μDatabase Annotated Reference Manual (Preliminary Draft)

Version 1.0

Peter A. Buhr and Anil K. Goel

September 1, 1998

Contents

Preface

The goal of this work is to provide an efficient methodology for constructing low-level database tools that are built around a single-level store implemented using memory mapping. The methodology allows normal programming pointers to be stored directly onto secondary storage, and subsequently retrieved and manipulated by other programs without having to modify the pointers or the code that manipulates them. File structures for a database, e.g., a B-Tree, built using this approach are significantly simpler to build, test, and maintain than traditional file structures. All access methods to the file structure are statically type-safe and file structure definitions can be made generic in the type of the record and possibly key(s) stored in the file structure, which affords significant code reuse. An additional design requirement is that multiple file structures may be simultaneously accessible by an application. Concurrency at both the front end (multiple accessors) and the back end (file structure partitioned over multiple disks) are possible and different forms of recovery can be used to build robust file structures.

This manual is strictly a reference manual for μDatabase. A reader should have an intermediate knowledge of concurrency and database issues to understand the ideas presented in this manual as well as some experience programming in μC++.

This manual contains annotations set off from the normal discussion in the following way:

> □ Annotation discussion like this is quoted with quads. □

An annotation provides rationale for design decisions or additional implementation information. Also a chapter or section may end with a commentary section, which contains major discussion about design alternatives and/or implementation issues.

Each chapter of the manual does *not* begin with an insightful quotation. Feel free to add your own.

Chapter 1

Introduction

The goal of this work is to provide an efficient methodology for constructing low-level database tools that are built around a single-level store implemented using memory mapping. A *single-level store* gives the illusion that data on disk (*secondary storage*) is accessible in the same way as data in main memory (*primary storage*), which is analogous to the goals of virtual memory. This uniform view of data eliminates the need for complex and expensive execution-time conversions of structured data between primary and secondary storage. A uniform view of data also allows the expressive power and the data structuring capabilities of a general purpose programming language to be used in creating and manipulating data structures stored on secondary storage. Although a single-level store is an old idea [Org72, IBM78], it has seen only limited use inside of operating systems, and it is only during the last few years that this idea has begun to receive new attention and approval from researchers in the database and programming language communities [CFW90, SZ90a, LLOW91]. For complex structures, a single-level store offers substantial performance advantages over conventional file access, which is crucial to database applications such as CAD/CAM systems, text management and GIS [vO90]. We argue that the performance advantage of a single-level store is lost if the pointers within it have to be relocated or swizzled [CAC+84, Mos90, Wil91].

One way of efficiently implementing a single-level store is by means of memory mapped files. Memory mapping is the use of virtual memory to map files stored on secondary storage into primary storage so that the data is directly accessible by the processor's instructions. Therefore, explicit read and write routine calls are not used to access data on disk. All read and write operations are done implicitly by the operating system during execution of a program. When the working set of the data structure can be kept in memory, performance begins to approach that of memory-resident databases. The memory mapping approach was not used in the past because of a lack of virtual memory hardware on most computers and/or limited access to memory-mapping capabilities by older operating systems. With today's large address-spaces (32-64 bits), memory mapping of secondary storage makes excellent sense, and operating systems are starting to provide access to this capability (e.g., `mmap` system call in UNIX and general access to virtual memory in Mach [TRY+87] and Sun OS 4.1 [Sun90]).

It is possible to implement a single-level store using memory mapped files in a variety of ways. A tool kit approach was adopted as the implementation strategy because it allows programmers to participate in some of the design activity; the tool kit is called μDatabase. Persistence in μDatabase is orthogonal because creating and manipulating data structures in a persistent area is the same as in a program. μDatabase is intented to provide easy-to-use and efficient tools for developing new databases, and for maintaining existing databases. While μDatabase shares the underlying principles of a single-level store with other recent proposals [STP+87, CFW90, SZ90a, LLOW91], it offers features that make it unique and an attractive alternative. It also fulfills a need for a set of educational tools for teaching operating system and database concepts. μDatabase is *not* an object store but it could be used to implement one.

The basic tenet in this work is that memory mapping provides a means of simplifying the implementation and improving the performance of file structures and their access methods built using it. Techniques for solving problems that arise from the use of memory mapping as a means of building a storage system are examined in detail. Many of the problems, such as consistency and concurrency control, have essentially the same implications in memory-mapped databases as they do in traditional databases. However, the use of memory mapping allows more efficient and straightforward solutions. At the same time, memory mapping techniques provide an enormous benefit in terms of simpler interfaces between the low-level database structures and the database designer, and subsequently, between the DBMS and the application developers. Thus, memory-mapped file structures turn out to be simpler to implement than their traditional counterpart.

In this manual, a *file structure* is defined to be a data structure that is a container for user records on secondary storage; a file structure relates the records in a particular way, for example, maintaining the records in order by one or more keys. An *access method* is defined to be a particular way that records are accessed. A file structure may have several access methods, for example, initial loading of records, sequential access of records, keyed access of records.

1.1 Motivation

A database programmer is faced with the problem of dealing with two different views of structured data, viz. the data in primary storage and the data on secondary storage. Traditionally, these two views of data tend to be incompatible with each other. The data in primary storage is usually organized using pointers, which are used directly by the processor's instructions. It is cumbersome and expensive to construct complex relationships among objects without the help of direct pointers. However, it is generally impossible to store and retrieve data structures containing pointers to/from disk without converting at least the pointers and at worst the entire data structure into a different format. Considerable efforts, both in terms of programming and execution time, have to be made in such systems to transform data from one view to the other. In general, these transformations are data structure specific and must be executed each time the data structure is stored or read from secondary storage. This situation is further hampered by the limited access to the file system provided by most operating systems. Finally, the powerful and flexible data structuring capabilities of modern programming languages are not directly available for building data structures on secondary storage. We have observed about a 30% reduction in code when not having to transform pointer and/or data, which correlates with others [ABC$^+$83].

In spite of these rather taxing difficulties, database implementors have traditionally rejected the use of mapped files and have chosen to implement the lower-level support for databases using traditional approaches (e.g., explicit buffer management). This rejection is not totally based on the lack of memory mapping facilities. The earliest use of memory mapping techniques can be traced back 20 years to the Multics system. However Multics provided these facilities in a framework that was very rigid and difficult to work with. Further objections are: [SZ90a, p. 90]

- Operating systems typically provide no control over when the data pages of a mapped file are written to disk, which makes it impossible to use recovery protocols like write-ahead logging [RM89] and sophisticated buffer management [CD85].
- The virtual address space provided by mapped files, usually limited to 32 bits, is too small to represent a large database.
- Page tables associated with mapped files can become excessively large.

These criticisms, while valid in the past, are no longer as strong now, as pointed out in [CFW90]. The rebuttal to these criticisms are:

- Newer operating systems, such as Mach [TRY$^+$87] and Sun OS 4.1 [Sun90], are considerably more liberal in what they allow users to do with the underlying virtual memory system.
- The address space provided by 32 bits, while not excessively large, is sufficient for the majority of applications. Additionally, processors with larger address spaces (up to 64 bits) are becoming available [Mip91].
- Memory management chips are becoming more sophisticated so that less memory is used for page tables, for example, N-level paging and page tables that are smaller than the size of the area they map by using subscript checking before indexing the page table [RKA92].

1.2 Memory Mapping

Memory mapping is the technique of using the underlying virtual memory support (both hardware and software) of the operating system to map some portion of the disk (e.g., a file) into the address space of a process, so that the data stored on disk apparently becomes directly accessible by the processor's instructions. Once mapped, the disk file has a one-to-one correspondence with its image in virtual memory. Memory mapping capabilities vary substantially among different computers and the access to this capability varies depending upon the operating system. In general, there are two major capabilities: segmentation and paging, which can be used independently or together. A *segment* is an area of memory that appears contiguous at the instruction fetch/store level and has a fixed starting address, usually 0. *Paging* is the ability to locate a segment non-contiguously in primary storage while maintaining the segment's contiguous property at the instruction fetch/store level. Depending on the system, memory mapping can map a file into a new segment or into a portion of an existing segment. As will be seen in further discussions, μDatabase only considers the case where a file is mapped into a new segment; otherwise the file mapping does not necessarily start at the same virtual address for each mapping, and memory addresses stored into the file cannot be used to access the data without first being modified. Demand segmentation and demand paging, which is the ability to copy only those segments/pages into primary storage that are referenced, is also essential because a database is almost always larger than the primary storage capacity of the machine. Notice that demand paging performs the job of a traditional buffer manager, except the buffering is implicit and tied into access at the instruction fetch/store level. Ideally, different page replacement algorithms are required for different kinds of access patterns to achieve maximum efficiency [Smi85], but we conjecture that the desired efficiency is possible with only a small number of different page replacement schemes. Currently, it is not possible to work at this level in the operating systems on which μDatabase is implemented.

1.2.1 Disadvantages of Memory Mapping

Larger Pointers Memory pointers may be larger than disk offsets, which increases the size of the file structure marginally, thus increasing access cost. However, only when the percentage of pointers to data is large is this a significant problem. This problem is further mitigated by the low cost and high density of secondary-storage media. As well, larger pointers have increased resolution for accessing data.

Non-Uniform Access Speed The apparent direct access of data can give a false sense of control to the file structure designer. While a file's contents are directly accessible to the processor, the access speed is non-uniform—when a non-resident page is referenced, a long delay occurs as for a traditional I/O operation; otherwise the reference is direct and occurs at normal memory speed. When programming a file structure using memory mapping, certain data structures may be inappropriate because of their access patterns in a non-uniform memory.

1.2.2 Advantages of Memory Mapping

Common Data Structure in Primary and Secondary Memory Use of programming-language data structures to organize the contents of a file eliminates the need to convert to a secondary storage format, which results in code that is substantially more reliable and easier to maintain. Also, for complex data structures, like an object in a CAD/CAM system, where a large percentage of the data is pointers, there is a significant performance advantage because no modification of pointers is required.

Reduced Need for Explicit Buffer Management Efficient buffer management is a major issue with traditional databases. A sophisticated buffer manager is crucial for the performance of a traditional database system. Furthermore, a file structure designer must be skilled in its use, explicitly invoking its facilities and pinning/unpinning buffers. This results in code that is complex, and difficult to understand and maintain. On systems without pinning support, double paging, i.e., paging of the buffers, is a serious drawback. A memory mapped access method is less complex because I/O management is largely transparent and is handled at the lowest possible level (instruction fetch and store). This transparency significantly reduces access method complexity.

Our results show that the buffer management provided by an operating system page-replacement algorithm produces results that are comparable to a hand-coded buffer-manager over a variety of access patterns [BGW92]. Allowing writers of memory-mapped access methods to affect their own page-replacement could produce even better performance for some specialized access methods. But this is still not buffer management, only hints to the operating system as to which pages are no longer needed or will be needed.

Simple Localization While locality of references is crucial for all data structures where access is non-uniform, memory-mapped access methods can easily take advantage of it by controlling memory layout; controlling disk layout through a traditional buffer manager and the file system can be difficult. Because the data structures on secondary storage can be manipulated directly by the programming language, tuning for localization is straightforward. Simple changes to memory allocation strategies can produce significantly better performance in memory-mapped access methods due to localization of accesses.

Rapid Prototyping of Access Methods Because a file structure designer works with a uniform view of data, a file structure can be reliably constructed in a short period of time, using all the available programming-language tools. Interactive debuggers make it possible to find errors quickly by examining data stored on secondary storage as easily as that stored in primary storage. In addition, the debugger knows the type of all the data structures, and hence, displays formatted output. Tools such as execution and storage management profilers, and visualization aids are also directly usable. Finally, polymorphism in a language is available to reuse existing file structure code.

Memory Mapping on a Loaded System In memory mapping, all I/O is done by the page-replacement algorithm so that the operating system can be fair to all users and dynamically respond to the system load either from database access or from non-database access. When the system load is light, it is perfectly reasonable to allow large portions (many megabytes) of the database to reside in memory. When the system load is heavy, database users have to share the machines resources with other users. In traditional database systems, the buffer manager is often in conflict with other applications, holding resources that it may not be using. It is our contention that, on shared systems, memory-mapped access methods can more easily achieve better overall performance than traditional database systems because the memory-mapped access methods can immediately take advantage of available storage to reduce I/O operations. This is because the operating system has knowledge about the entire state of the machine, and therefore, has the potential to make the best decisions to achieve good overall performance.

Contiguous Address Space Memory mapping provides the file structure designer with a contiguous address space even when the data on secondary storage is not contiguous. A single object within a given file structure may be split into several extents on one disk or across multiple disks, and a file structure designer may see no difference or only a sparse address space. In traditional systems, the buffer manager has to be designed to support this seamless view of individual objects in a file structure consisting of non-contiguous fixed-size blocks on secondary storage.

1.3 Organization

This manual is organized as a series of tutorials that illustrate the design methodology behind μDatabase and the different parts of the μDatabase tool kit. It is hoped that users of μDatabase can work directly from the tutorials in building their own persistent file structures.

Chapter 2

μDatabase Design Methodology

μDatabase uses the notion of a persistent area, in which data objects can be built or copied if they are to persist [BZ86, BZ89]. (The major alternative approach is reachability [PS-87, MBC$^+$89]; see Section 10.1.2.) A persistent area is currently implemented by an operating system file. If data is to be transferred from one persistent area to another, the data must be copied through primary storage. However, techniques are available to modify data directly in a persistent area. In all cases, the user interface to the file structure can provide encapsulation of the persistent area to ensure its integrity.

An application may need to access several persistent areas simultaneously. To accomplish this requirement, each persistent area is mapped into its own segment, while allows each file structure to use conventional pointers without having to adjust them. This approach is in contrast to systems that provide simultaneous access by mapping multiple persistent areas into the same segment. In these systems, all pointers are relocated when portions of an area are mapped. In general, this requires access to the type information of the file structure at runtime, which is not usually possible in programming languages that do not have runtime type-checking. Also, significant execution overhead may be incur in relocating pointers.

Currently, μDatabase does *not* cover pointers among persistent areas (see [BZ89] for a possible solution). Nor does it deal with distributed persistent areas; we believe that distributed shared memory [SZ90b, WF90] will allow our current design to scale up to a distributed environment. Object-oriented programming techniques are employed in the implementation of μDatabase, but are not essential. μC++ [BDS$^+$92] is used as the implementation language, which is a superset of C++ with concurrency extensions, because it allows immediate technology transfer. However, the fundamental ideas are implementable in any imperative programming language.

The following two properties evolved during the design and implementation of μDatabase. First, data associated with accessing a file structure, such as current location in the file, concurrency data or transient recovery information are not mapped into the file structure. Second, a deliberate attempt is made to retain the conventional semantics of *opening* and *closing* a file. A persistent area must be made accessible explicitly because its content is *not* directly accessible to the processor(s) until it is memory mapped, and therefore, this should be reflected in the semantics of the constructs and not hidden away by making the file implicitly accessible at all times. Implicit schemes, like pointer-swizzling on first access, have problems detecting the first access but more importantly is the problem of knowing when an access can be terminated (garbage collectors may be too slow). The two properties involve several levels, each performing a particular aspect of the storage or access management of the file structure. This structure is illustrated in Figure 2.1 and the components at each level are discussed in detail.

