Exact Positioning of Data Approach to Memory Mapped Persistent Stores: Design, Analysis and Modelling

by

Anil K. Goel

A thesis

presented to the University of Waterloo

in fulfilment of the

thesis requirement for the degree of

Doctor of Philosophy

in

Computer Science

Waterloo, Ontario, Canada, 1996

Abstract

One of the primary functions of computers is to store information, i.e., to deal with long lived or *persistent* data. Programmers working with persistent data structures are faced with the problem that there are two, mostly incompatible, views of structured data, namely data in primary and secondary storage. Traditionally, these two views of data have been dealt with independently by researchers in the programming language and database communities.

Significant research has occurred over the last decade on efficient and easy-to-use methods for manipulating persistent data structures in a fashion that makes the secondary storage transparent to the programmer. Merging primary and secondary storage in this manner produces a *single-level store*, which gives the illusion that data on secondary storage is accessible in the same way as data in primary storage. In complex design environments, a single-level store offers substantial performance advantages over conventional file or database access. These advantages are crucial to unconventional database applications such as computer-aided design, text management, and geographical information systems. In addition, a single-level store reduces complexity in a program by freeing the programmer from the responsibility of dealing with two views of data.

This dissertation proposes, develops and investigates a novel approach for implementing single-level stores using *memory mapping*. Memory mapping is the use of virtual memory to map data stored on secondary storage into primary storage so that the data is directly accessible by the processor's instructions. In this environment, all transfer of data to and from the secondary store takes place implicitly during program execution. The methodology was motivated by the significant simplification in expressing complex data structures offered by the technique of memory mapping. This work parallels other proposals that exploit the potential of memory mapping, but develops a unique approach based on the ideas of segmentation and exact positioning of data in memory. Rigorous experimentation has been conducted to demonstrate the effectiveness and ease of use of the proposed methodology vis-a-vis the traditional approaches of manipulating structured data on secondary storage.

The behaviour of high-level database algorithms in the proposed memory mapped environment, especially in highly parallel systems, has been investigated. A quantitative analytical model of computation in this environment has been designed and validated through experiments conducted on several database join algorithms; parallel multi-disk versions of the traditional join algorithms were developed for this purpose. An analytical model of the system is extremely useful for data structure and algorithm designers for predicting general performance behaviour without having to construct and test specific algorithms. More importantly, a quantitative model is an essential tool for database subsystems such as a query optimizer.

Acknowledgements

The production of this dissertation would have been so much harder, if not impossible, without the guidance, encouragement, mentoring, unbounded patience, hard work, friendship and generosity of my supervisor Dr. Peter Buhr. I thank Peter for everything he has done for me since my arrival at Waterloo.

I am grateful to Dr. Prabhakar Ragde and Dr. Naomi Nishimura for co-supervising the theoretical aspects of this dissertation. Their contribution was instrumental in making this dissertation into a reality.

I thank my wife Aparna, daughter Ankita and son Anirudh, who have all contributed to the dissertation in more ways than one. They provided the flicker of light during the darkest hours and ensured that I did not get lost somewhere along the way.

I thank my external examiner, Dr. John Rosenberg of the University of Sydney, Australia and the other members of my examining committee, Dr. Paul Larson, Dr. Frank Tompa and Dr. Bruno Preiss for their valuable suggestions.

Dr. Paul Larson, Dr. Bernhard Seeger, Andy Wai and David Clark provided valuable help during the early experiments. Dr. Ian Munro was always willing to discuss matters of theory. Several other people, in particular, N. Asokan, Gopi Attaluri, Lauri Brown and Glenn Paulley, offered suggestions and encouragement that was instrumental in preserving my senility and I am thankful to all of them.

I am thankful to the Math Faculty Computing Facility at the University of Waterloo for maintaining a top notch computing environment and for employing me during the last three years of my stay at UW. MFCF and the wonderful people that work there certainly made my life much easier. In particular, I thank Bill Ince for never saying no.

Finally, I thank all the people I have interacted with at Waterloo for making my stay here most enjoyable.

To Aparna

Contents

List of Tables

List of Figures

List of Programs

Chapter 1

Introduction

Researchers and programmers working with complex and possibly large persistent data structures have traditionally dealt with two different views of data, viz., the data stored on secondary storage (e.g., a disk) and the structured data in primary storage as seen by the processor's instructions. These two views of data are largely incompatible with each other. In primary storage, physical or virtual memory pointers are used to construct complex relationships among data; establishing these relationships without memory pointers is often cumbersome and expensive. On the other hand, data on secondary storage is organized without the use of memory pointers. The traditional approach of maintaining and manipulating these two disparate views of essentially the same data has resulted in a dichotomy that is quite artificial – researchers in the programming language community have dealt primarily with the primary storage view of data, while the database researchers have concerned themselves with the handling of data on secondary storage. In addition, when dealing with secondary storage data, the programming language community has tended to use tools, such as file systems, made available by the operating system designers whereas the database community has designed and used its own alternative tools. The dichotomy has meant, among other things, that the programming language, database and operating system communities have spent significant effort duplicating each others' work, albeit in separate environments and with different immediate goals. A prime example of this replication of effort is the page replacement strategies developed by the operating system designers as compared to the extensive buffer management strategies developed by their counterparts in the database community; the two strategies are often in conflict with, rather than enhancing, one another. An additional consequence of maintaining the two views of data has been development of applications that spend significant amounts of execution time converting data back and forth from one view to the other. To be fair, it needs to be pointed out that the mentioned dichotomy was borne out of, and sustained by, a lack of essential architectural tools being available at the user level. However, during recent years many of these historical deficiencies have been removed or made less restrictive at the hardware and operating system levels. This development, in conjunction with an increased appreciation of the benefits provided by a merging of the two views of data, especially for complex emerging database applications, has resulted in a significant increase in collaboration among the programming language, database and operating system communities.

1.1 The Single-Level Store

Significant research has occurred over the last decade, starting with the seminal work by Atkinson, *et al* [ABC$^+$83, AM85], on efficient and easy-to-use methodologies for constructing, storing, and subsequently retrieving and manipulating persistent data in a fashion that makes the secondary storage transparent to the programmer. This research extends primary storage practices and tools so that they also apply to secondary storage. Merging primary and secondary storage in this way produces a *single-level store*, which gives the illusion that data on secondary storage is accessible in the same way as data in primary storage. This uniform view of data eliminates the need for complex and expensive execution-time conversions of structured data between primary and secondary storage and allows the use of the expressive power and the data structuring capabilities of a general purpose programming language for creating and manipulating data on secondary storage, which is analogous to the goals of virtual memory. Although a single-level store was investigated as far back as the Multics system (1968) [Org72], it has seen only limited use, even in the field of operating systems. Only in the last few years has the use of single-level stores blossomed in both the database and programming language communities [CFW90, SZ90a, LLOW91, RCS93]. In complex design environments, a single-level store offers substantial performance and programming advantages over conventional file or database access. These advantages are crucial to complex database applications such as computer-aided design, text management, and geographical information systems.

While there are several ways to implement a single-level store, some projects do so using memory mapping. Memory mapping is the use of virtual memory to map data stored on secondary storage into primary storage so that the data is directly accessible by the processor's instructions. In this environment, there are no explicit read and write routine calls to access data on secondary storage. All I/O operations are done implicitly by the operating system during execution of a program when pointers are calculated and dereferenced. Hence, data structures related by pointers (e.g., a linked list) can be stored onto secondary storage and still be manipulated directly via pointers. When the working set of a database program can be kept entirely in memory, performance begins to approach that of memory-resident databases.

While there are few disadvantages in using memory mapping, it is still uncommon to see it used for accessing secondary storage in traditional file and database systems. One explanation is a lack of general virtual memory hardware on many computers and limited access to memory mapping capabilities by older operating systems. Stonebraker concluded [Sto81] that the DBMSs made little or no use of services offered by the operating system because these services were either inefficient or inappropriate. However, with today's large virtual address spaces (32-64 bits and more) and powerful memory management co-processors, memory mapping of secondary storage makes excellent sense, and operating systems are beginning to provide access to this capability, e.g., the mmap system call in UNIX and more general access to virtual memory in the Mach [TRY$^+$87] and SunOS [Sun90] operating systems.

1.2 Exact Positioning of Data Approach to Memory Mapping

All single-level stores support addresses in some form to relate data, and these addresses directly reference the data. The particular addressing mechanism used is central to the design and performance of each persistent storage system; Cockshott [Coc85] gives a general overview of a number of possible addressing schemes. Fundamental to all persistent storage systems is the following addressing problem. When data is copied from secondary to primary (or virtual) storage, either the data must be positioned exactly where it was originally created to maintain integrity of embedded pointers, or the embedded pointers must be modified to reflect the new location of data in primary storage. The former is difficult to handle because data from multiple files or databases may need to be copied to the same locations, thereby producing an irreconcilable conflict. The latter case is difficult to handle because it must be possible to locate all embedded pointers so they can be updated, and there is the additional complexity and runtime cost of reliably modifying the pointers. Pointer modification in this manner is called *pointer swizzling* [CAC$^+$84, Mos90]. Pointer swizzling is essentially a software version of virtual memory. A reference through a pointer to data on disk is detected by a software or hardware check, storage is allocated in primary storage, the data from disk is copied into that storage, and finally, the dereferenced pointer is updated (swizzled) to refer to the primary storage location of the data read from secondary storage. Depending upon the actual swizzling technique used, future uses of the same pointer may need no further checking to access the primary storage copy of the data directly. Pointer swizzling is called *lazy* when done only for the pointer being dereferenced, i.e., when the pointer is actually used during execution, and *eager* when done for the dereferenced pointer as well as all pointers embedded in data read into primary storage. In the latter case, the amount of data read in, over and above that needed by the specific pointer dereference, varies; it can be the size of a logical unit such as an individual record, or a fixed size such as a page or disk block. The smaller the amount of data read at one time, the greater the total I/O cost; alternatively, the larger the amount of data read, the more pointers that may need swizzling even if they are never dereferenced. In lazy pointer swizzling, there is normally some additional cost for each dereference of a pointer to determine if the pointer has already been swizzled. Eager swizzling of pointers, on the other hand, eliminates the per dereference check at the cost of swizzling some pointers that may never be dereferenced. Two other types of pointer swizzling techniques that have been proposed recently are called *ad hoc* and *hybrid* pointer swizzling.

Ad-hoc pointer swizzling uses memory mapping techniques coupled with swizzling of pointers, as described below. In ad-hoc schemes, the persistent pointers are the same size as the virtual memory pointers and the two may be identical. Whenever possible, the page containing the referent data is copied into the virtual memory indicated by the persistent pointer being dereferenced, i.e., an attempt is made to memory map the disk page to the virtual memory locations where the page was last memory resident so that pointers to data contained in the page remain correct. If the desired virtual memory locations are already occupied, e.g., when two objects with identical persistent pointers from different persistent storage areas need to be accessed simultaneously, swizzling is employed. One major problem exhibited by ad-hoc schemes is their greedy allocation of virtual memory.

Hybrid pointer swizzling combines the benefits of lazy and eager schemes, and attacks the greedy virtual allocation problem by dividing the swizzling process into two phases. In the first phase, embedded pointers are swizzled into an intermediate format called *partly swizzled pointers*. A partly swizzled pointer is converted into its final memory format called *fully swizzled pointer* when it is actually dereferenced. Details of the various pointer swizzling techniques are presented in chapter 2. In general, direct pointer manipulation, e.g., pointer arithmetic, is impossible in most swizzling schemes.

The first persistent storage systems to appear [PS-87, Bro89] used lazy pointer swizzling implemented entirely in software. In recent times, schemes have been proposed that perform eager pointer swizzling at page fault time [Wil91a] or employ hybrid swizzling of pointers [VD92]. ObjectStore [LLOW91] is a commercial database system that uses ad hoc pointer swizzling, and other similar schemes have recently appeared, such as QuickStore [WD94]. However, a significant performance advantage of a single-level store is lost if all or most of the pointers embedded in data have to be swizzled. This loss of performance is especially significant for operations that incur high overhead in data preparation; examples include operations like sequential scans, where the data is accessed only once, and operations that deal with large data structures with small primary storage, where the data is implicitly fetched and prepared multiple times. Therefore, I have pursued an alternative approach to memory mapping, called *exact positioning of data* (EPD) that eliminates the swizzling overhead for pointer dereference.

1.2.1 EPD Approach

As part of this pursuit, I have developed a toolkit, called μDatabase (pronounced micro-database), for building persistent data structures using the EPD approach to memory mapping. The EPD approach employs a novel technique that allows application of an old solution to the problem of address collisions when multiple files or databases are accessed simultaneously by an application. The old solution is hardware segmentation; each hardware segment is an address space, starting at a virtual zero, in which persistent data structures can be built, stored, and subsequently retrieved and updated. Data stored in multiple segments can be simultaneously accessed by an application program because each segment has its own non-conflicting address-space. When a segment is mapped into primary memory, embedded pointers that refer to data *within* the segment do not require modification and are treated like normal memory pointers; *inter-segment* or *inter-database* pointers require special treatment, but in general, these pointers represent a small percentage of the total number of pointers accessed during a typical database computation. The issue of intra and inter-segment pointers implemented in software was addressed by van Dam and Tompa [vDT72] in 1971. More recently, the MONADS architecture [Ros90] employs similar ideas at a hardware/architectural level in its object store layer. The current implementation of μDatabase is based on the UNIX operating system and uses the system call mmap to mimic segmentation on conventional hardware without user accessible support for segmentation; Brown's stable store [Bro89] predates this work and also uses mmap to implement a *single* repository persistent store.

1.2.2 Multiple Accessible Databases and Inter-Database Pointers

The EPD approach, and some other memory mapping schemes, support multiple simultaneously accessible persistent areas or databases, each of which can be viewed as an independent single-level store by a program. This support is fundamental to the world view adopted in this work that there will always be multiple, independent data repositories motivated by the desire to cluster related data, enhance security, make it easier to distribute data and simplify addressing. All schemes that support simultaneous access to multiple databases have to deal with the problem of inter-database pointers that are required to construct relationships among objects stored in different databases. Although the complexity of implementing inter-database pointers varies with individual schemes, there is usually some cost to be paid for the processing of inter-database pointers over and above the cost of dereferencing intra-database pointers. Some schemes bypass the problems related with accessing multiple databases by only supporting a single persistent address space, i.e., all persistent objects live in a single repository and are uniformly accessed on disk(s). Such systems usually require a format for persistent pointers that allows a very large persistent space. However, the support is provided at the cost of precluding the notion of multiple repositories, which I believe is unrealistic. Further, these systems incur high execution time costs associated with pointer swizzling.

As a consequence of the additional cost imposed by the processing of inter-database pointers, the performance of all multi-database approaches degrades when an application program dereferences a relatively large number of inter-database pointers as compared to intra-database pointers. It is typical for a computation to dereference many more intra-database pointers than inter-database pointers. The clustering of *related* objects in both traditional and emerging database applications is a widely accepted phenomenon that supports the above assertion. It would be interesting to conduct a study of existing applications to determine the number of *near objects* and *far objects* referenced during a computation. Such a study, however, is beyond the scope of this work and I was unable to find any published reports to contradict the popular wisdom as it applies to this aspect. In other words, the degenerate case for multi-database approaches is quite atypical for real applications and the cost of supporting multiple databases is completely justified by the benefits derived from such support.

It should also be pointed out that the performance of multi-database memory mapping schemes approaches the performance of single-database memory mapping schemes for applications that only access a single database if the memory mapping approach optimizes the case where data can be copied into its previous memory locations. The differences among various memory mapping schemes arise when an application needs to access multiple databases simultaneously.

When compared with other multi-database approaches, the main disadvantage of the EPD approach is the additional cost of accessing data between segments under certain scenarios. In the EPD approach, an inter-database pointer is always dereferenced within the virtual space of the segment corresponding to the database containing the referent data. Thus, the additional cost of dereferencing an inter-database pointer is the cost of establishing a connection to the appropriate segment, where the actual dereferencing of the pointer takes place at normal intra-database pointer dereference speeds. However, sometimes data from one database is needed simultaneously with data from another database, e.g., when data objects from different databases need to be compared during the execution of a program. This situation can be handled in three different ways, depending upon the facilities provided by the hardware and the operating system:

1. by using hardware segment instructions, e.g., an inter-segment compare instruction,

2. by copying data directly from one segment to the other by means of block move instructions, which implies an inter-segment copy instruction,

3. by copying data into and out of a shared memory area that is accessible to all segments.

On hardware that does not support segmentation, no inter-segment instructions exist and it is necessary for segments to share some portion of their address space for transferring information, possibly for further processing. Therefore, μDatabase segments have an address space that is divided into private and shared portions implemented using shared memory.

The lack of segment hardware forces inter-segment copying of data, resulting in poor performance of the EPD approach when the nature of computation requires copying large amounts of data out of containing segment(s) to shared memory. Other multi-database approaches have similar degenerate cases. For example, schemes that map data from multiple databases into a single segment do not have to copy data in a manner similar to the EPD approach. However, such schemes have to deal with the issue of virtual address collisions and the solutions to this problem impose additional costs; e.g., White and DeWitt[WD94, p. 406] showed that for the worst case scenario involving relocations, the performance of their storage system worsened by a factor of three. What is essential is that the degenerate case does not occur often. In subsequent chapters, different techniques will be demonstrated that significantly reduce copying in the EPD approach, further reducing degenerate situations.

1.2.3 EPD Persistence Model

The following terms are used in this dissertation. A *file structure* is defined to be a data structure that is a container for user records or arbitrarily complex data structures on secondary storage; a file structure may relate the contained data in a particular way, for example, maintaining a set of records in order by one or more keys. A file structure is conceptually similar to a database and the two terms are used inter-changeably in this dissertation. An *access method* is defined to be a particular way the data objects in a file structure are accessed; each access method provides a particular interface to the file structure. Examples of different access methods are: one time reading of a set of records, sequential access of records, keyed access of records, depth first traversal of a B-Tree.

The EPD approach to memory mapping uses the notion of a separate persistent area in which data objects are built or copied if they are to persist; this decision was influenced by ideas presented by Buhr and Zarnke [BZ86, BZ89]. From the user's perspective the approach is largely traditional, as user data must be copied to and from the persistent area through a traditional interface (e.g., get() and put()) that provides encapsulation of the file structure to ensure its integrity. Therefore, in this design, memory mapping normally comes into play only for the file structure designer, which is necessary to support multiple accessible file structures in a single application while allowing each file structure to use conventional memory pointers without having to perform any pointer modification whatsoever.

The major alternative to this persistence model is *reachability*. Reachability is the notion that once a pointer has been made persistent, all data reachable from that pointer also persists. In other words, a data item persists as long as some active data item refers to it, directly or indirectly. In systems based on reachability, the executing program creates an arbitrarily complex data structure in its address space and stores a root pointer to that data structure into a *persistent store*. Upon program termination, the system performs a recursive traversal of the data structure, storing it in some way in the persistent store; retrieval occurs in a simpler fashion by dereferencing the root pointer, which causes the data structure and all of its referenced elements to become accessible. The concept of reachability relies on the existence of a special object, usually called the *persistent root*. All objects in the transitive closure of the persistent root are reachable from the root and, therefore, persist.

Reachability is a powerful abstraction with some very beneficial properties especially from the user's point of view, because the user is completely relieved of the responsibility of having to manage object storage. The user does not have to explicitly specify persistence for the objects, neither does the user have to worry about freeing storage for objects that are no longer needed because deallocation happens automatically in a system based on reachability. However, these benefits come at a hefty price because reachability imposes complex storage management and garbage collection requirements upon the system, which in my view, constitute a prohibitive cost. As well, while system handled storage management is useful for most applications, it prevents the implementation of sophisticated user-defined schemes, which are essential in certain circumstances. Also, it may be impossible to delete objects from the store if references to the objects "leak out", which may be an unacceptable prohibition from an object management point of view. Reachability also implies that there can never be a dangling reference in the store. I do not believe that such a requirement is scalable without an exorbitant implementation and/or run time cost. Finally, underlying the notion of reachability is a world view that consists of a single *ether*, i.e., all objects live in a single address space that spans all physical storage devices in the system, the network and even the entire universe. The notion of the ether is in direct contrast to my chosen world view comprising of related objects stored in independent collections; a view I believe is much closer to the real world of objects. While it is possible to selectively apply the abstraction of reachability to independent collections of objects, such an attempt dilutes the concept and results in a diminution of the benefits of reachability. In view of these reasons, I have chosen the explicit persistent areas model over reachability for this work.

1.3 Motivation

I was motivated to investigate a memory mapped single-level store based on the EPD approach because it seemed like the perfect solution to the dichotomy between programming language and database techniques, and yet I found very little evidence of its use in universities or in industry. At the beginning of my investigation, there were only a few systems in various stages of development, and these systems either were based on a world view I considered unrealistic (single persistent address space) or did not apply the memory mapping technique as directly as I envisaged. In addition, there was a complete lack of experimental evidence for or against the effectiveness of single-level stores in general and memory mapped single-level stores in particular. A sound practical and theoretical framework in which to measure and evaluate this emerging research was also missing. Consequently, I have pursued the EPD approach to memory mapping, outlined in section 1.2, for building single-level stores. I have designed and developed a methodology for implementing memory mapped single level stores based on the EPD approach, performed rigorous experimentation to demonstrate the effectiveness of the approach, developed algorithms tuned for performance in an EPD environment and constructed a theoretical framework within which the EPD and other related approaches can be studied and evaluated. In addition to findings presented in this dissertation, the approach followed by this work has been vindicated by the emergence of other systems that have followed similar approaches.

1.4 The Thesis

The thesis of this dissertation is that the EPD approach to memory mapping provides a means of simplifying the implementation and improving the performance of the methods used for manipulation of persistent data. The major issues and problems arising from the use of the EPD approach to memory mapping as a means of building a persistent storage system or database are examined. Many of the problems, such as parallelization of I/O, have essentially the same implications in memory mapped systems as they do in traditional databases. However, the use of memory mapping allows more efficient and straightforward solutions and provides 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 end user. Memory mapped databases are simpler to implement than their traditional counterparts, while eliminating the need for a traditional buffer manager as the operating system manages all I/O operations. On the other hand, some problems, such as recovery control, are much harder to solve in a memory mapped system, largely due to the lack of essential support at the architectural level. As well, some inefficiencies can be introduced because of lack of control over page replacement in most contemporary architectures.

In order to demonstrate the thesis, a prototype implementation, an experimental testbed and a theoretical model were designed and developed:

- to allow experiments to be conducted for comparing the construction and performance of memory mapped data structures with their traditional counterparts,
- to identify fundamental problems related to the memory mapping approach and its implementation on conventional architectures,
- to provide strong empirical evidence that traditional database techniques can be efficiently implemented in a memory mapped environment with significantly reduced programming effort,
- to show that the solutions presented are stable enough to allow the construction of analytical models for predicting behaviour,
- to provide necessary theoretical and experimental tools that can be used for studying high level sequential as well as parallel database algorithms and for performance tuning.

In parallel with the work presented in this dissertation, a few other proposals have been published that exploit similar ideas and contain some common features. However, each of the other proposals has differences that make this work novel. These differences have a profound impact on how each proposal works. In some cases, the differences are largely in the way the overall system is constructed. The important thing to note is that these systems have been developed independently and most have been commissioned only in the last few years. Further, since memory mapping technology is still in a nascent state, there are few measures by which to judge memory mapped systems, making it impossible to evaluate and compare these proposals. All the approaches have to be considered viable and pursued much further before a consensus or a clear winner emerges. It is partially for this reason that I decided to do extensive modeling work; no other project has developed a quantitative analytical model of a memory mapped system. It is my belief that the model will prove extremely useful for studying and evaluating various memory mapped and related systems. In addition to the conceptual differences with other work, another unique feature of this work is the extensive experimentation that has been carried out on a number of different database structures.

There are some aspects of the thesis that have not been implemented due to the size of the undertaking. A deliberate decision was made to concentrate efforts on building the core module of the system and on performing an extensive performance analysis, both comparative and quantitative, of the system. Further, the emphasis of my work has been on the storage aspects of a persistent system and, as such, language design issues were not examined in detail. As a result, I chose to add persistence mechanisms to an existing language, μC++ [BDS+92], by means of a set of library classes that can be linked with the applications that need to manipulate persistent data.

1.4.1 Dissertation Overview

The dissertation is divided into the following parts:

- **Single-Level Stores:** Chapter 2 motivates and introduces memory mapping and the EPD approach to building a single-level store in more detail. Advantages and disadvantages of memory mapping and single-level stores over the traditional approaches are outlined, followed by an extensive survey of related work.

- **The EPD Approach to Memory Mapped Stores:** Chapter 3 presents the main body of this work; the contributions have been divided into four parts:

  1. The EPD approach proposed and developed by this dissertation is presented along with a detailed design and critique of the system. The presentation includes a comparison with related work.
  2. The EPD approach allows general primary storage programming languages tools to be applied, with equal ease, to secondary storage data and its manipulation. It is demonstrated how these techniques are used in building file structures based on the EPD approach. A detailed description of the programming interface to μDatabase is also provided.
  3. In addition to building sequential file structures, partitioned file structures and parallel access methods were designed, developed and analyzed. Details of an investigation into the issue of parallelism in an EPD based system are presented. Parallelism is exploited both at the storage and retrieval levels.
  4. An analytical model of computation for making accurate predictions is an important tool that goes a long way towards demonstrating the thesis. A survey of the existing I/O and memory models revealed that none of these models applied well to the system proposed in this dissertation. Consequently, significant effort was devoted to the design and development of a reliable analytical model of the proposed system. To make the effort even more useful, the model that has been developed is quantitative as opposed to qualitative.

- **Experimental Analysis of EPD File Structures and Access Methods:** Chapter 4 presents the results of a series of experiments conducted on a carefully designed testbed. The experience gained from conducting these experiments suggests not only that EPD file structures and their access methods can be built more easily than their traditional counterparts, but also that, in most cases, memory mapped structures perform as efficiently or better than their traditional counterparts. Further results presented in the chapter show that memory mapped parallel access methods perform quite admirably in an EPD system and offer some distinct advantages.

- **Applying and Validating the Analytical Model:** Parallelized multi-disk versions of several database join algorithms were designed and implemented. The analytical model developed as part of this work was employed to perform a quantitative analysis of these algorithms when run on a specific machine. The analysis and its verification by means of experiments are presented in chapter 5.

- **Unresolved Aspects of the System:** Two important services provided by a DBMS are concurrency and recovery control. These aspects have not been dealt with in the current phase of this work. Chapter 6 contains a discussion of problems associated with providing these services in EPD systems. The discussion includes a survey of related work highlighting approaches taken by related memory mapped single-level stores. Some of these solutions can be applied to systems based on the EPD approach.

Chapter 2

Memory Mapping and Single-Level Stores

The main objective of this work is to investigate issues surrounding a single-level store based on the exact positioning of data (EPD) approach to memory mapping. The main reasons for using a single-level store are:

- A single-level store eliminates the need for expensive execution time conversions of structured data that are essential in a traditional multi-level store. As well, the cost of referencing persistent data is the same as a normal memory reference once the initial transfer of data from secondary to primary memory has occurred.
- The uniform view of data afforded by a single-level store has various other implications, the most important of these being reduced programming complexity, and the availability of the expressive power and the data structuring capabilities of a general purpose programming language for creating and manipulating data stored on secondary storage.

In a single-level store based on the EPD approach, the contents of a mapped file structure are accessible by a program just like the contents of a data structure in primary storage. What differentiates a mapped file structure from primary memory data is that the file structure data persists after a program using it terminates and during its use, the time to access its data is non-uniform because the file structure is kept on secondary storage but is implicitly cached in primary storage by memory mapping. In μDatabase, a file structure is maintained in a named UNIX file. A mapped file structure and its access methods need to be optimized to achieve good performance in the face of non-uniform access time, usually by improving locality of references by clustering related objects.

2.1 Motivation for Using Memory Mapping

Complex data structures in primary storage are usually organized with memory pointers used directly by the processor's instructions, rather than organized physically, such as elements of an array or records in a disk block. It is extremely difficult and cumbersome to construct complex relationships among data objects without the help of direct pointers. Thus, it is highly desirable to be able to use pointers in organizing and relating data in a file structure. However, it is generally impossible to store and retrieve data structures containing direct pointers from secondary storage without converting (at best) the pointers or (at worst) the entire data structure into a different format. In other words, the data structure in primary storage has to be reorganized into a form (e.g., a stream of bytes) that is suitable for secondary storage; the reverse must take place when the stored data structure is retrieved into primary storage. Considerable effort, both in terms of programming and execution time, is required to transform data from one format to the other in this manner. As an example, figure 2.1 illustrates the transformations that occur for restructuring a tree data structure into a stream of bytes suitable for secondary storage and vice-versa. The transformations **X** and **Y** are data structure specific and must be executed each time the data is written to or read from secondary storage. Consequently, the use of powerful and flexible data structuring capabilities of modern programming languages are not directly available for manipulating secondary storage data.

In spite of these rather taxing difficulties, database implementors have traditionally

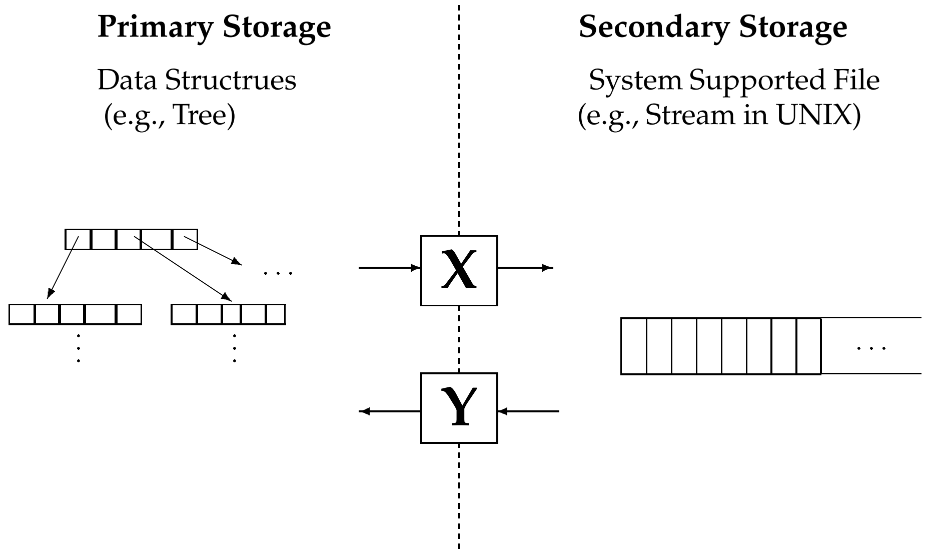


Figure 2.1: Two Views of Data

rejected the use of memory mapped files and have chosen to implement the storage level support for databases using traditional approaches (e.g., explicit buffer management). This rejection is not entirely based on a lack of availability of memory mapping facilities. The earliest use of memory mapping techniques can be traced back more than 20 years to the Multics system [BCD72]. However, earlier operating systems, including Multics, provided these facilities in a framework that was very rigid and difficult to work with. There are other reasons given to explain why memory mapping has not been popular with database designers. Among the most notable of these reasons 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. The rebuttals to these criticisms, as pointed out in [CFW90], are:

- Newer operating systems, such as Mach [TRY$^+$87] and SunOS [Sun90], are considerably more liberal in what they allow users to do with the underlying virtual memory system. Mach provides user-level facilities to better control when the data pages of a mapped file are written back.
- The address space provided by 32 bits, while not excessively large, is sufficient for many emerging and traditional applications. Additionally, processors with larger virtual address spaces (up to 64 bits) have become commercially available, e.g., the MIPS R4000 [Mip91] and the DEC Alpha [Sit92] microprocessors.
- Memory management schemes are becoming more sophisticated so that less memory is used for page tables. For example, some implementations employ 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].

