes integers, floating point numbers, characters, and enumerations, and the available type constructors include pointers, arrays, structures, and unions. New abstract data types (e.g., rectangle, complex number, or image) and their associated operations can be defined as E dbclasses. In addition, new data model type constructors (e.g., list or set) can be modeled by implementing them as parameterized E dbclasses. Since E provides pointers together with a rich collection of type constructors, even complex, recursive, end-user object types can be modeled in E without too much difficulty. We thus expect that the internal type system of EXODUS will be powerful enough to satisfactorily model most any application area's type system.

## 5. 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 a query 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 [Nguy82] 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 [Cloc81] and OPS5 [Forg81] 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 [Grae87a, Grae87b] which produces an optimization procedure in the C programming language [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. It is not part of the optimizer's task to produce an initial algebraic query tree from a non-procedural expression; this is done by the user interface and parser. An access plan is a tree with operator methods as internal nodes (e.g., a nested loops join method) and with files or indices as leaves. Once an access plan is obtained, it is then transformed into an iterator-based E program by a procedure that walks the access plan tree (in a manner loosely related to that of [Frey86]).

### 5.1. Basic Optimizer Generator Inputs

There are four key elements which the optimizer generator requires in a description file in order 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 are unaffected 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 prototype relational query optimizer:

```
%operator 2 join
%method 2 nested-loops-join merge-join
join (1, 2) <-> join (2, 1);
join (1, 2) by nested-loops-join (1, 2);
```

In this example, `join` is declared to be a binary operator, and `nested-loops-join` and `merge-join` are declared to be two binary methods. The symbol `<->` denotes equivalence (i.e., a potential two-way transformation) in the context of a transformation rule, and *by* is a keyword used for implementation rules. The transformation rule in the above example states that `join` is commutative, and the implementation rule says that `nested loops-join` is a join method. If `merge-join` is a method that is only useful for joining sorted relations, then its implementation rule would have to include a condition to test whether or not each input relation is sorted appropriately.

### 5.2. Optimizer Support Functions

In addition to a declarative description of the data model, the optimizer generator requires the DBE to provide a collection of support procedures in a code section of the optimizer description. These procedures are C routines that access and/or manipulate the optimizer's data structures. The generated optimization procedure employs two principal data structures, *MESH* and *OPEN*. *MESH* is a directed acyclic graph that holds all of the alternative operator trees and access plans that have been explored so far, employing a rather complex pointer structure to ensure that transformations can be identified and performed quickly (and also that equal subexpressions will be processed only once). *OPEN* is a priority queue containing currently applicable transformations, as described further in Section 5.3.

The overall cost of an access plan is defined as the sum of the costs of the methods involved, and the objective of optimization is to minimize this sum. Thus, for each method, the DBE must provide a cost function for calculating the cost of the method based on the characteristics of the method's input. The method's cost function will be called by the generated optimizer whenever it considers using the method for implementing an operator (or a pattern of operators). The input arguments for method cost functions are pointers to the root node of the relevant portion of the query tree in *MESH* and to the nodes in *MESH* that produce the input streams according to the associated implementation rule.

In addition to method cost functions, a property function is needed for each operator and each method. The DBE is permitted to define properties (as C structures) that are to be associated with each operator and method node in *MESH*. Operator properties are logical properties of intermediate results, such as their cardinalities and schemas. Method properties are physical properties (i.e., method side effects), such as sort order in our merge-join example. Operator property functions are called by the generated optimizer when transformations are applied, and method propery functions are invoked when methods are selected.

Each node in *MESH* contains arguments associated with the operator represented by the node and with the best method that has been found for the subquery rooted at the node. In a relational optimizer, for example, the select and join operators have predicates as arguments, and the project operator has a field list as an argument. A method that implements a combined select-project operation would have both a predicate and a field list as arguments. As with properties, it is necessary for the DBE to define this data-model-dependent aspect of a node (typically as a C union). By default, operator and method arguments are copied automatically between corresponding *MESH* nodes, which is fine for many simple transformation and implementation rules (e.g., join commutativity, which simply reorders operators in the query tree). However, for rules where such copying is insufficient, the DBE must provide argument transfer functions to manipulate the arguments. For example, consider the following transformation rule:

```
select 9 (product (1, 2)) -> join (select(1), select(2)) xpj_arg_xfer
{{
    if (no_join_predicate(OPERATOR_9.oper_argument))
      REJECT;
}};
```