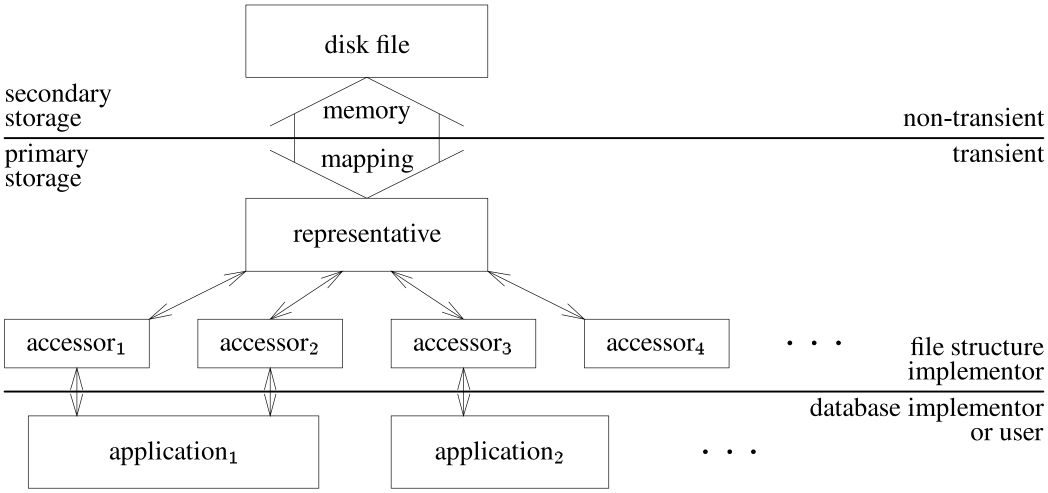

Figure 2.1: Basic Structure of the Design Methodology

2.1 Representative

A *representative* is responsible for creation and initialization of the file structure, for the storage management of access method data in primary storage, for concurrent accesses to the file's contents, and for consistency and recovery. Each file structure has a unique representative. The representative is created on demand, during creation of a file and for subsequent access by a user, and exists only as long as required by either of these operations. Thus, the representative is created when the first explicit access request is received for the file structure and terminates when all the access requests are completed.

In μDatabase, the representative is implemented by a UNIX process, which has its own virtual address space in which transient information is maintained and the file is mapped, and its own thread of control. A representative's memory is divided into two sections: private and shared (see Figure 2.2). Private memory can only be accessed by the thread of control associated with the UNIX process that created it, i.e., the representative. The disk file is mapped into the private memory while all data associated with concurrent access and consistency are contained in the representative's shared memory; such data is always transient. Shared memory is accessible by multiple threads associated with UNIX processes that interact with the representative. There is no implicit concurrency control among threads accessing shared memory; mutual exclusion must be explicitly programmed by the file structure designer using the facilities in μC++.

To allow addresses to be stored directly into the file and subsequently used, the following convention is observed by all representatives: the disk file must be mapped into memory starting at a fixed memory location, called the *segment base address*. The segment base address is conceptually the *virtual zero* of a separate segment; this is how μDatabase uses a UNIX process as a separate segment. In μDatabase, the value 16M has been chosen for the segment base address as the starting location of all mapped files; this leaves a sufficiently large space for the application and the representative(s). Note that the address space is sparsely filled and uses only as much virtual memory as that referenced in the representative. In an ideal situation, where independent creation of new virtual memory segments is allowed, each disk file would be mapped into its own segment, as in Multics, instead of a separate process. Separate segments are supported by several processors, like the Intel 386/486, HP-PA, and IBM-RS6000 computers; however, access to the segmentation capabilities may not exist in the operating systems.

An application in μDatabase can have multiple file structures accessible simultaneously. This capability is possible because each representative has its own private mapping area. Figure 2.3 shows the memory organization of an application using 3 file structures simultaneously. Note each representative uses some of

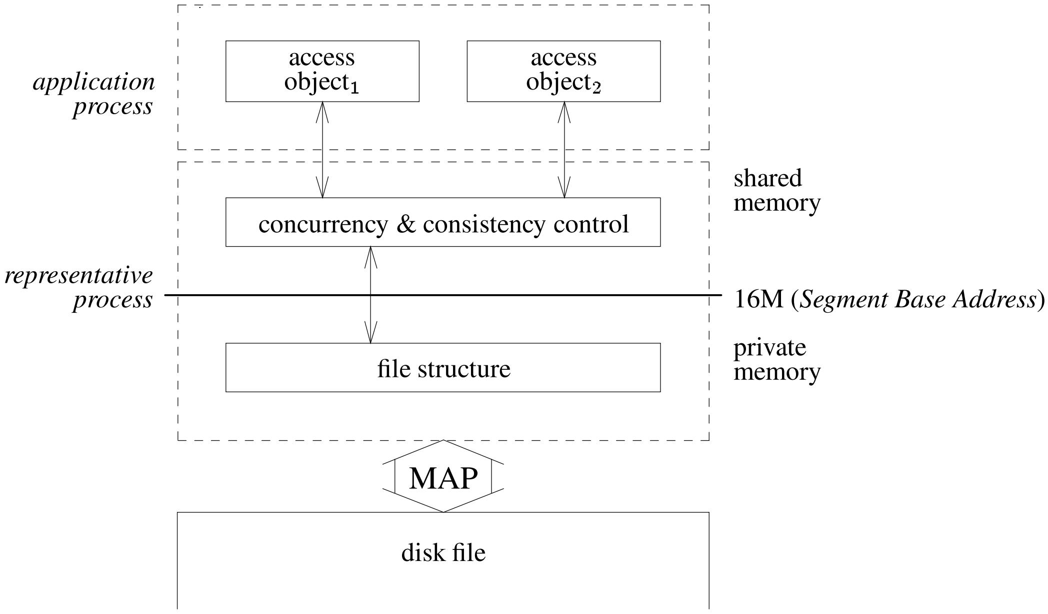

Figure 2.2: Storage Model for Representative

the shared memory for concurrency and consistency data. This memory organization is implemented using four page tables, one of each UNIX process and each page table maps the same shared memory.

The disadvantage of this approach is that there can never be pointers from the shared area to any of the private mapped areas and vice versa. However, addresses from one file structure can be stored in another file structure, but such addresses can only be dereferenced in the file structure they come from. Hence, either data must be copied out of a file structure to be manipulated by an application and copied back again, or an application light-weight task must migrate to the UNIX process implementing the representatives to perform a series of operations. As well, in object-oriented languages, objects with virtual members cannot be instantiated in the file structure as the virtual pointer refers back to data in shared memory (virtual routine vectors).

> □ Dynamic linking would solve this problem. Our solution is to use free routines and generic definitions instead of virtual routines (see Section 3.5.1). □

2.2 Access

The mechanisms for requesting and providing access to a file structure are provided in the form of another abstract data-type, which is implemented as a class called an *access* class. Declaration of an access class instance, called an *access object*, constitutes the explicit action required to gain access to a file's contents (i.e., create the mapping). Creating an access object corresponds to opening a file in traditional systems but it is tied into the programming-language declaration mechanism (like block structure). As well, the access object contains any transient data associated with a particular access (e.g., the current record pointer), while the representative contains global transient information (e.g., the type of access for each accessor). Because the access object is in the application process, communication between it and the representative process is done by synchronous calls passing data through shared memory. At least one access class must be provided for each file definition. It is possible to have multiple access classes, each providing a distinct form of access

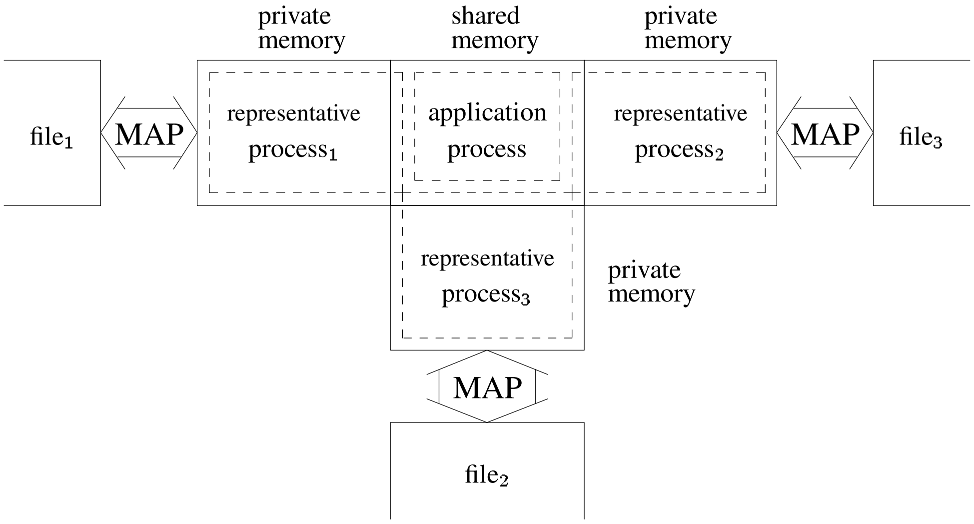

Figure 2.3: Accessing Multiple File Structures

(e.g., initial loading, sequential, keyed). It is also possible to have multiple access objects communicating with the same representative. This capability allows an application to have multiple simultaneous views of the data (see Figure 2.1).

Depending on the particular kind of concurrency control, the declaration of the access object may block until it is safe to access the file contents and/or individual access routine calls may block. Once instantiated, the access object can be used by an application to perform operations on the file structure.

Chapter 3

Storage Management

One of the most complex parts of any data structure is its efficient storage management. In fact, much of a file structure designers time is spent organizing data in memory and on secondary storage. For memory mapped file structures, organizing data in memory indirectly organizes the data on secondary storage.

This chapter discusses the conventions and software tools used to organize and manage a file structure's storage. By following these conventions and using the appropriate tools, it is possible to significantly reduce the amount of time it takes to construct a complex file structure. The first half of the section details the programming interface to the memory mangement tools and the second half is a tutorial in which a simple persistent linked-list data structure is build using the tools.

3.1 Memory Organization

In μDatabase, memory is divided into three major levels for storage management:

address space is a set of addresses from 0 to N used to refer to bytes or words of memory. This memory is conceptually contiguous from the address-space user's perspective, although it might be implemented with non-contiguous pages. The address space is supported by the hardware and managed by the operating system.

segment is a contiguous portion of an address space. There may be a one-to-one correspondence between an address space and a segment, or an address space may be subdivided into multiple segments. In μDatabase, a segment is also mapped onto a portion of the secondary storage. The segment is supported by the hardware and managed by the address-space storage manager (usually the representative).

heap is a contiguous portion of a segment whose internal management is independent of the storage management of other heaps in a segment, but heaps at a particular storage level will interact. The heap is *not* supported by the hardware and is managed by its containing storage manager.

Since μDatabase is capable of creating multiple mappings simultaneously, multiple segments can exist at the same time. In a traditional programming environment with only a single heap, dynamic memory management routines for the heap are usually provided by the programming language system (e.g., `new` and `delete` operators). This facility is, unfortunately, no longer adequate for the multiple segments in μDatabase. This lack of capability results from two major differences between a heap area in shared memory and a private memory-mapped segment:

1. If there is only one heap, any space allocated must come from that heap. When multiple segments can be present at the same time, a target segment must be specified each time a memory management request occurs.

2. The shared memory heap is a general purpose storage area. A mapped segment, on the other hand, is almost always dedicated to a particular data structure, e.g., a linked list or a B-Tree. Therefore, there is an opportunity for optimizing the storage management scheme based on the contained data structure. In addition, many data structures require special action to be taken when overflow and underflow occurs. The storage management facility has to be able to accommodate application specific actions for these cases.

To achieve the above, μDatabase memory management facilities are provided in the form of genetic *memory manager classes*. Memory manager objects instantiated from these classes are self-contained units capable of managing a contiguous piece of storage of arbitrary size, starting at an arbitrary address. If a segment is managed by a given memory manager object, invoking member routines within that object implicitly perform the desired operation on that segment. And since the different managed areas are controlled by different memory managers, it is possible to create memory management classes with different storage management schemes to suit the needs of different data structures. Finally, a programming technique is provided that allows application specific overflow action.

3.2 Address Space Tools

As mentioned, an address space is managed by the operating system so there is usually little or no control over it by the file structure designer. However, some operating systems support specifications like sequential or random access of an address space, providing different paging schemes for each; facilities to control which page is replaced would be extremely useful, but are almost never available. The point here is that if address-space management tools are available, they can make a significant difference in the performance of a file structure, but currently such tools are not portable across different UNIX operating systems so this manual does not discuss such mechanisms.

3.3 Segment Tools

The following tools create, manage and destroy segments in an address space. Furthermore, flexible capabilities are provided for mapping one or more disk files into a segment. This latter capability is discussed in detail in Chapter 6 where it is used for parallel I/O. In this chapter, only a single file is mapped into a segment. All segment capabilities are provided through the representative for a file structure.

3.3.1 Representative Interface

The header file:

```
#include "/u/usystem/software/udb/Rep/src/Rep.h"
```

defines the representative facilities and it is mandatory for any application requiring μDatabase facilities. The basic representative functionality is provided by three related classes: Rep, RepAccess and RepWrapper.

Rep is the representative data structure. It is responsible for mapping and unmapping files to/from segments, and controlling the size of the segment and hence the size of the file. The basic interface of Rep is as follows:

```
class Rep {
  public:
    virtual void *start();              // starting address of mapping
    virtual int size();                 // current size of mapping
    virtual void resize(int size);      // resize mapping
    virtual int created();              // UNIX file created by this rep?
};
```

The member routine start returns the starting address of the mapping, which is currently 16M, i.e., the segment base address. The member routine size reports the current size of the mapped space and thus the size of the mapped file. The routine resize sets the size of the mapped space, and indirectly, the file size to the value given in its argument. Finally, the routine created return 1 if the requested UNIX file was created by the current representative, and 0 if the file was present before the representative was created.

The class Rep is not intended to be instantiated by file structure code, which is why it has no constructor. Instead, a representative is created indirectly through the class RepAccess, which may create an instance of Rep for a file or use an existing one. Thus, the only way to control file mapping and unmapping is through an instance of RepAccess. The representative access object takes part in maintaining the μDatabase global representative table that guarantees a one-to-one relationship between representative and file mapping in an application. Therefore, application programs do not have direct access to the representative creation process and must rely on the service of RepAccess.

The interface to RepAccess is:

```
class RepAccess {
  public:
    RepAccess( char *filename );
    virtual void *start();              // starting address of mapping
    virtual int size();                 // current size of mapping
    virtual void resize(int size);      // resize mapping
    virtual int created();              // UNIX file created by this rep?
};
```

The constructor's parameters is the name of the UNIX file to be mapped. Upon the creation of an instance of RepAccess, the global representative table is searched in an attempt to locate an active representative for the requested UNIX file. If a representative for the file is present, the file is already mapped so no new mapping is necessary. A pointer to this representative is stored in the RepAccess instance, the representative's use count in the global table is incremented, and the creation is complete. If no representative is found, a representative is created for the file (i.e., an instance of Rep) and entered into the representative table. If the file does not already exist in the UNIX file system, it is created.

The member routines start, size, resize and created are covers for those in the Rep class. They perform the same function as their counterparts in Rep. They are present so the full functionality of the representative is available to the file structure designer via the representative access class. This approach serves to completely isolate the representative objects from the file structure code. However, this presents a problem for the objects within the persistent area for the following reasons:

1. A persistent object within the file space cannot reliably refer to an existing RepAccess object created outside the mapped area because a RepAccess object is created on a per access basis and has a many-to-one relationship with the file space.

2. A RepAccess object cannot be created from inside the mapped area because that would result in a pointer out of the mapped area, which is a pointer to a transient object from a persistent area.

3. The constructor for RepAccess takes the name of the UNIX backing file as an argument. To supply

the argument, the name of the file has to be stored inside the persistent area. That means, the UNIX backing file must not be renamed once it is created by μDatabase. This limitation is unacceptable.

Therefore, the only access to mapping control for objects within the persistent area is by a direct pointer to the Rep structure.