Using memory mapping to implement a single-level store offers a number of advantages that significantly simplify the development of file structures in complex design environments, such as CAD/CAM systems. These advantages are described in detail in section 2.2.2, and clearly outweigh any disadvantages of memory mapping described in section 2.2.3.

2.2 Memory Mapping and the EPD Approach

As illustrated in Figure 2.2, memory mapping is the technique of using the underlying hardware and software architectural support for virtual memory to map some portion of the secondary storage (e.g., a disk file) into the virtual address space of a program, so that the data stored on secondary storage becomes directly accessible by the processor's instructions. Once mapped, the secondary storage data has a one-to-one correspondence with its image in virtual memory.

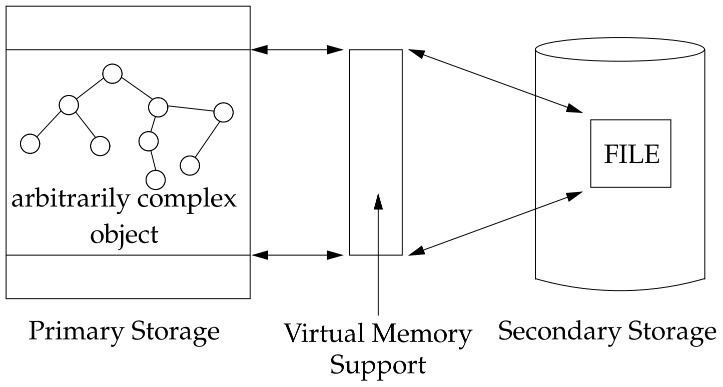


Figure 2.2: Memory Mapping

The concept of virtual memory has been expounded upon in detail in the literature (see [Den70]) and a basic understanding of virtual memory is assumed in this dissertation. Virtual memory capabilities and their accessibility vary substantially among different computer architectures. In general, there are two major capabilities: segmentation and paging, which can be used independently or together. A *segment* is a variable sized contiguous area of virtual memory with a fixed starting address, usually 0. This starting address is called the *virtual zero*. Conceptually, a segment is a set of contiguous *pages* in virtual memory, where a page is a fixed size range of virtual addresses. The physical memory analogue of a virtual memory page, called a memory frame, is a fixed size range of contiguous physical memory locations. *Paging* is the ability to map a non-contiguous set of physical memory frames onto a contiguous set of virtual memory pages.

Depending on the system capabilities, memory mapping can map a file structure into a new segment or into a portion of an existing segment. Because this work adopts the EPD approach to memory mapping, a file structure is mapped into its own dedicated segment; otherwise the mapping cannot be guaranteed to start at the same virtual address each time the file structure is mapped, and memory pointers embedded in persistent data cannot be used to access the referent data without first being *relocated* or *swizzled* (address consistency problem). The ability to store direct memory pointers to relate data in a file structure and to use these pointers, without modification, to access the referent data is essential to the design presented in this dissertation. Demand segmentation and paging, the abilities to copy only those pages of a segment into primary storage that are referenced during execution, are also essential to this design because a file structure is almost always larger than the primary storage capacity of the machine.

Notice that demand paging conceptually performs the function of a traditional buffer manager, except that 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, but the desired efficiency is possible with relatively few different page replacement schemes [Smi85]. Although commercial operating systems have traditionally supported a single system-wide page replacement scheme, many systems are beginning to provide tools that allow application programs to influence the underlying page replacement strategy. In addition, some research projects are building operating systems with specialized paging support geared towards persistent systems.

2.2.1 Non-Uniform Access Speed

When constructing a memory mapped file structure, it is imperative to understand that certain access behaviours can be expensive. While the contents of a file structure are made directly accessible to the processor, the access speed is non-uniform – when the page containing a reference has to be read in, a long delay occurs as for a traditional disk read operation, otherwise the reference is direct and occurs at normal memory speed. Non-uniform access is an aspect of performance that a file structure designer will never be able to control in its entirety as the use of memory mapping involves a deliberate decision to let the operating system be in control of the demand aspect of segmentation and paging to make efficient use of primary storage and other system resources. Unless different page replacement schemes can be selected by individual applications, a file structure designer can, at best, influence the effects of paging by controlling the management of primary storage and, to a lesser extent, by controlling disk allocation. While this lack of control seems like a fundamental flaw, experimental work has shown that it presents few practical problems, except for certain specialized access patterns, depending upon the particular page replacement strategies available. The problem is further mitigated by advancements in the operating system technology mentioned earlier.

2.2.2 Advantages

The following are some of the benefits that are derived from using single-level stores, especially memory mapped stores, to build file structures and their access methods.

Uniformly Accessible Data: The dichotomy resulting from maintaining two disparate views of data in traditional programming systems has been mentioned earlier. Pointers embedded in a data structure must be transformed in order to be compatible with secondary storage before they can be stored and the reverse must occur during retrieval. For example, a CAD/CAM designer may want to store data organized in the form of a CSG tree [Req80] on a conventional file system, e.g., the UNIX file system. The access method designer must devise ways of expressing the CSG tree in the form of a stream of bytes, and either graph operations execute on this stream of bytes emulating a graph, or the primary memory graph must be regenerated during retrieval. Both approaches result in significant overhead in terms of program complexity and execution time. A single-level store greatly reduces and even eliminates these deficiencies by allowing the use of programming language constructs for organizing persistent data. No conversion of primary storage data structures is necessary to store them on secondary storage, which results in significantly improved performance.

A Single Pointer Type: Single-level stores based on pointer swizzling schemes present a uniform view of all data to an application program but these systems use different formats for pointers to persistent and transient data; the conversion of pointers from one form to the other is transparent to the executing program. In the EPD approach, however, normal memory pointers are stored directly on secondary storage without any transformation, and used subsequently to access the referent data. When pointers to persistent data are dereferenced during execution of a program, the I/O necessary to bring the referent data into primary storage occurs implicitly through the virtual memory mechanisms.

For rather simple data structures, like a B-Tree, the elimination of pointer swizzling does not result in a major performance improvement. However, for complex data structures, such as graphical objects in a CAD/CAM system or in a geographical information system, where a large proportion of the data consists of pointers, there is a significant performance advantage resulting from the elimination of the transformation of pointers from one format to the other.

Elimination of Explicit Buffer Management: Efficient buffer management is crucial for the performance of a traditional database system and writing a good buffer manager is complex. Further, given the availability of a buffer manager, a file structure designer must be skilled in its usage to achieve good performance, explicitly invoking the buffer manager's facilities correctly, possibly pinning/un-pinning buffers, which results in code that is complex and difficult to write, understand and maintain.

In a memory mapped system, all data is implicitly buffered, with the I/O being done by the underlying operating system. This model of I/O results in significantly less complex access methods. I/O management is completely transparent and is handled at the lowest possible level, where it has the potential to have the greatest effect on the overall efficiency of the system, particularly on a shared machine.

Unfortunately, most contemporary operating systems do not allow an application to select its own page replacement strategy. This lack of choice nullifies the advantage of memory mapping for some specific applications. However, subsequent results will show that, in general, the buffer management provided by a typical operating system page replacement algorithm produces results that are comparable to a hand-coded buffer-manager for a number of varied access patterns.

Simple Localization of Access: The apparent direct access of all memory locations implies that any data structure can be stored on secondary storage. This feature, which gives a false sense of control to the file structure designer, and coupled with non-uniform access of locations can result in data structures that are not appropriate for the memory mapping (or any other) approach from a performance view-point. The main design criterion for constructing memory mapped access methods is localization of data access. 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. Simple changes to memory allocation strategies can produce significantly better performance in memory mapped access methods due to localization of accesses.

Because the data structures on secondary storage can be manipulated directly by the programming language, tuning for localization is straightforward. Also, this capability provides a wider spectrum of choices for the designer. A trade off between complexity (of increasing locality of references) and performance can be exploited to achieve a desirable balance.

Rapid Prototyping: By relieving the file structure designer of the responsibility of dealing with two different views of data in essentially two different environments, a file structure can be reliably constructed in a shorter period of time. The file structures discussed later in this dissertation were constructed and debugged quickly.

Building a file structure based on the EPD approach is further assisted by the ability to use all the available programming language tools. For example, language polymorphism can be exploited to reuse existing code, and an interactive debugger can facilitate quick detection of errors. When debugging, it is possible to examine secondary storage data as easily and efficiently as primary storage data. Other programming language tools such as execution and storage management profilers, and visualization tools are also directly usable for secondary storage data.

Better Utilization of a Shared System: In memory mapped systems, all I/O is performed by the underlying page replacement algorithm, which allows the operating system to be fair to all users and to dynamically respond to the system load both from database and non-database access. When the system load is light, it is perfectly reasonable to allow large portions of the database to reside in memory. During times of heavy load, database applications share available resources with other applications. In traditional database systems, the buffer manager is often in conflict with other users of the machine, holding resources it is not currently using when access to the database is low.

In general, access methods built using memory mapping cannot make guarantees about absolute performance during execution on a shared system any more than traditional access methods and buffering strategies can. In both cases, statements about performance are only valid if there are no other applications running on the machine. In reality, most database systems share the machine with other applications that affect performance in unpredictable ways. It is my contention that tying file access into the paging mechanism allows better overall system resource utilization and that memory mapped access methods have the potential to achieve better performance on a shared system than traditional database systems. This assertion is based on the fact that the operating system has knowledge about the entire state of the machine, and therefore, has the potential to make informed decisions to achieve good overall performance. Further, memory mapped access methods can immediately benefit from any extra memory that becomes available in the system, even on machines elsewhere in the local-area network [FMP$^+$95].

Improved Support for Large Objects: Memory mapping provides the file structure designer with a contiguous address space even when persistent data is not stored contiguously, which means that a large single object may be split into several extents on one disk or several disks and the application does not need to be aware of this splitting. In traditional systems, the buffer manager has to be designed to explicitly support this seamless view of individual objects in a file structure consisting of non-contiguous fixed-size blocks on secondary storage.

Further, systems based on conventional pointer swizzling schemes are constrained to read at least a whole object. An entire object must become memory-resident when a pointer to it is dereferenced in order to maintain integrity of references to persistent objects. In virtual memory based systems, partial objects can be memory-resident; only those pages of a multi-page object that are referenced need to be read into virtual memory resulting in a significant performance advantage for applications that make sparse use of large objects.

Elimination of Double Paging: In traditional systems, the buffer management provided by the database system can be at odds with the underlying virtual memory management of the operating system. This conflict can result in excessive and unnecessary I/O, unless facilities are provided to instruct the operating system not to manage the buffer space in virtual storage. Not all operating systems provide such a facility nor will they guarantee to honour such a request. In a memory mapped system this problem is eliminated.

Pointer Arithmetic: An important advantage of the EPD approach for building a single-level store is the ability to perform normal pointer arithmetic on pointers to persistent data structures. While all programs may not need to perform pointer arithmetic, certain specialized storage management schemes require this capability. The fact that pointer arithmetic works on persistent data structures in a manner similar to data structures in primary memory illustrates the level of transparency in the single-store provide by the EPD approach.

2.2.3 Disadvantages

Rigid Page Replacement Schemes: Commercial operating systems have tended to be quite inflexible in terms of allowing application programs to control the page replacement strategy used. Typically, a single system-wide page replacement scheme, usually a variant of the LRU scheme, is employed. While LRU is quite suitable for a wide variety of access patterns, it can result in excessive paging under certain circumstances, e.g., when it is known in an algorithm that a page will never be used again, the LRU scheme must still let the page age before it becomes a candidate for removal. This problem can be removed to a large extent by providing operating system facilities that allow an application to influence page replacement decisions taken by the operating system. Operating system designers are beginning to take notice and some application level control over page replacement is already available in a few commercial operating systems, most notably the Mach operating system.

Timing of Dirty Page Write Back: Another major irritant with conventional operating systems is the lack of control over the time at which modified (or dirty) pages in the virtual space of an application are written back to disk. Premature writing of dirty pages in the middle of a transaction results in the data on disk being in an inconsistent state, which increases the difficulty of implementing transactional support for memory mapped systems. One suggested solution to overcome this difficulty is based on page comparing techniques. The basic idea is to keep a before copy of all pages that need to be modified during a transaction. When the transaction commits, the current state of the modified pages is compared against the before copies and any differences are used to maintain recovery logs. This issue is discussed further in chapter 6.

2.3 Survey of Related Work

The earliest use of memory mapping techniques can be found in the Multics system [BCD72]. However, earlier operating systems were not flexible enough to allow exploitation of memory mapping 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 salient work on memory mapping as well as work on single-level stores in general.

2.3.1 Software Approaches Based on Conventional Architectures

In these systems, the emphasis is on using software systems to build a single-level store without requiring new hardware and making no or little changes to the operating system kernel. The main advantages of following this approach are simplicity, cost effectiveness, immediate availability on existing architectures, and wide applicability. On the other hand, there are certain aspects of building a single-level store that are difficult or inefficient to implement without the availability of specialized hardware or operating system support. Nevertheless, the convenience of the software approach makes it an attractive pursuit, particularly in view of recent and imminent advancements in hardware and operating system technologies. The experiences gained by pursuing the software approach also provide valuable input in the design of desired features for future commercial architectures.

While some projects have designed new or modified compilers (PS-Algol, E, ObjectStore), many of the systems have been implemented as language (particularly C++) class libraries that can be linked with applications, and require no special compiler support. Some systems support *orthogonal persistence* [ABC$^+$83] implying that the same compiled code can be used to manipulate both transient and persistent data, and objects of all data types definable in the language can be made persistent. On the other hand, many of the non-orthogonal systems provide a subset of the benefits of orthogonal persistence, e.g., a unified type system for all data without considerations of longevity. Finally, some systems, such as the EXODUS Storage Manager, support the concept of object identity at an additional cost, whereas most memory mapped systems do *not* fully support object identity because non-garbage collected memory management is used.

In addition to the systems described below, there are other varied object managers and storage systems that have been proposed in the last decade, e.g., the O2 Object Manager [D$^+$91] and the GemStone database system [BOS91]. The emphasis and design of these systems is, however, quite different from my work and, as such, they are not described here.

PS-Algol / POMS

PS-Algol [PS-87] was the first effort to add persistence to a conventional programming language and the Persistent Object Management System (POMS) [CAC$^+$84], written in PS-Algol, can be considered the first persistent object system. An implementation of POMS in the C language [Bro89], called CPOMS, provides the underlying support for PS-Algol in Unix environments. In POMS, pointers to objects resident in virtual memory have a format different from pointers to objects stored on disk; the former are called *local object numbers* (LONs[1]) and the latter are referred to as *persistent identifiers* (PIDs[2]). Although PIDs can be arbitrarily large, in the actual implementation of CPOMS, the PIDs were the same size as normal pointers in PS-Algol. PS-Algol's persistence model is based on reachability (see section 1.2.3) – all objects stored in a database on disk are

[1]In CPOMS implementation, a LON is simply a virtually memory address, called a local address.

[2]A PID may be a simple offset within a file for single database implementations or a more complex entity, e.g., a long pointer that identifies the disk object in a (single) universe of objects comprising all objects stored on all disks on a network.

reachable from a distinguished object called the *root* of the database. At the beginning of program execution, the root object is loaded in virtual memory and assigned to a global pointer accessible to all programs. The root object contains pointers (PIDs) to other objects on disk. When a program tries to dereference an embedded PID, the system loads the referent object into virtual memory from disk and replaces (i.e., swizzles) the dereferenced PID with the LON of the newly loaded object in memory. Thus, all pointers embedded in an object on disk are represented by PIDs, whereas for an object in memory, the embedded pointers can be either LONs or PIDs. When the program finishes, all embedded LONs are converted back (i.e., de-swizzled) to PIDs before the objects in memory are written back to disk. All of this address translation is handled in software and is lazy pointer swizzling because a pointer is swizzled only when it is actually used. Since LONs and PIDs co-exist in primary memory, it is necessary for the two types of pointers to be distinguishable. In POMS, this distinction is achieved by using the most significant bit (MSB) of the pointer fields; PIDs have a MSB value of 1 whereas the LONs contain a zero in that bit. At each dereference, the MSB is checked to see if the pointer being dereferenced is a PID or a LON. These checks, called *residency checks*, are a potential performance problem. Another problem with the scheme is that an object has to be loaded in its entirety in order to avoid problems with referential integrity. Also, the size of the PIDLAM, described next, can become a problem since it contains one entry for every object.

During program execution, POMS maintains a two way mapping between LONs and PIDs to facilitate swizzling and de-swizzling of pointers. The mapping is implemented in a memory-resident data structure called the Persistent Identifier to Local Address Map (PIDLAM), which is a two way index implemented by means of two hash tables. When an embedded pointer in local memory is dereferenced, the dereferencing operation does a residency check and consults the PIDLAM if the pointer is a PID. If an entry for the

PID exists, the referent object is already in memory and, therefore, execution can resume as soon as the dereferenced PID is replaced by the LON of the referent object from the PIDLAM. If, however, the PIDLAM contains no entry for the PID, the referent object is fetched from disk and loaded into local memory. An entry linking the PID with the new LON of the object is added to the PIDLAM, the dereferenced PID is replaced by the LON and execution continues. In both cases, any subsequent deference of the same pointer field continues without delay after the residency check because the pointer is a LON.

Napier / Brown's Stable Store

The Napier88 system used a stable store by A. L. Brown [Bro89, DRH$^+$92]. Brown's store is one of the earliest proposals to exploit memory mapped files for implementing a persistent store. The store is implemented by mapping a single, fixed length Unix file to a single virtual address space. The mapping of the file is done at a fixed virtual zero; the mapped data in the file starts at an offset from the beginning of the file. Since the mapping occurs at a fixed address (EPD approach), there is no need to relocate or swizzle memory pointers embedded in the data objects. Brown's store does not support multiple simultaneously accessible persistent stores; all persistent objects live and are addressed in a single persistent store. Also missing is support for disk partitioning of the single-level store. The Napier88 system has since been extended considerably by integration of concurrency and distribution mechanisms into the system. This extension was carried out by Munro [Mun93] as part of his doctoral thesis and is described briefly in section 6.1.1.

E / EXODUS Storage Manager

E [RCS93] is a persistent programming language, developed as part of the EXODUS project at the University of Wisconsin, that relies on the EXODUS storage manager for providing basic support for objects, files and transactions. E extends C++ by adding persistence and some other language features; the latest version of the E compiler is based on the Gnu C++ compiler. The support for persistence in E is provided by means of new data types (called database or db types) and a persistent storage class. Any C++ type/class can be defined as a db type/class, thereby defining the type of objects in the database. The persistent storage class provides the mechanism for storing objects in a database. In order to persist after a program is run, an object of a db type needs to have the persistent storage class property. Additional language constructs are provided for manipulation of persistent objects, e.g., the built-in db class collection[Type] provides a mechanism for creating and deleting objects in a persistent collection. The support for persistence in E is implemented in software similar to PS-Algol; each dereference of a pointer incurs a residency check implemented by means of in-lined code.

The style of persistence provided by E is called *allocation-based* persistence; the designers of E rejected the notion of reachability (see section 1.2.3) for reasons outlined in [RCS93]. In E, the persistence of an object of db type needs to be explicitly specified (either by declaring a persistent variable or by placing the object in a persistent collection). Another aspect in which E differs from PS-Algol and other systems is its rejection of orthogonal persistence. Only objects of db types can persist, i.e., E has a dual type system: objects of db types *may* persist whereas objects of normal types are all transient. One of the implications of this dual approach is that only pointers to db types incur the cost of a run time residency check. Another reason cited by the designers of E for rejecting orthogonal persistence is the wasted space that results by making all pointers long pointers.

Other Language Efforts

In addition to PS-Algol, Napier and E, there have been several other language design efforts that add persistence to a traditional programming language. One of the more prominent of these languages is O++ [DAG93, BDG93] based on the Ode/EOS object manager. O++ is a database programming language based on C++ that provides support for orthogonal persistence. The compiler is implemented as a front end called `ofront` that translates O++ code into C++ code to be compiled and linked together with the Ode Object Manager, which is implemented on top of the EOS storage system. EOS manipulates data on disk in units of disk pages and objects are essentially uninterpreted sequences of bytes with some header information. One of the important features of O++ is its support for making the virtual pointers of C++ persist, i.e., it allows objects with virtual members to be persistent.

Texas: Pointer Swizzling at Page Fault Time

Paul Wilson [Wil91a] has developed a scheme that combines the concepts of pointer swizzling and run-time page faulting to support huge persistent address spaces with existing virtual memory hardware. Texas [SKW92] is a persistent store based on Wilson's scheme of pointer swizzling at page fault time. In Wilson's scheme, pointers on secondary store have a format different from the pointers in primary storage, which allows for a persistent store that is larger in size than the virtual space supported by a given hardware. Wilson's scheme requires a special page fault handler that is responsible for swizzling pointers.

The basic strategy is to fetch pages as opposed to objects as is done in classical lazy pointer swizzling. When a page fault occurs, i.e., the virtual memory hardware detects an attempt to dereference a pointer to a location in a non-resident disk page, virtual memory for the disk page is allocated (if not done already) and the page is fetched in memory. An address translation table maintains the current mapping of virtual memory to disk pages. The table contains one entry per page rather than one entry per object resulting in a considerably smaller and fixed size table.

During fetching, a disk page is scanned and all embedded persistent pointers are translated into virtual memory pointers, which requires knowledge of all pointers. Thus, memory resident pages in Wilson's scheme never contain persistent pointers, only virtual memory pointers. Extra information is maintained on disk to permit the finding of all pointers embedded in data and there is an associated run-time cost of processing this extra information. Furthermore, objects that cross page boundaries require additional language support.

For embedded persistent pointers that refer to disk pages seen previously during the current execution, the translation table is used to swizzle the pointers into corresponding virtual memory values. To facilitate the translation of other embedded persistent pointers, all the referent disk pages are greedily allocated virtual memory space and appropriate entries are made in the translation table. However, the disk pages are not actually loaded at this time. Thus, the faulting of a single page can result in virtual memory being allocated for a rather large number of other pages. Some of these pages may never be used and, therefore, Wilson's scheme can result in underutilization of virtual space. Wilson has proposed some solutions to these problems, such as periodically invalidating all the mappings and rebuilding them; however, the solutions increase significantly the complexity and cost of his basic scheme. For example, if virtual memory space is exhausted during execution of a transaction, some of the memory-resident pages have to be written back to disk in order to recover virtual memory space. Evacuation of memory-resident pages in this manner requires a significantly complex de-swizzling process, which results in a serious degradation of performance.

Hybrid Pointer Swizzling

Vaughan and Dearle [VD92] have presented a hybrid pointer swizzling scheme that retains the salient features of both the lazy swizzling employed by POMS and the eager swizzling of Wilson's scheme. The hybrid scheme splits the pointer swizzling process into two phases in order to avoid the problems associated with Wilson's greedy virtual memory allocation. The hybrid scheme mandates that persistent pointers be at least twice the size of virtual memory pointers. The low order bits in a pointer field are used for machine addressing and the extra space available in a swizzled pointer is used to maintain some additional information used to simplify the de-swizzling process. Pointer fields in memory-resident pages can contain valid virtual memory pointers to either actual objects or entries in a memory-resident translation table; the former are called fully swizzled pointers and the latter partially swizzled pointers. To dereference a partially swizzled pointer, the system consults the translation table to see if the corresponding disk page is memory-resident; if not, a page fault occurs and the disk page is brought into virtual memory. At this time, each pointer embedded in the newly read page is changed to a fully swizzled pointer if it refers to a memory-resident page, or to a partially swizzled pointer if the referent page has not yet been fetched into memory. Dereferencing of fully swizzled pointers proceeds without interruption, thereby avoiding the per reference cost associated with lazy swizzling. Since references to non-resident pages are not immediately translated into virtual memory pointers, the greedy allocation of virtual memory is avoided; a disk page is allocated virtual memory only when actually used. The main additional costs with the hybrid scheme are a special software dereference operator and the increased size (at least double length) of pointers.

ObjectStore

The ObjectStore Database System [Atw90, LLOW91], developed by Object Design Inc., is a commercial object oriented DBMS that makes use of conventional hardware in conjunction with an ad hoc pointer swizzling scheme for storing virtual memory pointers on secondary storage. The swizzling scheme used is similar to Wilson's scheme except that in ObjectStore, normal programming language pointers are used to refer to persistent as well as transient objects. ObjectStore supports simultaneous access to multiple databases and individual databases are allowed to be larger than the virtual address space. These capabilities are achieved by maintaining a mapping between disk pages and the virtual memory addresses assigned when the pages were last memory-resident. The exact details of the scheme used by ObjectStore are proprietary[3]. However, the QuickStore system, described below, is believed to use the same scheme.

Every new persistent object in ObjectStore is explicitly created in a particular database. Individual databases are subdivided into segments[4] and the application can cluster related objects by specifying the segment within a database where the new object is to be created. An inter-database pointer in ObjectStore is treated differently from an intra-database pointer. In general, an inter-database pointer is transient, i.e., it is valid only during the scope of the assigning transaction. A persistent inter-database pointer needs to be explicitly distinguished and is implemented as a long pointer.

ObjectStore is based on the client/server paradigm; the server maintains the persistent store and provides all fundamental support services including concurrency and recovery control. The server makes available, on demand at page fault time, the necessary pages of secondary storage, which are then mapped by the client into its virtual address

[3]The description in this section is based on information obtained from [LLOW91] and [Obj93]

[4]The term segment in this section refers to a logical sub-division of a larger database and does not mean a hardware supported segment as discussed in section 1.2.

space; the granularity of server transfers can be changed from a page to a segment, with the latter resulting in *en masse* transfer of a complete segment. For each transaction, only those parts of the database(s) that are accessed by the transaction are mapped into the address space of the client. This strategy introduces a restriction on the total amount of data that can be referred to by any single transaction. Large operations need to be broken down into a series of smaller transactions.

When a page is mapped into the virtual address space, ObjectStore dynamically assigns a virtual address where the mapping is to take place. An attempt is made to assign the address so that the pointers stored on the server continue to be valid in virtual memory of the client, which is possible when the page being mapped as well as all the pages referred to by pointers embedded in the mapped page can be assigned the same virtual addresses where the pages were last resident. No pointers need to be swizzled in this scenario and execution can continue as soon as the page is mapped. In all other cases, the server has to find all the pointers embedded into the page and swizzle the pointers as needed, which requires that some portion of the type system be available at run time in order to locate all embedded pointers. ObjectStore keeps this information in an auxiliary data structure called the tag table, which records the location and type of every object in the database. The tag table is used in conjunction with the database schema to locate all pointers embedded in objects stored in the page being mapped.

At the end of a transaction, all pages in the client's address space are unmapped and any modified pages are transmitted back to the server; the client blocks until the pages are written back to the server's disk(s). Unmapped pages stay in the client's cache until room is needed for other new pages. A client cache coherency scheme is used to accommodate sharing of pages by multiple clients. In this aspect, ObjectStore's storage management is similar to the one employed by the EXODUS Storage Manager.

Cricket

Cricket[SZ90a] is a storage system that uses the memory management primitives of the Mach operating system [TRY+87] to provide the abstraction of a "shared, transactional single-level store that can be directly accessed by user applications" [SZ90a, p. 89]. Cricket follows the client/server paradigm and, upon an explicit request, maps the database directly into the virtual space of the client application. Cricket uses direct memory pointers and the database is mapped to the same range of virtual addresses so that pointer modification is unnecessary. However, 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 into one collection is not supported in Cricket. Cricket takes the view that everything an application needs to use is placed in a single large persistent store. The designers of Cricket did acknowledge the need to support files and planned on providing an implementation for files in future work. However, it may be almost impossible to support a truly general implementation of files within the framework of Cricket's architecture.

QuickStore

QuickStore [WD94] is a storage system for persistent C++ that is built on top of the EXODUS Storage Manager (ESM), offers nearly the same functionality as E, makes use of memory mapping, and performs pointer swizzling at page-fault time similar to ObjectStore. Because of its use of ESM, QuickStore has a client-server architecture with support for transactions. There are no limits placed on the size of a database; the amount of data accessible by a single transaction is limited to the size of virtual memory. The persistent pointers in QuickStore are the same as virtual memory pointers. The value of a pointer to a persistent object in QuickStore is the virtual memory address of the object when the page containing the object was last memory resident.

Dereferencing a pointer to a non-resident object causes a page fault to be detected by hardware causing the QuickStore page fault handler to request ESM to fetch the page containing the object into the ESM client buffer pool; from there the object is memory mapped into the virtual address space of the application program for direct manipulation. While fetching, the page fault handler performs actions such as swizzling of embedded pointers before the client application resumes. Virtual memory is greedily allocated for the page being fetched as well as for all the other pages that are referred to by embedded pointers. Like ObjectStore, an attempt is made to assign the same virtual frames as used previously. If all of the pages can be assigned their old virtual frames, no swizzling of pointers is needed and the application can resume execution. If some disk pages get mapped to new virtual addresses, however, the faulted page is scanned and any embedded pointers that refer to the relocated page(s) are updated.

In order to perform swizzling, QuickStore maintains extra information for memory-resident and persistent data. The main memory-resident data structure is a table that keeps track of the current logical mapping from virtual memory frames to disk pages. This table contains one entry, called a page descriptor, for every page that has been fetched into memory or is referred to by pointers embedded in memory-resident pages. Page descriptors contain the virtual memory and disk addresses of corresponding pages. The page descriptor table is consulted during allocation of virtual memory addresses to disk pages.