This (one-way) transformation rule replaces a selection over a cross-product with a join of two selections. The function `xpj_arg_xfer`, which the DBE must provide, will be called when the rule is applied by the generated optimizer in order to rearrange the operators' predicate arguments; it will need to split the predicate associated with the select operator on the left-hand side of the rule into a join predicate and two select predicates for the operators on the rule's right-hand side. This example also illustrates several other features that were mentioned earlier. First, it shows how a condition can be associated with a rule, as the transformation will be rejected if the `no_join_predicate` function provided by the DBE determines that no join predicate exists. Second, it shows how access is provided to operator arguments. The select operator on the left-hand side of the example rule is numbered to identify it uniquely within the rule so that its `oper_argument` field can be passed to the function employed in the condition. Method arguments and properties of operators and methods can be accessed in a similar manner.

Finally, in some cases the DBE may wish to assist the optimizer in estimating the benefit of a given transformation before the transformation is actually performed. A function can be named in the transformation rule using the keyword *estimate*, in which case this function will be called by the optimizer to estimate the expected cost of a transformed query based on the relevant portion of the query tree, the operator arguments, and the operator and method properties.

At first glance, it may appear that there is quite a bit of code for the DBE to write. However, not all of the functions outlined above are required. In particular, only the cost functions are absolutely necessary. If the DBE does not specify a type for operator property fields, then operator property functions are not necessary; similarly, method property functions are only needed if a method property type is specified. Argument transfer functions and estimation functions are optional, and need not be specified except for rules where their functionality is required. Finally, remember that a key design goal for the EXODUS optimizer generator was data model independence, so these functions really cannot be built into the optimizer generator. However, we do intend to provide libraries of generally useful functions (such as predicate manipulation routines) that the DBE can use in cases where they are appropriate.

### 5.3. Operation of the Generated Optimizer

The generated optimization procedure starts by initializing the MESH and OPEN data structures. MESH is set up to contain a tree with the same structure as the original query. The method with the lowest cost estimate is then selected for each node in MESH using the implementation rules. Finally, the transformation rules are used to determine possible transformations which are inserted into OPEN. Once MESH and OPEN have been initialized in this manner, the optimizer 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 the methods with the 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. To insure that a matching transformation rule with a low expected cost factor will be applied first, entries in OPEN are prioritized by their expected cost decrease. 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 smaller inputs. Second, it is sometimes necessary to apply equivalence transformations even if they do not directly yield cheaper solutions, as they may be needed as intermediate steps to even less expensive access plans. Such transformations represent hill climbing, and we limit their application through the use of a *hill climbing factor*. Lastly, when a transformation results in a lower cost, the parent nodes of the old expression must be reanalyzed to propagate any cost advantages; a *reanalyzing factor*, similar to the hill climbing factor, limits this propagation in cases where the new plan's cost is worse than the best equivalent subquery by more than this factor.

It is a non-trivial problem to select values for the optimization parameters so as to guarantee optimal access plans together with good optimizer performance. Thus, it is would be nice if they could be determined and adjusted automatically. We have not yet automated the selection of the hill climbing or reanalyzing parameter values, but we have successfully automated the choice of expected cost factors. Our current prototype initializes the expected cost factors of all transformation rules to 1, the neutral value, and then adjusts them using sliding geometric averages. This has turned out to be quite effective in experiments with several prototype relational optimizers [Grae87a, Grae87b]; our experience has been that generated optimizers are fast enough for production use, and that the access plans that they produce are consistently very close to optimal. We have also found that the EXODUS optimizer generator provides a useful, modular framework for breaking the data-model-dependent code of a query optimizer into small but meaningful pieces, which aids in the rapid prototyping and development of new query optimizers.