After the representative is created (indirectly by RepAccess), the file is mapped into a new segment, and by convention, the representative writes a pointer to itself at the beginning of the newly mapped space. This action is done for the following reason. A storage manager for a segment or heap must exist before the area it manages so there is at least somewhere to store a pointer to the new segment or heap. Therefore, the storage manager is allocated out of an existing storage area and the new storage area is conceptually nested in the storage area that contains its storage manager. In general, the nesting relationship needs both a pointer from parent to child and vice versa. Without the back pointer from child to parent or a pointer to the root of the storage hierarchy, it is not possible to find the parent storage manager when a child needs more storage. The pointer inserted at the beginning of a segment for a newly mapped file provides the back pointer for storage managers in the segment to communicate with the representative's storage management operations.

Figure 3.1 shows the organization of representatives and their access classes and segments. The representatives are chained together to allow them to be searched when a representative's access class is created to see if there is already a representative for a particular file. Notice, also, the pointer from the segment to the representative. Having a pointer from persistent memory to the transient memory for the representative violates a previous design restriction because a pointer to the transient representative from the persistent file is invalid as soon as the application that created the representative terminates. However, this scheme works because the representative pointer is relocated on each access to a persistent area by an application. Hence, there is a trivial amount of dynamic relocation in this scheme.

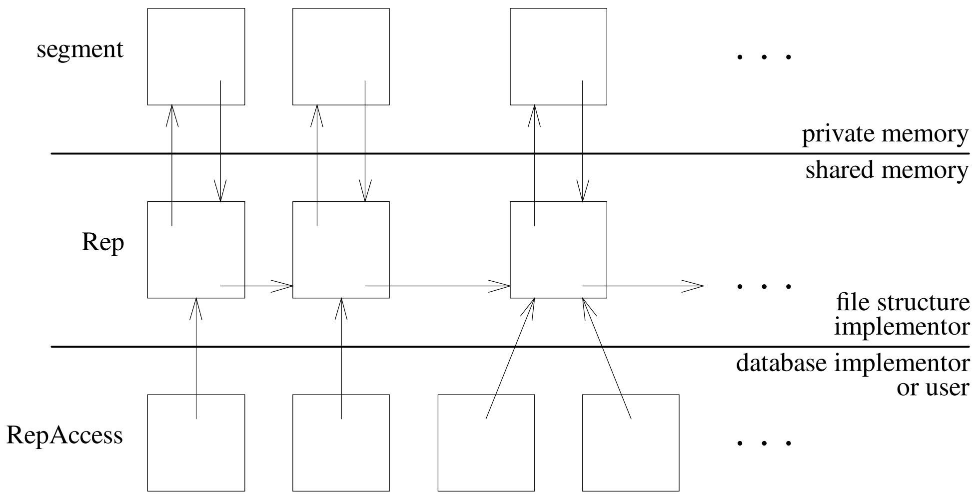

Figure 3.1: Organization of Representatives

Upon the destruction of an instance of RepAccess, the use count for the representative in the global table is decremented. If the use count reaches zero, then all access requests for the file have been closed. The mapping is then terminated and the representative destroyed.

Since the file is mapped into the representative's private memory, user application code does not normally have direct access to the contents of the mapped file in private memory; the application code only has access to shared memory. The class RepWrapper provides the mechanism to allow application code to access both shared and private memory for a particular representative's segment:

```
class RepWrapper {
  public:
    RepWrapper( RepAccess &repacc );
};
```

The class is a *wrapper* class and therefore does not have any member routine; all actions of the wrapper class are carried out by the constructor and destructor of the wrapper. When a wrapper is declared in a block, both halfs of the wrapper's operations are guaranteed to be performed, even if the block is terminated by an exception. RepWrapper's constructor takes an instance of RepAccess as an argument, which indirectly refers to a representative's address space and any segments mapped into it.

As soon as an instance of RepWrapper is created, that representative's address space is accessible, as well as shared memory; the duration of accessibility is the duration of the wrapper. Note however that two wrappers cannot be active at the same time because only one address space can be in effect at a time so only segments in that address space are accessible. Therefore, a task cannot have direct access to two mapped file simultaneously. One way to ensure this restriction is to only create one instance of RepWrapper per block and make the wrapper the first declaration to ensure the segment is accessible before operations are performed on it, as in:

```
void list::rtn() {
    RepWrapper( repacc );    // rep's address space becomes accessible

    // may access data in shared segment and rep's segment only

} // rep's address space, and hence, its segments become unaccessible
```

This convention ensures that the wrapper's actions occur as the first and last operations of a routine block.

□ Accessibility to a representative's segment is accomplished by migrating the light-weight task that is performing the operation from the application address space to the representative's address space where the segment(s) is mapped. The cost is a light-weight context switch and possibly a heavy weight context switch if the UNIX process associated with the representative's address space has blocked. □

3.4 Heap Tools

As mentioned, a segment has no facilities to manage allocation and deallocation of its memory. This section discusses the heap tools that can be used to manage a segment's memory. If none of these tools are appropriate, it is possible to build specialized heap management tools.

3.4.1 Heap Storage Management Schemes

While there are a large number of storage management schemes possible, the following three basic schemes are provided in μDatabase. More will be added as time permits. They are ordered in increasing functionality and runtime cost.

uniform has fixed allocation size. The size is specified during the creation of the memory manager object and cannot be changed afterwards. Uniform memory management is often used to divide a segment into fixed sized heaps (e.g., B-Tree fixed-sized nodes).

variable has variable allocation size. The size is specified on a per allocation basis but once allocated, cannot be changed. This is a general purpose scheme very similar to C's malloc and free [KR88] routines.

dynamic has variable allocation size. The size is specified on a per allocation basis and can be expanded and contracted any time as long as the area remains allocated. Because of this property, the location of allocated blocks are not guaranteed to be fixed. Therefore, an allocation returns an *object descriptor* instead of an absolute address. An allocated block does not have an absolute address and must be accessed indirectly through its descriptor. Because of this indirection, it is possible to perform compaction on the managed space. Therefore, fragmentation can be dealt with in an application independent manner.

It is believed that these three storage management schemes should be able to cope with most application demands. Should special needs arise, special purpose memory management schemes can be created and easily integrated into μDatabase.

The programming interface for each is defined in the header files:

```
#include "/u/usystem/software/udb/inc/usm.h"
#include "/u/usystem/software/udb/inc/vsm.h"
#include "/u/usystem/software/udb/inc/dsm.h"
```

3.5 Heap Overflow Control

When a heap fills, there are 3 actions that can be sensibly taken by a generic storage manager:

1. enlarge the heap by adding additional storage at the end of the heap (a heap is a contiguous area). However, when there are multiple heaps at a particular nesting level, this may necessitate moving one or more heaps after the full heap. Moving a heap requires relocating any pointers to its contents.

2. allocate a new heap, which is larger than the existing heap, and copy the contents of the old heap to the new heap and delete the old heap. However, moving a heap requires relocating any pointers to its contents.

3. allocate a new heap and copy some portion of the contents to the new heap so that each heap has some free space. However, moving a heap's contents requires relocating any pointers within the contents and there are now two independent heaps that must be managed.

Since generic memory managers are independent of the type of data they manage, it is impossible for them to take these actions on behalf of the file structure. Therefore, a generic memory manager does not deal with expansion.

Instead, a generic memory manager is designed with an *expansion exit*, which is activated when a heap fills, so that a data-structure specific action can be performed in this situation. The following are some examples of data-structure specific actions. When a B-Tree node fills, an additional node is allocated and some of the contents of the old node is migrated to the new node. When a variable-size character string heap fills, the heap may be copied to a new heap that is larger and the previous heap freed.

To encapsulate this application specific dependency, the concept of an expansion exit is implemented using an *expansion object*. An expansion object is written as part of a file structure definition and it contains enough information to deal with overflow. All expansion objects are derived from a special *expansion abstract class* and one must be passed to the generic memory manager when it is created. When the generic memory manager detects that a heap is full during an allocation operation, it calls member routines in the expansion object to deal with this situation.

3.5.1 Expansion Object

As mentioned, a basic memory manager does not deal with heap overflow. In order to handle overflow, an application specific heap expansion definition must be created to perform application specific overflow action. The class uExpand below is the interface between the memory manager and the overflow handler:

```
class uExpand {
  public:
    virtual bool expand( int ) = 0;
}; // uExpand
```

The member routine expand is called from within the memory manager whenever more storage is needed. The routine takes an integer argument that specifies the number of additional bytes requested. Every application specific expansion class must redefine the expand routine to perform its required overflow action, adding more private variables to the class definition if necessary. The expand routine's return code controls the future action of the memory manager. If the expand routine returns false, the memory manager gives up and returns NULL to the caller (or raises an exception). If the expand routine returns true, the memory manager once again attempts to allocate memory out of the expanded heap. It fails if there is still insufficient storage after the expansion and the first action is taken.

Because the expand routine is defined as a C++ virtual routine, so that it can be replaced by specialized derived expansion classes, and stored in the persistent area together with the data they manage, the virtual routine pointer must be relocated each time the segment is made accessible. And as mentioned in Section 2.1, virtual members are normally not allowed in a persistent area. Therefore, the expansion object must be handled as a special case and relocated as part of connecting to a persistent store.

The following is the interface portion for the uniform memory manager:

```
class uUniform {
  protected:
    void *mstart;                    // starting addess of heap
    void *hend;                      // current high water mark in heap
    void *mend;                      // ending address beyond the end of the heap
    int usize;                       // size of heap allocation
    freeblk *freel;                  // start of free list
    uExpand &expn;                   // reference to expansion object
  public:
    uUniform( void *mstart, int msize, uExpand &expn, int usize );
    void *alloc( int );
    void *alloc();
    void free( void *p );
    void sethsize( int newsize );
};
```

The constructor takes four arguments.

mstart is the starting address of the managed space (i.e., the heap)

msize is the initial heap size

expn is a reference to the expansion object

usize is the size of the uniform sized blocks being managed.

Once initialized, the member routines alloc and free can be used to allocate and free uniform sized blocks of storage. The member routine sethsize is used to inform the memory manager of the new heap size when the heap size has changed. This routine is intended to be called by the expansion object.

A specific uniform memory manager is created in the following way. First an expansion class is defined and a specific uniform memory manager is created using it, as in:

```
class myExpand : public uExpand {
        // variables necessary to perform expansion
    public:
        myExpand( ... );             // pass data needed for expansion
        bool expand( int ) {
            // code to perform expansion
        }
};

myExpand myExpObj;               // create expansion object

uUniform myUniSM( repacc.start(), 1000, myExpObj, 100 );
```

This creates a uniform memory manager whose storage starts at the beginning of the mapped area, is initially 1000 bytes in size, is allocated in 100 byte blocks and overflow is handled by myExpObj.

For more flexible storage management, the variable or dynamic memory manager may be required. The following is the interface portion for the variable memory manager:

```
class uVariable {
    protected:
        void *mstart;                    // starting addess of heap
        void *hend;                      // current high water mark in heap
        void *mend;                      // ending address beyond the end of the heap
        int usize;                       // default size of heap allocation
        vfb *freel;                      // start of free list
        uExpand &expn;                   // reference to expansion object

        void *_alloc( int size );
    public:
        uVariable( void *mstart, int msize, uExpand &expn, int usize = 0 );
        void *alloc( int size );
        void *alloc();
        void free( void *fb );
        void sethsize( int newsize );
}; // uVariable
```

The following is the interface portion for the dynamic memory manager:

```
class uDynamic {
    public:
        uDynamic( void *mstart, int msize, T &expn );
        Descriptor alloc( int size );
        void free( Descriptor p );
        void sethsize( int newsize );
};
```

The constructors takes three arguments mstart, msize and expn, which specify the starting address, the initial size of the heap and the expansion object, just like they do in the uniform manager constructor. The member routines alloc, free and sethsize perform the same functions are those in the uniform manager. The dynamic manager deals with movable memory blocks, and therefore the alloc and free routines make use of the indirect pointer type Descriptor instead of the direct pointer type void *. Specific variable and dynamic managers are created in the same manner as uniform managers.

Chapter 4

Persistent Linked List

The following example illustrates all the basic techniques and tools for constructing a persistent data structure by building a generic singlely linked list with nodes containing a variable length string value.

4.1 List Application

At the application level, three data structures are provided: one to form the nodes of the list, one to declare a persistent linked list and one to access it. Figure 4.1 shows a simple application program using the persistent linked list. First, there is a definition of the list node, myNode, which must inherit from listNode to get the appropriate link fields added. Since the data in each node is a variable length string, the node structure only defines a place holder field, value of zero size, and storage for the string is allocated as each node is created. Second, is the declaration of the persistent list, l, with UNIX file name abc. The persistent list is generic in the type of the node so that all accesses to the list can be statically type checked. Third, is the declaration of the access class, la, for persistent list l. The access class is generic in the type of the node so that all accesses to the access class can be statically type checked.

The next three loops add, change and remove nodes using the access class routines add, get and put, and remove, respectively. The generic generator, listGen, returns a consecutive sequence of pointers to the nodes in the persistent list. However, these pointers cannot be dereferenced in the application program; they can only be used as place holders to nodes and passed to other access routines, like get and put.

> □ It is possible to create a special list pointer type that restricts dereferencing to authorized list objects. □

4.2 Linked List Implementation

Figure 4.2 shows all the list data structures created and their inter-relationships; it should be useful to refer to this figure during the following discussion.

4.3 List Node

The abstract class, PQueueNode, contains the fields needed by each node in a linked list to relate the data:

```
#include <uC++.h>
#include "/u/usystem/software/udb/LinkedList/src/LinkedList.h"

class myNode : public listNode {               // inherit from list node
  public:
    char value[0];                             // variable sized string
}; // myNode

void uMain::main() {
    list<myNode> l( "abc" );                   // create persistent list
    listAccess<myNode> la( l );                // open list
    char *name = "xxx";
    int i;

    for ( i = 1; i <= 100; i += 1 ) {          // create nodes in list
        la.add( name );
    } // for

    listGen<myNode> gen;                       // used to scan through list
    myNode *p;
    char name[25];

    for ( gen.over( la ), i = 0; gen >> p; i += 1 ) {  // modify the list indirectly
        la.get( p, name );                     // copy out information
        if ( i % 2 == 0 ) {                    // change every 2nd node
            name[0] = 'a';
            la.put( p, name );                     // copy information back
        } // if                                // (size cannot change)
    } // for

    for ( gen.over( la ); gen >> p; ) {        // destroy the list
        la.remove( p );
    } // for
} // uMain::main
```

Figure 4.1: Linked List Example

```
class PQueueNode {                             // abstract class containing link field
    PQueueNode *nxt;
  public:
    PQueueNode *&next() {                      // access to link field
        return nxt;
    } // PQueueNode::next
}; // PQueueNode
```

The member routine next allows indirect access to the link field.