The information maintained on disk for each disk page includes a mapping object and a bitmap. The mapping object for a disk page, say p_i, records the mapping between virtual frames referred to by pointers embedded in p_i and the corresponding disk pages at the time when p_i was last memory resident. The size of a mapping object can vary and, therefore, the mapping object is not stored as part of its disk page; instead, a pointer to the mapping object is kept in a special fixed-size meta object stored at the beginning of the disk page. The bitmap for a disk page is maintained by means of the type information made available at run time and is used to locate all embedded pointers so that they can be swizzled, if necessary. The bitmap is also stored independently of its disk page because the bitmap for a disk page is only needed if pages are relocated at page fault time.

2.3.2 Architectural Approaches

This section describes major projects on single-level stores other than Multics, which has been mentioned before, that focus on memory mapping at the hardware and operating system level. By its very nature, this work takes an entirely different approach than the software based systems described earlier. Architectural approaches are significantly more expensive to investigate and represent important work that provides insights into, and hopefully guides the development of future hardware and operating system support in commercial systems.

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 several 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. In Bubba, the Flex/32 version of the AT&T UNIX System V Release 2.2 was extensively modified to build a single-level store, which makes the store highly unportable. The programming interface to Bubba is FAD, a parallel database programming language.

MONADS Architecture

The MONADS project [RK87, Ros90] started in 1976 at the University of Newcastle, Australia developed a new computer architecture that supports orthogonal persistence by means of a uniform virtual memory as one of its central design goals. The MONADS architecture provides explicit support for objects, both at the architectural and the system software levels. The implementation of the MONADS architecture took the route of employing a combination of hardware, microcode and software. The virtual memory in MONADS is uniformly addressable using a segment addressing scheme. Segments are essentially arbitrary size chunks of addresses in a very large virtual address space (up to 128 bits).

The virtual store in MONADS, unlike many other architectures, is divided into regions called *address spaces* and as such is not flat. A non-flat store was motivated by a desire to make the store as efficient and flexible as a conventional file system, which allows related data objects to be grouped together and managed independently of other groups of objects (see section 1.2). To facilitate efficient and easy addressing, each virtual address in MONADS consists of two components, the address space number and the offset within the address space. The segment addressing scheme in MONADS builds upon the conventional segmentation schemes such as the one used in Multics.

Model for Address-Oriented Software

In [SW92], Smith and Welland introduce a concept called *address-oriented software* to describe any software that makes use of the *value* of memory addresses it references. The authors further propose a general model of the operations, called *address-management*, such software uses. The model is being used to design a new operating system and hardware that is more conducive to supporting the class of address-oriented software. The address-oriented services in their model relevant to this work include support for virtual memory and a persistent object store.

The model proposes a single large persistent address space, larger than the address space of the processor. As such, object pointers are not the same as memory addresses and address translation must take place before a pointer can be dereferenced. At the time of opening the store, the root objects of the store are copied into memory and converted to the memory format. At the same time, virtual address space is allocated to all objects that are referenced in the root objects; no physical memory is allocated yet. When a new object is referenced, it is read into memory using virtual addresses that have previously been allocated for the object and new virtual space is allocated for all objects that have references in the newly read object. This process repeats as the computation progresses, as for Texas, ObjectStore and QuickStore.

In the same paper, Smith and Welland presented a hardware model for implementing their address-management model of address-oriented services and describe the implementation of a subset of their hardware design in the memory management unit of the ARM 600, a processor being built by Advanced RISC Machines, Ltd. of Cambridge, England.

Single Address Space Operating Systems (SASOSs)

Quite recently, some research has been done on developing a new class of operating systems called *Single Address Space Operating Systems* (see [CFL93] for a description of some issues and problems with SASOSs; this section includes a brief relevant discussion). An SASOS is fundamentally different from traditional operating systems in that it uses a single global virtual address space for all protection domains as opposed to assigning each protection domain (e.g., a UNIX process) its own private virtual space. The single address space approach has become viable with the commercial availability of workstations with large virtual address spaces because it is now realistically possible for all computation on a node to occur within a single address space. The global address space is shared by all threads executing on the system. Thus, all threads work with the same virtual to physical mapping of addresses and any virtual address in the system can be dereferenced by any thread. Access to data, however, is determined by the thread's protection domain.

The goal of an SASOS is not to provide persistence but to facilitates sharing of the *transient* address space; in some sense, an SASOS does for transient data what μDatabase does for persistent data. Nevertheless, SASOSs have some relevance to this work because of their promotion of EPD and the use of large virtual address spaces. One of the best known SASOSs is Opal [CLBHL92, CLFL94] developed at the University of Washington. Opal is built on top of the Mach operating system and thus, co-exists with UNIX. The virtual memory allocation unit in Opal is a virtual memory segment, which is a variable sized set of contiguous virtual memory pages. Opal supports recoverable as well as distributed virtual memory segments. Upon allocation, a virtual memory segment is assigned a unique range of addresses in the global address space in order to avoid address conflicts in the event of sharing. One of the major similarities between Opal and μDatabase is support for sharing of pointer-based data structures. μDatabase facilitates sharing via the EPD approach to persistence, whereas in Opal, the use of a single address space allows independently developed tools (e.g., editors and debuggers) to pass and manipulate transient pointer-based data structures.

Grasshopper Operating System

The Grasshopper operating system [DdBF$^+$94] is an attempt to develop an orthogonally persistent operating system that runs on conventional hardware. The desire to develop a new operating system is motivated by the fact that it is often inefficient or too hard to build an orthogonal persistence system on top of a conventional operating system due to their fundamentally different natures. A persistent operating system like Grasshopper removes this inefficiency by providing support for orthogonal persistence in the operating system itself. Nevertheless, there are some limitations that remain due to the lack of required features in conventional hardware.

The fundamental abstractions used in Grasshopper to support orthogonal persistence are called *containers* (storage), *loci* (computation) and capabilities (*access control*). The three abstractions are orthogonal in nature and, as such, can be applied independently. Grasshopper adopts a fully partitioned address space model, i.e., there is no global address space. Instead, there are fully independent address spaces each of which can be arbitrarily sized. Processes execute within one of these address spaces (*host address space*) and access is limited to data stored within the host address space. Further, the conventional association of address spaces with processes is non-existent because processes (loci) are orthogonal to address spaces (containers). A Grasshopper system contains a set of address spaces and a set of loci executing within the set of address spaces. A locus can execute in and access data stored within one container at a time. Unlike other operating system designs, loci or processes in Grasshopper are inherently persistent. The orthogonality of loci and containers facilitates support for multi-threaded programming because a number of loci can execute within one container simultaneously. Finally, access control over containers and loci in Grasshopper is capability based.

IBM RS 6000 and AS/400

Malhotra and Munroe [MM92] have proposed schemes to support persistent objects on the architectures of IBM RS/6000 and IBM AS/400 computer systems. Both these systems incorporate support for single large virtual memories that are subdivided into segments, although segments have slightly different semantics in the two systems. The authors argue that using virtual memory references to access objects is both more efficient than other approaches, including swizzling, and easier to implement since the operating system does most of the work.

Recoverable Virtual Memory

Thatte [Tha86] has described a persistent memory system based on a uniform memory abstraction for a storage system in which both transient and persistent objects are managed in a uniform manner. The memory is viewed as a collection of variable size blocks of consecutive addresses, in a single large virtual space, interconnected with virtual memory pointers. Reachability (see section 1.2.3) is used for the persistent model; an object in the virtual space persists as long as it is reachable from a persistent root. Thatte's proposal includes a recovery scheme at the level of virtual memory itself because his scheme assumes no separate file system.

In [Kol90], Kolodner presents a critique of Thatte's persistent memory and proposes an alternative scheme.

Camelot Distributed Transaction System

The Camelot project [STP$^+$87] used the memory management facilities of the Mach operating system to provide a single-level store. However, the store was not directly accessible to the application processes but was to be used within a "data server" for storing persistent data. In this regard, Camelot differs from Cricket and the approach I have developed lies somewhere between these two extremes.

IBM's 801 prototype hardware architecture

IBM's 801 architecture [CM88] incorporated an operating system that provided mapped files with automatic concurrency control and recovery. The major share of the support for operations on mapped files was provided by adding special hardware, which resulted in a solution lacking both flexibility and portability.

Clouds Distributed Operating System

The Clouds project [DLA87, PP88, DC90] was an attempt to build a general purpose distributed computing environment for a wide variety of user communities. An "object" in Clouds is the fundamental entity used to build the system. A Clouds object is conceptually a persistent virtual space and lightweight threads are used to perform computations through code stored in objects. The persistent objects and threads, give rise to a programming environment composed of a globally shared permanent memory.

2.3.3 Others

The following are some other interesting efforts at exploiting mapped files and single-level stores. The focus of these works is quite different from my work, and they are described here for completeness.

- Some other notable projects that proposed new architectures to address problems faced by persistent programming community include EOS, an environment for object-based systems [GADV92] and work done on Choices, an object-oriented operating system [CRJ87, RC89, MC92].

- Inohara, *et al* [ISU$^+$95] describe an optimistic page-based concurrency control scheme for memory mapped persistent object systems.
- Peter van Oosterom [vO90] used shared mapped files to introduce persistent objects in the object-oriented programming language Procol [vdBL89] as part of his Ph.D. thesis on reactive data structures for Geographical Information Systems.
- The Hurricane Operating System [SUK92] is a shared memory multi-processor operating system that runs on The Hector Multiprocessor [VSWL91] and uses memory mapping to implement its file system.
- Orran Krieger, *et al* [KSU91] describe a stream I/O interface for Unix using memory mapping facilities.

2.4 Summary

This chapter presented the *raison d'être* for this work: how a single-level store based on the EPD approach is beneficial for managing persistent data. An extensive survey of all major software and architectural approaches to building persistent systems is presented as background material for the design of an EPD based system presented in the next chapter. The software approach to building EPD persistent stores imposes some restrictions on what can be achieved realistically and efficiently, but it provides an excellent opportunity to explore new ideas and to establish a solid framework in which these and other related ideas can be evaluated and analyzed.

Chapter 3

Using the EPD Approach to Build a Single-Level Store

As stated earlier, this work has resulted in the design and development of a toolkit called μDatabase to implement the EPD approach to memory mapped persistent stores on conventional hardware running the UNIX operating system. μDatabase has been used to study a number of sequential and parallel access methods. Also, a theoretical framework has been established by means of an analytical model of computation in μDatabase. This chapter presents all these contributions in detail.

3.1 μDatabase Design Methodology

The design methodology developed as part of this work provides the necessary environment to build efficient file structures and their access methods in a memory mapped environment based on the EPD approach. As stated earlier, a toolkit approach has been adopted, which allows file structure designers to participate in some of the design activity; μDatabase allows extensible additions or simple replacement of low-level components by file structure designers. While μDatabase shares the underlying principles of a single-level store with other proposals [CM88, CFW90, SZ90a, LLOW91, STP$^+$87, WD94], it offers features that make it unique and an attractive alternative. μDatabase is intended to provide easy-to-use and efficient tools for developing new databases, and for converting and maintaining existing databases. It also fulfills a need for a set of educational tools for teaching operating system and database concepts. The design is based on some important decisions, described next, that were made at the outset of this work.

3.1.1 Design Objectives

Employ the EPD approach to memory mapping. μDatabase develops and exploits the EPD approach so that an arbitrary programming language data structure can be stored on secondary storage as is; neither restructuring of data nor pointer swizzling is required for accessing and manipulating the data structure. In the case of a B-Tree, for example, the tree structure can be stored in its entirety on disk using programming language pointers, to be retrieved and navigated at a later date. The routines for performing B-Tree operations in primary storage are used directly on persistent data, which allows *conventional programming techniques for data structures* that happen to be stored in a file. The absence of any transformation of pointers has a beneficial impact on execution costs.

Retain conventional user interfaces. A deliberate design decision was made to retain the conventional semantics of *opening* and *closing* a file. A file structure must be made accessible explicitly because the file content is *not* directly accessible to the processor(s) until it is memory mapped, and therefore, this aspect should be reflected in the semantics of the constructs and not hidden by making the file implicitly accessible at all times. Like pointer-swizzling, there is some problem with efficiently detecting the first access to a file structure so that the additional data structures needed during access can be created. However, the most difficult problem with implicit access is knowing when the access can be terminated, which is a particular concern with concurrency control.

Use light-weight threading. To provide highly concurrent access to the file structures, it was decided to use a concurrent shared-memory thread library as the basis of the design. This decision ensured that concurrency issues were dealt with starting at the lowest levels of the design, thereby avoiding the problems associated with trying to add concurrency post hoc on an inherently sequential design.

Provide library of routines. The current implementation of μDatabase is designed as a multi-level modular system, based on the toolkit approach, with each level performing a particular aspect of the overall system. The system is available as a C++ library that can be linked with user applications. This route was preferred, at least for the present, over making language extensions via a new front end parser or a modified compiler.

Use conventional hardware and software. μDatabase follows the software approach of building a persistent store on top of conventional hardware and operating system. It is one of the fundamental goals of this work to base the design on a standard commercial architecture while keeping the design flexible enough to cope with advances in architecture design. This decision makes the research immediately available to theoreticians and practitioners alike. Also, the use of a standard system has highlighted certain problems that need to be addressed at the architecture level before memory mapped systems can become mainstream.

Multiple simultaneously accessible databases. μDatabase allows individual applications to simultaneously access multiple file structures or databases. Conversely, multiple applications can share the same file structure. This decision has been influenced by the recognition of the merits of a world view that tends to relate objects by their functionality and groups related objects together. Groups of functionally related objects are shared freely among applications in this world view while providing the necessary abstraction, protection and efficiency. The current scope of μDatabase does *not* cover inter-database pointers, only intra-database pointers are fully implemented.

Reachability does not extend to world view. Many current systems (e.g., Cedar, Lisp, Smalltalk, PS-Algol, Napier) use reachability as the fundamental mechanism that determines the persistence of data. Conceptually, reachability 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 feature permits special programming language constructs, such as files, databases, directories, names spaces, etc., to be replaced by simpler arrays or linked-list structures. However, reachability places complex storage-management requirements [BDZ89] on the file structure designer. As explained in section 1.2.3, instead of reachability, μDatabase uses the notion of a separate persistent area, in which data objects are built or copied if they are to persist. Reachability has also been rejected by some other systems described earlier, e.g., E and ObjectStore provide a style of persistence similar to that of μDatabase.

The world is not flat. μDatabase envisions a pragmatic non-flat view of the world. The persistent store in μDatabase consists of a collection of files stored in a conventional file system. To become accessible to a program, each file is mapped into an independent segment with its own virtual space. Thus, each inter-segment persistent address in μDatabase is conceptually divided into a handle for a conventional file (e.g., the Unix file name) and an address within the virtual space corresponding to the file. This model extends to a hierarchical address, including machine or node id on which the file structure resides.

Separate transient and persistent data. Data associated with accessing the file structure, such as traversal locations in the structure, are not mapped in the file structure. This organization facilitates performance and recovery after system failure by ensuring that this data is never written to the disk.

Finally, persistence in the current implementation of μDatabase is not orthogonal because of the restrictions imposed on the use of persistent and transient pointers. However, μDatabase provides some of the features and benefits associated with an orthogonal persistence system. For example, creating and manipulating data structures within a persistent area is the same as in a program, i.e., all data within a persistent area is treated uniformly irrespective of its longevity, which means that code written for primary memory data structures can be used, without re-compilation, to manipulate persistent data.

3.1.2 Basic Structure

In order to simplify the specification of file structures, object-oriented programming techniques are used in the implementation of μDatabase, but are not essential; an implementation may be done in a non-object-oriented programming language. C++ [ES90] is used as the concrete implementation programming language. The constructors and destructors in C++ eliminate the need to have explicit initializations and clean ups, and allow an implementor to make certain assumptions about correct usage (e.g., pre- and post-conditions). Concurrency facilities employed by μDatabase are provided by μC++ [BDS+92], which is the preferred implementation language for writing access methods in μDatabase. μC++ is a superset of C++ with concurrency extensions. However, the fundamental ideas are implementable in any imperative programming language.

The design involves several levels, each performing a particular aspect of the storage or access management of the file structure. The design structure is illustrated in Figure 3.1 and the components at each level are discussed below in detail.

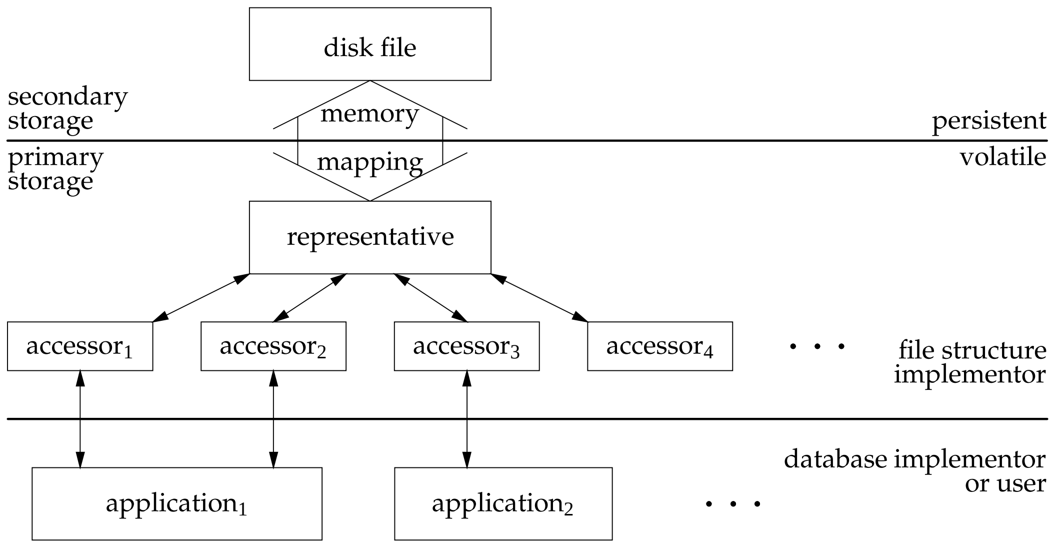


Figure 3.1: μDatabase Design Methodology: Basic Structure

3.1.3 Representative

To allow virtual memory pointers to be stored in the file structure and subsequently used without modification, the system maps each file into its own segment. None of the operating systems available for this work provide the facility of segmentation even when the underlying hardware is capable of supporting such a facility. To overcome this deficiency, μDatabase mimics a segment by using a UNIX process. The disk file is mapped into the virtual space of the UNIX process starting at a fixed memory location, called the *Segment Base Address*. The segment base address is conceptually the virtual zero of a separate segment. In the current implementation, the value 16M has been chosen as the Segment Base Address, which leaves a sufficiently large space for the process's transient data and program code. In an ideal situation, where independent creation of new virtual memory segments is allowed, each disk file would be mapped into its own segment; separate segments are possible, at least at the hardware level, on the Intel 386/486, IBM RS/6000 and the IBM 400 series computers.

The object that manages the segment is called its *representative* in μDatabase, and it is responsible for the creation and initialization of the file structure, the storage management of access method data in primary storage, concurrent accesses to the file structure's contents, consistency and recovery. Each file structure has a unique representative. The representative is created implicitly on demand, during creation of a file structure and for subsequent access by a user application, and it exists only as long as required by either of these operations. Thus, the representative for a file structure is created when the first access request is received for the file structure and terminates when all the access requests have completed. For each application program, there is a one to one correspondence between a file structure and its representative, i.e., at any given time there is at most one active representative for a file structure irrespective of the number of different tasks accessing the file structure simultaneously (as the scope of μDatabase is expanded, there will be only one representative per file structure across an entire system).

A representative's memory is divided into two sections: private and shared; Figure 3.2 illustrates this storage structure. Private memory can only be accessed by the thread of control associated with the UNIX process that created it, i.e., the representative. The persistent file structure is mapped into the private memory of the representative while all data associated with concurrent access to the file structure is contained in the representative's shared memory; such data is always transient. Shared memory is accessible by

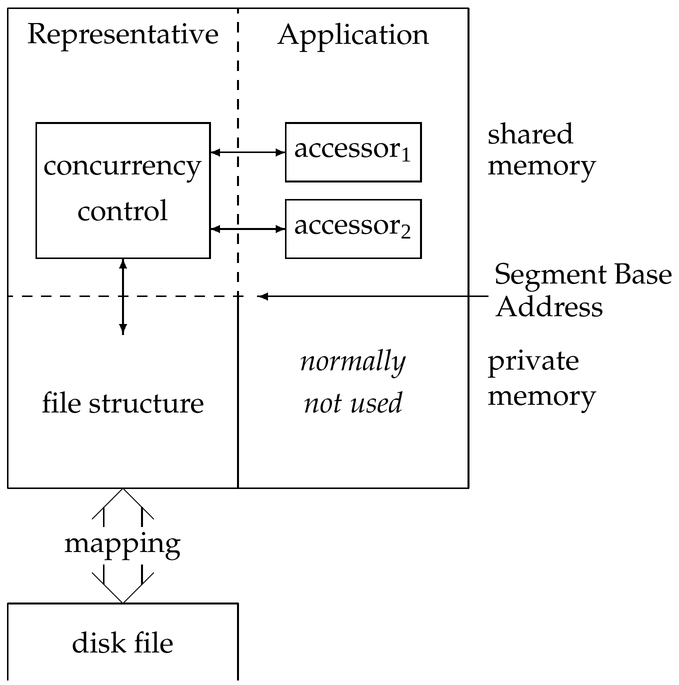


Figure 3.2: Storage Model for the Representative

multiple threads that interact with the representative's thread. There is no implicit concurrency control among threads accessing shared memory except at the memory word level where synchronization is enforced by the hardware. Mutual exclusion must be explicitly programmed by the file structure designer using the light-weight tasking facility of μC++ [BS90, BDS+92]. The light-weight tasking facilities allow virtually any concurrency control scheme to be implemented in the representative.

Some operating systems arbitrarily restrict the maximum accessible virtual space size of a process (e.g., DYNIX has a restriction of 256M), while the hardware is capable of supporting much larger virtual spaces (4G or more). Because of the EPD approach used by μDatabase, the maximum allowable size of any single file structure is the maximum accessible virtual space size minus the Segment Base Address. For databases larger than the maximum allowed virtual space size, it is possible to subdivide the database into multiple files, which the representative can handle by creating sub-representatives; however, this strategy increases the complexity of the file structure. With the increasing number of 64 bit processors becoming available, the restriction imposed on the database size is no longer an issue.

It is possible for an application in μDatabase to have multiple file structures accessible simultaneously because each file structure is mapped into its own representative that has an independent private mapping area. Figure 3.3 shows the memory organization of an application using 3 file structures simultaneously. Since each representative is a separate segment, relocation of pointers in a file structure is never required.

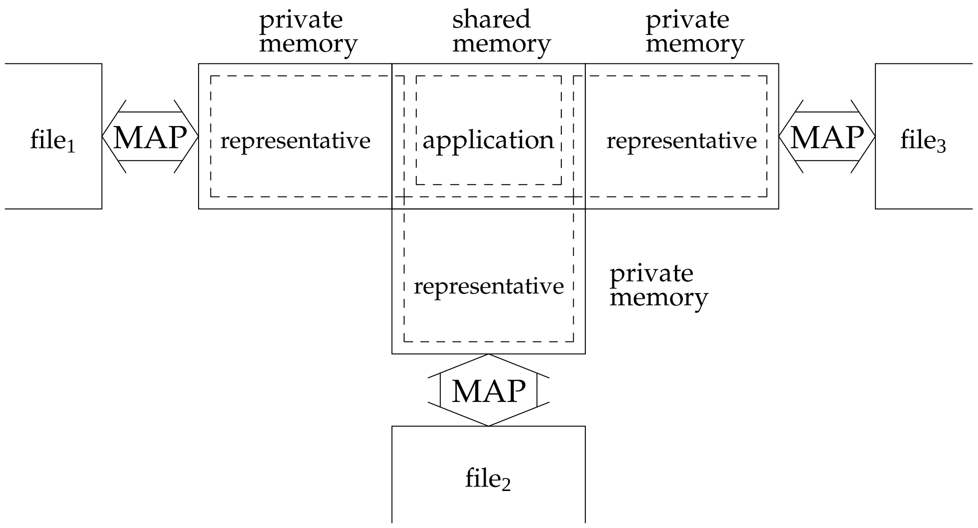


Figure 3.3: Simultaneously Accessing Multiple File Structures

The effect of mapping each disk file into a different virtual space is to move the abstraction of a single-level store from the domain of the application (which is the case in systems such as Cricket [SZ90a]) into the domain of the file structure designer. This strategy *may* compromise performance and complexity slightly, but any potential loss is significantly offset by the added protection for the file structure and the flexibility provided by multiple accessible disk files. Nevertheless, for specialized situations, μDatabase makes it possible to have a single file structure mapped into the application address space, which allows the application direct access to the file structure data at the cost of having only one specialized file structure accessible at a time. However, the use of this facility is intended mostly for temporary data that is generated by the program doing the mapping. The design philosophy of μDatabase discourages general use of this feature.

The representative for a file structure partitioned across multiple disks works by mapping the individual partitions into a single address space similar to single-disk non-partitioned file structures. A partitioned file structure consists of a list of UNIX files each of which may exist on a different disk. The representative partitions its address space based on the list of files and maps each file into a partition of its address space as depicted in figure 3.4. This scheme makes many of the details of partitioning transparent to the application program. One obvious drawback of the scheme is that the size to which each partition can grow is restricted.

3.1.4 Accessors

The representative of a file structure provides all the low-level support and the file structure is “hidden” in that the representative does not provide any direct access to it. The mechanisms for requesting and providing access to a file structure are provided in the form of another entity called an *accessor*. Declaration of an instance of an accessor, called an *access object*, constitutes the explicit action required to gain access to a file structure’s contents (i.e., create the mapping). Creating an access object corresponds to opening a file in traditional systems but is tied into the programming language block structure. As well, the access object contains any transient data associated with a particular access (e.g., the current locations in the file structure), while the representative contains global transient information (e.g., the type of access for each accessor). At least one accessor

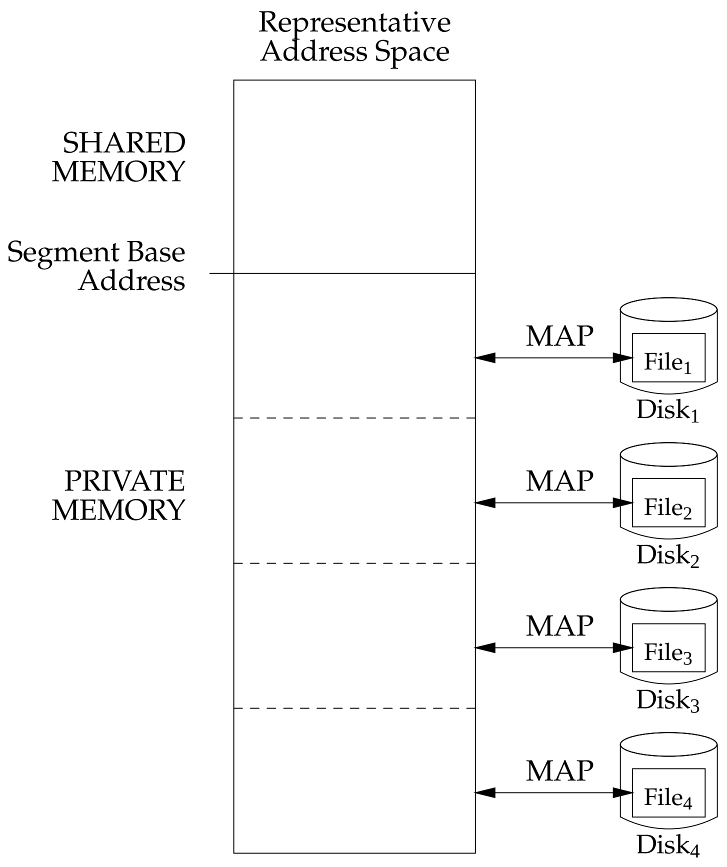


Figure 3.4: Storage Layout for a Partitioned File Structure

must be provided for each file structure definition. However, it is possible to have multiple accessors, each providing a particular form of access, e.g., initial loading, sequential access, keyed retrieval. An application can choose the particular accessor it wants to use for a given transaction depending upon the type of access needed and the functionality provided by the available accessors. It is also possible to have multiple access objects communicating with the same representative, which allows an application to have multiple simultaneous views of the same data, as illustrated in Figures 3.1 and 3.2.

In order to gain access to a file structure, an application program creates an instance of an accessor available for the file structure. The connection between accessor and file structure is established by passing the file object to the access object. The file object also contains a transient pointer to the representative for the file structure. The access object initiates the creation of the representative if it does not already exist. The execution of the termination code of the last access object for a file structure terminates the corresponding connection to the representative. Further details of the programming interface to the representative and the accessors are presented later in section 3.4.8.

Once instantiated, the access object can be used by the application to perform operations on the file structure by invoking the methods of the access object. For example, in order to read from the file structure, a call may be made to a method called read() provided by the accessor. The method read() communicates with the representative, to perform the desired operation. Depending on the particular kind of concurrency control requested, the declaration of the access object or individual access object method calls may block until it is safe to access the file contents.

Using the techniques discussed earlier, a library of several memory mapped access methods has been built. Currently, these access methods are used for comparison purposes among themselves and with traditional access methods, but they will ultimately provide database access-method designers with a starting point for construction of new databases. The programming interface for the μDatabase library, file structures and their access methods is presented in section 3.4.

3.1.5 Critique of μDatabase

A pointer to an object stored in a file structure can be dereferenced only within the address space of the file structure's representative. Therefore, dereferencing an inter-segment pointer may require the object to be copied out of the representative segment via shared memory. This copying involves an extra cost but is necessary, in general, to protect the integrity of the file structure.

Currently, μDatabase does not provide support for the virtual pointers of C++ because these pointers refer to data in the text segment of an executing program and the referential integrity of these pointers cannot be guaranteed across multiple invocations of the program. Virtual pointers are an important mechanism used by C++ to support inheritance and dynamically linkable code. Partial solutions to this problem have been reported in the literature (see [BDG93]) and can be applied to μDatabase.

Currently, μDatabase is based on a shared-memory design and is not distributed. [RD95b] contains a collection of papers that deal with the issue of distribution in persistent object stores. Further, an object store can be built on top of μDatabase but in its current form, μDatabase does not incorporate an object management system.

Many persistent systems make code as well as data persist due to the advantages that arrive from it. Some systems even provide support for making the current state of execution persistent, i.e., they allow for the current state of a program to be preserved for resuming at a later time. μDatabase, in its current form, does not deal with storing code in the persistent store; see section 6.3.1 for a brief discussion of the issues involved.

3.2 Comparison of μDatabase with Related Approaches

μDatabase is more closely related, both in scope and intent, to the software based approaches described in section 2.3.1 than to the architectural approaches described in section 2.3.2.