## 6. A DBMS SUPPORTING COMPLEX OBJECTS AND OBJECT ORIENTATION

As a number of the components of EXODUS are now ready for an initial trial, we recently turned our attention to the process of selecting a target data model to implement using the EXODUS toolkit. The goals of this implementation effort are to validate the EXODUS approach to DBMS development, to serve as a forcing function for developing a library of access methods and operator methods, and to provide a system that can serve as a demonstration of the use of EXODUS for potential users. Since no single data model and query language was quite what we were looking for (in terms of our goals), we decided to design our own data model and query language. The EXTRA data model and EXCESS query language are the result of this design effort. The EXTRA data model includes support for complex objects with shared subobjects, a novel mix of object- and value-oriented semantics for data, support for persistent objects of any type in the EXTRA type lattice, and user-defined abstract data type extensions (ADTs). The EXCESS query language provides facilities for querying and updating complex object structures, and it can be extended through the addition of ADT functions and operators (written in E) and procedures and functions for manipulating EXTRA schema types (written in EXCESS). This section of the paper presents an overview of the key features of EXTRA and EXCESS; more details can be found in [Care88].

### 6.1. The EXTRA Data Model

An EXTRA database is a collection of named persistent objects of any type that can be defined using the EXTRA type system. EXTRA separates the notions of type and instance. Thus, users can collect related objects together in semantically meaningful sets and arrays, which can then be queried, rather than having to settle for queries over type extents as in many other data models (e.g., [Mylo80, Ship81, Bane87, Lecl87, Rowe87]). EXTRA provides a collection of type constructors that includes tuple, set, fixed-length array, and variable-length array. In addition, there are four flavors of instance values, **own**, **ref**, **own ref**, and **own unique ref** (although casual users such as query writers need not be concerned with this distinction). Combined with the EXTRA type constructors, these provide a powerful set of facilities for modeling complex object types and their semantics. Finally, EXTRA provides support for user-defined ADTs, derived attributes, and automatic maintenance of inverse relationships

among attributes.

Figure 6 shows a simple database defined using the EXTRA data model. In EXTRA, the tuple, set, and array constructors for complex objects are denoted by parentheses, curly braces, and square brackets, respectively. The figure should be fairly self-explanatory, with the exception of the keywords **own**, **unique**, and **ref**. In EXTRA, subordinate entities are treated as values (as in nested relational models [Sche86]), not as objects with their own separate identity, unless prefaced by **ref**, **own ref**, or **own unique ref** in a type definition or an object creation statement. The declaration **ref x** indicates that **x** is a reference to an extant object. **Own ref x** indicates that **x** has object identity but its existence is dependent on the existence of at least one object that refers to it via an **own ref** reference. **Own unique ref x** indicates that **x** has object identity but its existence is dependent on a unique owning object.

Briefly, Figure 6 defines four types, all of which happen to be tuple types in this example: Person, Student (a subtype of Person), Employee (another subtype of Person), and Department. It then defines a university database consisting of two named, persistent objects: Students, which is a set of Student objects, and Departments, which is a set of Department objects. Since both are of the form "{ **own unique ref** ... }", these two sets own their member objects and these member objects will be deleted when their owning sets are subsequently **destroy**ed. Each Department object in the set Departments contains a set of Employee objects that work in that Department, and it owns these objects (i.e., a given Employee object can be thought of as being part of a "composite" [Bane87] Department object). Both Employee and Student objects contain (references to) a Department object that they work for or major in, respectively, and the management structure for the employees is captured by the manager and sub_ords attributes of Employee objects.

Two concepts are central in the design of EXTRA/EXCESS: extensibility and support for complex objects. In addition, the model incorporates the basic themes common to most semantic data models [Hull87, Peck88]. Extensibility in EXTRA/EXCESS is provided through both an abstract data type mechanism, where new types can be written in the E programming language and then registered with the system, and through support for user-defined functions and procedures that are written in the EXCESS query language and operate on user-defined EXTRA types. Complex objects are objects in the database, possibly composed of other objects, that have their own unique identity. Such objects can be referenced by their identity from anywhere in the database. In [Bato84], four useful varieties of complex objects are identified: disjoint-recursive, disjoint-nonrecursive, nondisjoint-recursive, and nondisjoint-nonrecursive.[11] The EXTRA data model is capable of modeling all four varieties.