□ Unfortunately, member next cannot be private and accessed through friend classes because friendship is not inherited. □

4.4 List Administration

Information pertinent to a particular linked list, e.g., the pointer to the head of the list and the mapped space's memory management information, must outlive the application program that creates the list. Therefore,

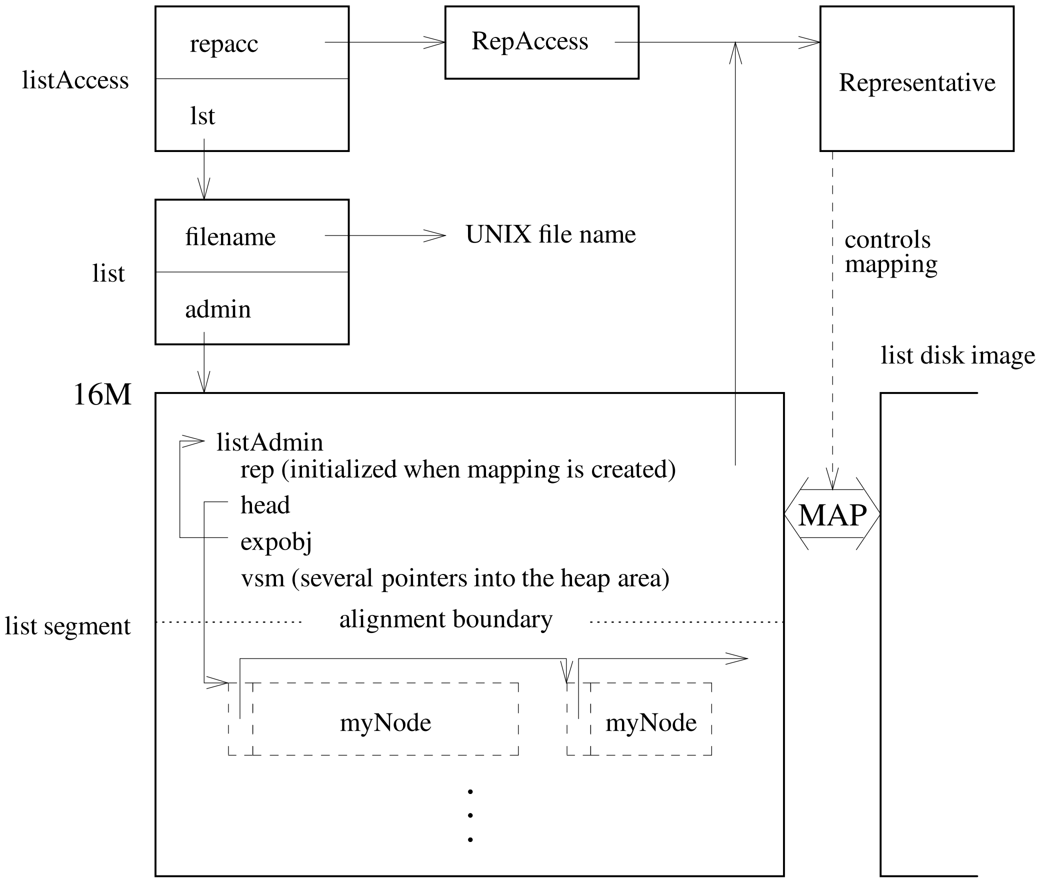

Figure 4.2: Linked List Storage

this information cannot reside in shared memory. Instead, it is stored in the same persistent area as the linked list nodes. By convention, all such persistent administrative information is grouped together into an *administrative object* at the beginning of the segment. Furthermore, the administrative type *must* inherit from the abstract type RepAdmin.

The list administrative class is defined in Figure **??**. The class contains a pointer to the head of the linked list, head, the expansion object for the current mapped space (discussed shortly), and the variable memory manager that manages the mapped space. As mentioned in Section 3.3.1, the representative initializes a pointer to itself at the beginning of every mapped space. This pointer can be accessed from subclasses of RepAdmin through the protected variable rep. The constructor for the administrative class takes an integer indicating the initial heap size as an argument. It initializes the expansion object, expobj, the variable memory manager, vsm, and then sets the list head pointer to NULL, indicating an empty list. The two private member routines alloc and free are utility routines that make use of the underlying variable memory manager. The routine alloc is important because it casts the untyped bytes returned from the variable memory manager into the type of the generic list node.

The expand class for the linked list is defined as follows:

```
template<class T, class SM> class PQueueAdmin : private RepAdmin {
#   define MagicCookie "PQueue<T,SM>"

    friend class PQueueExpType<T,SM>;
    friend class PQueue<T,SM>;

    char typeName[sizeof(MagicCookie)];        // sizeof includes the ´\0´
    T *head, *tail;                            // first and last node of the queue
    PQueueExpType<T,SM> expobj;                // expansion object to extend memory for the queue
    SM sm;                                     // storage manager for the queue

    T *alloc( int size) {
        return (T *)sm.alloc(size);
    } // PQueueAdmin<T,SM>::alloc

    T *alloc() {
        return (T *)sm.alloc();
    } // PQueueAdmin<T,SM>::alloc

    void free( T *p ) {
        sm.free( p );
    } // PQueueAdmin<T,SM>::free

    void checkType( char *filename ) {
        if ( strcmp( typeName, MagicCookie ) != 0 ) {        // dynamic type check
            uAbort( "PQueueAdmin<T,SM>(0x%x)::checkType: File \"%s\" is not type \"%s\".",
                    this, filename, MagicCookie );
        } // if
    } // PQueueAdmin<T,SM>::checkType
  public:
    PQueueAdmin( int fileSize ) : expobj( *this ), sm( (void *)this+sizeof(PQueueAdmin<T,Storage Management>),
                    fileSize-sizeof(PQueueAdmin<T,SM>), expobj, sizeof(T) ) {
        strcpy( typeName, MagicCookie );       // copy the file´s type into the file as a magic cookie
        head = NULL;
    } // PQueueAdmin<T,SM>::PQueueAdmin
}; // PQueueAdmin<T,SM>


template<class T, class SM> class PQueueExpType : public uExpand {
    PQueueAdmin<T,SM> &admin;
  public:
    PQueueExpType( PQueueAdmin<T,SM> &admin ) : admin( admin ) {
    } // PQueueExpType<T,SM>::PQueueExpType

    bool expand( int extension ) {
        if ( extension < 8 * 1024 ) {
            extension = 8 * 1024;              // extend at least 8K each time
        } // if
        admin.rep->resize( admin.rep->size() + extension ); // extend the segment
        admin.sm.sethsize( admin.rep->size() - sizeof(PQueueAdmin<T,SM>) ); // extend the storage manager
        return true;
    } // PQueueExpType<T,SM>::expand
}; // PQueueExpType<T,SM>
```

The constructor initializes a pointer to the administrative object so that it can access the containing storage manager, listAdmin::rep. The member routine expand first resizes the current space by calling the representative's resize routine. It then informs the variable memory manager of the change by calling its sethsize routine and it returns 1 indicating that the allocation operation should be reattempted.

4.5 List File Structure

The purpose of the list file structure is to establish a contention between the program and the UNIX file that contains the list data structure. It does not make the file accessible unless it is creating the file, and then the file is made accessible only long enough to initialize the file structure. Figure 4.3 contains the definition of list.

list's constructor takes two arguments. The first one indicates the name of the UNIX file that contains the linked list. The second one indicates the initial size of the persistent storage that contains the linked list nodes if the file is being created; otherwise this parameter is ignored. list's constructor first makes a copy of name of the UNIX backing file in shared memory, which is a valid action because the duration of this pointer is at most the duration of the program. The constructor then opens up a mapping by creating a RepAccess object, makes the segment accessible by creating a RepWrapper, initializes a pointer to where the administrative object should be at the beginning of the segment (currently 16M), and checks to see if the file was created on access. If it is newly created, the segment size is first set to that indicated by the initSize parameter, then an administrative object is created at the beginning of the segment, which initializes itself through its constructor, creating an empty list.

list's destructor releases the storage taken up by the UNIX filename string in shared memory.

The private member routines first, add and remove are the routines that manipulate the list nodes. These routines are in the list object so that the list can be modified by other objects within the segment. The first routine returns a pointer to the beginning of the list. The add routine calls the uniform storage manager in the administrative object to obtain storage for a myNode that can contain the string parameter, copies the parameter into the new list node, and chains it onto the head of the list. The remove routine removes the give node from the list and frees the storage for the node. These routines make use of standard singly linked-list algorithms.

4.6 List Access Class

An access class defines an access object. An access object defines the duration that a file structure segment is accessible. The class listAccess is the access class for list. It provides routines to operate on the list. It is the sole means for application code to access the list data. It also contains per access information, in a manner similar to a UNIX file descriptor. The list access class is defined in Figure 4.4.

The constructor takes a reference to a list object as an argument. A reference to the list object is retained for subsequent access to list routines and a file structure mapping is creating by creating a RepAccess object. The member routines add and remove are covers for the same routines in the list object. The member routines get and put copy data out of or into the value field of a list node, respectively. All these member routines begin by making the list segment accessible by creating a RepWrapper object and then performing an operation on the list.

4.7 List Generator

A generator iterates over an ordered data structure, returning some or all of the elements of the data structure. Generators provide access to the elements of a data structure without having to use or access the particular data structure's implementation; hence, generators enforce the notion of abstract data types. Depending on the data structure, there may be multiple generators that iterate over the data structure in different ways

```
template<class T, class SM> class PQueue {
    friend class PQueueAccess<T,SM>;
    friend class PQueueGen<T,SM>;

    char *fileName;                     // UNIX file containing the queue
    PQueueAdmin<T,SM> *admin;                     // administrator for the queue segment

    PQueue( const PQueue & );                     // prevent copying
    PQueue &operator=( const PQueue );

    T *first() {                        // return pointer to first node in queue
        return admin->head;
    } // PQueue<T,SM>::first

    T *last() {                         // return pointer to last node in queue
        return admin->tail;
    } // PQueue<T,SM>::last

    void addCommon( T *newNode, const T *value ) {
        new( newNode ) T();
        *newNode = *value;
        if ( admin->head == NULL ) {                // first node in queue ?
            admin->head = admin->tail = newNode;
        } else {                        // general case
            admin->tail->next() = newNode;
            admin->tail = newNode;
        } // if
        newNode->next() = NULL;
    } // PQueue<T,SM>::addCommon

    void add( const T *value, int size ) {          // add node to end of the queue
        T *newNode = admin->alloc( sizeof(T) + size );
        addCommon( newNode, value );
    } // PQueue<T,SM>::add

    void add( const T *value ) {                    // add node to end of the queue
        T *newNode = admin->alloc();
        addCommon( newNode, value );
    } // PQueue<T,SM>::add

    void remove( T *p ) {                           // remove node from queue
        if ( p == admin->head ) {                   // remove first node
            admin->head = (T *)p->next();
            if ( admin->head == NULL ) {            // empty queue ?
             admin->tail == NULL;
            } // if
        } else {                        // remove node in queue
            T *pred, *curr;
            for ( pred = admin->head, curr = (T *)pred->next(); curr != p; pred = curr, curr = (T *)curr->next() ) {
            } // for
            pred->next() = curr->next();
            if ( admin->tail == curr ) {            // last node ?
             admin->tail = pred;
            } // if
        } // if
        admin->free( p );
    } // PQueue<T,SM>::remove
  public:
    PQueue( char *name, int initSize = 4 * 1024 ) {
        fileName = new char[strlen( name ) + 1];          // allocate storage for file name
        strcpy( fileName, name );                   // copy file name
```

```
template<class T, class SM> class PQueueAccess {
    friend class PQueueWrapper<T,SM>;
    friend class PQueueGen<T,SM>;

    RepAccess<Rep> repacc;                // access class for representative
    PQueue<T,SM> &queueRoot;               // queue being accessed

    PQueueAccess( const PQueueAccess & );      // prevent copying
    PQueueAccess &operator=( const PQueueAccess );
  public:
    PQueueAccess( PQueue<T,SM> &queueRoot ) : queueRoot( queueRoot ), repacc( queueRoot.fileName ) {
    } // PQueueAccess<T,SM>::PQueueAccess

    void add( const T *value, int size ) {
        RepWrapper wrapper( repacc );          // migrate to file segment

        queueRoot.add( value, size );
    } // PQueueAccess<T,SM>::add

    void add( const T &value ) {
        RepWrapper wrapper( repacc );          // migrate to file segment

        queueRoot.add( &value );
    } // PQueueAccess<T,SM>::add

    void get( const T *p, T &value ) {
        RepWrapper wrapper( repacc );          // migrate to file segment

        value = *p;
    } // PQueueAccess<T,SM>::get

    void put( T *p, const T &value ) {
        RepWrapper wrapper( repacc );          // migrate to file segment

        *p = value;
    } // PQueueAccess<T,SM>::put

    void remove( T *p ) {
        RepWrapper wrapper( repacc );          // migrate to file segment

        queueRoot.remove( p );
    } // PQueueAccess<T,SM>::remove
}; // PQueueAccess<T,SM>
```

Figure 4.4: List Access Class

```
template<class T, class SM> class PQueueGen {
    const PQueueAccess<T,SM> *qa;
    T *curr;

    PQueueGen( const PQueueGen & );          // prevent copying
    PQueueGen &operator=( const PQueueGen );
  public:
    PQueueGen() {
    } // PQueueGen<T,SM>::PQueueGen

    PQueueGen( const PQueueAccess<T,SM> &qa ) {
        RepWrapper wrapper( qa.repacc );         // migrate to file segment

        PQueueGen::qa = &qa;
        curr = qa.queueRoot.first();
    } // PQueueGen<T,SM>::PQueueGen

    void over( const PQueueAccess<T,SM> &qa ) {
        RepWrapper wrapper( qa.repacc );         // migrate to file segment

        PQueueGen::qa = &qa;
        curr = qa.queueRoot.first();
    } // PQueueGen<T,SM>::over

    bool operator>>( T *&p ) {
        RepWrapper wrapper( qa->repacc );        // migrate to file segment

        p = curr;                      // return current node
        if ( curr != NULL ) {                    // if possible, advance to next node
            curr = (T *)curr->next();
        } // if
        return p != NULL;
    } // PQueueGen<T,SM>::operator>>
}; // PQueueGen<T,SM>
```

Figure 4.5: List Generator

and/or a generator may have several parameters that control the precise way the generator iterates over the data structure. It is highly recommended that every persistent data structure provide a generator. The list generator is defined in Figure 4.5.

The list generator has two constructors. The first constructor allows the specification of a list access object, and it initializes the generator to the beginning of the list. This constructor is used when the generator object is not going to be reused multiple times, possibly with different lists, as in:

```
for ( listGen<myNode>( la ) gen; gen >> p; ) { ... }
```

Here the generator is only used once in the loop.

□ Beware of using this form because the scope of the variable gen is not the loop body but the entire block containing the loop. □

The second constructor is used to create a generator that is subsequently specified to work with a particular list access object, as in:

```
listGen<myNode> gen;
for ( gen.over( la ); gen >> p; ) { ... }
for ( gen.over( ma ); gen >> p; ) { ... }
```

In this case, when the list generator is declared, it is not associated with a particular list access object. The association occurs through the over member routine, which initializes the generator to the beginning of the list. Notice that the generator, gen, is used to iterate over two different list access objects, la and ma, which may be accessing the same or different lists; the only requirement is that both list access-objects access lists that contain nodes of type myNode. Finally, the operator >> is used to extract the next place holder to an element in the data structure. While the place holder may be declared to be a normal pointer, in general, it cannot be dereferenced in the application program because it points into the list segment, which is not accessible from the application (Section 4.8 discusses exceptions to this rule). Instead the place holder is used by other member routines in an access object to transfer element data out of or into appropriate list nodes in the list segment.

4.8 List Wrapper

Figure 4.1 showed how an application modifies the linked list data by copying data out of a list node, changing it, and copying it back; hence, the data is modified indirectly from the original list nodes. The reason for copying is that a pointer returned by a list generator cannot be used in the application program because it points into the list segment, which is not directly accessible from the application. As mentioned in Section 3.3.1, a wrapper is used to make the representative's address space accessible. This technique can be extended to the application program by providing a wrapper that makes the list segment directly accessible; pointers from the list generator can then be used to directly modify data in list nodes, as in:

```
{
    listWrapper<myNode> dummy( la );              // make la's segment accessible

    for ( gen.over( fa ), i = 0; gen >> p; i += 1 ) {  // modify the list directly
        if ( i % 2 == 0 ) {                        // change every 2nd node
            p->value[0] = 'a';
        } // if
    } // for
}
```

A new block is started, { ... }, to define the duration of the list segment access and the list wrapper is declared. Within the block, pointers returned from the generator can be directly dereferenced to read and modify the list node data. A substantial performance gain can be achieved by this technique, because the list segment is only made accessible once for all accesses to the list data and the copying of the list data is eliminated.