The basic memory mapping scheme used in Brown's stable store is identical to the one employed by a single μDatabase representative. The major difference lies in the fact that Brown's store is flat while the store in μDatabase consists of a collection of address spaces that all start at the same virtual zero. Thus, μDatabase provides support for multiple simultaneously accessible persistent areas for an application. Both schemes suffer from the same problems insofar as operating system support is concerned, e.g., less control over the time at which dirty pages are actually written out.

ObjectStore and μDatabase share many goals and objectives. Some of these include ease of learning, no translation[1] between the disk-resident representation of data and the main memory-resident representation used during execution, full expressive power of a general purpose programming language when accessing persistent data, re-usability of code and statically type-checked access to data. In ObjectStore, there is a limitation on the size of data a single transaction can access simultaneously because ObjectStore maps only portions of a file at a given time. In μDatabase, an entire file structure is mapped into an individual segment, which restricts the maximum 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 multiple 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 small virtual address spaces. In ObjectStore, all pointers embedded in a data page may need to be found and relocated when the page is mapped. This problem is non-existent in μDatabase because each database is mapped into a different virtual space and relocation of pointers is unnecessary. However, in μDatabase additional copying of data has to occur for inter-segment pointers from the file structure segment to the application segment. Some of this copying is unavoidable in any mapped system, including ObjectStore. Overall, I believe that any additional copying costs will be less than the total cost of doing relocation and, in general, is required to protect the integrity of the file structure, anyway.

The ObjectStore server is conceptually analogous to the μDatabase representative.

[1]Note that, unlike μDatabase, ObjectStore only achieves this goal in the special case when it can reload data into memory where the data was last manipulated; otherwise, pointers must be modified (swizzled).

There is some difference in the treatment of inter-segment pointers between ObjectStore and μDatabase. In ObjectStore, inter-database pointers can be short-lived (created and valid during the scope of a transaction) or long-lived; the former are implemented using normal virtual memory pointers whereas the latter are long pointers. It is possible for ObjectStore to use short-lived normal pointers because a transaction maps all databases it accesses into the same address space. μDatabase, on the other hand, maps each individual database into its own segment, and, therefore, it must use long pointers for *all* inter-database pointers except for one special file mapped into the application segment. Clustering of objects in μDatabase can be attained by means of simple storage management primitives.

Cricket is similar to μDatabase in its use of direct memory pointers that are always mapped to the same locations in the virtual address space, thereby eliminating the need for relocation. The fundamental difference between these two systems lies in their view of the address space. Unlike Cricket, μDatabase uses a structured rather than a flat virtual space. In Cricket, everything that an application ever needs to use must exist in a single database whereas μDatabase allows the application(s) to group related data in independent collections that can be used and shared as desired as long as there are few data inter-relationships; otherwise, the performance of the system degrades. μDatabase builds on top of the concept of files to provide multiple, individually sharable collection areas, and provides for sharing of information stored in files, with each different file being mapped into its own individual segment.

Texas and similar schemes are clear winners for applications that require extremely large persistent address spaces, larger than the address space of currently available hardware. For applications that do not require persistent address spaces that are larger than the virtual space of modern computers (32-64 bits), the simplicity and efficiency of μDatabase put it at a distinct advantage over the scheme used by Texas. In other cases, I believe that splitting a very large database into smaller databases of related objects is a very good approach simply for organizational reasons while eliminating the need for complicated swizzling mechanisms.

QuickStore uses memory mapping to create the mapping between virtual memory frames and disk pages in its client buffer pool. However, its use of memory mapping techniques is quite different from the EPD approach followed in μDatabase. QuickStore is essentially a pointer swizzling scheme that manages to avoid swizzling the pointers in some cases, as does ObjectStore. And like ObjectStore, it has to maintain all the information needed by a swizzling system. Also, since it is built on top of another storage system (EXODUS storage manager), it does not gain from all the benefits afforded by a fully memory mapped implementation.

Texas and QuickStore have many of the same problems as ObjectStore with regard to dynamic relocation and multiple accessible databases, and as such, the critique of ObjectStore is applicable here as well. Also, the need to relocate pages in these systems has the potential of seriously degrading performance for certain access patterns.

Systems described in section 2.3.2 make use of new hardware and operating systems and, as such, are usually not portable to other systems. μDatabase can run on any UNIX system that supports the mmap system call. Unlike Bubba, the current design of μDatabase is based on a multiprocessor shared-memory architecture and is not intended to be used in a distributed environment unless the environment supports distributed shared memory[SZ90b, WF90] which, I believe, will allow the current design to scale up to a distributed environment. The representative in μDatabase is quite similar to the Clouds object, except that the representative has its own thread of control while the Clouds object is totally passive. Also, the focus of Clouds was to build a new operating system, while μDatabase is an attempt to make memory mapping ideas available to database designers in the form of a toolkit that can be supported on any operating system that provides support for segmentation. Malhotra and Munroe make some of the same arguments that I have made for μDatabase when it comes to using virtual memory references to access objects. Their proposed long object identifier for the RS 6000 system is similar to the one employed by μDatabase: a long identifier consists of a file id and an address within the virtual space corresponding to the file. Like MONADS, the virtual store in μDatabase is not flat; it consists of a collection of independent areas as described in section 3.1.1. The segmentation addressing scheme employed in MONADS is similar to the one envisioned for μDatabase. However, due to the nature of the current implementation of μDatabase, it is not possible to make use of an addressing scheme identical to MONADS.

3.3 Parallelism in μDatabase

Generally speaking, there are two distinct forms of parallelism that can be exploited in database systems to achieve better performance and functionality. The two forms of parallelism, as depicted in figure 3.5, are:

Concurrent Retrieval of Data. The slowest link in accessing a file structure is transferring data to and from secondary storage. Secondary storage speeds range from 1,000 to 100,000 times slower than primary storage. Further, there does not appear to be any imminent technological advancements that will significantly reduce this ratio in speed between secondary and primary storage; in fact, the difference has only increased over the last decade. Therefore, the only approach that is currently available to improve performance is to partition data onto multiple secondary storage devices and access these devices in parallel. Disk arrays (RAIDs) are the most common implementation of this idea [PGK88, WZS91]. Once the data is partitioned, significant performance advantage can be obtained by partitioning indi-

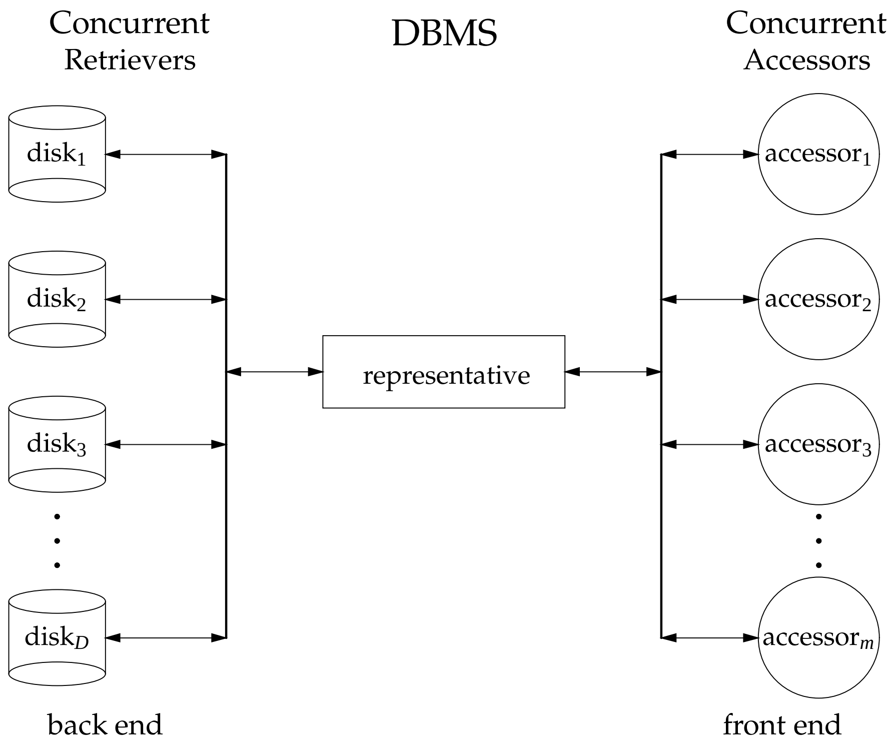


Figure 3.5: Two Forms of Concurrency in a File Structure

vidual queries and executing the resulting sub-queries in parallel.

Concurrent Accessors. Supporting concurrent access to a database improves its utilization, in the same way that multiprogramming operating systems improve utilization of a computer—by having several simultaneous requests to execute, it is possible to perform some of the requests in parallel if the requests access data in a non-conflicting manner. 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 same request, but the total 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 processors and the I/O devices. As well, such systems can quickly saturate because of the I/O bottleneck; therefore, attempts to achieve optimal parallelism in accessing persistent data are often fruitless.

Note that concurrent retrieval and concurrent access are orthogonal aspects of parallelism; systems exist that provide one or the other or both.

The question addressed in this section is how to use the EPD techniques for partitioning file structures across multiple disks and accessing partitioned data to achieve concurrent retrieval. The issue of concurrent accessors for EPD file structures is addressed in chapter 6. Further, the design, implementation and analysis of the parallelized multi-disk versions of three database join algorithms is presented in chapter 5.

3.3.1 Partitioned File Structures and Concurrent Retrievals

Concurrent retrieval attempts to deal with the CPU-I/O bottleneck by partitioning data across multiple disks and then accessing the data in parallel. A typical disk-array system partitions a file structure into several *stripes*, 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 issues is that the striping or partitioning algorithm should partition the data so that the access time for a particular access method is minimized and the I/O load is balanced across the disks. The partitioning algorithm can be application specific or general. While partitioning, balancing the I/O load does not imply a physically even distribution of data across several disks. The goal is to partition data in such a manner that during retrievals, the data units that need to be accessed are as evenly distributed as possible across disks.

In the discussion here and in chapter 4, the general concern is not about access to the index portion of the file structure. Normally the index is relatively small and highly accessed so that most of it remains resident in main memory, and consequently, does not contribute significantly to disk accesses. Nevertheless, the discussion can be easily extended, if necessary, to include the index portions of file structures.

A B^+-Tree and an R-Tree based on the EPD approach were modified so that insert operations automatically partition the data across several disks and the query operations retrieve data from multiple disks in parallel. The partitioning of the two file structures is discussed in chapter 4, while some general issues are presented below.

3.3.2 Query Types and Parallelism

Exact match queries (e.g., retrieving a single record to match a specified key) usually cannot take advantage of partitioning because the index is already in memory and there is only one disk access required to service the request. Note, however, that in some disk array based implementations, individual data records are also split across multiple disks. Retrieving a record in such a system involves a parallel reading of the individual pieces of the record from multiple disks.

Range queries, on the other hand, can exploit partitioned file structures to perform parallel retrievals if the data needed for responding to a query is distributed across multiple disks. A specified range query can be broken down into multiple smaller sub-queries that can be executed in parallel inside the DBMS. By dividing the original query so that the resulting sub-queries access data on different disks, the overall query can be processed much faster. Similarly, if the file structure is aware of the access pattern for its data blocks, it can employ pre-reading techniques to increase the parallelism in reading blocks of data from secondary storage.

Several different kinds of range queries common in database applications were designed, implemented and evaluated using μDatabase to demonstrate the effectiveness of parallelism in a memory mapped environment based on the EPD approach. In particular, the following three types of range queries were investigated:

1. A range query, called a <K,K> range query, described by two key values K_1 and K_2. All records with key values between K_1 and K_2 inclusive are returned as a result of the query. Neither K_1 nor K_2 need be in the database.

2. A range query, called a <K,C> range query, described by a key value K and a signed integer C. The query result consists of $|C|$ consecutive records starting with the record with key value K. The direction traversed from K is specified by the sign of C. A positive C indicates traversal in the order that the keys are sorted in the file structure—for ascending order, values greater than K are returned and for descending order, values less than K are returned. A negative C causes traversal to occur in the opposite direction.

3. A range query, called a <K,C,C> range query, described by a key value K and two positive integer values C_1 and C_2. C_1 records with keys before and C_2 records with keys after the record with the key value K are returned.

In general, the records returned from a range query are unordered.

3.3.3 Range Query Generators or Iterators

In μDatabase, an application program performs a range query by using a programming construct called a *generator* or an *iterator* [LAB+81, Sha81, RCS93]. A typical range query generator provides at least two methods, namely, an initialization routine and an iterative operator. The initialization routines are used to specify a range query. Once the generator has been initialized, each successive invocation of the iterative operator returns another object from the result of the specified query until all records have been returned. Further discussion of generators is presented in section 3.4. The generators developed for the partitioned file structures perform parallel retrieval of data needed to process the specified range query. Each generator provides an iterative operator >> that can be invoked to retrieve, one record at a time, the result of the specified range query. As an example, consider the following code fragment: This code fragment initializes an instance of a B-Tree generator object, gen, to process a <Key,100> range query on the B-Tree structure referred to by the access object, accessObject. Each successive call to the >> operator of gen returns a pointer to another record from the result of the <Key,100> range query. When all the records have been exhausted, the NULL pointer is returned, which causes the loop to terminate.

```
// <K,C> range query application
for ( BTreeRangeQuery gen(accessObject, Key, 100); gen >> rec; ) {
        // process rec
}
```

3.3.4 Generic Concurrent Retrieval Algorithm

Once a file structure has been partitioned, the issue of accessing it while employing as much parallelism as possible can be addressed. The main concern is increasing the degree of parallelism at the back end of the DBMS. A concurrent retrieval algorithm can take advantage of the potential parallelism, but only if sufficient hardware is available. First, multiple disks must be accessible in parallel, which implies that disks must be capable of concurrent seeks. Second, if multiple processors are available, they must be capable of performing any file-structure administration in parallel with the application program processing the results of the range query. The Sequent Symmetry computer configuration described in section 4.1 satisfied both of these hardware requirements.

The following algorithm was developed to perform concurrent retrievals on EPD file structures. The algorithm is generic in nature and can be easily specialized for performing concurrent retrievals on any arbitrarily complex indexed file structure. Recall, in μDatabase, a file structure is a single object with one representative. When the file structure is partitioned across D disks, the file structure segment is also divided into D contiguous partitions (see section 3.1.3) and the *D* file structure partitions are memory mapped, one after the other, into the divided segment by the representative. Then *M* kernel threads (UNIX processes in μC++) are created to share the representative segment containing the mapped file structure partitions; *M* is a control variable specified by the experimenter. The M kernel threads execute $D+1$ light-weight tasks that are created to perform retrieval requests. *D* of the tasks, called retrievers, copy records from the private memory of the representative to shared memory and the remaining task is called the *leaf retrieval administrator* (LRA) as depicted in figure 3.6. When a generator object is instantiated for executing a specified range query, the generator allocates a buffer area of specified size to be shared between the specified accessor object and its representative.

In addition, another task, called the *file structure traverser* (FST), is created. The FST organizes and maintains the allocated buffer space as a sharable buffer pool. As well, the FST traverses the file structure index and generates a list of pointers to leaf nodes that contain all the records satisfying the query being executed, without actually dereferencing any leaf node pointers. For each pointer in the list, the FST communicates with the LRA specifying the leaf node pointer, number of records needed from the referent leaf node (obtained from an appropriate index entry), and a handle for the buffer pool. The LRA partitions and distributes the FST requests among the retriever tasks. A retriever task dereferences the specified pointer causing the referent leaf node to be retrieved from disk, allocates a buffer from the buffer pool, and copies as many records from the leaf node as will fit into the buffer. The last step is repeated until all the selected records in the leaf node have been copied into the buffer pool and then the retriever task waits for

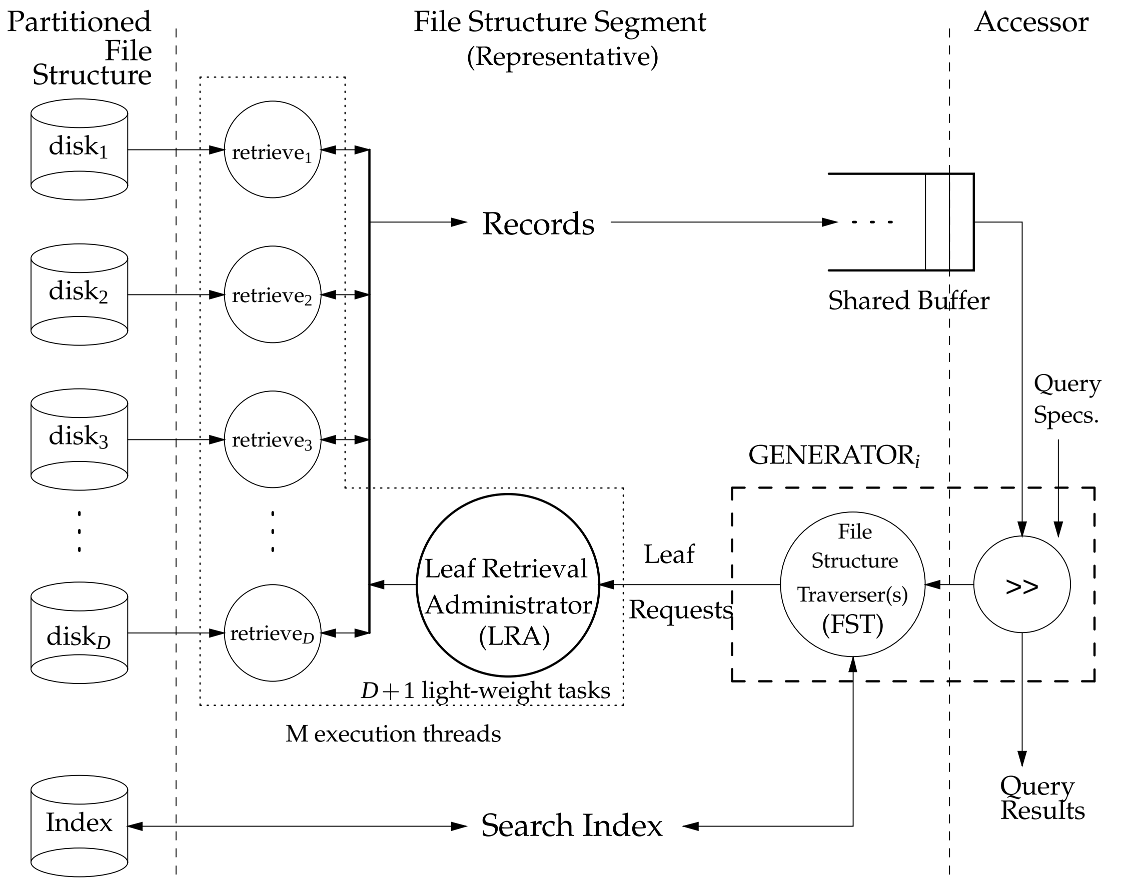


Figure 3.6: Generic Concurrent Retrieval Algorithm

more work from the LRA. This design ensures that the only bottleneck in parallel processing of the specified query is the speed with which the bounded buffer can be filled and emptied. In general, an application program can keep ahead of a small number of disks (1-7 disks), depending upon the complexity of data processing involved, because the data processing time is significantly less than the I/O time.

A retriever task may or may not be tied to a particular disk. In EPD terminology, being tied to a particular disk means accessing only that part of the address space that contains the mapping for the disk. When a retriever task is not tied to a particular disk, the task can be asked to process any leaf node by the LRA. In this strategy, the LRA maintains a single FIFO queue of requests from the FST. When any of the retrievers is free, the LRA passes on the FST request at the front of the queue, causing the leaf nodes to be processed in FIFO order. The problem with this scheme is that parallelism is compromised when several consecutive leaf node requests in the queue are for the same disk. In this case, many or all of the retriever tasks may block on a single disk while other disks (and their controllers) remain un-utilized. Therefore, it is usually more efficient to tie each retriever task to a particular disk by making the LRA create D queues, one for each disk. Upon arrival at the LRA, a leaf retrieval request from the FST is queued on one of the D queues depending upon the disk containing the leaf node to be processed. Each retriever task processes requests from one queue only. In this scheme, the throughput is directly controlled by the slowest disk in the chain, resulting in a near optimal solution. This observation was verified by running experiments on both of these strategies (see chapter 4).

Finally, double buffering can be exploited by tying more than one retriever task to each disk so that while one retriever is processing data for a leaf node, another one is reading the next leaf node to be processed from the same disk. Note that in order to gain from this strategy, the number of kernel threads must be at least the number of retriever tasks, because in a memory mapped system, the kernel thread causing a page fault blocks until the faulted page has been brought into memory.

The generic concurrent retrieval algorithm described above can be used for different indexed file structures by specializing the FST and the components of a retriever task responsible for processing of individual leaf nodes to extract information.

3.4 Programming Issues and Interfaces

A memory mapped file structure should be able to use all the capabilities of the implementation programming language. This chapter illustrates some of the ways in which the EPD approach to memory mapping achieves this goal in μDatabase.

3.4.1 Polymorphism

The polymorphic facilities of C++ can be applied to generalize the definitions of file structures and to allow reuse of the file structure's implementation by other file structures. Generalization allows existing file structure code to be specialized by users and reuse allows file structure designers to write new file structures in a shorter time with fewer errors (on the assumption that the old file structure is debugged). The desire to generalize and reuse code arose during the construction of the file structures used in the experiments described in chapter 4. During this process, two issues were noticed:

1. A file structure and its access methods are usually polymorphic, that is, they can handle a number of different record (and possibly key) types. However, this polymorphism is usually achieved at the loss of type safety by dealing with blocks of untyped bytes. Some systems [GR83, CLV91] provide dynamic type/format checking to tackle this problem. I believe that the interface to an access method of a file structure should be statically type checked to permit early detection of errors and efficient code generation (as in E [RCS93]). Therefore, there is a need to be able to generalize a file structure and its access-method interface across the record (and possibly key) type.

2. Many file structure algorithms incorporate both a data structure and a storage management scheme, e.g., a B-Tree is an N-ary tree with fixed or variable sized

data records stored in uniform sized nodes. Storage management deals with untyped blocks of bytes of a segment and, therefore, it is not possible to perform static type-checking at the storage management level [BDZ89]. Among different file structures, there is a significant amount of duplicated code dealing with storage management that can be abstracted out and reused. By factoring out the storage management aspect, it is possible to deal with the data structure independently of storage management, which can be encapsulated into a separate tool that can be used in varying ways with different data structures. In μDatabase, only file structure designers work at this level; users usually work at a statically type-safe level.

The rest of this section describes work done to achieve the above two goals. Object-oriented programming techniques are employed, but languages with other forms of polymorphism, e.g., parametric polymorphism, are equally applicable. C++ [ES90] is used as the concrete implementation programming language. A general knowledge of object-oriented programming is assumed throughout this discussion. In addition, a basic familiarity with C++ is assumed, although most of the examples are self-explanatory.

3.4.2 Generic File Structures and Access Methods

From a code reuse standpoint, the code to manage a file structure is largely independent of the record (and possibly key) types. A simple example is an ordered linked-list. The linked-list data structure is independent of the type of elements stored in a node of the list, requiring only assignment on the record type if stored by value, and comparison on the key type. However, the access-method routines used to modify a file structure need to be specialized in the record (and possibly key) type so that static type-checking is possible. Therefore, an access method needs to be generic in these fields and possibly generalized in other aspects. To accomplish these design requirements, I initially used a preliminary version of the C++ template facility [FON90] to define generic file structures and their access methods; the code was subsequently changed to use the standard C++ templates when they became available.

The template facility allows all components of a file structure to be statically type-safe. A user application specializes a generic file structure by the data stored in it. For example, a B-Tree file structure is declared as follows:

```
BTreeFile<int,float> db( "testdb", greater );
```

which creates a B-Tree stored in a UNIX file named `"testdb"`, with `int` keys and `float` data records, and the B-Tree is structured by a user supplied key comparison routine `greater()`. Generic linked-list and B-Tree file structures are presented later in this chapter to demonstrate the basic concept, and I have applied this approach to construct R-Tree, general N-ary Tree, generalized graph and other file structures.

3.4.3 Storage Management

One of the most complex parts of any data structure is efficient storage management. In fact, much of a file structure designer's 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 section 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 time it takes to construct a complex file structure. The details of the programming interface to the memory management tools are presented and then a tutorial in which a simple persistent linked-list data structure and a generic B-Tree are built using the tools.

To a large extent, this is the approach of many garbage collection systems that provide system- or program-wide storage management [Wil91b]. The criterion used to judge the success of this approach is whether an independent facility for storage management can provide performance that is close to traditional schemes that incorporate storage management directly with the data structure.

Memory Organization

In the EPD approach, memory is conceptually 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 user's perspective, although it might be implemented with non-contiguous pages. An address space is supported by hardware and managed by the operating system.

segment is a contiguous portion of an address space. Usually, there is a one-to-one correspondence between an address space and a segment, but it is possible for an address space to be subdivided into multiple segments, e.g., with segmented hardware addressing. In μDatabase, a segment is also mapped onto a portion of the secondary storage. A segment is supported by hardware and managed by the address-space storage manager (the representative).

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

Since μDatabase is capable of creating multiple mappings simultaneously (see section 1.2.2), 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 no longer adequate for the multiple segments in μDatabase for the following reasons:

1. When multiple segments are present simultaneously with each having its own heap, a target segment must be specified each time a memory allocation request occurs.

2. The programming language 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 storage overflows and underflows, e.g., when a node in a B^+-Tree fills up during insert, the data structure requires the creation of a new node and the movement of a subset of data from the old node to the new one. The storage management facility must be able to accommodate application specific actions for these cases. The basic concept of using multiple independent heaps has been employed by many other systems, e.g., the area variables in PL/I [IBM81].

To achieve the above, μDatabase memory management facilities are provided in the form of generic *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 the object implicitly performs the desired management on its segment. Since the different managed areas are controlled by independent 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.

3.4.4 Nested Memory Structure

All segments are nested in an address space. All heaps are nested in a segment. Further, since a heap is simply a block of storage, it is possible for heaps to be nested within one another. This structure is illustrated in Figure 3.7. The form of an address for each level may depend on the storage management scheme at that level.

In theory there is no limit on the depth of nesting of heaps, but in practice there is a limit imposed by the number of bits in the address used to reference data in the lowest level heap. In general, a small number of sub-heaps are sufficient for most practical problems; see [BZ88] for a further discussion of expressing nesting.

3.4.5 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. 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 almost non-inexistent in commercial systems.

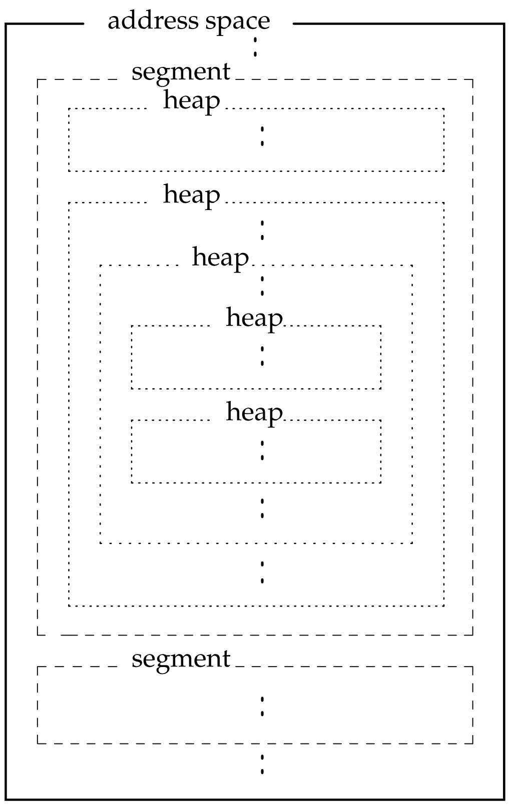


Figure 3.7: Nested Memory Structure

3.4.6 Segment Tools

Segment 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. The capability to map multiple disk files is discussed in chapters 3 and 4 where it is used for partitioned file structures. In this chapter, the focus is on mapping a single file into a segment. All segment capabilities are provided through the representative for a file structure. The programming interface for these facilities is discussed next.

3.4.7 μDatabase Programming Interface

An application program that uses μDatabase consists of the following basic modules that are linked together to form an executable program.

1. **μDatabase library:** This library is the core module that contains the basic implementation of the representative and the accessor, both of which are generic and provide all low-level support needed to employ the EPD approach to memory mapping for building file structures.

2. **File structures and their access methods:** A specific file structure and its access methods are implemented as classes that inherit from the specialized versions of the base representative and the accessor respectively. An alternative to inheritance is to make a specialized instance of the representative a member object of the file structure class. It is the responsibility of the file structure designer to provide definitions and implementations for the file structure and its access methods. A library consisting of several different file structures, both sequential and parallel, has been developed as part of this work.

3. **Application program:** In order to manipulate data stored in a file structure, the application program declares instances of the file structure and the accessor objects and uses the interface provided by the accessor class to perform operations on the file structure.

Code in the μDatabase library as well as file structure code is executed in the representative segment, while application program code is executed in its own address space. With the use of *wrappers* provided in the μDatabase library and described in section 3.4.8, application code can be executed in the context of the representative segment.

3.4.8 Representative Interface

The representative interface in the μDatabase library is provided by three related classes: Rep, RepAccess and RepWrapper.

Class Rep

Rep is the representative data structure. It is responsible for mapping and un-mapping files to/from segments, and controlling the size of the segment, which determines the size of the file. The basic public interface of Rep is shown in Program 3.1; some details have been omitted to simplify the following discussion.

```
uMonitor Rep {
  public:
     virtual void *start();              // starting address of mapping
     virtual int size();                 // current size of mapping
     virtual void resize(int size);      // resize mapping
     virtual bool created();             // UNIX file created by this rep?
     void createExtraProcs(int Num);     // add extra virtual processors
     void deleteExtraProcs();            // remove extra processors
};
```

Program 3.1: Basic Representative Interface

uMonitor is a μC++ artifact that declares a *monitor class*. Briefly, a monitor class is a normal C++ class except that concurrent execution of the public member routines of a monitor class is serialized (see [BFC95] for further details on monitors). The member routine start returns the starting address of the mapping, the segment base address, which is currently 16M. 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 specified as its argument. The routine created returns true if the requested UNIX file was created by the current representative, and false if the file was present before the representative was created. The routine createExtraProcs creates extra virtual processors that are attached to the address space of the representative segment. Extra virtual processors are useful for increasing parallelism and can be destroyed when not needed by invoking the companion routine deleteExtraProcs.