EXTRA also provides many of the capabilities found in semantic data models. Four primitive modeling concepts fundamental to most semantic data models [Hull87, Peck88] are: the *is-a* relationship (also known as generalization), the *part-of* relationship (often called aggregation), the *instance-of* relationship (also referred to as classification), and the *member-of* relationship (called association in some models). Each of these concepts is easily modeled using the facilities of EXTRA. Generalization is modeled in EXTRA by using the **inherits** keyword to indicate that a type inherits attributes and functions from another type. For example, an Employee is a Person in Figure 6. (Note that our notion of generalization is often called "specialization" in the semantic modeling literature.) Aggregation is easily modeled using the tuple constructor — for instance, a Department is an aggregation of its employees, its manager, etc. (We ignore the distinction between attributes, which merely describe an object, and components of an object). Classification is simply the notion of type-instance dichotomy, and is present in EXTRA in the distinction between the **define type** and **create** statements in Figure 6. Finally, association is modeled by the set constructor of EXTRA. An example of this is the set of employees which are subordinate to a manager.

### 6.2. The EXCESS Query Language

EXCESS queries range over objects created using the **create** statement. EXCESS is based on QUEL [Ston76], GEM [Zani83], POSTQUEL [Rowe87], and SQL extensions for nested relations [Dada86, Sche86]. EXCESS is designed to provide a uniform query interface to sets, arrays, tuples, and single objects, all of which can

[11] Two objects are disjoint if they share no subobjects; an object is recursive if it contains other objects of the same object type.

```
define type Person:
(
        ssnum:          int4,
        name:           char[ ],
        street:         char[20],
        city:           char[10],
        zip:            int4,
        birthday:       Date
)

define type Student:
(
        gpa:            float4,
        dept:           ref Department
)
inherits Person

define type Employee:
(
        jobtitle:       char[20],
        dept:           ref Department,
        manager:        ref Employee,
        sub_ords:       { ref Employee },
        salary:         int4,
        kids:           { own Person }
)
inherits Person

define type Department:
(
        name:           char[ ],
        floor:          int4,
        employees:      { own unique ref Employee }
)

create Students:        { own unique ref Student }
create Departments:     { own unique ref Department }
```

Figure 6: A simple EXTRA database.

be composed and nested arbitrarily. In addition, user-defined functions (written both in E and in EXCESS) and aggregate functions (written in E) are supported in a clean and consistent way. A few examples should suffice to convey the basic flavor of the language.

As a first example, the following query finds the names of the children of all employees who work for a department on the second floor:

```
range of E is Employees
retrieve (C.name) from C in E.kids where E.dept.floor = 2
```

Our second example illustrates the use of an aggregate function over a nested set of objects. The following query retrieves the name of each employee, and for each employee it retrieves the age of the youngest child among the children of all employees working in a department on the same floor as the employee's department.

```
range of EMP is Employees
retrieve (EMP.name, min(E.kids.age
    from E in Employees
    where E.dept.floor = EMP.dept.floor))
```

In this example, the variable E ranges over Employees within the scope of the min aggregate, and within the aggregate it is connected to the variable EMP through a join on Employee.dept.floor. The query aggregates over Employee.kids, which is a set-valued attribute. Here, age is assumed to be defined by a function that computes the age of a Person from the current date and their birthday, so it is a virtual field of Person objects.

User-defined functions and procedures written in EXCESS are supported, and are handled uniformly in the syntax (as illustrated by the use of the age function in the example above). Functions can be invoked using either the syntax for denoting attributes (for functions defined on particular types) or the user-defined function syntax (which is similar to the aggregate function invocation syntax shown in the previous example). EXCESS procedures are invoked in a manner consistent with the EXCESS syntax for performing updates. Procedures differ from functions in that functions return a value and have no side-effects, while procedures usually have side-effects and return no value. Further details and examples are presented in [Care88].

### 6.3. The EXTRA/EXCESS System Architecture

The EXTRA/EXCESS environment will consist of a frontend process and a backend process, as illustrated in Figure 7 (which is essentially an EXTRA/EXCESS-specific version of Figure 1). The frontend process parses a query, converts it to an optimizable form, optimizes it, converts the optimized query to E, and sends this E program to the backend for execution.[12] The optimizer will be generated using the EXODUS Optimizer Generator. The frontend also interfaces with the EXTRA/EXCESS data dictionary for processing data definition language requests, performing authorization, etc. The data dictionary is itself designed as an EXTRA database, and thus will be stored by the EXODUS Storage Manager like all other data. It is drawn separately in Figure 7 simply to clarify its function.