The list wrapper is defined as follows:

```
template<class T, class SM> class PQueueWrapper {
    RepWrapper wrapper;                     // migrate to file segment

    PQueueWrapper( const PQueueWrapper & );    // prevent copying
    PQueueWrapper &operator=( const PQueueWrapper );
  public:
    PQueueWrapper( const PQueueAccess<T,SM> &qa ) : wrapper( qa.repacc ) {
    } // PQueueWrapper<T,SM>::PQueueWrapper
}; // PQueueWrapper<T,SM>
```

and it is simply a cover definition for declaring a RepWrapper to the particular list segment.

4.9 Summary

The simple generic linked-list illustrates all the basic conventions and tools for building a persistent file structure. The conventions are:

- The representative writes a pointer to itself at the beginning of the newly mapped segment.
- All persistent administrative information is grouped together at the beginning of the segment and must inherit from RepAdmin to ensure there is space for the back pointer to the representative.
- A block is started before declaring a wrapper so that the wrapper's actions occur as the first and last operations of the block.
- Only one access wrapper can be declared in a block, because only shared memory and one segment's memory canbe accessible at a time.

Each basic file structure should have the following classes at the application level: a node abstract class, a file structure class, one or more access classes, and (usually) one or more generator classes. At the file structure level, there is the administrative class.

In the next chapter, more advanced techniques are demonstrated, such as nested storage management and mapping multiple files into a single segment.

Chapter 5

Persistent Binary Search Tree

5.1 BianryTree Node

The abstract tree node class, Treeable, contains two child pointers from which all tree nodes must inherit.

```
class Treeable {                    // abstract class containing child pointers
    friend class TFriend;

    Treeable *leftChild, *rightChild;
  public:
    Treeable() {
     leftChild = NULL;
     rightChild = NULL;
    }
}; // Treeable
```

To abstract and encapsulate the direct access to a tree node, a friend class of Treeable, TFriend, is defined so that a binary tree object can access Treeable::leftChild and Treeable::rightChild indirectly by inheriting from TFriend. The binary tree class cannot be a friend class of Treeable, because it is a template class.

```
class TFriend {                     // TFriend and its descendants have access to tree node
  protected:
    Treeable *&left( Treeable *tp ) const {
     return tp->leftChild;
    } // TFriend::left

    Treeable *&right( Treeable *tp ) const {
     return tp->rightChild;
    } // TFriend::right
}; // TFriend
```

5.2 BianryTree Administration

5.3 BianryTree File Structure

5.4 BianryTree Access Class

5.5 BianryTree Generator

Chapter 6

Parallelism

Supporting simultaneous access to a database can improve utilization of a database in the following ways:

Retrieval The slowest link in accessing a file structure is retrieving and storing data on secondary storage. Secondary storage ranges from 1,000 to 100,000 times slower than primary storage. Furthermore, there does not seem to be any immediate technological advancements that will significantly reduce this ratio in speed between primary and secondary storage; in fact, the difference has only been increasing over the last decade. Therefore, the only approach that is currently available is to partition the data onto multiple secondary storage devices and performing accesses to these devices in parallel. Disk arrays are the most common implementation of this idea [PGK88].

Accessors Supporting simultaneous access to a database can improve utilization of a database, in the same way that multiprogramming operating systems improve utilization of a computer—by having a number of requests to execute, it is possible to perform some of the requests in parallel if the requests access data in different areas of the database. There is no difference in the turnaround time of an individual request (in fact, there may be a slight increase in turnaround) in comparison to serial execution of the requests, but the throughput of requests is improved. However, there is a high cost in complexity that must be paid to ensure proper access to shared data. Problems such as livelock, deadlock, and starvation must all be dealt with, while attempting to achieve as much parallelism between the CPU(s) and the I/O device(s). Unfortunately, systems with multiple accessors can quickly saturate because of the I/O bottleneck in retrieval of data from the database.

Currently, μDatabase allows a file structure designer to build whatever form of concurrency is appropriate. Concurrency control can be specified at a low-level, where semaphores are used to protect data, or at a high-level, where light-weight server tasks control access to data. While concurrency often tied into a particular data structure, we believe it is possible to provide some general concurrency abstractions to the file structure designer to aid in this process. While a large number of concurrency techniques exist, they can be classified into two distinct forms of parallelism:

Backend concurrency deals with the I/O bottleneck, a file structure is partitioned across multiple disks and access is performed in parallel.

Frontend concurrency allow a number of requests to execute in parallel if the requests access data in different areas of the database.

Figure 6.1 shows these two forms of concurrency. Note that backend and frontend concurrency are mostly orthogonal aspects of concurrency; systems exist that provide one or the other or both. The question to be addressed is how to use memory mapping with both backend and frontend concurrency.

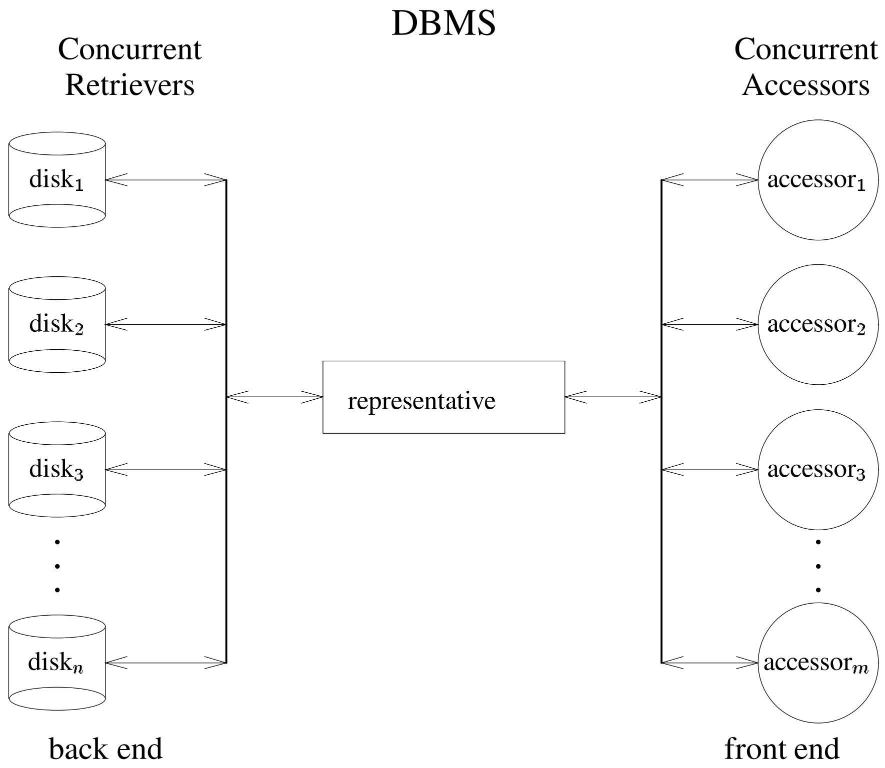

Figure 6.1: Two Forms of Concurrency in a File Structure

6.1 Backend Concurrency

Backend concurrency attempts to deal with the CPU-I/O bottleneck by partitioning data across multiple disks and then accessing the data in parallel [PGK88]. This bottleneck has been examined thoroughly using disk arrays [PGK88, WZS91]. A typical disk-array system partitions a file structure into several strips each stored on a different disk. Both static and dynamic allocation of file structures across several disks have been addressed in the literature. One of the major points is that the striping algorithm, called *partitioning*, should partition the data so that the access time for a particular file is minimized and the I/O load is balanced across the disks. The partitioning can be application specific or general. In partitioning, *balancing the I/O load* does not imply a physically even distribution of data across several disks. The goal is to distribute the data in such a manner that under a typical access request, the data units that need to be touched are as evenly distributed as possible across disks.

Once data has been partitioned, the issue of accessing it while employing as much parallelism as possible must be addressed. Exact match queries usually cannot take advantage of parallelism possible from partitioning because there is usually only one disk access to service the request. Range queries can take advantage of the parallelism possible from partitioning if the data is distributed so that portions of the range can be accessed in parallel. A range query may be broken down into a number of smaller range queries so that each can be executed in parallel. If the division of the query is done judiciously so that the respective domains of the individual smaller queries are on different disks, the overall query can be processed much faster than would have been possible otherwise. Similarly, if the file structure is aware of the access pattern of different blocks, it can employ pre-reading techniques to increase the parallelism in reading blocks of data from the disk. In general, the records returned from a range query are unordered. If records must be returned in a specific order, that can significantly reduce the amount of parallelism. In μDatabase, the generator types for each file structure can be written to manage all concurrent retrieval of records implicitly.

In the following discussion, the main concern is not about access to the index portion of the file structure. Normally the index is relatively small so that most of it remains resident in main memory, and consequently, does not play a significant role as far as disk accesses are concerned.

6.2 Frontend Concurrency

Here the concern is with allowing multiple client accessors to simultaneously traverse and manipulate the file structure. Currently, μC++ provides a number of language mechanisms for a file designer to build concurrency control. Many options will be built, tested and provided as part of μDatabase tool kit, however these will be used to build file-structure specific concurrency control. It is also our intention to study and develop a general purpose low-level concurrency control facility that will be automatically available to applications written in μDatabase. For example, allowing multiple versions of data to co-exist allows a high degree of concurrent and can be implemented in a general way.

Chapter 7

N-Tree Example

The following example illustrates advanced techniques and tools for constructing a persistent data structure by building a generic N-Tree file structure. An N-Tree is simply a generailzed B-Tree, where the nodes of the N-Tree index can have N pointers instead of just two.

7.1 N-Tree Application

A generic N-Tree file structure is presented to demonstrate the basic concept. The template facilities of C++ allow the creation of generic file structures (as in E [RCS93]). The generic B-Tree definition has 2 type parameters and 1 conventional parameter. The type parameters provide the type of the key and the type

```
class Record {                                    // data record
  public:
    float field1, field2;
    Record &operator=(const Record &rhs) {      // define assignment
        field1 = rhs.field1;
        field2 = rhs.field2;
        return( *this );  }
};
int greater( const int &op1, const int &op2 ) { // key ordering routine
    return op1 > op2;
}
void uMain::main() {                              // uMain uC++ artifact
    BTree<int, Record, 4 Kb> db( "testdb", greater, 30 Kb );
    BTreeAccess<int, Record, 4 Kb> dbacc( db ); // open B-Tree
    int key;
    Record rec, *recp;
    // insert records
    for ( key = 1; key <= 1000; key += 1 ) {
        rec.field1 = key / 10.0;
        rec.field2 = key / 100.0;
        dbacc.insert( key, &rec );  }             // static type-checking
    // retrieve records
    for ( BTreeGen<int, Record, 4 Kb> gen(dbacc); gen >> recp; ) {
    uCout << recp->field1 << " " << recp->field2 << endl;  }
}
```

Figure 7.1: Example Program using a Generic B-Tree

of the record for the B-Tree. The optional conventional parameter provides the size of the B-Tree nodes in bytes. Each B-Tree instance generated from a generic B-Tree type has 3 conventional parameters: the backing-store UNIX file name, the routine used to compare the keys and the initial space allocated for the B-Tree in bytes. The following creates two specialized B-Trees:

```
BTree<int, Record, 4 Kb> db1( "db1BTree", less ),  // default initial size
         db2( "db2BTree", greater, 30 Kb );          // 30 K initial size
```

Both B-Trees have int keys, Record records and a 4K node size. One instance is sorted in ascending order (less) and the other one in descending order (greater). Unfortunately, this B-Tree instantiation requires the UNIX file name and the name of the comparison routine be re-specified at each subsequent usage of the file structure, which is type unsafe. However, once these two aspects of a file structure are specified correctly, all subsequent access to the database file structure can be statically type-checked.

There are several requirements on the key type, the record type and the comparison routine. As well, some additional routines must be supplied. For example, the type of the key and the record must provide an assignment operator, among other things, and the comparison routine must have a specific type. A complete example showing the creation of a B-Tree and insertion and retrieval of records is presented in Figure 7.1.

In μDatabase, each file structure can provide range queries using a generator or iterator [RCS93, LAB$^+$81], e.g., BTreeGen. The *generator* is an object whose arguments define the kind of range query and it returns one record at a time from the set of records that satisfy the requirements denoted by the information provided to the generator. The operator >> returns a pointer to some record within the specified range, but successive records are not normally ordered. If all records in the range have been returned, the NULL pointer is returned. By iteratively invoking the operator >>, the individual records of the range query are obtained.

To define a file structure, e.g., BTreeFile, an abstract data-type is defined with two operations that are implicitly performed: initialization and termination; no other operations are available. A B-Tree is defined as follows:

```
class BTreeFile {
  public:
    BTreeFile( char *DiskFileName, ... ) { initialization code };
    ~BTreeFile( void ) { termination code };
};
```

The initialization routine BTreeFile and the termination routine ~BTreeFile are invoked automatically whenever an instance of BTreeFile is created and deleted, respectively. An instance of a B-Tree file structure is created using type BTreeFile, as in: BTreeFile f("StudentData", *other arguments*), where StudentData is the name of the UNIX file in which the data are stored and retrieved from. There are no user visible routines, which ensures that after the declaration of an instance of BTreeFile, the corresponding file structure is not accessible to the user/application program.

7.2 Access

For BTreeFile, the access object is called BTreeFileAcc.

```
class BTreeFileAcc {
  public:
    BTreeFileAcc( BTreeFile *f, char *access ) { initialization code };
    ~BTreeFileAcc() { termination code };
    read( ... ) { ... };
    ... { other appropriate access routines };
};
```

To gain read access to a file structure object f, an application program declares an instance of BTreeFileAcc, as follows: BTreeFileAcc pf(&f, "r"). The pointer to f specifies the file structure that is to be accessed through pf, and `"r"` specifies the kind of access for concurrency control purposes. Once instantiated, the access object can be used by an application to perform operations on the file structure by invoking the public member routines of BTreeFileAcc. For example, in order to read from f, a call is made to the member routine read of BTreeFileAcc, as in pf.read(...). The routine read communicates with the representative to perform the desired operation.

7.3 Generic B-Tree

The polymorphic facilities of a programming language can be applied to generalize the definitions of file structures and to allow reuse of the file structure's implementation by other file structures.

7.4 Nesting Heaps

With many applications, a segment has to be subdivided into multiple heaps that are managed independently from each other. The nodes of a B-Tree are examples of such heaps. Since the heaps are themselves pieces of storage that are usually allocated and released dynamically, it is logical to have a higher level memory manager that deal with these heaps. The segment then becomes an upper level heap with dynamically allocated subheaps nested inside.

In theory, there is no limit on the nestings of heaps, but the form of address for each level may depend on the storage management scheme at that level. In practice, there is a limit imposed by the number of bits in the address used to reference data in the lowest level heap. We believe three levels of sub-heaps should be sufficient for most practical problems. (See [BZ88] for a further discussion of expressing nesting.)

A heap may be accessed in two ways: by the file structure implementor and by a nested heap. For example, the storage management for a B-Tree has 3 levels: the segment, which is managed by the representative, within which uniform-size B-Tree nodes are allocated, within which uniform or variable sized records are allocated. Depending on the particular implementation of the storage manager at each level, different capabilities are provided. A file structure implementor makes calls to the lowest level (uniform or variable storage manager) to allocate records. A uniform or variable memory manager can then be created within the node. After that, the lower level memory manager for the node can be called to allocate data records in that node. Figure 7.2 illustrates this storage structure.

7.5 Nested Memory Manager Example

As discussed in Section 7.4, heaps managed by memory managers can be nested within each other. A B-Tree data structure is a good example where such nesting is useful. The file space is divided into uniform sized B-Tree nodes. A uniform memory manager is created to manage these nodes. And then a variable memory manager is created within each node to manage the variable sized B-Tree records contained within. (See Figure 7.2 on page 40.)