Class Rep is not intended to be instantiated directly by the file structure code, which is why it has no public constructors. Instead, a representative is created indirectly through class RepAccess, which may create a new instance of Rep for the file structure, if one does not exist yet, or use an existing one. Thus, the only way to control file mapping and un-mapping 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 representatives and file structures in an application.

Class RepAccess

The basic interface to RepAccess is shown in Program 3.2. RepAcess is generic in the type of a specialized representative that is created by inheriting from Rep. Usually, a specialized representative is not needed and RepAcess is specialized with class Rep itself. The constructor's parameter for RepAccess is the name of a UNIX file for a file structure or a list of the names of UNIX files comprising a partitioned file structure. Upon the creation of an instance of RepAccess, the global representative table for the application is searched in an attempt to locate an active representative for the file structure specified by the given UNIX file(s). If a representative is present, the corresponding file structure is already mapped and a new mapping is unnecessary. A pointer to the existing representative is stored in the newly created RepAccess instance, the representative's use count in the global table is incremented, and the creation is complete. If, however, no representative is found for the file structure, an instance of class RepType is created and entered into the representative table. If the file structure does not already exist in the UNIX file system, it is created and initialized. The file structure is always mapped at the same starting location, the segment base address (see section 3.1.3). However, an advanced facility is provided for specifying the starting address for the mapping. This facility must be used with caution and only when no dereferencing of embedded pointers is to take place during execution.

```
template<class RepType> class RepAccess {
  public:
     RepAccess( char *filename );
     RepAccess( char *filename[ ], int NumPartitions);
     void *start();                  // starting address of mapping
     int size();                     // current size of mapping
     void resize(int size);          // resize mapping
     int created();                  // UNIX file created by this rep?
};
```

Program 3.2: Basic Access Class Interface

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

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

2. A RepAccess object cannot be created from inside the persistent 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 RepAccess constructor 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, which means that the UNIX backing file cannot be renamed once it is created by μDatabase. This limitation is quite unacceptable.

Because of the above problems, the only access to mapping control for objects *within* the persistent area is by a direct pointer to the Rep structure.

Organization of Representative and Access Classes

After the representative is created (indirectly by an accessor object), 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 for the following reason. The 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 instantiated mapped file structure provides the back pointer for storage managers in the segment to communicate with the representative's storage manager. For abstraction, this pointer is contained in an instance of a pre-defined μDatabase class, RepAdmin, which is stored at the beginning of the segment by convention.

Figure 3.8 shows the organization of representatives and their access classes and segments. The representatives are chained together to allow them to be searched when an access object is created to see if there is already an active representative for the specified file structure. Notice, also, a pointer from the segment to the representative. Having a pointer from persistent memory to 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 or destroys the representative object. However, this scheme works because the representative pointer is dynamically initialized on the first access to the corresponding persistent area during an application's execution, i.e., when the representative object is created and the persistent area is mapped, the representative segment pointer is initialized. Once the file structure has been unmapped, the representative segment pointer at the beginning of its persistent area becomes meaningless.

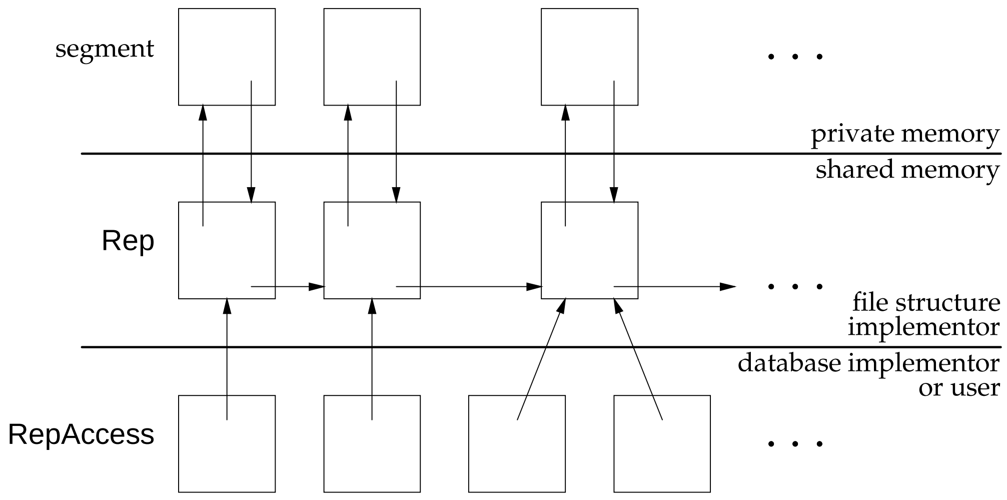


Figure 3.8: Organization of Representatives

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

Class RepWrapper

Since the file structure is mapped into the representative's private memory, user application code does not have direct access to the contents of the file structure; the application code only has access to shared memory. The class RepWrapper, with an interface shown in Program 3.3, provides the mechanism to allow application code to access private memory for a particular representative's segment.

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

Program 3.3: Basic Wrapper Interface

RepWrapper is a *wrapper* class and, therefore, does not have any member routine of its own; all actions of the wrapper class are carried out by the constructor and destructor of the wrapper. When a wrapper is declared inside a program block, both of the wrapper's operations are guaranteed to be performed, even if the block is terminated by an exception. The RepWrapper constructor takes an instance of RepAccess as an argument, which indirectly refers to a representative's address space and any segment(s) mapped into it. The main action taken by the constructor is to reset the current segment pointer, to a value corresponding to the specified RepAccess object, from which addresses are implicitly related. In effect, the current thread of control is migrated to another segment in which addresses have a new meaning, except for those addresses that refer to data in the common shared area of each segment. However, since μDatabase executes on an architecture without segmentation capabilities, switching the current segment pointer is achieved indirectly by explicitly migrating the current thread of control to another UNIX process, which has a different page table and, hence, a different mapping for the private part of the address space. In practice, the UNIX process executing the thread stops and the other UNIX process continues executing the thread's code. The destructor executes the reverse action, i.e., it migrates the current thread of execution back to the address space where the constructor was executing. The cost of either operation is a light-weight context switch and possibly a heavy weight context switch if the UNIX process associated with the destination address space is currently blocked.

As soon as an instance of RepWrapper is created, the specified representative's address space becomes accessible to the executing program in addition to the already accessible shared memory; the duration of accessibility is the life 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 process cannot have direct access to two or more mapped files 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 shown in Program 3.4. This convention further ensures that the wrapper's actions occur as the first and last operations of a program block.

3.4.9 Heap Tools

As mentioned earlier, a segment has no inherent facilities to manage allocation and deallocation of its memory. This section discusses heap tools that can be used to manage a segment's memory. If none of the tools presented in this section is appropriate for a

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

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

} // back to previous address space and rep's address space is inaccessible
```

Program 3.4: Using a Wrapper

given application, it is possible to build specialized heap management tools.

Storage Management Schemes

While there are a large number of storage management schemes possible, three basic schemes are provided in μDatabase. The first version of these schemes was implemented by A. Wai as part of his M.Math essay [Wai92]. The schemes presented below 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 the malloc and free routines of C [KR88].

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 locations 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.

These three storage management schemes should cope with most application demands. Should special needs arise, special purpose memory management schemes can be created and easily integrated into μDatabase, possibly reusing code from the supplied schemes.

Nesting Heaps

With many applications, a segment has to be subdivided into multiple heaps that are managed independently of 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 to deal with these heaps. The segment then becomes an upper level heap with dynamically allocated subheaps nested inside.

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 3.10 on page 116 illustrates this storage structure.

Overflow Control

When a heap fills, a generic storage manager can sensibly take three actions:

1. enlarge the heap by adding additional storage at the end of the contiguous heap. However, when there are multiple heaps at a particular nesting level, this may necessitate moving one or more other heaps.

2. allocate a new heap which is larger than the existing heap, copy the contents of the old heap to the new heap, and delete the old heap.

3. allocate a new heap and copy some portion of the contents to the new heap so that each heap has some free space. This action results in two independent heaps that must be managed.

Moving heaps or their contents requires finding and relocating pointers to data being moved. 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 to deal with heap overflow. The following are two examples of such data-structure specific actions. When a B-Tree node fills during an insert operation, an additional node is allocated and some of the contents of the old node are 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 intelligence to deal with overflow. All expansion objects are derived from a special *expansion base class* and one must be provided to the generic memory manager when the latter 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 the situation.

Note that heap underflow can also be dealt with in a similar manner, but is not discussed here.

Expansion Object

As mentioned earlier, a basic memory manager does not deal with heap overflow. In order to handle overflow, a specialized heap expansion definition must be created to perform application specific overflow action. The class uExpand, shown in Program 3.5, is the interface between the memory manager and the overflow handler.

```
class uExpand {
  public:
    virtual bool expand( int ) {              // default expand routine
        cerr << "uExpand::expand(" << this <<
            "): no expand routine defined." << endl;
        uExit( -1 );
    }
};
```

Program 3.5: Heap Expansion Object

The member routine expand is called from within the memory manager whenever more storage is needed due to a heap overflow. The routine takes an integer argument that specifies the amount of additional storage requested. A file structure specific expansion class must derive from class uExpand and redefine the expand routine to perform the desired overflow action, adding more private variables to the class definition as necessary. The expand routine's return code controls the future action of the memory manager. If the expand routine returns false, the allocation process fails. If the expand routine returns true, the memory manager re-attempts to allocate memory out of the expanded heap and fails if there is still insufficient storage after the expansion.

As shown in program 3.5, ordinarily the expand routine would be defined as a C++ virtual routine so that it can be replaced by specialized derived expansion classes. However, expansion objects associated with persistent data structures are stored in the persistent area together with the data they manage. As mentioned in Section 3.1.5, virtual members are currently not supported in a persistent area. Consequently, the expand routine is defined as a regular member routine and specialized using the generic (template) facility in C++. The memory managers, which invoke the file structure specific version of expand, are parameterized based on the file structure expansion class. The interface of the generic uniform memory manager is shown in Program 3.6.

```
template<class T> class uUniform {
  public:
     uUniform( void *mstart, int msize, T &expn, int usize );
     void *alloc();
     void free( void *p );
     void sethsize( int newsize );
};
```

Program 3.6: Interface for Uniform Storage Manager

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 specialized expansion object and usize is the allocation size for the uniform heap. Once initialized, the member routines alloc and free are used to allocate and free uniform sized blocks of storage in the heap. The member routine sethsize is used to inform the memory manager of a change in heap size and is intended to be invoked by the expansion object.

To create a specialized uniform memory manager for use in a file structure, a specialized expansion class is defined first, as shown in Program 3.7. This program creates a uniform memory manager to manage storage that starts at the beginning of the persistent area referred to by the access object repacc, is initially 1000 bytes in size, is allocated in 100 byte blocks and overflow is handled by myExpObj.

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

myExpand myExpObj;                // create specialized expansion object

// create and initialize the storage manager
uUniform<myExpand> myUniSM( repacc.start(), 1000, myExpObj, 100 );
```

Program 3.7: Specializing a Uniform Storage Manager

For more flexible storage management, a variable or dynamic memory manager may be required. The interfaces of these two parameterized classes are shown in Program 3.8. The constructors takes three arguments mstart, msize and expn, which specify the starting address, the initial size of the heap and the expansion object, respectively, just as they do in the uniform manager constructor. Further, member routines alloc, free and sethsize perform the same functions as 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 *. Specialized variable and dynamic memory managers are created in the same manner as specialized uniform managers described earlier.

```
template<class T> class uVariable {
  public:
     uVariable( void *mstart, int msize, T &expn );
     void *alloc( int size );
     void free( void *fb );
     void sethsize( int newsize );
};

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

Program 3.8: Interfaces for Variable and Dynamic Storage Managers

3.4.10 Linked List Example

This section illustrates basic techniques and tools for constructing a persistent file structure by building a generic singly linked list with nodes containing a variable length string value. Note that for a more flexible linked list, the type of the data stored in the nodes can also be parameterized.

List Application

At the application level, the file structure designer makes available four data structures: one to form the nodes of the list (listNode), one to declare a persistent linked list (list<class nodeType>), one to access it (listAccess<class nodeType>) and one to traverse it (listGen<class nodeType>). Program 3.9 shows a simple application program using the persistent linked list.

There are several distinguishable components of the persistent linked list application. First, there is a definition of the specialized list node, myNode, which must inherit from

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

char *next_string() { ... };                  // a random string generator

void process_string(char *p) { ... };         // modify contents of string

void uMain::main() {
     list<myNode> l( "abc" );                 // create persistent list
     listAccess<myNode> la( l );              // open list

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

     listGen<myNode> gen;                     // used to scan through list
     myNode *p;
     char name[MAX_STRING_LEN];               // buffer space for strings

     for ( gen.over( la ); gen >> p; ) {      // modify the list indirectly
          la.get( p, name );                  // copy out information
          process_string( name );             // modify contents as needed
          la.put( p, name );                  // copy information back
     } // for

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

Program 3.9: Linked List Example

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 the actual storage for the string is allocated as each node is created. Second is the creation of the persistent list file structure, l, with UNIX file name "abc". Third is the declaration of the access class object, la, for persistent list object, l. Both the persistent list class and its access class are generic in the type of the node so that all accesses to the two classes can be statically type checked.

The next three loops add, update and remove nodes using the access class routines add, get and put, and remove, respectively. The generic linked-list generator, listGen, returns a sequence of pointers to nodes stored 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.

3.4.11 Linked List File Structure

Figure 3.9 shows all the list data structures created and their inter-relationships in the persistent linked list file structure.

List Node

The abstract class, listNode, shown in Program 3.10, contains the fields needed by each node in a linked list to relate the data. The member routine next allows indirect access to the link field.

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

Program 3.10: Abstract List Node Class

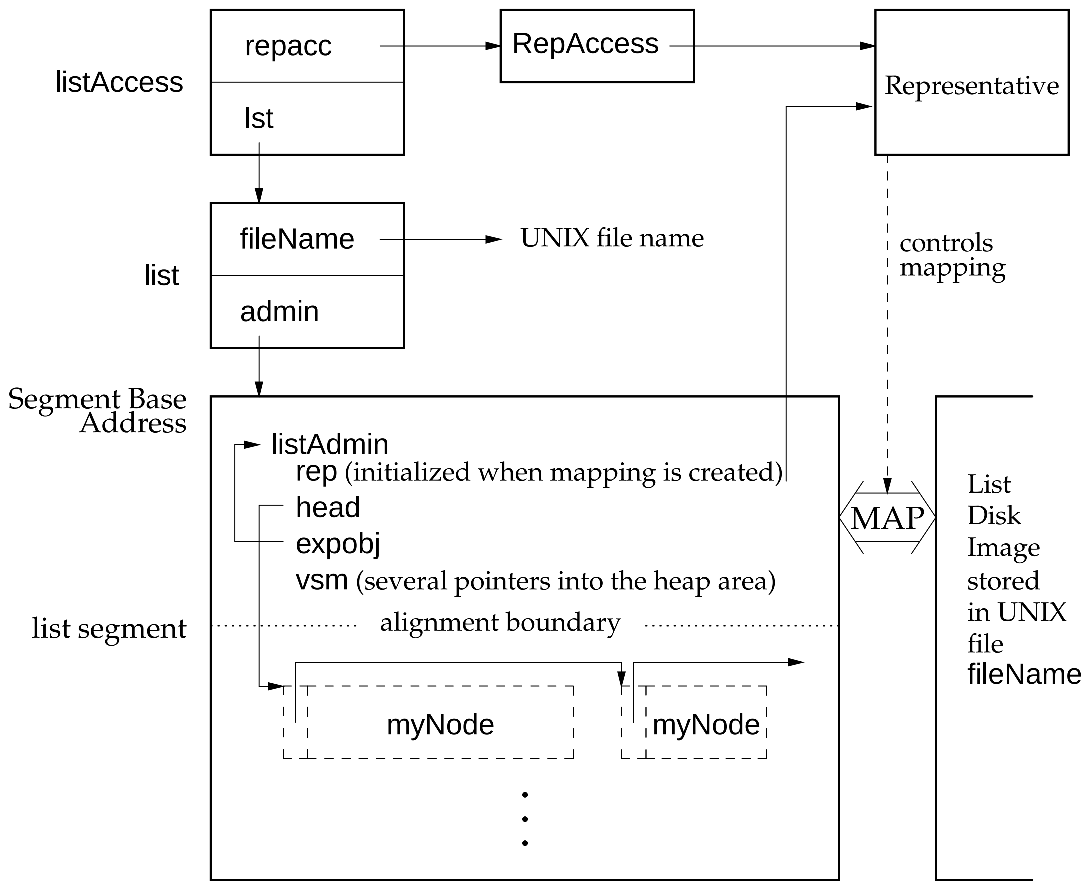


Figure 3.9: Linked List Storage Structure

Administration

Information pertinent to a particular linked list, e.g., the pointer to the head of the list and the memory management information for the persistent area, must outlive the application program that creates the list, i.e., information other than the linked list data itself must persist. Therefore, this information must be stored in the same persistent area as the linked list itself. By convention, all such persistent administrative information is encapsulated into an *administrative object* stored at the beginning of the persistent area or the segment. Furthermore, the administrative type *must* inherit from the pre-defined abstract type RepAdmin (see section 3.4.8).

The code for the list administrative class is presented in Program 3.11. The class contains a pointer, head, to the root of the persistent linked list, the expansion object for the persistent area for the list, and the variable memory manager that manages the persistent area. As discussed in section 3.4.8, the representative initializes a pointer to itself at the beginning of its persistent area. This pointer can be accessed from subclasses of RepAdmin through the protected variable rep and is the reason for the convention requiring the administrative class to inherit from RepAdmin and for the administrative object to be stored at the beginning of the persistent area or segment. The constructor for the administrative class takes an integer, indicating the initial heap size, as an argument, initializes the expansion object, expobj, the variable memory manager, vsm, and then sets the list root 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, thereby providing type-safe access to the routines of the linked list file structure.

Expansion Class

The expansion class for the linked list is defined in Program 3.12. The constructor initializes a reference to the administrative object so that the expansion object can access both the containing storage manager, listAdmin::vsm, and the list representative, listAdmin::RepAdmin::rep. The member routine expand first extends the persistent area by calling the representative's resize routine. It then informs the variable memory manager of the change by calling its sethsize routine and finally, returns true indicating that

```
template<class T> class listAdmin : public RepAdmin {
    friend class listExpType<T>;              // give access to expansion class
    friend class list<T>;

    T *head;                                  // root node of the list
    listExpType<T> expobj;                    // expansion object to extend list memory
    uVariable< listExpType<T> > vsm;          // variable sized list storage manager

    T *alloc( int size ) {
        return (T *)vsm.alloc( sizeof(T) + size );
    } // listAdmin<T>::alloc

    void free( T *p ) {
        vsm.free( p );
    } // listAdmin<T>::free
  public:
    listAdmin( int fileSize ) :
            expobj( *this ),
            vsm((void *)this + sizeof(listAdmin<T>),
                fileSize - sizeof(listAdmin<T>),
                expobj
            ) {
        head = NULL;
    } // listAdmin<T>::listAdmin<T>
}; // listAdmin<T>
```

Program 3.11: List Administration Class

the original allocation operation should be re-attempted[2].

File Structure Class

The purpose of the list file structure class is to establish a connection between the executing 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. Program 3.13 contains the definition of the list class, list.

[2]An additional error check is required to deal with failure to obtain sufficient storage from the segment, but has been removed for clarity.

```
template<class T> class listExpType : public uExpand {
    listAdmin<T> &admin;
  public:
    listExpType( listAdmin<T> &adminobj ) : admin( adminobj ) {
    }; // listExpType<T>::listExpType

    bool expand( int extension ) {
        // extend the segment
        admin.rep->resize( admin.rep->size() + extension );
        // inform the storage manager
        admin.vsm.sethsize( admin.rep->size() - sizeof(listAdmin<T>) );
        return true;
    } // listExpType<T>::expand
}; // listExpType<T>
```

Program 3.12: List Expansion Class

The constructor of list takes two arguments. The first one indicates the name of the UNIX file that contains the persistent linked list. The second argument is optional and indicates the initial size of the persistent storage that contains the linked list nodes, if the file structure is to be created; otherwise this parameter is ignored. The constructor makes a copy of the UNIX file name in shared memory, establishes a mapping to the file by creating a RepAccess object, makes the resulting segment accessible by creating a RepWrapper, obtains a pointer to the beginning of the segment to use as the location of the administrative object, and checks to see if the file was created on access. If the file has been newly created, the segment is extended to the specified size and the administrative object is created at the beginning of the segment, which initializes itself through its constructor, creating an empty list.

The private member routines first, add and remove manipulate the list nodes. These routines are in the list object so that the list can be modified by other objects within the persistent area. The first routine returns a pointer to the beginning of the list. The add routine calls the variable storage manager in the administrative object to obtain storage for a node of type myNode that can contain the string parameter, copies the parameter

```
template<class T> class list {
    friend class listAccess<T>;
    friend class listGen<T>;

    char *fileName;                              // UNIX file containing the list
    listAdmin<T> *admin;                         // administrator for the list segment

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

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

    void add( char *value ) {                    // add name to the beginning of the list
        T *newNode = admin->alloc( strlen( value ) );
        . . . // initialize newNode with value and put at head of list
    } // list<T>::add

    void remove( T *p ) {                        // remove node from list
        if ( p == admin->head ) {                // remove first node
            admin->head = (T *)p->next();
        } else {                                 // remove node in list
            . . . // search for and remove node p from list
        } // if
        admin->free( p );
    } // list<T>::remove

  public:
    list( char *name, int initSize = 4 * 1024 ) {
        fileName = new char[strlen(name) + 1];       // allocate storage for file name
        strcpy( fileName, name );                // copy file name
        RepAccess<Rep> repacc( fileName );       // map file
        {
            RepWrapper wrapper( repacc );        // migrate to file segment

            admin = (listAdmin<T> *)repacc.start(); // admin object at start of segment
            if ( repacc.created() ) {            // file created when mapped ?
                repacc.resize( initSize );       // initialize segment
                new(admin) listAdmin<T>( repacc.size() ); // initialize admin object
            } // if
        }
    } // list<T>::list

    ~list() {
        delete [] fileName;
    } // list<T>::~list
}; // list<T>
```

Program 3.13: Linked List Class

into the new node, and chains the node onto the head of the list. The remove routine removes the given node from the list and frees the storage for the node. These routines make use of standard singly linked-list algorithms using pointers.

Access Class

An access class defines the duration for which a file structure segment is accessible. The access class for the list file structure, called listAccess, is shown in Program 3.14 and provides routines to operate on the list. It is the sole means for the application code to access list data. listAccess also contains per access information, in a manner similar to a UNIX file descriptor.

The constructor of listAccess takes a reference to a list class object as an argument. The reference is retained for subsequent access to the list routines and a file structure mapping is established by creating a RepAccess object. The member routines add and remove are covers for the corresponding routines in the list object whereas get and put are cover routines that copy data out of or into the value field of a list node, respectively. All these routines make the list segment accessible by creating a RepWrapper object before performing an operation on the list.

Generator

As discussed in section 3.3.3, a generator iterates over a 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 and/or a generator may have several parameters that control the precise

```
template <class T> class listAccess {
    friend class listGen<T>;
    friend class listWrapper<T>;

    RepAccess<Rep> repacc;                  // access class for representative
    list<T> &lst;                           // list being accessed

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

  public:
    listAccess( list<T> &lst ) : lst( lst ), repacc( lst.fileName ) {
    } // listAccess<T>::listAccess

    void add( char *value ) {
        RepWrapper wrapper( repacc );

        lst.add( value );
    } // listAccess<T>::add

    void get( T *p, char *value ) {
        RepWrapper wrapper( repacc );

        strcpy( value, p->value );
    } // listAccess<T>::get

    void put( T *p, char *value ) {
        RepWrapper wrapper( repacc );

        strcpy( p->value, value );
    } // listAccess<T>::put

    void remove( T *p ) {
        RepWrapper wrapper( repacc );

        lst.remove( p );
    } // listAccess<T>::remove
}; // listAccess<T>
```

Program 3.14: List Access Class

way the generator iterates over the data structure.

The list generator, as defined in program 3.15, has two constructors. The first constructor allows the specification of a list access object and initializes the generator to the beginning of the list. This constructor is employed when the generator object is going to be used only once for one particular list object, as in:

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

The second constructor is employed to create a generator that is subsequently re-initialized to work with a particular list access object, as shown in Program 3.16. In this case, when the list generator object is created, 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 specified list. Notice that the same generator object, `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 access objects refer to lists that contain nodes of type `myNode`. Finally, the iterative 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 (exceptions to this rule are discussed next). 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.

Wrapper

Program 3.9 showed how an application can modify linked list data by copying data out of a list node, changing it, and copying it back by invoking the access class routines; hence, the data is modified indirectly in the original list nodes. The reason for copying is that a pointer returned by a list generator cannot be used in the application program

```
template<class T> class listGen {
    listAccess<T> *la;
    T *curr;

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

  public:
    listGen( const listAccess<T> &la ) {
        RepWrapper wrapper( la.repacc );

        listGen::la = &la;
        curr = la.lst.first();
    } // listGen<T>::listGen

    listGen() {
    } // listGen<T>::listGen

    void over( const listAccess<T> &la ) {
        RepWrapper wrapper( la.repacc );

        listGen::la = &la;
        curr = la.lst.first();
    } // listGen<T>::over

    int operator>>( T *&p ) {
        RepWrapper wrapper( la->repacc );

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

Program 3.15: List Generator

```
listGen<myNode> gen;           // one generator
listAccess<myNode> la, ma;     // two lists
. . .
for ( gen.over( la ); gen >> p; ) { . . . }          // generator used with different lists
for ( gen.over( ma ); gen >> p; ) { . . . }
```

Program 3.16: Using the List Generator

since it points into the list segment, which is not directly accessible from the application. As mentioned in Section 3.4.8, 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 directly to modify data in list nodes, as shown in Program 3.17. 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.

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

    for ( gen.over( la ); gen >> p; ) {  // modify the list directly
        process_string( p->value );
    } // for
}
```

Program 3.17: Using a Linked List Wrapper

The list wrapper is defined in Program 3.18 and is simply a cover definition for declaring a RepWrapper for the specified list segment.

3.4.12 Programming Conventions

The simple generic linked-list illustrated 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.

```
template<class T> class listWrapper {
    RepWrapper wrapper;

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

  public:
    listWrapper( const listAccess<T> &la ) : wrapper( la.repacc ) {
    } // listWrapper<T>::listWrapper
}; // listWrapper<T>
```

Program 3.18: Definition of a Linked List Wrapper

- All persistent administrative information is encapsulated into an administrative object that is stored at the beginning of the segment. Further, the type of the administrative object inherits 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 action occurs 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 can be accessible at a time.

Each basic file structure should provide 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.

3.4.13 B-Tree Example

The following example further illustrates advanced techniques and tools, such as nesting storage managers, for constructing a persistent data structure by building a generic B-Tree file structure. However, the basic structure of the B-Tree file structures follows the persistent linked list exactly.

B-Tree Application

Similar to the persistent linked list, the B-Tree makes available three data structures: one to declare it (BTreeFile), one to access it (BTreeAccess), and one to traverse it (BTreeGen). The nodes of the B-Tree are not created directly by users and, hence, this structure does not exist. All these class definitions are parameterized on two classes KeyType and RecordType to specify the types of the key and the data records, respectively, for the B-Tree.

Program 3.19 illustrates the usage of these classes to write a small application program that creates a persistent B-Tree, inserts a number of records into the B-Tree and finally, retrieves the records from the B-Tree in their sorted order. The program first defines a class Record to describe the structure of the data records to be stored into the B-Tree. The type of the key used to index the records in the B-Tree is the built-in type int. In general, both KeyType and DataType can be defined as arbitrarily complex data structures with the requirement that there exist an assignment operator that can be invoked to copy objects. This requirement is necessary so that records and keys can be copied to and from shared and private memory.

Next, a comparison routine is defined to specify a function that takes two objects of type KeyType and returns a true or false value depending upon whether the first object is "greater" or "smaller" than the second. The comparison routine provides the mechanism necessary to order keys in the B-Tree. In Program 3.19, the comparison routine greater results in the records being arranged in descending order by their keys.

The program then creates the B-Tree, if it does not already exist, with an initial size of 30K by creating a BTreeFile object, db, which is passed as an argument to a newly created B-Tree access object. Once the access object has been created, its member routine insert is invoked to insert a number of records into the B-Tree. Finally, a B-Tree generator

```
struct Record {                                        // data record
    float field1, field2;
    Record &operator = (const Record &rhs) {           // define assignment
        field1 = rhs.field1;
        field2 = rhs.field2;
        return( *this );
    }
};

bool greater( const int &op1, const int &op2 ) { // key ordering routine
    return op1 > op2;
}

void uMain::main() {                                   // uMain is a uC++ artifact
    BTreeFile<int, Record> db( "testdb", greater, 30 Kb ); // create B-Tree
    BTreeFileAccess<int, Record> dbacc( db ); // open B-Tree
    Record rec, *recp;

    // insert records
    for ( int 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> gen(dbacc); gen >> recp; ) {
        . . . // process recp
    }
}
```

Program 3.19: Example Program using a Generic B-Tree

object gen is invoked to retrieve the records stored in the B-Tree in order.

Nested Memory Manager

As discussed in Section 3.4.9, heaps managed by memory managers can be nested within each other. A B-Tree file structure is a good example where nesting is needed. The file space is divided into uniform sized B-Tree nodes managed by a uniform memory manager. A variable memory manager is created within each node to manage the variable sized B-Tree records contained within the node (See Figure 3.10).

The administrative class for the B-Tree, shown in Program 3.20, is defined in the same manner as the linked list structure in section 3.4.10. Note that the class definitions in this section are *not* presented as generic classes for simplification of presentation. In practice, these classes are parameterized in the types of keys and records. The administrative class contains a uniform memory manager and an expansion object for the manager.

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

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

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

Program 3.20: Administrative Class for the B-Tree

The expansion class is defined in Program 3.21. The expansion 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 while the latter is called a leaf node. Both types keep their infor-

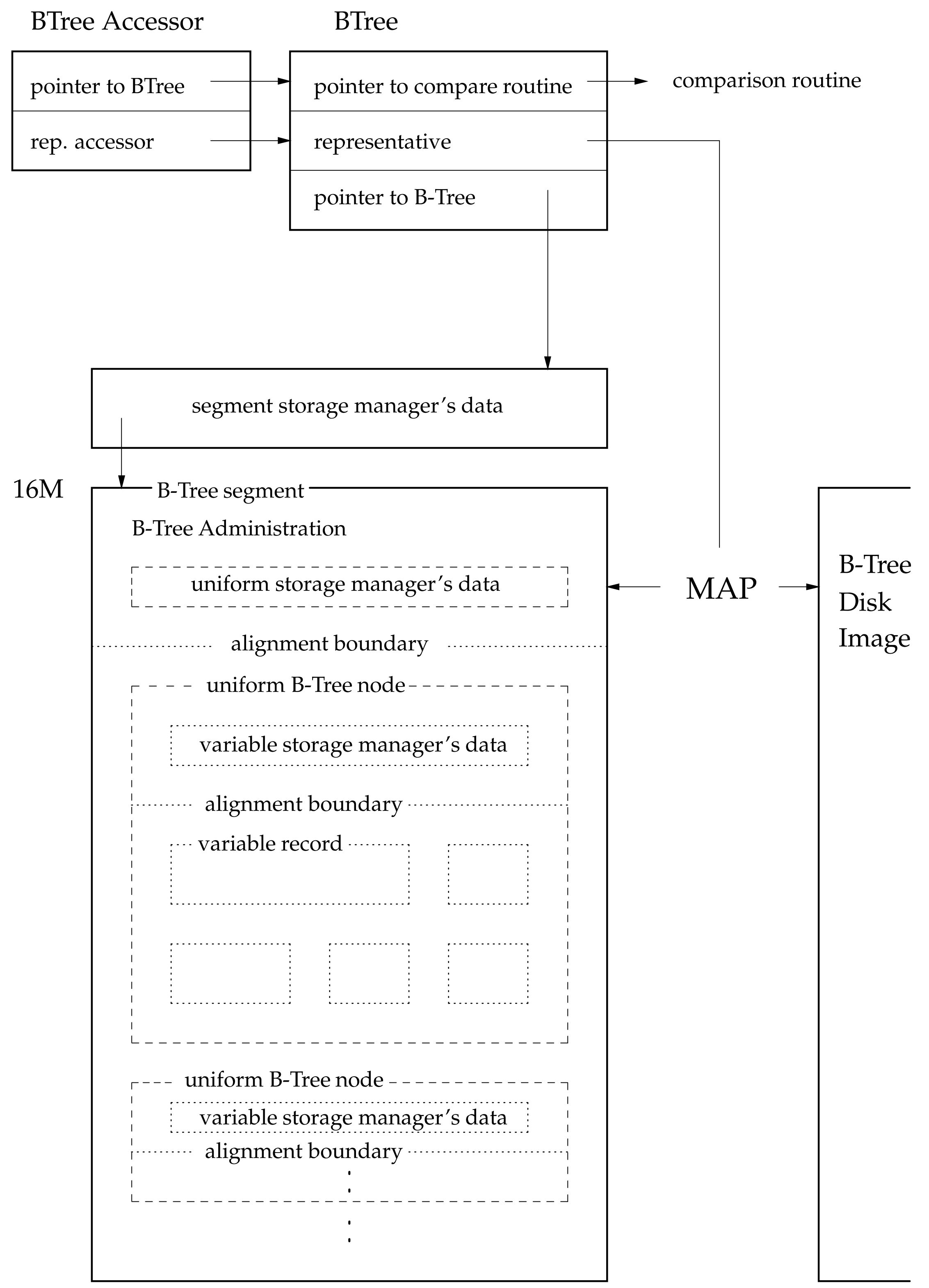


Figure 3.10: B-Tree Storage Structure

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

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

Program 3.21: Expansion Class for the B-Tree Storage Manager

mation within variable sized records managed by a variable memory manager. The leaf node class BTreeLeaf is shown in Program 3.22 and the expansion class for the memory manager vsm is shown in Program 3.23.

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

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

Program 3.22: B-Tree Leaf Node Class

Because all B-Tree nodes are fixed size, a node cannot be enlarged when full. Instead, the member routine SplitLeaf, shown in Program 3.24, within the BTreeLeaf class is

```
class BTreeLeafExpType : public expand_obj {
     BTreeLeaf &leaf;
     . . . .
     BTreeLeafExpType( BTreeLeaf &lf ) : leaf( lf ){}
  public:
     int expand( int );
}; // BTreeLeafExpType

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

Program 3.23: Expansion Class for the B-Tree Leaf Node Storage Manager

called to split the node into two. 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. Note that the leaf nodes in a B-Tree are usually chained together in the form of a doubly-linked list. Existing generic linked list code can be reused to implement linking of the B-Tree leaf nodes, thereby avoiding the need to implement linked lists in B-Tree code.