The backend process consists of several components. The E compiler compiles E code into executables which contain calls to the EXODUS Storage Manager. There is also a loader to dynamically load compiled queries into the E run-time system (ERTS). ERTS contains operator methods and access methods written in E by the DBE as well as methods taken from the generic method libraries provided by EXODUS. The EXODUS Storage Manager is also part of the backend, as it serves as the repository for persistent E objects; it is the only component of the system which directly manipulates persistent data. Finally, the backend will send query results to the frontend for formatting and output.

## 7. SUMMARY AND CURRENT STATUS

In this paper we have described EXODUS, an extensible database system project aimed at simplifying 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 engineers, or DBEs, 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 DBEs. 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 DBE in his or her work.

---

[12] We also plan to support precompiled queries, but their execution path is not shown in Figure 7.

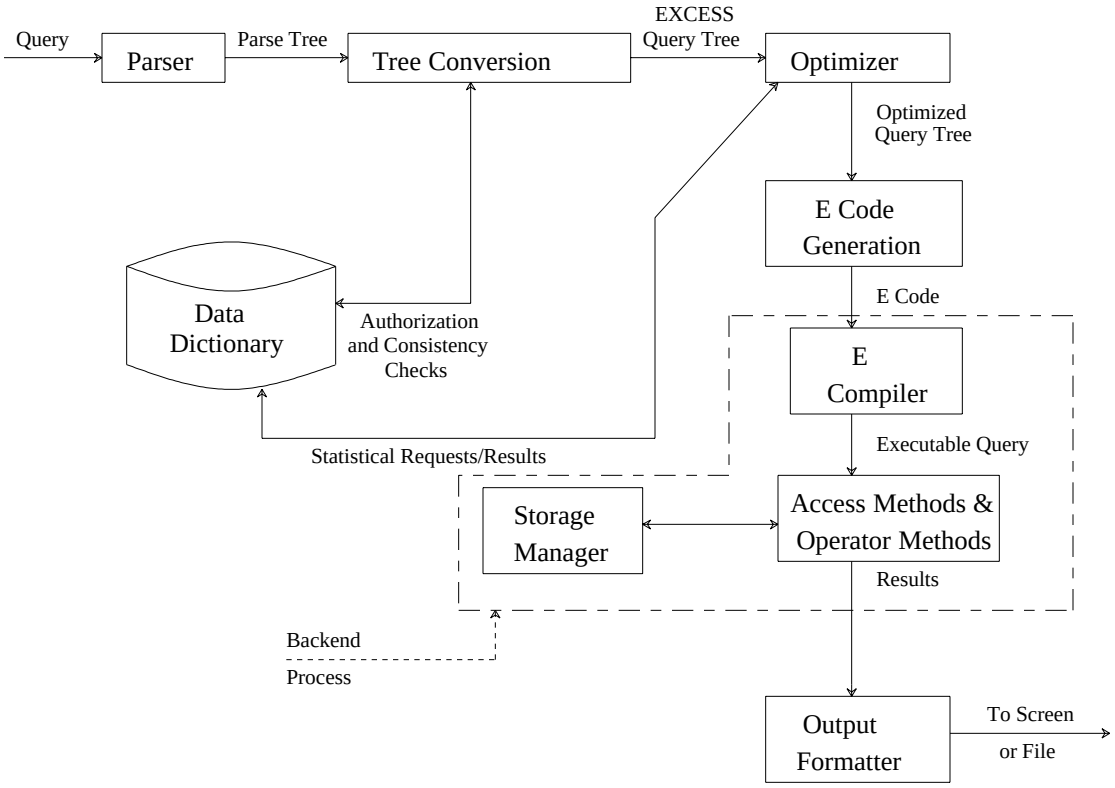


Figure 7: The EXTRA/EXCESS system architecture.