The administrative class for the B-Tree is defined in the same manner as the linked list structure in Section 4:

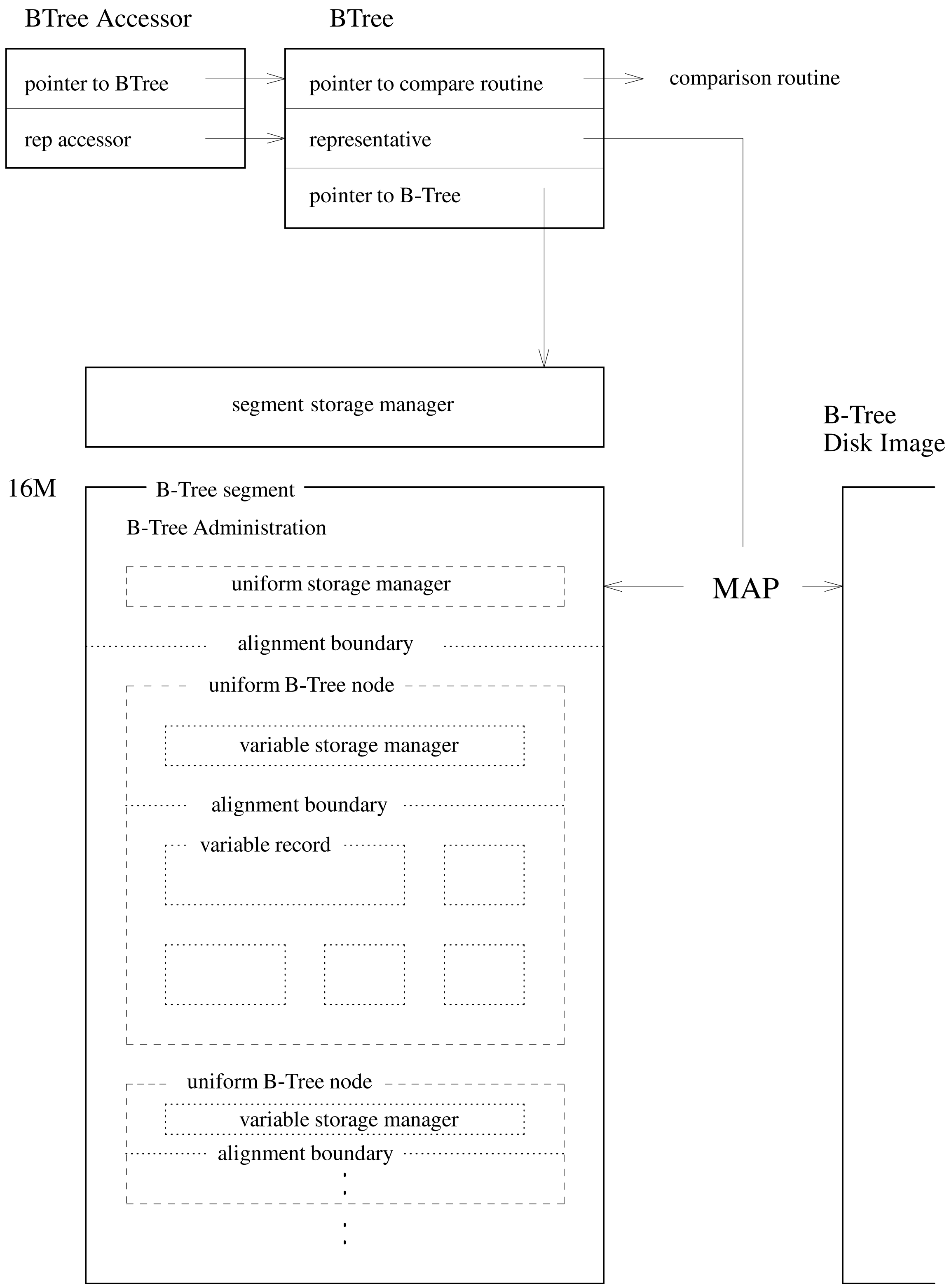

Figure 7.2: B-Tree Storage Structure

```
class NTreeAdmin {
  public:
    Rep *rep;                                   // initialized automatically
    ....                                        // at beginning of mapping
    void *Root;                                 // root node of the N-Tree
    NTreeExpType expobj;
    uniform<NTreeExpType> usm;

    NTreeAdmin( int FileSize, char *TypeName, int BlkSize );
    ....
}; // NTreeAdmin

NTreeAdmin::NTreeAdmin( int FileSize, char *TypeName, int BlkSize ) :
        expobj( *this ),
        usm( (void *)this + sizeof(NTreeAdmin),
            FileSize - sizeof(NTreeAdmin), expobj, BlkSize ) {
    Root = NULL;
} // NTreeAdmin::NTreeAdmin
```

The administrative class contains a uniform memory manager and an expansion object for the manager. The expansion class is defined as follows:

```
class NTreeExpType : public expand_obj {
    NTreeAdmin &admin;
  public:
    NTreeExpType( NTreeAdmin &adm ) : admin( adm ) {};
    int expand( int extension );
}; // NTreeExpType

int NTreeExpType::expand( int extension ) {
    admin.rep->resize( admin.rep->size() + extension );
    admin.usm.sethsize( admin.rep->size() - sizeof(NTreeAdmin) );
    return 1;              // retry allocation
} // NTreeExpType::expand
```

The expand object attempts to expand the size of the mapped file by calling the representative's resize routine, which is the typical action taken by the top level expansion object.

A B-Tree node can be used to hold B-Tree indices or data records. The former is called an index node the latter is called a leaf node. Both types keep their information within variable sized records managed by a variable memory manager. The leaf node class NTreeLeaf is shown below:

```
class NTreeLeaf {
    friend NTreeLeafExpType;
    NTreeLeafExpType expobj;
    variable<NTreeLeafExpType> vsm;
    . . .
    void MoveRecords( . . . );
    retcode SplitLeaf( . . . );
  public:
    NTreeLeaf();
}; // NTreeLeaf

NTreeLeaf::NTreeLeaf() : expobj(*this), vsm( (void *)this +
        sizeof(NTreeLeaf), NodeSize - sizeof(NTreeLeaf), expobj) {
    . . . .
} // NTreeLeaf::NTreeLeaf
```

And the expansion class for the memory manager vsm is shown below:

```
class NTreeLeafExpType : public expand_obj {
    NTreeLeaf &leaf;
    . . . .
    retcode rc;
    NTreeLeafExpType( NTreeLeaf &lf ) : leaf( lf ){}
  public:
    int expand( int );
}; // NTreeLeafExpType

int NTreeLeafExpType::expand(int) {
    rc = leaf.SplitLeaf( . . . . );
    return 0;          // done, give up allocation
} // NTreeLeafExpType
```

Because all B-Tree nodes are fixed size, a node cannot be enlarged when full. Instead, the member routine SplitLeaf within the NTreeLeaf class is called to split the node into two:

```
retcode NTreeLeaf::SplitLeaf( . . . ) {
    // create a new node
    NTreeLeaf *NewLeafPtr = new{SegZero->usm.alloc(NodeSize)} NTreeLeaf();
    // move some records out the current node and into the new node
    MoveRecords( . . . );
    . . .
    return 1;
} // NTreeLeaf::SplitLeaf
```

First, the SplitLeaf routine allocates a new node by calling the top level memory manager. Then, the tree is reorganized by moving some of the data records into the newly created empty node, thus making more space available in the current node.

7.6 Backend Concurrency Algorithm

Once a file structure is partitioned, a retrieval algorithm can take advantage of the potential parallelism, but only if sufficient hardware is available. First, the disks must be able to be accessed in parallel, which implies that there must be multiple disk controllers. Second, if multiple processors are available, they must be able to be used to perform any file-structure administration in parallel with the application processing the records from the range query.

The algorithm used for backend concurrency is as follows. For a file structure partitioned across N disks, the N disk files are memory mapped into one contiguous segment. Then M (a control variable) kernel threads (UNIX processes) are created that all share the data segment containing the mapped file. $N+1$ light-weight tasks are created to perform the retrieval requests and they execute on the M kernel threads. N of the tasks are retrievers and the $(N+1)^{th}$ task is the *leaf retrieval administrator* (LRA). For each generator created, a buffer is allocated by the generator, which is shared between the application and the file structure. As well, another task, the *file structure traverser*, is generated, which partitions the range query. The size of the buffer can be specified as an optional parameter when creating the generator. The default buffer size is 32K bytes. The traverser task assumes the responsibility of organizing the buffer space in the form of a sharable buffer pool in some suitable manner. Then the traverser task searches the index structure finding the leaf nodes that contain records in the range. For each leaf node, the traverser communicates with the LRA specifying the leaf, number of records in the leaf, and the buffer pool. The LRA farms out the generator requests to its retrieval tasks. A retrieval task accesses the specified leaf page, allocates a buffer from the buffer pool, and copies as many records as will fit from the leaf page to the buffer. The last step is repeated until all the records have been copied into buffers and then the retriever task gets more work from the LRA. The structure of this algorithm is illustrated in Figure 7.3. This structure ensures that the only bottleneck in the retrieval is the speed that the buffer can be filled or emptied. In general, an application program can keep ahead of a small number of disks (1-7 disks). This generic backend concurrency algorithm can be used for different file structures by specializing the file structure traverser and the component responsible for processing of individual leaves to extract information.

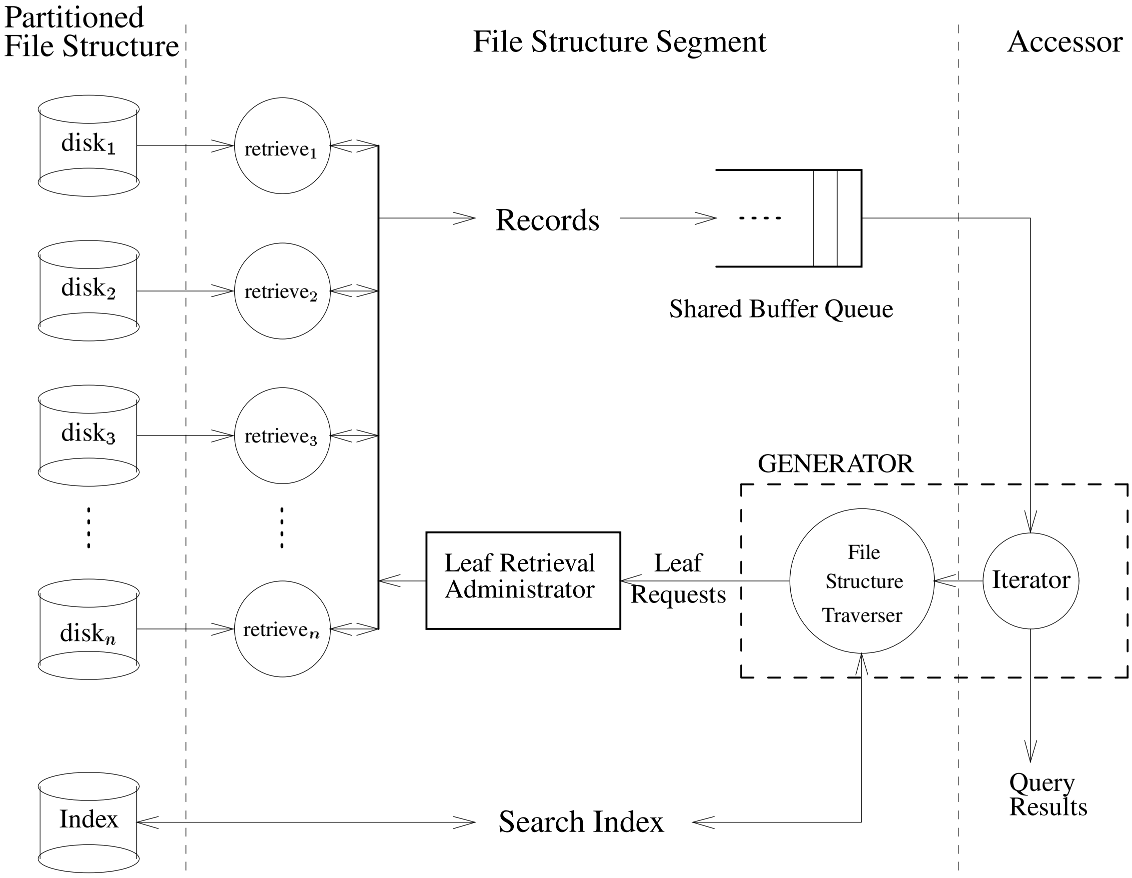

Figure 7.3: Backend Concurrency Structure

Chapter 8

Recovery

Implementing recovery is difficult in memory mapping and a satisfactory solution is still a research issue. If there is operating-system support to pin pages, traditional schemes can be used (however, with all the associated disadvantages). With no operating-system support, new techniques must be developed. We will be examining the use of dual memory maps to allow shadow write pages. One mapping represents the consistent database, which can be read at any time. The shadow mapping is for pages that are currently being modified. By precisely controlling when the shadow pages are copied back to the consistent mapping, it is possible to mimic traditional recovery schemes without operating-system support. The main problem to overcome is premature writing of modified pages by the operating system.

8.1 Experimental Analysis of General Storage Management

The criterion used to judge the general storage management approach is whether it can provide performance that is close to traditional schemes that amalgamate storage management directly with the data structure. Both an independent and integrated storage management B-Tree were constructed and creation tests were run. The results were virtually identical, with timings varying by $\pm$ 2%.

Chapter 9

Experimental Proof

A number of compelling arguments have been made in [CFW90] and other publications for the use of single-level stores for implementing databases. In spite of these arguments, it is clear there is still resistance and skepticism in the database community. Furthermore, our contention is that mapped files can be used advantageously for building databases not only in the new single-store environment but also in the traditional environment. Traditional databases can be accessed using memory-mapped access methods without requiring any changes to the file structure. In all cases, the mapped access methods should provide performance comparable to traditional approaches while making it much easier to augment the access methods of the file structure in the future by greatly reducing program complexity.

At the start of our work, there was little published experimental evidence available to support the view that memory-mapped file structures could perform as well as or better than traditional file structures. Therefore, it was necessary to implement a number of different memory-mapped file structures and to compare their performance against equivalent traditional ones.

9.1 Experimental Structure

To demonstrate the benefits of memory mapping, different experiments were constructed. The general form of an experiment was to implement a file structure in the traditional and memory-mapped styles, perform retrievals from a file, which is the most common form of access in a database, and compare the results. While every effort was made to keep the two file structures as similar as possible, some system problems precluded absolutely identical execution environments. In particular, the traditional file structures were stored on disk as character-special UNIX files and an LRU buffer manager was used. The memory-mapped files could not be mapped from a character-special file and had to be accessed from the UNIX file system, which performs all I/O in 8K blocks even though the system page size is 4K. Therefore, to make the comparisons equal, all of the file structures had 8K node sizes and all I/O was done in 8K blocks.

All experiments were run on a Sequent Symmetry with 10 i386 processors, which uses a simple page-replacement algorithm. The page-replacement algorithm is FIFO per page table plus a global LRU cache of replaced pages so there is a second chance to recover a page before it is reallocated. The maximum total size of the resident pages for a program is determined by the user and upon exceeding that size, pages are removed from the resident set on a FIFO basis. Upon removal, a page is put into the global cache where it can be reinstated to the resident set if a fault occurs for the page before the page is reused. This page replacement algorithm was matched against an LRU buffer-manager used by the traditional databases.