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

Program 3.24: Leaf Node Member Routine for Splitting

At the file structure level, an access-method implementor makes calls to the lowest level (variable storage manager BTreeLeaf::vsm) to allocate records as shown in program

3.25. The variable storage manager in turn calls the higher level, BTreeAdmin::usm, if necessary, as described earlier.

```
retcode BTreeLeaf::InsertRecord( . . . ) {

  // call lowest level variable storage manager to allocate space within node
  BTreeLeafRecord *FreeRecPtr = (BTreeLeafRecord *) vsm.alloc( /* leaf rec. size */ );

    . . .

} // BTreeLeaf::InsertRecord
```

Program 3.25: Leaf Node Member Routine for Inserting a New Record

3.5 Analytical Modelling of the System

Chapter 4 presents an experimental framework for studying file structures based on the EPD approach to memory mapping. Conducting experiments of this magnitude is exorbitantly expensive in terms of both human and machine resources. Consequently, I felt that an important research contribution could be made to the study of EPD persistent stores by developing a mathematical model for the system. After surveying a number of theoretical models for memory and I/O systems, none of the existing models was found to represent the EPD system closely enough to make accurate predictions about the behaviour of real experiments.

This section describes work done towards the development of an accurate quantitative model for an EPD system that can be employed to accurately predict performance of algorithms in the EPD environment. A related goal is to investigate the behaviour of database algorithms in a memory mapped environment based on the EPD approach, especially in highly parallel systems. I believe that results from this work should apply to other kinds of memory mapped single-level stores as well.

The model can be used to analyze and study sequential and parallel algorithms on a physical machine. My hope is that the model can act as a high-level filter for data structure and algorithm designers to predict general performance behaviour without having to construct and test specific approaches. Only those approaches that look promising from the model need to be more fully tested. Further, a quantitative model is an essential tool for subsystems such as a database query optimizer where the model can be used to compute costs for alternative execution strategies in order to plan optimal schemes for executing specified queries.

3.5.1 Survey of Related Work

The influences on this work stretch across many areas within computer science. The following survey of the modelling literature is divided into two areas: theoretical I/O modelling and other relevant studies on database joins, particularly in shared-memory environments.

Theoretical Models

Classical theoretical models of computation in random access machines have, in recent years, been extended to cover hierarchical memories and the resulting I/O bottleneck as well as spatial and temporal locality. This section presents a brief survey of this work, both in sequential and parallel shared-memory settings.

The classical model of a *Random Access Machine*, or RAM [AHU83] consists of a processor executing instructions on data stored in a uniformly accessible collection of memory cells. The *Parallel Random Access Machine*, or PRAM [FW78] is an extension of RAM for a parallel shared memory machine, which consists of a number of processors communicating through shared-memory. Each processor has access to two types of memories: local and shared (or global) and is capable of performing standard RAM operations as well as reading and writing of cells in global memory. There are several aspects of PRAM that make it unsuitable as a practical model of computation. Nevertheless, the PRAM has served as a useful platform for several subsequent models that are more realistic.

Refinements of the older models have resulted in increasingly complex models. One of the major problems with PRAM as a realistic model is its lack of distinction between local and global memory [PU87, AC88]. The *Block PRAM*, or *BPRAM*$_{p,\ell}$ [ACS89], makes this distinction by assigning different access times to local and global memory, resulting in a two-level memory. Further, a block size based cost model is introduced by the notion of start-up memory transfer costs – words in local memory are uniformly accessible whereas the cost of accessing a block of b contiguous cells in global memory is $b+\ell$, where ℓ is the machine dependent latency. However, BPRAM like models fail to capture the real life notion of a fixed block size and block boundaries. Moreover, by its very nature, the two-level model does not account for differential costs in accessing different sections of memory from the point of view of multiple processors.

Further models have recently been proposed for multi-level memory [AACS87, ACS87, ACFS94], both in the sequential and parallel settings. In the *Hierarchical Memory Model*, a hierarchical organization of memory cells is modelled by assigning access time for location x as $f(x)$, for functions such as $f(x) = \log x$ and $f(x) = x^{\alpha}$ [AACS87]. Block transfer capability is added to the basic model by computing the cost of accessing a block of b bits starting at location x as $f(x) + (b-1)$ [ACS87]. The notions of block transfer and hierarchy are developed further by modelling the memory as a tree of modules, where computation takes place at the leaves [ACFS94]. In this model, data is transfered between modules by buses; parameters of the model include size of blocks, bandwidths of buses, and branching at each level.

I/O complexity models [HK81, AV88, VS94a, VS94b] take a slightly different approach, e.g., Aggarwal and Vitter [AC88] consider a two-level memory model in which a single CPU communicates with a single disk; several blocks of memory can be transferred in a single I/O operation. Vitter and Shriver [VS94a, VS94b] changed this model so that secondary storage consists of several disks and each disk can transfer a single block in one operation.

The memory mapped analytical model presented in this chapter draws on ideas from several of the above papers, though the intent is not to characterize the complexity of problems, but rather to predict performance on many real architectures.

Database Studies

Many database modelling efforts related to this work use the join algorithm for analysis and validation purposes. Joining is a merging of data from two collections of data objects, *R* and *S*, where an *R* object contains a join attribute that refers to an *S* object, and data from each is combined to form the join.

This work builds on the framework proposed by Shekita and Carey [SC90], which presents an analytical single-processor, single-disk model that can be viewed as a simpler version of my subsequent multiprocessor, multi-disk model. In their model, three pointer-based join algorithms are analyzed: nested loops, sort-merge and hybrid hash. However, no experimental data is presented to validate their model.

Shekita and Carey make a number of simplifying assumptions some of which are removed or modified in my analysis. For instance, for joining of a relation *R* with another relation *S*, they assume that every object in relation *S* is referenced by exactly one object in *R*. While my analysis retains this assumption, it leaves open the possibility for a one-to-many relationship between the two relations. They assume the cost of I/O on a single byte to be a constant, not taking into account seek times or the possibility of savings using block transfer; they do not distinguish between sequential and random I/O; they do not consider the fact that the minimum I/O transfer unit on virtually all computers is at least a disk sector and more commonly a virtual memory page.

Two assumptions made in their paper need to be extracted from the analysis: constant-time hashing, and clustering of identical references in a single hash chain during the hybrid-hash algorithm so that a given object from *S* need only be read once to perform the join. My analysis replaces the second assumption with a weaker assumption that all of the objects of *S* referenced in one hash chain can fit into the portion of memory not used by the hash table. In the traditional hybrid-hash algorithm, only one object (or one block) of *S* is present in memory at any given time.

Shapiro [Sha86] analyzes sort-merge and three hash-based algorithms and also provides a discussion of various memory management strategies. Again, no experimental data is provided to validate the model.

Lieuwen, DeWitt and Mehta [LDM93] analyze parallel versions of Hash-Loops and Hybrid-Hash pointer-based join algorithms (see section 5.1) and compare them to a new algorithm, the Probe-child join algorithm. Their work also builds upon Shekita and Carey [SC90] but has a different emphasis from my work in that I develop a validated model for a shared memory architecture based upon the EPD approach.

Martin, Larson and Deshpande [MLD94] present a validated analytical model for a multi-processor, single disk situation. Their model makes a number of assumptions that can introduce unpredictable amounts of both positive and negative error. For instance, the assumptions of perfect inter-process parallelism and perfect processing-I/O parallelism tend to decrease the model's estimate of elapsed time, but the assumption of maximum processor contention for spin locks tends to increase the estimate.

I have extended the work in the above papers in several ways: by allowing multiple processors and multiple disks (resulting in further algorithm design decisions in the course of parallelizing the standard join algorithms), by drawing a distinction between private and shared memory, and of course by using an EPD environment. The parallelization used in my algorithms has been influenced by ideas presented in [SD89]. In addition, my analysis is quantitative as opposed to the qualitative analysis in other models. The model uses measured parameters that quantify the computing environment in which the join occurs, such as how disk I/O is affected by all aspects of the join.

Munro, *et al* [MCM$^+$95] have, quite recently, reported some early work on validating an I/O cost model, called MaStA, for database crash recovery mechanisms. Like this work, MaStA takes into account the peculiarities of a persistent system and attempts to provide more realistic and finer grained estimation of I/O costs than previous attempts. One of the major areas where MaStA differs from this work is the modelling of disk transfer time, $dtt()$ (see section 3.5.2). MaStA divides the I/O costs into a number of different access pattern categories (sequential, asynchronous, clustered synchronous, etc.) with each category assigned a different cost model. The μDatabase model, on the other hand, estimates I/O costs on the basis of a single unified cost model. The amortized cost model developed in this work implicitly incorporates effects of disk access patterns by defining average cost as a function as opposed to a constant. Both models work by assigning an average cost per disk access for a specific I/O category. In the μDatabase model, the average cost function, $dtt()$, is obtained by experiment. Finally, MaStA concentrates exclusively on I/O costs whereas the μDatabase model models CPU costs as well. It is my experience that in a database computation, while the CPU costs are usually not dominant, they can be quite substantial.

3.5.2 Modelling

This section presents the basic model, developed for EPD based systems, and its parameters. The model has as components a number of processes, each having its own segment with a private area of memory, a shared area of memory accessible to all processors through which communication takes place, and a number of disks allowing parallel I/O. The parameters of the model are shown in figure 3.11 and table 3.1.

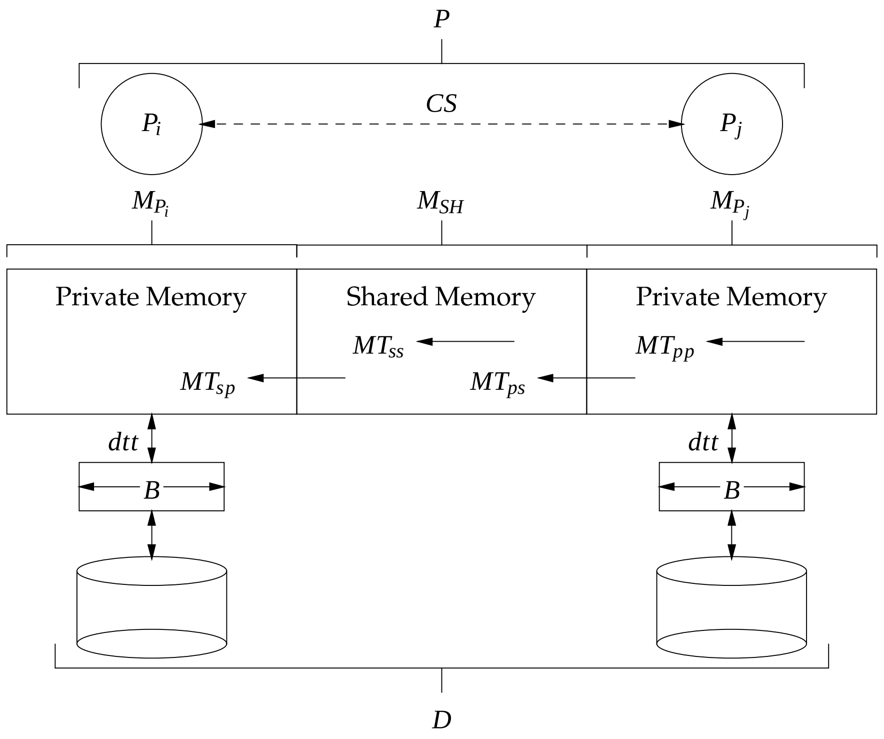


Figure 3.11: Basic Structure of the Analytical Model

The parameter D usually refers to the number of disk controllers, not disks, since there is a one-to-many relationship between controllers and disks (see results in section 4.4 concerning performance effects from disk controllers). When simultaneous requests arrive for the same disk, the disk arbitration mechanism is left unspecified. Alternatives for future refinement of the model include denying algorithms simultaneous access, serializing overlapping requests, and a priority scheme for simultaneous requests.

Memory transfer times are given in the form of combined read/write times because

Parameter	Description
P	number of processes used by a given algorithm
CS	amount of time for a context switch from one process to another
M	number of bytes of memory, private and shared
M_{P_i}	number of bytes of private memory used by process P_i
M_{SH}	number of bytes of shared memory available for use to P processes
B	size, in bytes, of a block or page of virtual memory
D	number of disks that can be operated in parallel
dtt	disk transfer time
dtt_r	disk transfer time – read
dtt_w	disk transfer time – write
MT_{sp}	shared to private memory transfer time
MT_{ss}	shared to shared memory transfer time
MT_{ps}	private to shared memory transfer time
MT_{pp}	private to private memory transfer time
$newMap$	time to create a mapping for new area of disk
$openMap$	time to create a mapping for existing area of disk
$deleteMap$	time to destroy mapping as well as disk area

Table 3.1: Parameters of the Model

all segment transfers move data using assignment statements, which read and then write. Furthermore, these transfer times can be used even if the architecture implements an explicit block move instruction that does not directly involve process registers; in this case, the transfer time may be parameterized by the length of the move because a block move may be more efficient for longer transfers. As an example of the use of memory transfer times, if one process transfers k bytes from shared memory to private memory, this takes time $k \cdot MT_{sp}$. For machine with block move instructions, this time could be $MT_{sp}(k)$.

Disk Transfer Time

Modelling disk transfer is complex because it is a function of the data access pattern due to the inherent sequentiality of the components of a disk access. The nature of join algorithms is such that data access is clustered into contiguous bands on the disk during certain parts of an algorithm. Intense (random or sequential) I/O occurs in a band followed by similar I/O occurring in the next band and so on. This clustering of accesses is modelled by measuring the average cost per block of sequentially accessing bands in which random access occurs, over a large area of disk. The size of the disk area is irrelevant; it only has to be large enough to obtain an average access time for the band size. The layout of data on disk is always given to explain the band size in further algorithmic discussion.

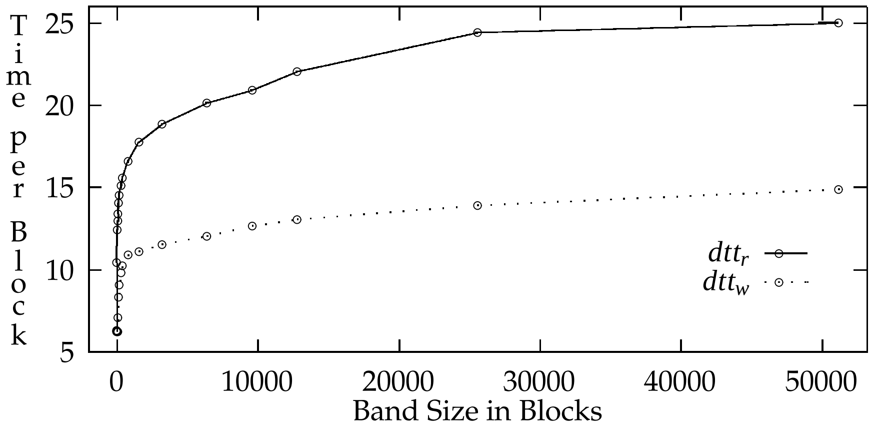
In general, the disk transfer time function, *dtt*, has two arguments: the unit of data transfer, and the span, in blocks, over which random disk accesses take place, i.e., the size of the band. In the physical machine used for this work, the first argument is always *B*, the virtual memory page size; therefore, the first argument of *dtt* is dropped from all of the subsequent formulas, i.e., *dtt* is considered to be a function of band size alone. Figure 3.12(a) shows the average time, for the Sequent Symmetry used for experiments (see section 5.5.1), to transfer a block (4K) to or from disk with respect to a given band size. When the band size is one, access is sequential; when the band size is greater, access is random over that area. Thus, average time increases as the band size increases. One curve is for random reading in a band with no repetition of blocks; the other curve is for random writing in the band with no repetition of blocks. One might intuitively expect the read and write times to be identical. However, while a read page fault must cause an immediate I/O operation, writing dirty pages can be deferred allowing for the possibility of parallel I/O and optimization using shortest seek-time scheduling algorithms. Thus, writes, on average, cost less than reads. The two curves are used to interpolate disk transfer times for reading, dtt_r, and writing, dtt_w, respectively. Both dtt_r and dtt_w are machine-dependent and must be measured for the physical environment in which the join is executed.

It needs to be emphasized that the band size in the *dtt* functions is the *not* the logical span in the database over which accesses takes place but rather the actual span on disk. In other words, the argument to the *dtt* function has to take into account the actual layout of the database file on disk (which includes non-contiguous layout of data by the operating system). In order to measure the *dtt* curves shown in figure 3.12(a), the test file is laid out contiguously on disk so that the logical bands in the file also correspond to similar bands on disk.

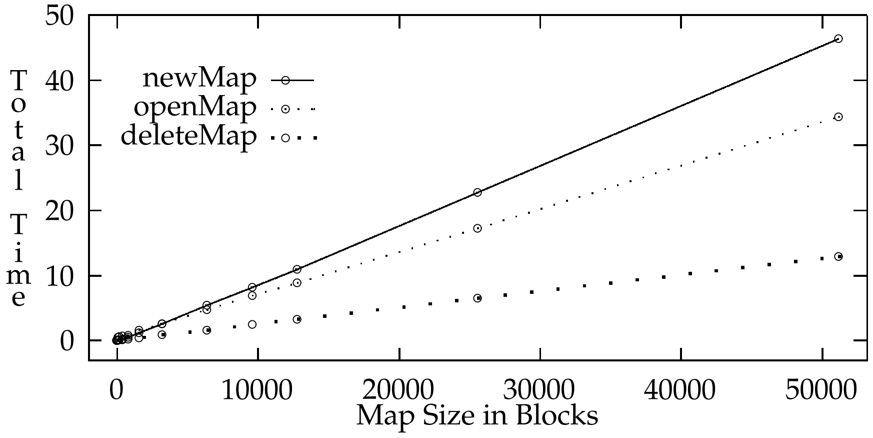
Finally, the shape of the *dtt* curves is determined by two distinct phenomenon, namely, the number of times the disk arm changes direction, which is an expensive operation, and the total amount of distance traveled. The latter increases linearly, after a threshold value is reached, with band size and is reflected in the rise in the *dtt* values in the upper portions of the curves. The total number of times the disk arm changes directions increases very rapidly when the band size is increased from 1 but soon reaches a saturation value and stays relatively constant after that. This behaviour is the main cause of the initial growth of the *dtt* curves in figure 3.12(a).

Memory Mapping Costs

The cost of three fundamental memory mapping operations, namely, creating a mapping for a new area of disk, establishing a mapping to an existing area of disk, and destroying a mapping as well as its data in an existing area of disk, is modelled by three measured functions, *newMap*, *openMap* and *deleteMap*. Each of these functions takes the size of the mapping as an argument.



(a) Disk Transfer Time (in msecs)



(b) Memory Mapping Setup Time (in secs)

Figure 3.12: Measured Machine Dependent Functions (for a Sequent Symmetry running DYNIX 3.1 with Fujitsu M2344K and M2372K disk drives)

Figure 3.12(b) shows the measured values, for the Sequent Symmetry used for experiments (see section 5.5.1), of memory mapping costs. All mapping costs increase with size because constructing the page table and acquiring disk space increases linearly with the size of the file mapped. New mappings are more expensive than existing mappings because new disk space must be acquired. Deleting is the least expensive because only the storage for the page table and disk space need to be freed.

In absolute terms these costs are very high and constitute a significant performance problem. However, the high cost is mostly a function of the particular memory mapping implementation in DYNIX and of the slow hardware. Measuring the same costs on a SUN SPARCserver 670MP running the SunOS 5.3 operating system results in much smaller values, e.g., the cost of creating a new mapping of 50,000 4K blocks on the SUN machine is less than 0.1 second.

3.5.3 Using the Model to Analyze an Algorithm

Chapter 5 contains a discussion on the design of three parallel join algorithms that were implemented and analyzed by means of the model presented earlier. This section outlines the general procedure for analyzing a given algorithm within the framework of the model. The analysis can be used to predict the performance of the algorithm on a physical machine. The specifics of the physical machine are incorporated into the analysis by means of the measured parameters such as disk transfer time and memory mapping costs. Additional parameters can be added as needed to analyze the given algorithm, e.g., in order to analyze heap-sort the cost of inserting and removing an element from the heap of pointers in memory must be measured. Once all the required parameters have been compiled and their values determined, they can be used for computing the costs of the individual steps performed by the algorithm.

One of the expensive activities in a database algorithm is I/O cost, which can vary substantially depending upon how the algorithm accesses disk blocks. As such, it is essential to identify the patterns of disk access in various steps or phases of the algorithm. After determining the nature of I/O in a particular phase of an algorithm, the appropriate *dtt* formulas are applied to compute the I/O cost of the specific phase. The nature of I/O for a phase is determined not only by the total amount of disk blocks read and written, but also by the type of disk access (sequential or random) and by the span of disk over which the I/O takes place. Finally, the memory, CPU and I/O costs of the various phases of the algorithm are summed, as appropriate, to predict the total cost of the algorithm on the physical machine. Any parallelism in the algorithm is accounted for by computing the serial cost of computations that occur in parallel.

3.6 Summary

The design of μDatabase is motivated by a desire to eliminate the complexity and expense of swizzling pointers, support persistence within a compartmentalized view of persistent objects in which individual programs are allowed to simultaneously manipulate data stored in multiple collections, and follow the software approach based on conventional architectures for its immediate accessibility and portability. This work identifies and quantifies some components of a persistent system that are quite difficult or inefficient to construct with conventional operating system and hardware support. The support for multiple persistent areas is provided by employing the notion of hardware segments, which are implemented on conventional architectures by a novel usage of Unix processes. This chapter also illustrated the programming interfaces and conventions used for developing applications in μDatabase. The ease with which powerful programming techniques such as polymorphism and storage management can be applied to persistent as well as transient data is amply demonstrated in the process. The lack of support at the compiler level means that programming in the current stage of μDatabase relies on certain conventions being strictly followed. The μDatabase programming interfaces can be simplified and made more secure by providing some language support, which is also needed for implementing services like recovery control (see chapter 6).

Presenting a generic concurrent retrieval algorithm for partitioning file structures demonstrates how parallelism can be exploited easily and naturally in the EPD approach. For instance, by mapping various partitions of a file structure into a single address space, many partitioning issues are made transparent to the executing program, resulting in code that is less complex and more efficient. Finally, this chapter presented the design and development of a quantitative analytical model of computation in the EPD environment. Once validated (see chapter 5), the model can be used to predict the performance of specific algorithms as the system and data parameters are tweaked, resulting in significant benefits when studying new algorithms.

Chapter 4

Experimental Analysis of EPD File Structures

One of the important goals of this work is to demonstrate the feasibility and viability of the EPD approach to memory mapped file structures. The most effective way of doing so is to design and construct illustrative EPD file structures and run experiments on them using a tightly controlled test bed.

As mentioned earlier, many of the traditionally cited reasons for rejecting the use of mapped files are no longer valid, and compelling arguments have been made for the use of memory mapped single-level stores for implementing databases. Further, memory mapping techniques can be used advantageously not only for complex data structures but also for simpler traditional database structures. Traditional databases can be accessed using memory mapped access methods without requiring any changes to the existing data. It is my thesis that memory mapping techniques can provide performance comparable to traditional approaches while making it much easier to construct, maintain and augment the access methods of a file structure (i.e., to support extensible databases) by greatly reducing program complexity. In spite of all these arguments, there is still resistance and skepticism in the database community to memory mapping. One reason for this skepticism is the lack of hard data to support arguments in favour of memory mapped file structures and their access methods.

At the beginning of this work, no major undertaking to conduct experiments on a memory mapped storage system had been reported and there was no experimental evidence available to support the view that memory mapped file structures could perform as well as or better than traditional file structures. Therefore, to demonstrate the feasibility of the EPD approach to memory mapping, I decided to implement several illustrative file structures using both the EPD approach and the traditional buffer management approach. The performance of these file structures in the two environments was measured and compared. For this purpose, an experimental testbed was designed and implemented. The testbed allowed the experiments to be conducted in a tightly controlled environment and was employed to make reliable performance measurements.

In addition to conducting experiments on sequential file structures, it was also decided to study the behaviour of partitioned file structures in an EPD environment since parallel access methods represent an important and active area of research in database technology. Two of the sequential single-disk EPD file structures were partitioned by using data striping techniques and algorithms were designed to perform parallel queries on these striped structures. Experiments were conducted to study the benefits obtained from data partitioning and parallel access methods in the memory mapped environments based on the EPD approach.

All these experiments are an important and somewhat unique aspect of this work that has been well received by other researchers working in this area [BGW92]. In addition to demonstrating the effectiveness of the EPD approach, this work establishes the beginnings of benchmarks against which other work in the area can be evaluated. The rest of this chapter presents the design of the experimental testbed and various file structures, the experiments conducted on the testbed and an analysis of the results obtained.

4.1 Testbed

4.1.1 Hardware/Software Platform

All the experiments presented in this dissertation were conducted on a 10-processor (Intel i386) Sequent Symmetry [Sym87], a shared-memory symmetric multi-processor, running the DYNIX 3.1 operating system. The system contained 64M of physical memory, one Sequent dual-channel disk controller (DCC) and eight Fujitsu M2344K/M2372K disk drives. The DYNIX operating system uses a simple page replacement algorithm that employs a FIFO queue per page table augmented by a global LRU cache of replaced pages so there is a second chance to recover a memory frame before it is reallocated. In order to analyze the experimental results, it is important to understand the organization of the DCC and the DYNIX page replacement algorithm. Therefore, a summary of these two aspects is presented before describing the other details of the testbed. The information presented in sections 4.1.2 and 4.1.3 has been derived from the Sequent Symmetry technical summary guide [Sym87]. Also, figures 4.1 and 4.2 have been reproduced from the same source.

4.1.2 Sequent Dual-Channel Disk Controller (DCC)

The Sequent dual-channel disk controller (DCC) controls 8 disk drives using the S MD-E (Storage Module Drive – Extended) disk interface. Transfer of data to and from the disks takes place at bursts of up to 3 megabytes per second. The DCC provides two independent data channels, each of which connects 4 disk drives to the system bus. The dual-channel design, depicted in figure 4.1, allows two drives, one on each channel, to transfer data simultaneously in each direction. All drives are capable of simultaneous seeks. The drives are connected to the data channels of the DCC via two multiplexors, with each multiplexor connecting two drives each to the two channels.