As we described, EXODUS includes one component that requires no change from application area to application area — the Storage Manager, a flexible, low-level storage manager that provides concurrent and recoverable access to storage objects of arbitrary size. In addition, EXODUS provides libraries of database system components that are likely to be widely applicable, including various useful access methods and operator methods. The corresponding system layers are constructed by the DBE 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 DBE from many of the low-level details of persistent programming. E is also the vehicle provided for defining new ADTs, which makes it easy for the DBE to write operations on ADTs even when they are very large (e.g., an image or voice ADT). At the upper level of the system, EXODUS provides a generator that produces a query optimizer from a description of the available operations and methods. Finally, we described EXTRA/EXCESS, a next-generation data model and query language with an object-oriented flavor that will drive further EXODUS developments.

An initial implementation of the EXODUS tools is basically complete, including all of the components described here. A single-user version of the Storage Manager is running, providing excellent performance for both small and large storage objects; work on versions, concurrency control, and recovery is underway. The implementation of the rule-based query optimizer generator was completed over a year ago, and it has been used to generate most of a full relational query optimizer. The E compiler is coming along nicely, with virtually all of the language features (except hints) working at this time; the E effort is currently focused on optimization issues (e.g., coalescing storage manager calls and improving our implementation of generic classes). At SIGMOD-88 we demonstrated a

relational DBMS prototype that was implemented using the EXODUS tools, and we are now working on an initial EXTRA/EXCESS implementation.

**REFERENCES**


[Andr87] Andrews, T., and Harris, C., "Combining Language and Database Advances in an Object-Oriented Development Environment," *Proc. of 1987 OOPSLA Conf.*, Orlando, FL, Oct. 1987.

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

[Atki87] Atkinson, M., and Buneman, O.P., "Types and Persistence in Database Programming Languages," *ACM Comp. Surveys*, 19, 2, June 1987.

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

[Bane87] Banerjee, J., et. al., "Data Model Issues for Object-Oriented Applications," *ACM Trans. on Office Info. Sys.* 5, 1, Jan. 1987.

[Bato84] Batory, D., and Buchmann, A., "Molecular Objects, Abstract Data Types, and Data Models: A Framework," *Proc. of the 1984 VLDB Conf.*, Singapore, Aug. 1984.

[Bato88a] Batory, D., "Concepts for a Database System Compiler," *Proc. of the 1988 ACM Principles of Database Sys. Conf.*, Austin, TX, March 1988.

[Bato88b] Batory, D., et al, "GENESIS: An Extensible Database Management System," *IEEE Trans. on Software Eng.* 14, 11, Nov. 1988.

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

[Care86a] Carey, M., and D. DeWitt, "Extensible Database Systems," in *On Knowledge Base Management: Integrating Artificial Intelligence and Database Technologies*, M. Brodie and J. Mylopoulos, eds., Springer-Verlag, 1986.

[Care86b] Carey, M., et al, "Object and File Management in the EXODUS Extensible Database System," *Proc. of the 1986 VLDB Conf.*, Kyoto, Japan, Aug. 1986.

[Care86c] Carey, M., et al, "The Architecture of the EXODUS Extensible DBMS" *Proc. of the Int'l. Workshop on Object-Oriented Database Sys.*, Pacific Grove, CA, Sept. 1986.

[Care87] Carey, M., and DeWitt, D., "An Overview of EXODUS," in [DBE87].

[Care88] Carey, M., DeWitt, D., and Vandenberg, S., "A Data Model and Query Language for EXODUS," *Proc. of the 1988 SIGMOD Conf.*, Chicago, IL, June 1988.

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

[Clif85] Clifford, J., and A. Tansel, "On An Algebra for Historical Relational Databases: Two Views," *Proc. of the 1985 SIGMOD Conf.*, 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," *Proc. of the 1984 SIGMOD Conf.*, Boston, MA, May 1984.

[DBE87] *Database Engineering* 10, 2, Special Issue on Extensible Database Systems, M. Carey, ed., June 1987.

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

[Dada86] Dadam, P., et al, "A DBMS Prototype to Support Extended $NF^2$ Relations: An Integrated View of Flat Tables," *Proc. of the 1986 SIGMOD Conf.*, Washington, DC, May 1986.

[Daya86] Dayal, U. and J. Smith, "PROBE: A Knowledge-Oriented Database Management System," in *On Knowledge Base Management: Integrating Artificial Intelligence and Database Technologies*, M. Brodie and J. Mylopoulos, eds., Springer-Verlag, 1986.