The execution environment was strictly controlled so that results between traditional and memory-mapped access methods were comparable. First, all experiments were run stand-alone to preclude external interference, except for those experiments that needed a loaded system. Second, the amount of memory for the experiment's address space and the global cache were tightly controlled so that both kinds of file structures had exactly the same amount of buffer space or virtual memory, respectively. The test files varied in size from 6-32 megabytes. The amount of primary storage available for buffer management or paging was restricted so that the ratio of primary to secondary storage was approximately 1:10 and 1:20. These ratios are believed to be common in the current generation of computers, supporting medium (0.1G-.5G) to large databases (1G-4G) but not very large databases.

The following experiments were implemented:

Prefix B$^+$Tree In this experiment, 100,000 uniformly distributed records were generated whose keys were taken from the unit interval. A record had a variable length with an average of 27 bytes. These records were inserted into a prefix B$^+$Tree [BU77]. For this B-Tree, 4 query files were generated, where the queries followed a uniform distribution. Each file is described by the tuple (n,m) where n is the number of queries and m is the number of records sequentially read from the B-Tree (range query). For example, (10,1000) means executing 10 queries with each query reading a set of 1000 records sequentially. A 5th query file contained 10,000 exact match queries that follow a normal distribution with mean 0.5 and variance 0.1.

R-Tree The R-Tree [Gut84] is an access method for multidimensional rectangles. It supports *point* queries and different types of *window* queries. A point query asks for all rectangles that cover a given query point whereas a window query asks for all rectangles which enclose, intersect or are contained in a given query rectangle. The window queries are similar to a range query in an ordinary B-Tree. However, there is one basic difference: index pages (internal nodes) are accessed more frequently in the case of the R-Tree than in the B-Tree case.

For this experiment, 2-dimensional data (100,000 rectangles) and queries from a standardized test bed [BKSS90] were taken. The maximum number of data rectangles was limited to 450 in the data pages, and to 455 in the directory pages. The query file consisted of 1000 point queries and 400 each of the three different types of window queries.

Graph To simulate the access patterns found in other data intensive applications (e.g., hypertext or object-oriented databases), a large directed graph was constructed consisting of 64,000 nodes, each of which was 512 bytes. The nodes were grouped into clusters of 64 where nodes in a cluster were physically localized. An edge out from a node had a high probability (85%, 90%, 95%) of referencing a node within the same cluster. Edges leaving a cluster went to a uniformly random selected node. Each experiment consisted of 40 concurrent random walks within the graph, consisting of 500 edge traversals each.

The results of the experiments are presented in Table 9.1. For each query file, three performance measures were gathered: the CPU time, the elapsed time, and number of pages or buffers read. The CPU time is the total time spend by all processors in a given test run, and hence, the CPU time may be greater than the elapsed time. Multiple processors were used in both traditional and memory mapped experiments. The retrieval application ran on one processor while the access method for the particular file structure ran on another processor. The elapsed time is the real clock time from the beginning to the end of the test run. Both times include any system overhead.

The results of the experiments confirm the conjecture that performance of memory-mapped file structures is equivalent or better than traditional file structures. For the read operations, the memory-mapped access methods are comparable ($\pm$ 10%) to their traditional counterparts. An exception occurs when the LRU buffer space is only 5% of the file size for sequential reads because the LRU algorithm is suboptimal in this case while the FIFO page-replacement algorithm is near optimal. For the CPU times, the memory-mapped access methods are generally better than the traditional ones because there is less time spent doing buffer management. For the elapsed times, the memory-mapped access methods are comparable ($\pm$ 10%)

Primary Memory Size 10% of Database Size

Access Method	Query Distr.	Memory Mapped CPU* Time (secs)	Memory Mapped Elapse Time (secs)	Memory Mapped Page Reads	Traditional CPU Time (secs)	Traditional Elapse Time (secs)	Traditional Disk Reads
Prefix B-Tree	1x10,000	35.7	19.7	61	32.2	32.9	53
	10x1,000	35.7	19.5	56	32.5	32.6	58
	100x100	37.5	22.4	147	35.4	35.7	150
	10,000x1	98.1	217.6	8789	240.5	223.6	8746
	normal	91.8	181.0	6777	202.3	183.6	6638
R-Tree	non-point	154.0	174.5	1414	330.4	334.1	1462
	point	109.4	124.1	934	230.5	234.4	896
Network Graph	85% local	318.1	476.1	15294	526.7	458.8	15004
	90% local	271.6	375.5	11278	449.0	370.7	11368
	95% local	207.0	243.8	6584	337.7	254.7	6539

Primary Memory Size 5% of Database Size

Access Method	Query Distr.	Memory Mapped CPU* Time (secs)	Memory Mapped Elapse Time (secs)	Memory Mapped Page Reads	Traditional CPU Time (secs)	Traditional Elapse Time (secs)	Traditional Disk Reads
Prefix B-Tree	1x10,000	35.5	19.5	61	35.3	35.5	117
	10x1,000	35.2	19.6	66	34.2	33.5	131
	100x100	37.0	22.1	155	37.4	36.6	216
	10,000x1	127.8	255.6	9415	260.7	224.1	9723
	normal	126.6	235.8	8250	253.8	217.6	9313
R-Tree	non-point	181.3	227.8	2913	367.1	374.5	3396
	point	136.8	184.5	2647	279.5	289.6	3491
Network Graph	85% local	383.3	565.8	17772	563.4	495.8	16550
	90% local	330.3	462.1	13602	484.0	403.9	12781
	95% local	264.9	316.6	8338	361.9	276.1	7400

CPU times may be greater than elapse time because multiple CPUs are used.

Table 9.1: Access Method Comparison : Node Size 8K

to their traditional counterparts. An exception occurs when memory-mapped access methods perform small sequential reads because the FIFO page-replacement algorithm is near optimal in this case. All of the results show that the Sequent page replacement scheme performed comparably to the LRU buffer-manager.

To verify the conjecture on the expected behavior of mapped access methods on a loaded machine, the previous B-Tree experiments were run during a peak-load period of 20-30 time-sharing users on the Sequent. The memory mapped and traditional B-Tree retrievals were started at the same time (3:00pm) and so were competing with each other as well as all other users on the system. The two file structures were on different disks accessed through different controllers so the OS could not share pages and retrievals were not interacting at the hardware I/O level. However, the amount of global cache could not be restricted during the day, so if there was free memory available, the memory-mapped access method would use it indirectly. Table 9.2 shows the averages of 5 trials. As can be seen, there was a difference only when there were a significant number of reads. In those cases, the memory-mapped access methods make use of any extra free memory to buffer data. This is particularly noticeable for the normal distribution because any extra memory significantly reduced the pages read, and hence, the elapse time. Clearly, the LRU buffer manager could be extended to dynamically increase and decrease buffer space depending on system load, but that further complicates the buffer manager and duplicates code in the operating system.

Primary Memory Size 10% of Database Size

		Memory Mapped			Traditional		
		CPU*	Elapse	Page	CPU	Elapse	Disk
Access Method	Query Distr.	Time (secs)	Time (secs)	Reads	Time (secs)	Time (secs)	Reads
Prefix B-Tree	1x10,000	35.8	21.6	60	34.0	35.7	53
	10x1,000	36.1	21.8	56	34.8	36.8	58
	100x100	37.4	25.24	143	37.0	38.68	150
	10,000x1	111.2	277.0	6677	263.4	263.3	8746
	normal	97.82	134.5	2063	221.5	217.0	6638

Table 9.2: Peak Load Retrievals : Node Size 8K

9.2 Experimental Analysis of Partitioned B-Tree

The machine used for these experiments was the same Sequent Symmetry with 8 disk drives, of which 4 were used. There were 2 disk controllers, each with 2 channels. The drives were equally divided between the controllers. The experiment was 1000 range queries with each query consisted of reading a random number of sequential records starting at a randomly selected initial key. The average query size was 2000 records. Two partitioned B-Trees were tested, one created using a round-robin partitioning (each block is created on the next disk) and one created using the Larson-Seeger algorithm [SL91]. The partitioned experiments were performed with 1–4 partitions and the application program received each record but did no processing on the record. The results of the experiments appear in the graphs of Figure 9.1. The largest decrease in elapsed time is from 1 to 2 partitions because there are 2 controllers. After that, the elapse time increases because of contention on the two controllers.

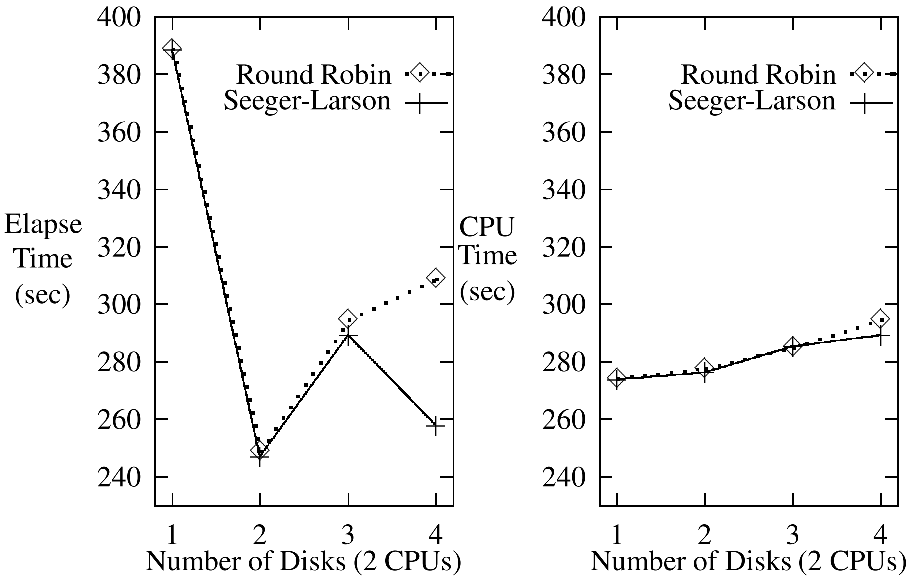

Figure 9.1: Backend Concurrency with B-Trees

Chapter 10

Related Work

10.1 Related Models

10.1.1 Pointer Swizzling

10.1.2 Reachability

Many current systems (e.g. Cedar, Lisp, Smalltalk, PS-Algol, Napier) use reachability as the fundamental mechanism that determines the persistence of data: a data item persists as long as some active data item refers to it, directly or indirectly. Conceptually, this mechanism can be applied as easily locally, to determine persistence of data items within a single program or process, as globally, to determine persistence of data that is independent of programs, such as conventional files and databases. This permits special programming language constructs, such as files, databases, directories, names spaces, etc., to be replaced by simpler arrays or linked-list structures.

Basing persistence on reachability, however, begs several crucial questions.

- Does reachability have a sensible definition in all circumstances?
- Can the fundamental property of reachability, namely, the prevention of dangling pointers, be guaranteed when systems are not necessarily perfect?
- Is it desirable to have all data persist until no references to it exist?

It is unclear to us what the effect of temporarily inaccessible nodes on a distributed system has on reachability and the programs that manipulate distributed data structures. If a node is inaccessible, then how can data structures stored in it be used in order to pursue reachable data structures, particularly when data structures on other nodes are reachable only through the inaccessible node. How long does a node have to be inaccessible before it is considered to have been removed from the network, thereby permitting data reachable through it to be reclaimed? What about nodes, such as portable workstations, that can be detached from the network and reattached at an arbitrary location and time.

We do not believe that any system can guarantee no dangling references over its lifetime. System failure, data corruption, and other problems will make ensuring this virtually impossible. Therefore, rather than spend an enormous effort in an attempt to achieve what we believe is not achievable, we suggest a conservative approach of assuming that dangling references can occur but guaranteeing that a dangling reference is detectable, at least at the time it is dereferenced. While ensuring that dangling references are detectable is still non-trivial and has a runtime cost, it is implementable. This is the philosophy taken in many network communication protocols concerning message delivery. It is more realistic for message senders and receivers to be ready to deal with a missing or corrupted message packet, then to design a protocol that guarantees all packets are delivered correctly under all possible circumstances.

Since the system assumes the existence of dangling pointers and checks for them, it is possible to take advantage of this and provide an explicit deletion operation. This would allow a user to explicitly delete data in the presences of outstanding references to it. This capability could be used to remove private data that has “leaked out”, or as a “crisis management” mechanism, for instance, as a way to recover space on an over-used node of a distributed system.

Underlying the notion of reachability is the concept of a single persistent storage space, the **ether**, that encompasses all the real storage devices (disks, etc.) on the system. A data item in the ether is conceptually always accessible; in other words, when a pointer to it is dereferenced it must be made accessible if it is not already. This can impose a very substantial burden on the system because the individual data items may be made accessible independently of one another. Data in the ether is usually made accessible on reference by copying it to a volatile memory where it is more easily accessed by the processor (i.e. reading data in from disk to real memory). If data is shared by several processors on a network, it may be quite difficult either to ensure that only a single copy of each data item is kept in volatile memory or to ensure that multiple copies are kept consistent with one another. Solving these and related problems may significantly reduce the performance of the system compared with one that uses more conventional means to deal with persistent data.

Another problem is in compacting the ether. It makes little sense to compact the ether onto one or more consecutive machine memories regardless of whether the nodes are distributed or not. This means that the ether must be subdivided according to the actual hardware memories in order to perform the compaction. If the ether is subdivided in this way, it is not really an ether but multiple memories with a common addressing scheme. It should be possible to generalize this dividing into user-definable logical memories, as described in [BZ89].

Having pointers maintained by the system essentially precludes having programmers write specialized storage management schemes, one of our objectives mentioned above. The assumption is that the system will be able to do a good (or, at least, adequate) job of storage management for all data structures.

10.2 Related Systems

As mentioned earlier, the earliest use of memory mapping techniques (or a single-level store) can be found in the Multics system [BCD72]; however, earlier operating systems were not flexible enough to allow exploitation of the memory techniques in a serious manner. In recent times, with the development of more open systems, a number of efforts have been made to use memory mapping. The following discussion covers the recent work on memory mapping. μDatabase is most closely related to the first four systems described below. All of these systems have some common features with μDatabase; nevertheless, there are significant differences that make μDatabase novel. In some cases, the differences are largely to do with the way the overall system is constructed. The important thing to note is that most of these systems are being developed independently and in parallel and have been commissioned only in the last few years. There are few measures yet to judge memory mapped systems by, and hence, all the different approaches have to be considered as viable.

10.2.1 The Objectstore Database System

The Objectstore Database System [Atw90, LLOW91], developed by Object Design Inc, includes work on memory mapping that is most closely related to μDatabase. μDatabase shares a number of goals and objectives with Objectstore. Some of these include ease of learning, no translation between the disk-resident representation of data and the the main memory-resident representation used during execution, full expressive power of a general purpose programming language when accessing persistent data, reusability of code and virtually all access to data is statically type-checked. μDatabase has progressed independently of Objectstore and over a largely overlapping period of time.

Objectstore differs significantly from μDatabase in how the goals and objectives are achieved. In both systems, normal programming language pointers (in virtual space) are used to refer to persistent objects as well as to transient ones. Objectstore does not restrict the maximum size of a single file structure by mapping portions of it; only the currently accessed pages used by a given transaction are mapped into the address space of the application. This approach introduces a limit on the number of different data pages that can be used simultaneously by any single transaction. An operation large enough to reach this maximum limit has to be broken down into a series of smaller transactions. In μDatabase, an entire file structure is mapped into an individual segment. Currently, this approach limits the size of any single file structure to be less than the virtual space supported by the available hardware; large file structures have to be split into smaller ones. There is, however, no restriction on how much data a single transaction can access simultaneously. Admittedly, the restriction imposed in Objectstore may be less severe than the one imposed in μDatabase, especially with 32-bit addresses.