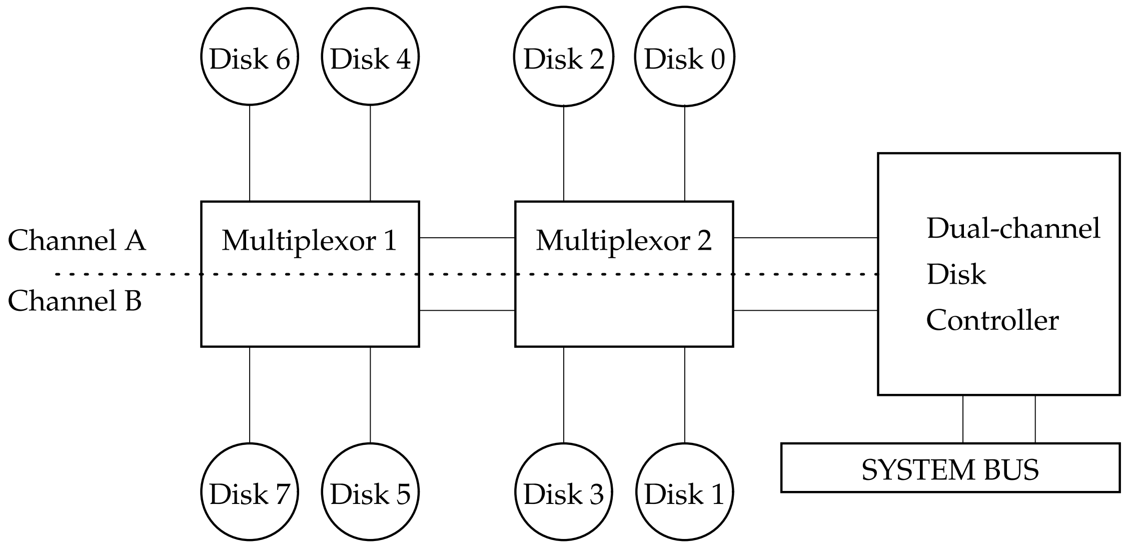
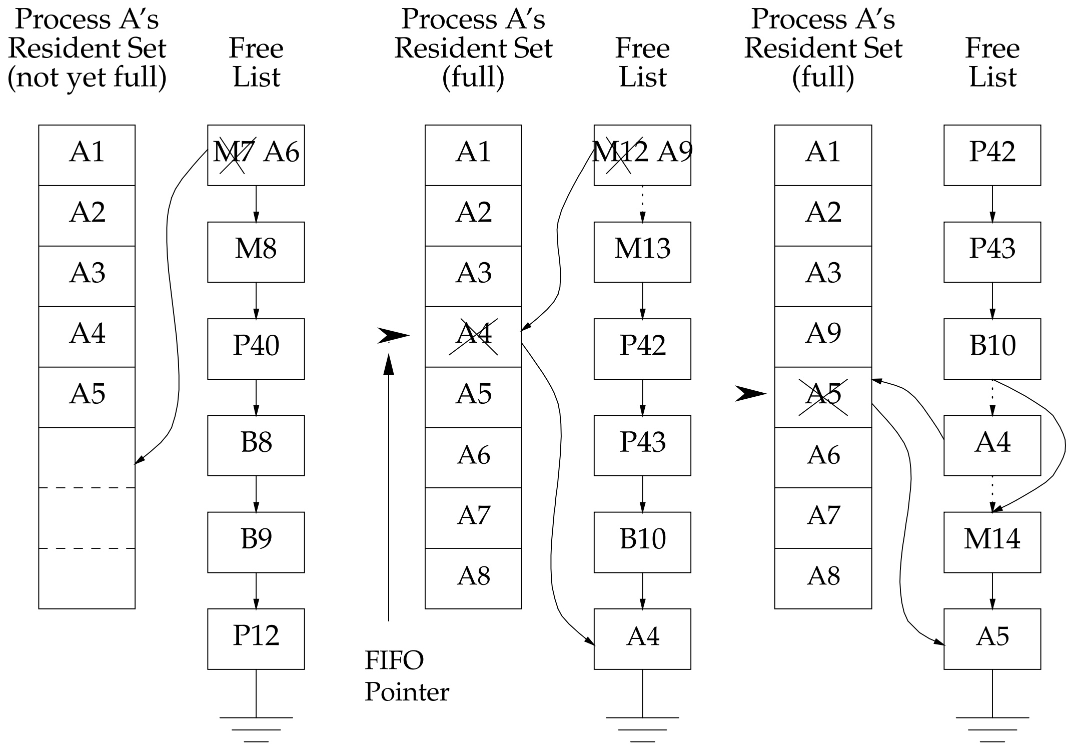


Figure 4.1: Sequent Dual-channel Disk Controller (DCC)

4.1.3 DYNIX Virtual Memory Implementation

DYNIX employs the virtual memory management implementation first used in the VAX/VMS operating system for the VAX-11/780 (see [LL82] for details of the VAX/VMS implementation). At boot time, the DYNIX kernel allocates physical memory for itself and its basic data structures. The remaining pages of physical memory are inserted into a queue called the *free list*. All memory needed for user processes is taken from the free list. When a process starts executing, the pages of virtual memory it needs are loaded on demand at page fault time. Each process has a *resident set*, which consists of the list of physical memory pages allocated to that process. The maximum size of the resident set for each process is limited, to prevent any one process from monopolizing physical memory, and can be specified by invoking a system call; otherwise the operating system uses a heuristic to determine the maximum resident set for the process. During the initial flurry of page faults after a process starts executing, the process obtains physical memory by depleting the free list (see figure 4.2(a)[1]). After the resident set is filled up, however, the page replacement algorithm is invoked at page fault time to trade a page from the resident set with one from the free-list (see figure 4.2(b)). When pages are added to the free list, they go to the tail of the list. A page that is not reclaimed by its process eventually reaches the head of the free list and is claimed by a new process.



(a) Filling the Resident Set of Process A

(b) Page A4 moved to the free list to make room for page A9

(c) Reclaiming page A4

Figure 4.2: DYNIX Page Replacement Algorithm

[1]The letter in a page label indicates a process that owns the page and the following digits indicate the page number within the address space of the process.

The page replacement algorithm is a modification of FIFO and is implemented with a pointer per process that cycles through the pages of the resident set of a process. When a page fault occurs, the page indicated by the pointer is swapped with a page from the free list. A record is maintained of all the pages a process has placed on the free list. If one of these pages is referenced again by the process, the resulting page fault, called a *minor page fault*, simply reclaims the page from the free list (see figure 4.2(c)), thereby avoiding the need to read the page from disk. By contrast, a *major page fault* results in reading the faulted page from disk. Thus, page faults in figures 4.2(a) and 4.2(b) are major page faults while the one in figure 4.2(c) is a minor page fault. Once a page is brought back to the resident set, it is not replaced until the FIFO pointer makes another pass through the resident set.

To handle pages modified during execution, the above process is modified slightly by the introduction of another queue called the *dirty list* that works in a manner similar to the free list. If a page being replaced has been modified during execution, the page gets added to the dirty list instead of the free list. When free memory gets low, an operating system daemon process writes out a subset of the pages in the dirty list to disk and transfers these pages to the free list as clean pages.

4.1.4 Experimental Testbed

The testbed designed and developed as part of this work allows experiments to be run in a controlled environment. In test mode, the only activities taking place on the system are the ones concerning the experiment. Thus, the experiments run without any external interference. This environment was made possible by the following manipulations of the DYNIX operating system mechanisms.

During test mode, the system runs in its normal multi-user mode but all operating system services except the ones needed by the experiments are disabled or shut down, which includes disabling external login, and all network services.

The parameters that can be controlled by the experimenter include:

Maximum resident set: The maximum resident set for each individual process can be specified at run time, which controls the maximum amount of real memory available to the experiment during execution. Upon exceeding that size, the page replacement algorithm described in section 4.1.3 is invoked to make room for a new page.

Total amount of free physical memory in the system: Due to the nature of the DYNIX virtual memory implementation and to avoid working with extremely large databases, it was important to control the total amount of free memory available to the experiment. This restriction was achieved by using non-swappable memory blocking programs. A blocking program causes a specified amount of physical memory to be allocated and goes to sleep. Because the blocking program is made non-swappable, the physical memory allocated to it is not available for any other computation; it is as if that memory was not in the system.

During experiments, the total free memory was kept at a level that left a very small amount of free memory in the global cache after memory had been allocated to the executing processes. This strategy allowed the processes to continue execution while ensuring that the experiment did not benefit from any extra available memory in the global cache.

During experiments, the virtual memory system was tuned (by means of the vmtune facility of DYNIX) to reduce the size of global free memory as much as possible and to turn off operating system optimizations such as disk read-aheads. Some additional changes made subsequently to the testbed are described in section 5.5.1.

For parallel experiments presented in this dissertation, a maximum of 4 disks, with one disk attached to each side of the two multiplexers of the DCC, were used. This configuration allowed the experiments to make parallel use of the 4 disks. Note, however, that there were only two channels with two disks on each channel, which introduced some contention for data transfer.

4.2 Experimental Structure for Feasibility Studies

Several experiments were constructed to demonstrate the feasibility of the EPD approach to memory mapping. The general form of an experiment is to implement a file structure in both the traditional and the EPD styles, create and populate the file structures, measure performance of retrievals from the file structures, and compare the results. While every effort was made to keep the two types of file structures as similar as possible, some system limitations precluded absolutely identical execution environments. In particular, the traditional file structures are accessed through a custom built LRU buffer manager that performed raw I/O to and from disk. DYNIX does not support memory mapping using raw I/O, and therefore, regular file system I/O is employed for the EPD file structures. Separate experiments were conducted to ensure that memory mapping through the file system did not result in any advantages due to buffering; it was found that a mapped file does not make use of file system buffers. To make the comparisons equal, all file structures used 8K node sizes and all I/O was performed in 8K blocks.

In order to use the experimental testbed described in section 4.1, the following general steps are taken. First, a set of blocking programs are run whose only purpose is to reduce the amount of available physical in the system so that it is just enough to be the total amount of memory needed for an experiment. The blocking programs sleep during the experiment so as not to cause any interference. After the amount of available system memory has been reduced to the desired level, the DYNIX limit command is used to restrict the maximum resident set size for the program(s) constituting the experiment. The experiment is run in this restricted environment. The DYNIX ptime utility is employed to obtain performance measurements such as the number of page faults and elapsed time for the program. If it is necessary to measure the performance of the individual phases of the program, appropriate system calls are embedded into the program code. For example, getusclk() can be invoked to access the micro-second clock.

The traditional file structures were implemented on top of the LauRel database [Lar88]. The DYNIX page replacement algorithm (see section 4.1.3) was matched against the custom-built LRU buffer-manager used by LauRel. Experiments were run both stand-alone to preclude external interference and on a loaded machine. The amount of memory available for the experiment and total free global memory were tightly controlled using blocking programs so that both types of file structures had exactly the same amount of buffer space during execution.

The test file structures varied in size from 6 to 32 megabytes. The total amount of primary storage available for the experiments was restricted to keep the ratio of primary to secondary storage as 1:10 and 1:20 respectively for two different sets of experiments. Thus, primary storage for the experiments ranged in size from .6M to 3.2M and .3M to 1.6M. These primary to secondary storage ratios are common in the current generation of computers, supporting medium (0.1G-.5G) to large databases (1G-4G) but not very large databases (1T).

4.3 Sequential File Structures

In order to show that the EPD approach to memory mapping is suitable and efficient for the implementation of traditional (pointer-less) and complex (incorporating many pointers) data structures alike, experiments were conducted on a prefix B$^+$-Tree [BU77] and an R-Tree [Gut84], which are pointer-less, and a complex network graph structure, which contains many pointers. In each case, the cost of performing representative queries or traversals was measured in a controlled environment. The results, presented in section 4.4, demonstrate that the EPD structures perform quite admirably when compared against their traditional counterparts.

This section presents the sequential single-disk file structures implemented for experimentation and the details of the actual queries performed on individual file structures. The queries are designed to cover many realistic access patterns.

4.3.1 Prefix B$^+$-Tree

The prefix B$^+$-Tree [BU77] is a well studied and widely used data structure for maintaining indexes, and, as such, was an ideal candidate for inclusion in this study.

For the experiments with the B$^+$-Tree, 100,000 uniformly distributed records were generated whose (order) keys were taken from the unit interval. Records had variable lengths with an average length of 27 bytes. The records were inserted into a prefix B$^+$-Tree in the order of their generation, i.e., the records were inserted into the B$^+$-Tree in a uniformly distributed order of their keys. For the resulting B$^+$-Tree, four different query files were generated, each file requiring that 10,000 records be read in total in response to a collection of range queries of a given size. An individual query in each file was specified by a random key (based on a uniform distribution) and a fixed number of records (the size of the range query) to be read sequentially starting from the specified key. In the rest of this chapter, each of the four query files is described by a tuple <n,m> where n is the total number of queries in the file and m is the size of each query. For example, <10,1000> implies 10 queries of size 1,000 records each – the query file consists of 10 keys from a uniform distribution and, for each key, the experiment searches for the key in the B^+-Tree and then reads 1,000 data records sequentially by following the leaf node links of the B^+-Tree.

An additional fifth query file contained 10,000 exact match queries obtained from a normal distribution with a mean of 0.5 and a variance of 0.1. For each query in this file, the experiment searched for the specified key in the B^+-Tree index and retrieved the corresponding data record.

4.3.2 R-Tree

The R-Tree [Gut84] is a data structure and an access method for multi-dimensional objects (e.g., points and regions) and is used for representing spatial data, e.g., in geographical information systems. An R-Tree is a natural extension of the B-Tree for multi-dimensional data. This discussion is restricted to 2-dimensional objects, referred to as 2-dimensional *rectangles*.

A 2-dimensional rectangle is a tuple containing two (x,y) pairs, which denote the lower-left and the upper-right corners of a rectangular area in a 2-dimensional space. Thus, a 2-dimensional rectangle might be used to represent an area on a planar surface while a 3-dimensional rectangle might represent a box in space. The structure of an R-Tree is similar to that of a B^+-Tree except that the leaf nodes, called data nodes, of an R-Tree contain pointers to data rectangles while index nodes, called directory nodes, contain minimum bounding rectangles instead of keys. A minimum bounding rectangle for a given set of rectangles is the smallest sized rectangle that completely *encloses* all the rectangles in the given set (see figure 4.3); a rectangle is said to enclose another rectangle if the former overlaps the latter along each dimension. Thus, an index entry in an R-Tree directory node consists of a pointer to a next level (data or directory) node and a rectangle, which is the minimum bounding rectangle for all rectangles contained in the sub-tree rooted at the referent next level node. The R-Tree supports *point* queries and several types of *window* queries. A point query on an R-Tree asks for all rectangles that cover a given query point whereas a window query asks for all rectangles that enclose, *intersect* or are *contained* in a given query rectangle. An R-Tree window query is similar to a B^+-Tree range query. However, in terms of data access, there is one basic difference: index pages are accessed more frequently and involve much more computation for the R-Tree than for the B^+-Tree.

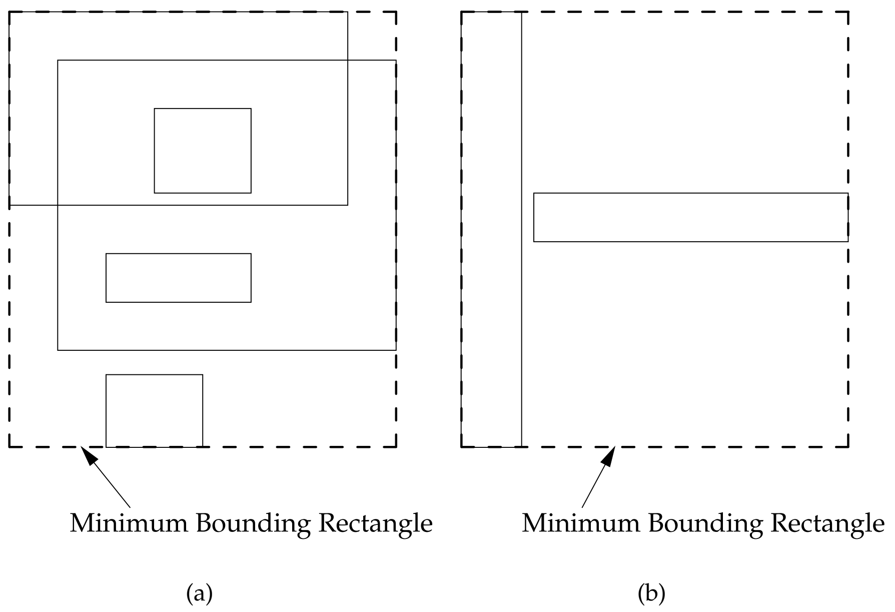


Figure 4.3: Minimum Bounding Rectangles in an R-Tree

For the R-Tree experiments, a 2-dimensional R-Tree was implemented in both the traditional and the EPD environments. The maximum number of entries in individual R-Tree nodes were limited to 450 in data nodes, and 455 in directory or index nodes. Each R-Tree was populated with data obtained from a standardized testbed [BKSS90]. The data consisted of 100,000 2-dimensional rectangles where each rectangle is assumed to be in the unit cube $[0,1]^2$. The centres of the rectangles follow a 2-dimensional independent uniform distribution; see [BKSS90] for further details of the test data. The query file used for the experiments was also taken from the same testbed and consisted of 1000 point queries and 400 each of the enclosement, intersection and containment window queries.

4.3.3 Network Graph

To simulate access patterns found in complex non-traditional data intensive applications (e.g. hypertext or object-oriented databases), a large directed graph was constructed consisting of 64,000 nodes of size 512 bytes each. The nodes were grouped into clusters of 64 nodes each; the nodes within a cluster were spatially localized on secondary storage. An edge going out from a node had a high probability (85%, 90% or 95%) of referencing another node within the same cluster. Inter-node edges were paired with randomly selected nodes. Figure 4.4 illustrates this structure. Each experiment consisted of 40 random walks within the graph; each walk traversed 500 edges. These traversals simulated a CAD/CAM system where multiple users access a particular part and then access the part information in different ways.

4.4 Results and Analysis of Experiments on Sequential File Structures

For each experiment, three performance measures were gathered: the CPU time, the elapsed time, and number of read operations from secondary storage. Multiple processors were used in both traditional and memory mapped experiments. The retrieval application process ran on one processor while the access method for the file structure ran on another processor. The measured CPU time is the total computing time spent by all processors in a given test run and the elapsed time is the real clock time from the beginning to the end of a test run. Hence, CPU time for an experiment may be greater

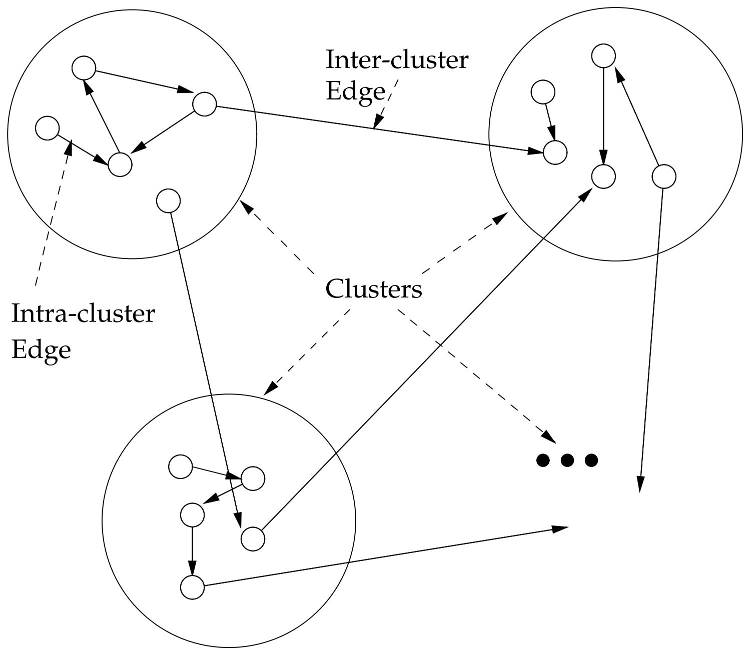


Figure 4.4: Network Graph Structure

than elapsed time. Both times include any system overhead.

4.4.1 Stand-alone System: No External Interference

The results of running the experiments on a stand-alone system are presented in table 4.1. For the CPU times, the memory mapped access methods are generally better than the traditional ones because there is less CPU time spent doing buffer management. For the elapsed times, the memory mapped access methods are comparable (± 10%) to their traditional counterparts. An exception occurs when the traditional LRU buffer space is only 5% of the file size for sequential reads because the LRU algorithm is suboptimal in this case and results in some extra input operations. The memory mapped FIFO page replacement algorithm is almost optimal in this case and can work with smaller amounts of primary memory without degrading performance. All of the results show that the DYNIX page replacement scheme performed well enough to be comparable to the traditional LRU buffer-manager.

These results confirm the thesis about memory mapped file structures, i.e., the EPD approach to memory mapping provides performance comparable to that obtained with traditional file structures for random queries.

4.4.2 Loaded System: External Interference

To verify the conjecture about the expected behaviour of mapped access methods on a loaded machine, the previous B^+-Tree experiments were repeated during peak-load periods. The memory mapped and traditional retrievals were started at the same time during peak load (3:00pm) and, hence, 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 retrievals were not interacting at the hardware I/O level. However, the amount of global cache was not restricted, so if free memory was available, the memory mapped access method would benefit from it. Table 4.2 shows the averages of trials on 5 different week-days.

Note that for the EPD file structures, the amount of *local* memory allocated to the experiment was 10% of the database size, i.e., the maximum resident set of the program was restricted to be 10% of database size plus an allowance for program code and data. However, the program's data can be cached by the operating system in any *global* memory that is not being used by other competing programs running on the system. As can be seen, there was a large difference when there were a significant number of random reads. In those cases, the memory mapped access methods make use of any extra free memory to buffer data. This effect is particularly noticeable for the normal distribution because any extra memory produces a significant improvement. Clearly, the LRU buffer manager could be extended to dynamically increase and decrease buffer space depending on system load, but doing so is non-trivial and further complicates the buffer

Block Size = 8K		Memory Mapped			Traditional		
Access Method	Query Distribution	CPU Time (secs)	Elapsed Time (secs)	Major Page Faults	CPU Time (secs)	Elapsed Time (secs)	Disk Reads
Prefix B^+-Tree	<1,10000>	35	19	61	32	32	53
	<10,1000>	35	19	56	32	32	58
	<100,100>	37	22	147	35	35	150
	<10000,1>	98	217	8789	240	223	8746
	normal	91	181	6777	202	183	6638
R-Tree	window	154	174	1414	330	334	1462
	point	109	124	934	230	234	896
Network Graph	85% local ref	318	476	15294	526	458	15004
	90% local ref	271	375	11278	449	370	11368
	95% local ref	207	243	6584	337	254	6539

(a) Primary Memory Size 10% of Database Size

Block Size = 8K		Memory Mapped			Traditional		
Access Method	Query Distribution	CPU Time (secs)	Elapsed Time (secs)	Major Page Faults	CPU Time (secs)	Elapsed Time (secs)	Disk Reads
Prefix B^+-Tree	<1,10000>	35	19	61	35	35	117
	<10,1000>	35	19	66	34	33	131
	<100,100>	37	22	155	37	36	216
	<10000,1>	127	255	9415	260	224	9723
	normal	126	235	8250	253	217	9313
R-Tree	window	181	227	2913	367	374	3396
	point	136	184	2647	279	289	3491
Network Graph	85% local ref	383	565	17772	563	495	16550
	90% local ref	330	462	13602	484	403	12781
	95% local ref	264	316	8338	361	276	7400

(b) Primary Memory Size 5% of Database Size

Table 4.1: Comparison of Memory Mapped and Traditional Access Methods

Allocated Primary Memory Size 10% of Database Size

Block Size = 8K		Memory Mapped			Traditional		
Access Method	Query Distribution	CPU Time (secs)	Elapsed Time (secs)	Major Page Faults	CPU Time (secs)	Elapsed Time (secs)	Disk Reads
Prefix B$^+$-Tree	<1,10000>	35	21	60	34	35	53
	<10,1000>	36	21	56	34	36	58
	<100,100>	37	25	143	37	38	150
	<10000,1>	111	277	6677	263	263	8746
	normal	97	134	2063	221	217	6638

Table 4.2: Peak Load Retrievals

manager while duplicating facilities provided by the operating system.

4.5 Partitioned B$^+$-Tree

A B$^+$-Tree based on the EPD approach was modified to become a partitioned B$^+$-Tree and evaluated. This section presents the modifications made to the B$^+$-Tree and the results of the experiments run using the methods presented in section 3.3.

4.5.1 Partitioning Algorithms

Two different partitioning algorithms, viz., a near-optimal algorithm by Seeger and Larson [SL91] and a simple round-robin algorithm, were studied. Given D disks, the Seeger-Larson algorithm guarantees that each leaf node in any sub-sequence of leaves of size $D/2$ or smaller is stored on a distinct disk. In the round-robin partitioning algorithm with D disks, numbered 0 through $D-1$, when a node splits, the new node is allocated on disk $(M+1) mod D$, where M is the disk containing the splitting node. The round robin algorithm distributes new nodes cyclicly over the D disks, and its performance was compared to the Seeger-Larson algorithm.

4.5.2 Modified File Structure

The prefix B$^+$-Tree (see section 4.3.1) was modified to achieve:

efficient partitioning of data: The B$^+$-Tree is partitioned across several disks during the insert operation; when a node splits, the new node is allocated on a disk different from the one containing the splitting node. The disk for the new node is determined by the partitioning algorithm being used.

efficient parallel execution of range queries: A parallel retrieval algorithm was designed and implemented for this purpose. The algorithm splits the specified range query into multiple smaller sub-queries that access data on different disks and are executed in parallel.

To achieve the first of the above goals, the structure of the B$^+$-Tree nodes is modified to store the cardinal number of the containing disk with each index entry and leaf node. This modification allows the traversal algorithm to determine the containing disk for a node referred to by an index entry without having to rely on a special format for the node pointers.

Partitioning a <K,K> range query involves searching the index for the two keys to determine the leaf nodes for the keys and then partitioning the set of leaf nodes by following the leaf node links. In order to allow the <K,C> and <K,C,C> range queries to be processed equally efficiently, the B$^+$-Tree node structure is further modified as follows. The total number of records stored in the sub-tree of an index entry is stored with the index entry as depicted in figure 4.5.

This structure allows a <K,C> style query to be changed into a <K,K> style query by using the record counts in index entries to locate the bounds of the <K,C> style range query. For example, execution of a <K,C> query first searches the index for key K and

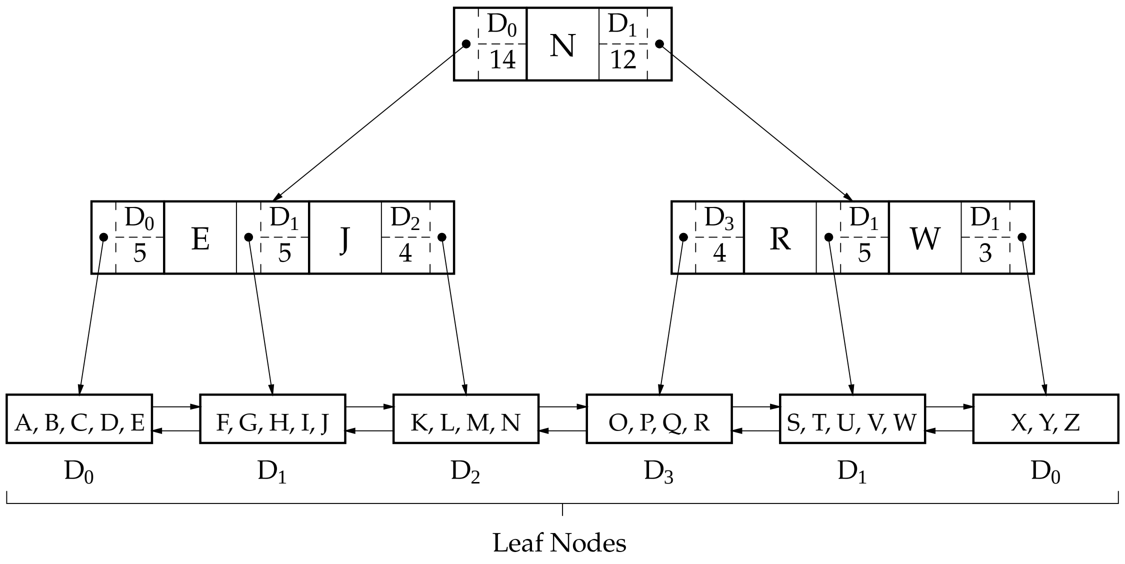


Figure 4.5: Modified B$^+$-Tree File Structure

then traverses right (assuming C has a positive sign) from K, summing record counts from index entries until C is equaled or exceeded at key K_b; the query <K,K$_b$> bounds the leaf nodes that must be retrieved to service the <K,C> query. Note that the above process traverses down the tree only up to the last index node level and does not dereference the leaf node pointers. With record counts stored in index nodes, the cost for searching the index structure to locate the bounds of a <K,C> style query is very low because the index pages for low to moderately sized B$^+$-Trees are usually cached in memory and thus require no disk accesses.

The cost paid for the above modifications to the B$^+$-Tree is reduced fan-out caused by the reduction in number of index entries per node because of the increased size of each entry. However, this overhead becomes significant only when the average length of the keys stored in the index nodes is relatively small.

4.5.3 Concurrent Retrieval Algorithm

For retrievals, a specialized form of the generic concurrent retrieval algorithm described in section 3.3.4 is used. During execution of a query on the B^+-Tree, very little processing takes place in the index nodes. Therefore, a single task is employed to traverse the index portion of the B^+-Tree.

4.5.4 Experimental Analysis

Recall, the machine used for experiments was a Sequent Symmetry with 10 processors and 8 disk drives, of which 4 were used (see section 4.1). In each experiment, 1000 <K,C> range queries were processed, where each individual query consisted of reading a random number of sequential records starting at a randomly selected key. The average query size was 2000 records. A control experiment was performed first, in which the B^+-Tree had only a single partition. The code executed is the same as in the partitioned case but there is no parallelism at the back end from the single partition. The partitioned B^+-Tree experiments were conducted with 4 partitions and the application program did no processing on query results. Thus, the application program did not introduce any delays.

As before, the performance parameters measured in each experiment included the elapsed time, the total CPU time over all processes and the total number of major page faults (see section 4.1.3). These parameters provide real time evaluation of the concurrent retrieval algorithm, and the partitioning algorithm. The results obtained for the partitioned B^+-Tree are presented in table 4.3 and figure 4.6. The largest decrease in elapsed time is from 1 to 2 processors because there are 2 channels allowing 2 disks to transfer data without contention. After that, the decrease is less because of the data transfer contention on the channels, until the elapsed time begins to rise because of this contention.