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

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

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

[Grae87a] Graefe, G., and DeWitt, D., "The EXODUS Optimizer Generator," *Proc. of the 1987 SIGMOD Conf.*, San Francisco, CA, May 1987.

[Grae87b] Graefe, G., "Rule-Based Query Optimization in Extensible Database Systems," Ph.D. Thesis, Comp. Sci. Dept., Univ. of Wisconsin, Madison, 1987.

[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, A., "R-Trees: A Dynamic Index Structure for Spatial Searching," *Proc. of the 1984 SIGMOD Conf.*, Boston, MA, May 1984.

[Hull87] Hull, R., and King, R., "Semantic Database Modeling: Survey, Applications, and Research Issues," *ACM Comp. Surveys* 19, 3, Sept. 1987.

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

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

[Klah85] Klahold, P., et al, "A Transaction Model Supporting Complex Applications in Integrated Information Systems," *Proc. of the 1985 SIGMOD Conf.*, Austin, TX, May 1985.

[Lecl87] Lecluse, C., et. al., "$O_2$, An Object-Oriented Data Model," *Proc. of the 1988 SIGMOD Conf.*, Chicago, IL, June 1988.

[Lind87] Lindsay, B., McPherson, J., and Pirahesh, H., "A Data Management Extension Architecture," *Proc. of the 1987 SIGMOD Conf.*, San Francisco, CA, May 1987.

[Lisk77] Liskov, B., et al, "Abstraction Mechanisms in CLU," *Comm. ACM* 20, 8, Aug. 1977.

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

[Lohm88] Lohman, G., "Grammar-Like Functional Rules for Representing Query Optimization Alternatives," *Proc. of the 1988 SIGMOD Conf.*, Chicago, IL, June 1988.

[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.

[Mano86] Manola, F., and Dayal, U., "PDM: An Object-Oriented Data Model," *Proc. of the Int'l. Workshop on Object-Oriented Database Sys.*, Pacific Grove, CA, Sept. 1986. Grove, CA, Sept. 1986.

[Mylo80] Mylopoulos, J., et. al., "A Language Facility for Designing Database-Intensive Applications," *ACM Trans. on Database Sys.* 5, 2, June 1980.

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

[Niev84] Nievergelt, J., H. Hintenberger, H., and Sevcik, K.C., "The Grid File: An Adaptable, Symmetric Multikey File Structure," *ACM Trans. on Database Sys.* 9, 1, March 1984.

[Peck88] Peckham, J., and Maryanski, F., "Semantic Data Models," *ACM Comp. Surveys* 20, 3, Sept. 1988.

[Rich87] Richardson, J., and Carey, M., "Programming Constructs for Database System Implementation in EXODUS," *Proc. of the 1987 SIGMOD Conf.*, San Francisco, CA, May 1987.

[Rich89a] Richardson, J., and Carey, M., "Implementing Persistence in E," *Proc. of the Newcastle Workshop on Persistent Object Sys.*, Newcastle, Australia, Jan. 1989.

[Rich89b] Richardson, J., and Carey, M., "The Design of the E Programming Language," Tech. Rep., Computer Sciences Dept., Univ. of Wisconsin, Madison, Jan. 1989.

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

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

[Rowe87] Rowe, L., and Stonebraker, M., "The POSTGRES Data Model," *Proc. of the 13th VLDB Conf.*, Brighton, England, Aug. 1987.

[Sche86] Schek, H.-J., and Scholl, M., "The Relational Model with Relation-Valued Attributes," *Information Sys.*, 11, 2, 1986.

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

[Schw86] Schwarz, P., et al, "Extensibility in the Starburst Database System," *Proc. of the Int'l. Workshop on Object-Oriented Database Sys.*, Pacific Grove, CA, Sept. 1986.

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

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

[Ston76] Stonebraker, M., Wong, E., and Kreps, P., "The Design and Implementation of INGRES," *ACM Trans. on Database Sys.* 1, 3, Sept. 1976.

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

[Ston83] Stonebraker, M., et al, "Document Processing in a Relational Database System", *ACM Trans. on Office Info. Sys.* 1, 2, April 1983.

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

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

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

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

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

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

[Zani83] Zaniolo, C., "The Database Language GEM," *Proc. of the 1983 SIGMOD Conf.*, San Jose, CA, May 1983.