The approach used in Objectstore results in an inferior solution to the problem of accessing multiple file structures from an application. Objectstore maps pages of all the databases used in an application into the same address space. Each page to be used is dynamically allocated a virtual address where it is mapped. This means that when a page is mapped into virtual memory, the correspondence of objects and virtual addresses may have changed. For the pointers stored in the page to be valid, they must be *relocated* to reflect the new virtual address of the object. This requires some portion of the type system to be available at runtime, which is used to locate all pointers stored in a page. Also, the need to relocate pointers has the potential of degrading performance of the database. This problem is non-existent in μDatabase because each database is mapped into a different virtual space and so no relocation of pointers is necessary.

However, in μDatabase additional copying of data may have to occur from the file structure segment to the application segment. Some of this copying is unavoidable in any mapped system, including Objectstore. Overall, we believe that this cost will be less than the total cost of providing relocation and, in general, is required to protect the integrity of the file structure.

10.2.2 Cricket: A Mapped, Persistent Object Store

Cricket is a database storage system that uses the memory management primitives of the Mach operating system to provide the abstraction of a "shared, transactional single-level store that can be directly accessed by user applications" [SZ90a, p. 89]. Cricket follows a client/server paradigm and, upon an explicit request, maps the database directly into the virtual space of the client application. Cricket is similar to μDatabase in that direct memory pointers are used and the database is mapped to the same range of virtual addresses so that relocation of addresses is not necessary. The fundamental difference from μDatabase is that the mapping takes place in the address space of the application, and hence, only one database at a time can be used by an application. Indeed, the concept of a disk file to group related objects in one collection is not a basic entity in Cricket and it takes the view that everything that an application needs to use is placed in a single large persistent store. The designers of Cricket do acknowledge the need to support files (i.e., multiple collection areas for objects) and plan on providing an implementation for them. It is our contention, though, that it will be almost impossible to support a truly general implementation of files within the framework of Cricket's architecture. We feel that this will lead to a certain amount of awkwardness in organizing various components of data and in sharing pieces of data across different projects. More importantly, this approach will not be able to handle partitioning adequately. μDatabase, on the other hand, builds on top of the concept of files to provide multiple, individually sharable collections areas.

10.2.3 Paul Wilson's work

In [Wil91], Paul Wilson describes a scheme that uses pointer swizzling at page fault time to support huge address spaces with existing virtual memory hardware. The basic scheme is very similar to the one employed by the Objectstore system except that in Wilson's scheme pointers on secondary store can have a format different from the pointers in primary storage. This allows for the maintenance of a persistent store that is much larger than the virtual space supported the hardware. Wilson's scheme requires a special page fault handler that is responsible for translating (swizzling) of persistent pointers into transient pointers. When a page fault occurs, all the persistent pointers in the page have to be located and translated, which requires runtime type information. Since some of the pointers in a page can refer to pages that have not yet been made available, the translation of these pointers requires that all the referent pages be *faulted* as well. To prevent a cascade of I/O operations, Wilson's scheme only reserves the addresses for these extra pages in the page table instead of actually mapping them to primary storage. However, this solution underutilizes the address space and an application can potentially run out of addresses. Wilson suggests periodically invalidating all the mappings and rebuilding them to deal with this problem. Furthermore, objects that cross page boundaries require additional language support. Wilson's scheme is a clear winner for applications that require extremely large persistent address spaces using existing virtual memory hardware. However, the scheme is complex and may result in significant overhead, especially for applications with poor locality of references. Finally, Wilson's approach has the same problems as Objectstore with regard to dynamic relocation and multiple accessible databases.

10.2.4 The Bubba database system

The designers of Bubba [BAC+90, CFW90], a highly parallel database system developed at Microelectronics and Computer Technology Corporation (MCC), exploited the concept of a single-level store to represent objects uniformly in a large virtual address space. Cricket borrowed a number of ideas from Bubba. The focus of Bubba was on developing a scalable *shared-nothing* architecture which could scale up to thousands of hardware nodes and the implementation of a single-level store was only a small, though important, portion of the overall project. The current design of μDatabase is based on a multiprocessor shared-memory architecture and is not intended to be used in a distributed environment. In Bubba, the Flex/32 version of AT&T UNIX System V Release 2.2 was extensively modified to build a single-level store, which makes their store highly unportable. μDatabase runs on any UNIX system that supports the `mmap` system call. Finally, the programming interface to Bubba is FAD, a parallel database programming language.

The Bubba database system The designers of Bubba [BAC+90, CFW90], a highly parallel database system developed at Microelectronics and Computer Technology Corporation (MCC), exploited the concept of a single-level store to represent objects uniformly in a large virtual address space. Cricket borrowed a number of ideas from Bubba. The focus of Bubba was on developing a scalable *shared-nothing* architecture which could scale up to thousands of hardware nodes and the implementation of a single-level store was only a small, though important, portion of the overall project. The current design of μDatabase is based on a multiprocessor shared-memory architecture and is not intended to be used in a distributed environment. In Bubba, the Flex/32 version of AT&T UNIX System V Release 2.2 was extensively modified to build a single-level store, which makes their store highly unportable. Finally, the programming interface to Bubba is FAD, a parallel database programming language.

10.2.5 Others

The following are other known efforts at exploiting mapped files that are quite different from this work and are not discussed here for lack of space. The Camelot Distributed Transaction System [STP+87], IBM's 801 prototype hardware architecture [CM88], The Clouds Distributed Operating System [DLA87, PP88], [Peter van Oosterom [vO90], The Hurricane Operating System [SUK92].

10.3 Conclusion

We have shown that memory mapping is an attractive alternative for implementing file structures for databases. Memory-mapped file structures are simpler to code, debug and maintain, while giving comparable performance when used stand-alone or on a loaded system than for traditional databases. Further, buffer management supplied through the page-replacement scheme of the operating system seems to provide excellent performance for many different access patterns. Our design for structuring the low-level portions of a DBMS for memory mapping provides the necessary environment to implement concurrency control and recovery. Finally, these benefits can be made available in tool kit form on any UNIX system that supports the `mmap` system call. Currently, μDatabase is only missing recovery facilities and these will be added in the near future.

Chapter 11

Miscellaneous

11.1 Contributors

While many people have made numerous suggestions, the following people were instrumental in turning this project from an idea into reality. Bob Zarnke and Peter Buhr kicked around many of the initial ideas in [Buh85]. Peter Buhr and Anil Goel firmed up the ideas and started building proof-of-concept experiments to see what could be implemented [BGW92]. The results of the experiments began to be integrated into a coherent package, which became μDatabase. Andy Wai wrote the first version of the heap memory-management tools [Wai92]. David Clarke helped with some of the early experiments and always kept us honest. Paul Larson and Berhard Seeger helped by vetting experimental results and in initially doubting that it would work at all.

Bibliography

[ABC+83] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, P. W. Cockshott, and R. Morrison. An Approach to Persistent Programming. *The Computer Journal*, 26(4):360–365, November 1983.

[Atw90] Thomas Atwood. Two Approaches to Adding Persistence to C++. In A. Dearle et al, editor, *Implementing Persistent Object Bases: Principles and Practise,* Proceedings of the Fourth International Workshop on Persistent Object Systems, pages 369–383. Morgan Kaufmann, 1990.

[BAC+90] H. Boral, W. Alexender, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez. Prototying Bubba, A Highly Parallel Database System. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):4–24, March 1990.

[BCD72] A. Bensoussan, C. T. Clingen, and R. C. Daley. The Multics Virtual Memory: Concepts and Design. *Communications of the ACM*, 15(5):308–318, May 1972.

[BDS+92] P. A. Buhr, Glen Ditchfield, R. A. Stroobosscher, B. M. Younger, and C. R. Zarnke. μC++: Concurrency in the Object-Oriented Language C++. *Software—Practice and Experience*, 22(2):137–172, February 1992.

[BGW92] Peter A. Buhr, Anil K. Goel, and Anderson Wai. μDatabase : A Toolkit for Constructing Memory Mapped Databases. In Antonio Albano and Ron Morrison, editors, *Persistent Object Systems*, pages 166–185, San Miniato, Italy, September 1992. Springer-Verlag. Workshops in Computing, Ed. by Professor C. J. van Rijsbergen, QA76.9.D3I59.

[BKSS90] N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *ACM SIGMOD*, pages 322–331, 1990.

[BU77] Rudolf Bayer and Karl Unterauer. Prefix B-Trees. *ACM Transactions on Database Systems*, 2(1):11–26, March 1977.

[Buh85] P. A. Buhr. *A Programming System*. PhD thesis, University of Manitoba, 1985.

[BZ86] P. A. Buhr and C. R. Zarnke. A Design for Integration of Files into a Strongly Typed Programming Language. In *Proceedings IEEE Computer Society 1986 International Conference on Computer Languages*, pages 190–200, Miami, Florida, U.S.A, October 1986.

[BZ88] P. A. Buhr and C. R. Zarnke. Nesting in an Object Oriented Language is NOT for the Birds. In S. Gjessing and K. Nygaard, editors, *Proceedings of the European Conference on Object Oriented Programming*, volume 322, pages 128–145, Oslo, Norway, August 1988. Springer-Verlag. Lecture Notes in Computer Science, Ed. by G. Goos and J. Hartmanis.

[BZ89] P. A. Buhr and C. R. Zarnke. Addressing in a Persistent Environment. In John Rosenburg and David Koch, editors, *Persistent Object Systems*, pages 200–217, Newcastle, New South Wales, Australia, January 1989. Springer-Verlag. Workshops in Computing, Ed. by Professor C. J. van Rijsbergen, QA76.64.I57.

[CAC+84] W. P. Cockshott, M. P. Atkinson, K. J. Chisholm, P. J. Bailey, and R. Morrison. Persistent Object Management System. *Software – Practice and Experience*, 14(1):49–71, 1984.

[CD85] Hong-Tai Chou and David J. DeWitt. An Evaluation of Buffer Management Strategies for Relational Database Systems. In A. Pirotte and Y. Vassiliou, editors, *Proceedings of the 11th International Conference on Very Large Data Bases*, pages 127–141, Stockholm, August 1985.

[CFW90] George Copeland, Michael Franklin, and Gerhard Weikum. Uniform Object Management. In *Advances in Database Technology – Proc. European Conference on Database Technology*, pages 253–268, Venice, Italy, March 1990.

[CM88] A. Chang and M. Mergen. 801 Storage: Architecture and Programming. *ACM Transactions on Computer Systems*, 6(1):28–50, January 1988.

[DLA87] P. Dasgupta, R. J. LeBlanc, Jr., and W. F. Appelbe. The Clouds Distributed Operating System: Functional Descriptions, Implementation Details and Related Work. Technical Report GIT-ICS-87/42, School of Information and Computer Science, Georgia Institute of Technology, 1987.

[Gut84] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *ACM SIGMOD*, pages 47–57, 1984.

[IBM78] *System/38 Services Overview*. IBM, 1978.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall Software Series. Prentice Hall, second edition, 1988.

[LAB+81] Barbara Liskov, Russell Atkinson, Toby Bloom, Eliot Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder. *CLU Reference Manual*, volume 114 of *Lecture Notes in Computer Science*. Springer-Verlag, 1981.

[LLOW91] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The Objectstore Database System. *Communications of the ACM*, 34(10):50–63, October 1991.

[MBC+89] R. Morrison, A. Brown, R. Carrick, R. Connor, A. Dearle, and M. P. Atkinson. The Napier Type System. In John Rosenberg and David Koch, editors, *Persistent Object Systems*, pages 3–18, University of Newcastle, New South Wales, Australia, January 1989. Springer-Verlag. Workshops in Computing, Ed. by Professor C. J. van Rijsbergen, QA76.64.I57.

[Mip91] *MIPS R4000 Microprocessor User's Manual*. MIPS Computer Systems Inc, 1991.

[Mos90] J. Moss. Working with Persistent Objects: To Swizzle or Not to Swizzle. Technical Report CS 90-38, CS Department, University of Massachusetts, May 1990.

[Org72] E. I. Organick. *The Multics System*. The MIT Press, Cambridge, Massachusetts, 1972.

[PGK88] D. A. Patterson, G. Gibson, and R. H. Katz. A Case for Redundant Arrays of Inexpensive Disks(RAID). In *ACM SIGMOD*, pages 109–116, June 1988.

[PP88] D. V. Pitts and Dasgupta P. Object Memory and Storage Management in the *Clouds* Kernel. *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 10–17, June 1988.

[PS-87] The PS-Algol Reference Manual, 4th Ed. Technical Report PPRR 12, University of Glasgow and St. Andrews, Scotland, June 1987.

[RCS93] Joel E. Richardson, Michael J. Carey, and Daniel T. Schuh. The Design of the E Programming Language. *ACM Transactions on Programming Languages and Systems*, 15(3):494–534, July 1993.

[RKA92] J. Rosenberg, J. L. Keedy, and D. A. Abramson. Addressing Mechanisms for Large Virtual Memories. *The Computer Journal*, 35(4):369–375, August 1992.

[RM89] K. Rothermel and C. Mohan. ARIES/NT: A Recovery Method Based on Write-Ahead Logging for Nested Transactions. In *Proceedings of the 15th International Conference on Very Large Data Bases*, pages 337–346, Palo Alto, Ca, August 1989. Morgan Kaufmann Publishers Inc.

[SL91] Bernhard Seeger and Per-Ake Larson. Multi-Disk B-trees. In *ACM SIGMOD*, pages 436–445, Denver, Colorado, USA, June 1991.

[Smi85] A. J. Smith. Disk Cache – Miss Ratio Analysis and Design Consideration. *ACM Transactions on Computer Systems*, 3(3):161–203, August 1985.

[STP+87] Alfred Z. Spector, D. Thompson, R. F. Pausch, J. L. Eppinger, D. Duchamp, R. Draves, D. S. Daniels, and J. L. Bloch. Camelot: A Distributed Transaction Facility for Mach and the Internet - An Interim Report. Technical Report CMU-CS-87-129, Carnegie Mellon University, 1987.

[SUK92] M. Stumm, R. Unrau, and O. Krieger. Designing a Scalable Operating System for Shared Memory Multiprocessors. *USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 285–303, April 1992.

[Sun90] *System Services Overview*. Sun Microsystems, 1990.

[SZ90a] Eugene Shekita and Michael Zwilling. Cricket: A Mapped, Persistent Object Store. In A. Dearle et al., editors, *Implementing Persistent Object Bases: Principles and Practise,* Proceedings of the Fourth International Workshop on Persistent Object Systems", pages 89–102. Morgan Kaufmann, 1990.

[SZ90b] M. Stumm and S. Zhou. Algorithms Implementing Distributed Shared Memory. *IEEE Computer*, 23(5):54–64, May 1990.

[TRY+87] A. Tevanian, Jr., R. F. Rashid, M. W. Young, D. B. Golub, M. R. Thompson, W. Bolosky, and R. Sanzi. A Unix Interface for Shared Memory and Memory Mapped Files Under Mach. In *Proceedings of the Summer 1987 USENIX Conference*, pages 53–67, Phoenix, Arizona, June 1987. USENIX Association.

[vO90] Peter van Oosterom. *Reactive Data Structures for Geographic Information Systems*. Ph.D. Thesis, Dept. of CS, Leiden University, December 1990.

[Wai92] Anderson Wai. Storage Management Support for Memory Mapping. Master's thesis, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, 1992.

[WF90] K.L. Wu and W.K. Fuchs. Recoverable Distributed Shared Virtual Memory. *IEEE Transactions on Computers*, 39(4):460–469, April 1990.

[Wil91] Paul R. Wilson. Pointer Swizzling at Page Fault Time: Efficiently Supporting Huge Address Spaces on Standard Hardware. *Computer Architecture News*, 19(4):6–13, June 1991.

[WZS91] Gerhard Weikum, Peter Zabbak, and Peter Scheuermann. Dynamic File Allocation in Disk Arrays. In *ACM SIGMOD*, pages 406–415, Denver, Colorado, USA, June 1991.

Index