Nevertheless, a speedup of 3.2 with 4 disks is quite noteworthy. Note that the graphs for the single disk case have only two data points. With only a single disk to work with, there is no benefit derived by having multiple CPUs to do the I/O because all I/O requests are serialized at the disk and a kernel thread blocks until a page fault is serviced. As a result, there is no saving in elapse time. In fact, the time increases slightly as the number of CPUs is increased because of the extra contention introduced by multiple CPUs accessing the single disk.

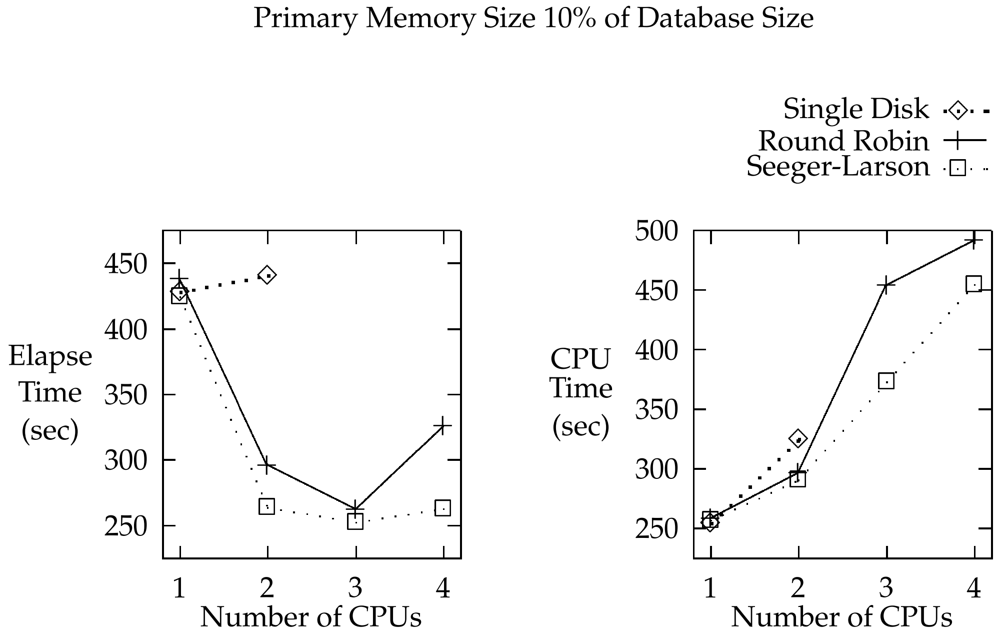


Figure 4.6: Comparison of Single Disk B$^+$-Tree with Four Disk B$^+$-Tree

During execution of the program, statistical information is collected in the LRA to measure the effectiveness of the two partitioning algorithms for the query set being studied. First, the total number of retrieval requests, indicating total I/O to be done, received by the LRA is maintained. Second, the distribution of these requests over various disks is

Primary Memory Size 10% of the Database Size

		Round Robin Partitioning			Seeger-Larson Partitioning		
Num Disks	Number CPUs	Total CPU Time	Elapsed Time	Num Page Faults	Total CPU Time	Elapsed Time	Num Page Faults
1	1	254	427	13458	254	427	13458
	2	324	440	13313	324	440	13313
4	1	259	438	13655	256	424	13623
	2	297	296	13545	290	263	13639
	3	454	262	13754	372	252	13646
	4	491	325	13650	454	262	13754

Table 4.3: Comparison of Single Disk B$^+$-Tree with Four Disk B$^+$-Trees

measured. An even distribution across all disks indicates that the partitioning algorithm achieved good overall load balance and good throughput. However, global load balance in itself is not sufficient to achieve good local load balance and response time for individual queries. For example, if each query retrieves data from a single disk, but individual queries are spread evenly over all disks, the statistical information will indicate an even overall distribution of disk retrievals. However, there would be no improvement in the response time of any one query and neither is there any improvement in throughput.

What is needed, therefore, is another criterion to measure the effective performance gain that takes into account the gain achieved by individual queries. For this purpose, a new parameter, called the *performance gain*, is defined to serve as a theoretical measure of the gain achieved by partitioning the file structure across multiple disks. Let Q_0, Q_1, ..., Q_{N-1} be the list of queries executed on a B$^+$-Tree partitioned across D disks. For the i-th query, let d_{ij} $(0 \leq j < D-1)$ be the number of leaf nodes retrieved from disk j. $D_j = \sum_{i=0}^{N-1} d_{ij}$ is the total number of leaf nodes retrieved from disk j to process all N queries and $Total = \sum_{j=0}^{D-1} D_j$ is the total number of leaves retrieved from all disks. Now, for the i-th query, $Total_i = \sum_{j=0}^{D-1} d_{ij}$ is the total number of disk reads, and $Max_i = \max_{j=0}^{D-1} d_{ij}$ is the *minimum* number of serialized disk reads required for executing the query; in other words, the disk with the largest number of I/O operations is the bottleneck and dictates the shortest possible time to process the query. Hence, $Total_i/Max_i$ indicates the maximum speedup possible by executing the i-th query in parallel. If an individual query accessed data equally from all disks, this number for the query is equal to D indicating a D fold speedup in the parallel execution of the query.

The performance gain over all queries is computed as $Total/\sum_{i=0}^{N-1} Max_i$ and provides a theoretical measure of the effectiveness of the partitioning algorithm alone. The performance gain, as computed for the round robin and the Seeger-Larson partitioning algorithms, is shown in Table 4.4. As can be seen, the Seeger-Larson algorithm performs much better than the round robin algorithm. However, in practice, the round robin algorithm performs reasonably well given its simplicity. Further, a sequential reading of the entire B$^+$-Tree indicated that the Seeger-Larson algorithm partitioned the B$^+$-Tree almost perfectly with a performance gain of approximately 3.95 with four disks. The corresponding performance gain for the round robin algorithm was slightly lower at 3.6.

Effect of Employing Extra Segments for Retrieving Data

In the experiments described so far, all the retriever tasks operated on a single mapping created by the representative (see Figure 4.7(a)) with its own page table and resident set. This arrangement can lead to some interference among the access patterns of individual retriever tasks because all the retriever tasks share the same resident set in primary storage. Thus, a page fault generated by one retriever task can potentially remove a page that might be needed immediately by another retriever task. For the B$^+$-Tree experiments, however, the effect is not likely to be large because of the uniform distribution of requests. To test this conjecture, another set of experiments was run, where additional

Num	Leaf	Disk Counts					Perf.
Disks	Count	D0	D1	D2	D3	Max	Gain
1	15189	15189	-	-	-	15189	1.000
4	15189	4065	4057	3534	3533	6144	2.472

(a) Round Robin Partitioning

Num	Leaf	Disk Counts					Perf.
Disks	Count	D0	D1	D2	D3	Max	Gain
1	15189	15189	-	-	-	15189	1.000
4	15189	3728	3824	3835	3802	4883	3.111

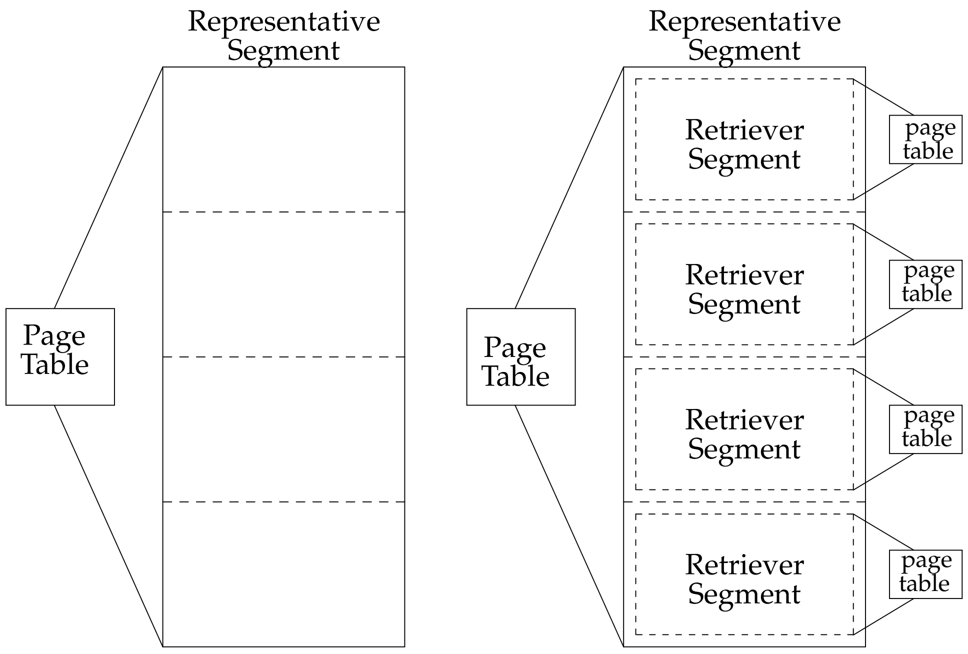
(b) Seeger-Larson Partitioning

Table 4.4: Expected Effectiveness of B^+-Tree Partitioning Algorithms

segments were created for the exclusive use of the retriever tasks, while the representative retained its own original segment (see Figure 4.7(b)). The retriever tasks were distributed evenly across the additional segments; the number of additional segments is a control variable. The additional segment created for a retriever task mapped the corresponding partition, of the representative segment, in its own address space, with its own page table and resident set.

The results of the experiments with extra retriever or worker segments are presented in Table 4.5[2]. With no worker segments, all the tasks execute on the representative segment, and therefore, any available CPU can execute any ready task. However, when worker segments are created, each with one CPU, the tasks are partitioned into disjoint subsets with each subset executing on a different segment; the retriever tasks are distributed uniformly across the available worker segments and all remaining tasks such as

[2]The last 4 rows from table 4.3 have been reproduced for easy reference.



(a) A single mapping (b) Extra mappings for Retrievers

Figure 4.7: Using Multiple Segments for Retriever Tasks (FSTs)

the LRA, the FSTs and the iterator stay on the representative segment.

Unfortunately, this partitioning of tasks is detrimental to parallelism afforded by multiple CPUs. When there is only one worker segment (rows 5 and 6 of table 4.5), all the retriever tasks use one CPU and the remaining tasks are executed by the CPU(s) on the representative segment. As a consequence, the representative segment CPUs spend significant amounts of time waiting for the worker segment CPU to retrieve data from disks via one of the retriever tasks. During this wait, a representative segment CPU either spins or goes to sleep depending upon the spin time configuration[3] and the actual

[3]Setting the spin time appropriately involves a trade-off between CPU and elapsed times. If the spin time is too small, the CPU may go to sleep too frequently resulting in lower CPU times but higher elapsed times caused by the cost involved in waking a CPU up. On the other hand, an excessively large value of spin time can result in wasted CPU cycles. For the experiment presented in this dissertation, the spin time

waiting time.

Primary Memory Size 10% of Database Size

Number of Disks = 4			Round Robin Partitioning			Seeger-Larson Partitioning		
Number Worker Segments	CPUs on Rep Segment	Total CPUs	Total CPU Time	Elapsed Time	Num Page Faults	Total CPU Time	Elapsed Time	Num Page Faults
0	1	1	259	438	13655	256	424	13623
	2	2	297	296	13545	290	263	13639
	3	3	454	262	13754	372	252	13646
	4	4	491	325	13650	454	262	13754
1	1	2	533	430	13720	533	362	13711
	2	3	743	506	14030	755	433	14090
2	1	3	338	361	13152	337	301	13131
4	1	5	304	426	13118	303	416	13137

Table 4.5: Effect of Extra Worker Segments on Concurrent B^+-Tree Retrievals

As can be seen from table 4.5, using a single worker segment with one CPU on the representative segment (rows 5-6) generates approximately the same number of page faults as using no worker segments (rows 1-4) because all the retriever tasks are still sharing the same segment. It is not clear why a slight increase in the number of page faults occurs with two CPUs on the representative segment when a single worker segment is employed (row 6). At the same time, the use of a single worker segment (rows 5-6) substantially increases the total amount of CPU time because of the spinning of the CPU(s) mentioned before. As well, the elapsed time (rows 5-6) increases because of the loss of data parallelism caused by reducing the number of CPUs available for the retriever tasks. When the number of worker segments is increased to 2 (row 7), there are

was set to 1 ms, which is the cost of waking up a UNIX process on the testbed used.

indeed fewer page faults generated, thus confirming the hypothesis about the interference among different retriever tasks. The total CPU time is reduced because now the representative segment CPU communicates in parallel with two worker segment CPUs, and therefore, spends significantly less time spinning. The elapsed time (row 7) also comes down because of the two fold data parallelism made possible by the two worker segment CPUs to execute the retriever tasks. Finally, when the number of worker segments is further increased to 4 (row 8), data parallelism increases so that retrieval requests generated by the tasks on the representative segment are processed faster; hence, the representative segment CPU(s) have higher utilization, which means less spinning, and hence, a lower CPU time. The elapsed time does not reduce any further because the amount of data parallelism is restricted by the number of available disk controllers.

Thus, the reduction in the number of page faults afforded by the extra worker segments is not substantial and does not offset the overhead introduced by the additional segments, as is evident from comparing the elapsed times in table 4.5 when the total number of CPUs employed is taken into account. For example, faster elapsed time is achieved by employing three CPUs on the representative segment instead of splitting the CPUs across one representative and two worker segments in spite of the slightly reduced number of page faults caused by the extra worker segments in the latter case.

These results seem to disfavour the use of additional segments for retriever tasks. Hardware systems with more advanced virtual memory capabilities might make this approach viable in the future. What is noteworthy is that μDatabase allows for all these different options to take advantage of available hardware.

4.6 Partitioned R-Tree

The R-Tree file structure (see section 4.3.2) was modified and partitioned to achieve parallel execution of queries. Note that point queries as well as window queries on an R-Tree are range queries capable of benefiting from parallel execution. Devising efficient partitioning algorithms for the R-Tree is much more complicated than for the B^+-Tree because there is a significant amount of computation that takes place in the index nodes of an R-Tree during traversal. The nature of computation depends upon the query being executed and influences how the tree should be partitioned to achieve good performance.

4.6.1 Partitioning Algorithms

For partitioning an R-Tree across multiple disks in the EPD environment and for executing queries (point, enclosure, intersection, containment) in parallel, a round-robin partitioning algorithm was used. For the purposes of this study, data rectangles, as opposed to pointers to data rectangles, are stored in leaf nodes of the R-Tree. This structure trivially ensures that the data portion of the R-Tree is spread over various disks without requiring any special attention and does not affect the outcome or the validity of the experiments.

As in the case of the B^+-Tree, the round robin partitioning algorithm is used for striping the R-Tree: when a node needs to be split, the next round robin disk is chosen for storing the new node. On average, the round robin algorithm is expected to provide reasonable performance and has minimal computational cost during partitioning.

4.6.2 Modified File Structure

The only modification made to Guttman's Linear R-Tree [Gut84] is the addition of an extra field to the next node pointer in each index entry. The extra field consists of the disk number containing the referent node and allows the LRA (see section 3.3.4) to determine, by analyzing an index entry, the disk on which a leaf node is stored. The resulting reduction in fan out of the R-Tree is not significant, because the extra space taken by the new field is small compared to the total size of an entry containing a multi-dimensional rectangle.

4.6.3 Concurrent Retrieval Algorithm

For retrievals, a specialized form of the generic concurrent retrieval algorithm described in section 3.3.4 is used. During execution of a query on the R-Tree, processing of index nodes is very computation intensive. Therefore, instead of using a single task to traverse the index portion of the R-Tree, provision is made to employ multiple file structure traverser (FST) tasks (see figure 3.6). The effect, on performance, of varying the number of these tasks is studied in the experiments conducted on the partitioned R-Tree.

4.6.4 Experimental Analysis

The round robin partitioning algorithm was implemented and studied. The experimental structure used is analogous to that used for the B^+-Tree partitioning experiments described in section 4.5.4. In all the experiments described in the rest of this section, the primary memory size is 5% of the database size. The results presented in Figure 4.8 provide a measured comparison of the single-disk R-Tree with a four-disk partitioned R-Tree. For all of these experiments 4 FST tasks were employed.

Expectedly, the partitioned R-Trees perform much better than a single-disk R-Tree. Another point to note about the results is the fact that, unlike for the B^+-Tree, elapsed time goes down for the single disk case when the number of CPUs is increased from one to two because of the CPU parallelism for processing that takes place at the directory

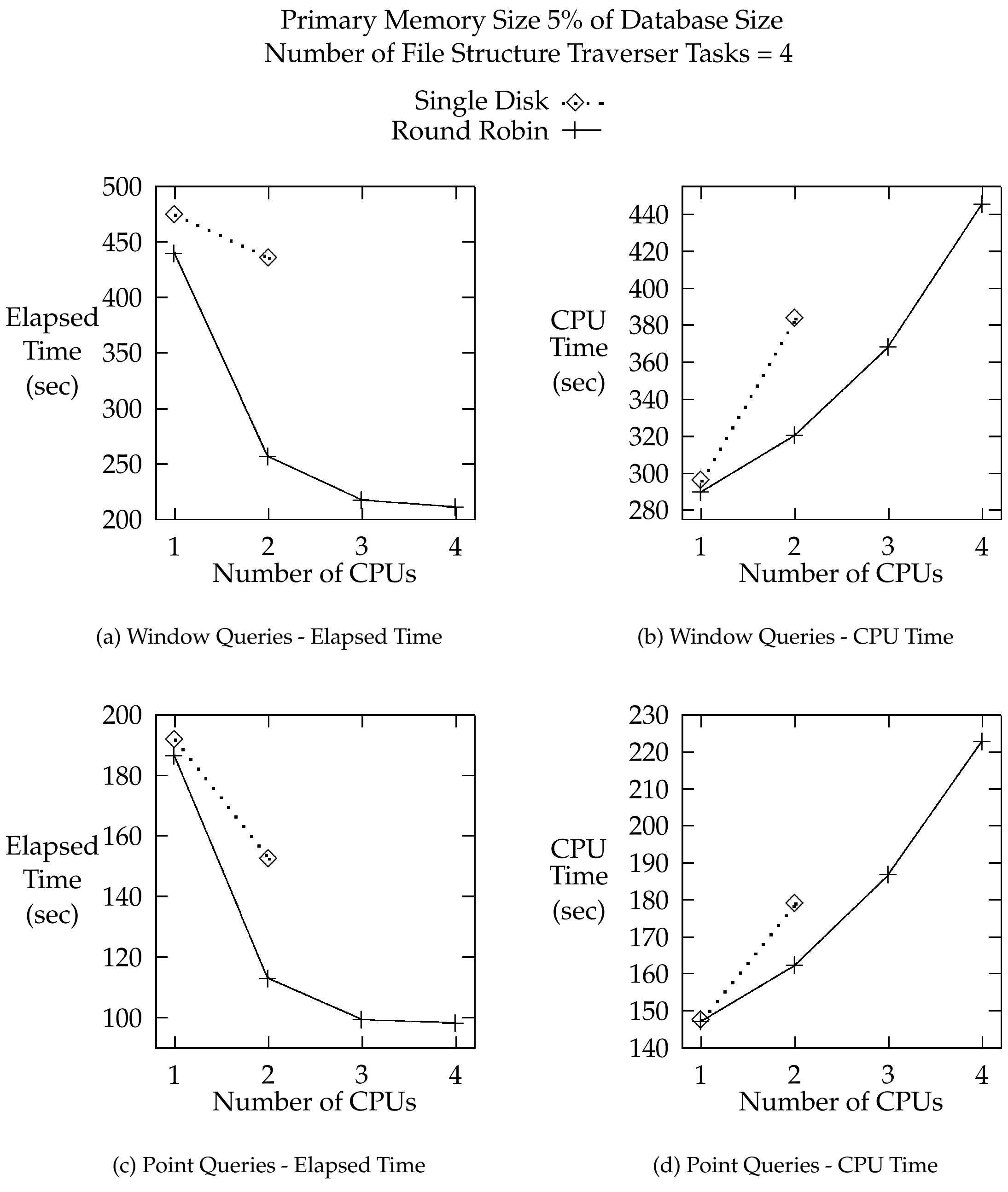


Figure 4.8: Comparison of Single Disk R-Tree with Four Disk R-Trees

nodes of an R-Tree.

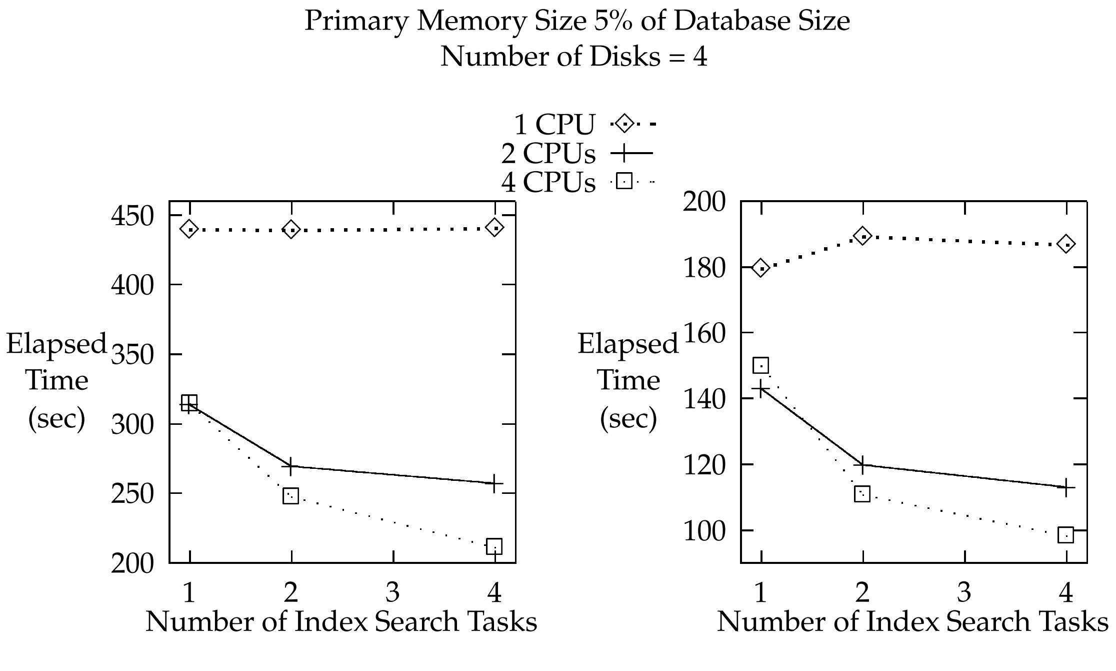
The performance gain parameter, as defined in section 4.5.4, is also computed for the R-Tree partitioning algorithm. These results are presented in Table 4.6.

Primary Memory Size 5% of Database Size

Num	Leaf	Disk Counts					Perf.
Disks	Count	D0	D1	D2	D3	Max	Gain
Window Queries							
1	32359	32359	-	-	-	32359	1.000
4	32359	7752	8291	7678	8638	10662	3.035
Point Queries							
1	11491	11491	-	-	-	11491	1.000
4	11491	2513	2986	2556	3436	4741	2.424

Table 4.6: Theoretical Effectiveness of Round-Robin R-Tree Partitioning Algorithm

Further experiments were conducted to investigate the effect of employing multiple FST tasks. In the case of the B^+-Tree, there is very little computation carried out in the index nodes during execution of a query. Therefore, a single task is able to traverse the index without creating a bottleneck. On the other hand, an R-Tree index search involves a significant amount of computation within each index node starting from the root of the tree. This computation raises the possibility that using a single task to traverse the index portion of the tree may create a bottleneck if the single task is unable to generate the list of leaf nodes for the LRA at a high enough speed. One solution is to divide the work of traversing the index portion among a number of tasks. For example, two concurrent tasks can be made to work on odd and even entries of an index node respectively. Figure 4.9 contains the results obtained by varying the number of tasks that search the index. In addition, tables 4.7 and 4.8 also tabulate the actual number of page faults generated in each case.



(a) Window Queries - Round Robin

(b) Point Queries - Round Robin

Figure 4.9: Using Multiple Index Search Tasks to Perform Parallel Queries

In all cases, where more than one processor is employed, an improvement in the elapsed time is obtained by using multiple index searching tasks. With only one CPU, there is not much to be gained by increasing the number of FSTs; indeed, the extra contention can even deteriorate performance slightly. With multiple CPUs, the results clearly establish that using multiple FSTs is beneficial. The most benefit is derived by increasing the number of FSTs from one to two, which causes an increase in the speed at which leaf node references are presented to the LRA for processing. A more detailed explanation of these results is provided later in this section.

As can be seen from tables 4.7 and 4.8, the number of page faults is not affected by an increase in the number of FSTs in most cases because the amount of I/O to be done is still the same. However, an apparent anomaly occurs for some cases (e.g., for the single disk

Primary Memory Size 5% of Database Size

Num Rep CPUs	Index Search Tasks	CPU Time	Elapsed Time	Num Page Faults
Single Disk				
1	1	282	446	14275
	2	291	470	13611
	4	295	473	12919
2	1	394	453	15073
	2	375	428	13934
	4	383	435	13597
4 Disks		Round Robin Partitioning		
1	1	283	439	13464
	2	285	438	12737
	4	290	440	12403
2	1	339	313	13520
	2	314	269	13045
	4	320	257	12815
3	1	431	295	13547
	2	379	239	13538
	4	368	217	13825
4	1	570	314	14261
	2	489	247	14206
	4	445	211	14292

Table 4.7: Using Multiple Index Search Tasks for Parallel Window Queries

case in table 4.7) whereby the number of page faults decreases slightly as the number of FSTs is increased from one. The cause of this anomaly needs to be investigated further.

Experiments were also conducted to study the effect of interference among retrieval tasks working on the same segment. These results are presented in tables 4.9 and 4.10. In order to analyze the results presented in tables 4.7 through 4.10, the algorithm of figure

Primary Memory Size 5% of Database Size

Num Rep CPUs	Index Search Tasks	CPU Time	Elapsed Time	Num Page Faults
Single Disk				
1	1	143	178	3318
	2	145	186	3259
	4	147	191	3309
2	1	187	174	3822
	2	173	149	3522
	4	178	152	3467
4 Disks		Round Robin Partitioning		
1	1	143	179	3252
	2	143	189	3273
	4	147	186	3278
2	1	173	142	3410
	2	158	119	3345
	4	162	112	3426
3	1	221	143	3621
	2	190	112	3623
	4	186	99	3651
4	1	284	149	3967
	2	240	110	3920
	4	222	98	3982

Table 4.8: Using Multiple Index Search Tasks for Parallel Point Queries

3.6 is reduced to a queueing system as shown in figure 4.10.

The LRA accepts leaf node references from the FST(s) and queues them up for the retriever tasks. The retriever tasks dereference the pointers and queue the resulting data for the iterator task to fetch on demand. The interface between the LRA and the FSTs can be considered a producer-consumer interface with a bounded buffer of size M, where M

Primary Memory Size 5% of Database Size

Number of Disks = 4			Round Robin Partitioning		
Num Worker Segments	Num Rep CPUs	Index Search Tasks	CPU Time	Elapsed Time	Num Page Faults
1	1	1	361	441	15562
		2	379	462	15485
		4	383	473	15526
	2	1	481	454	15809
		2	452	435	15841
		4	455	441	15756
2	1	1	377	340	14315
		2	394	338	14203
		4	406	344	14374
	2	1	498	349	14730
		2	462	306	14729
		4	473	305	14719
4	1	1	414	307	14962
		2	438	290	15004
		4	444	272	14978
	2	1	534	314	15310
		2	493	251	15344
		4	514	231	15306

Table 4.9: Using Multiple Representative Segments for Parallel Window Queries

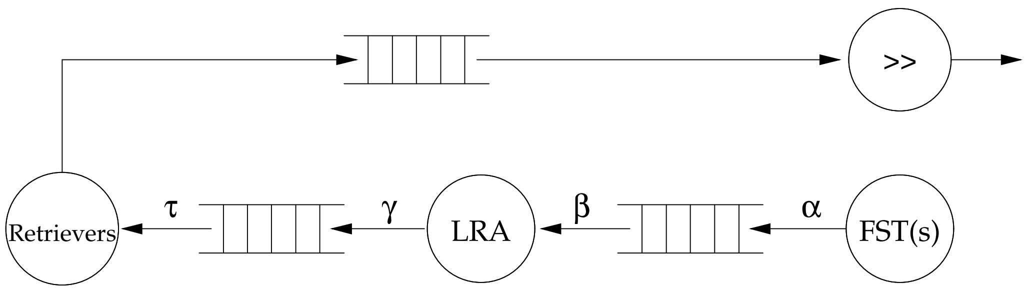


Figure 4.10: Queueing System for the Generic Concurrent Retrieval Algorithm

Primary Memory Size 5% of Database Size

Number of Disks = 4			Round Robin Partitioning		
Num Worker Segments	Num Rep CPUs	Index Search Tasks	CPU Time	Elapsed Time	Num Page Faults
1	1	1	171	177	4585
		2	179	191	4601
		4	184	196	4573
	2	1	226	184	4783
		2	207	168	4776
		4	211	170	4773
2	1	1	181	152	4029
		2	194	161	4107
		4	194	165	4074
	2	1	235	156	4188
		2	214	134	4245
		4	220	134	4187
4	1	1	189	145	4042
		2	204	151	3054
		4	209	151	3068
	2	1	249	153	4285
		2	226	123	4236
		4	235	120	4338

Table 4.10: Using Multiple Representative Segments for Parallel Point Queries

is the number of FSTs, i.e., each FST acts like a node in a buffer after it has computed a request but blocks because the LRA is busy. The rate, α at which items are placed in the buffer for the LRA is a linear function of M, i.e., doubling the number of FSTs doubles the rate of arrival at the buffer. A similar producer-consumer relationship exists between the LRA and the retriever tasks, and between the retriever tasks and the iterator task. The number of effective consumers for the data generated by the LRA is the number of CPUs (or the number of worker segments) executing the retriever irrespective of the number of retriever tasks because a retriever task is blocked on I/O most of the time, which correspondingly blocks the CPU it is executing on. This situation is analogous to having a single consumer whose rate of consumption, τ, is a linear function of the number of worker segments, N. Finally, the LRA, all the FST tasks and the iterator share the P CPU(s) available on the representative segment.

An informal analysis of the queueing system in figure 4.10 is employed to explain the results. There are three control variables, M, N and P. The effect of changing each of these variables is discussed next.

Effect of changing N, the number of worker segments: Increasing N makes more CPUs available for performing parallel I/O from the disks resulting in reduced elapsed time. The reduction in elapsed time diminishes as the number of segments approaches the number of available disk controllers. Also, in order for the parallelism to be fully exploited, the rate of production, γ, for the LRA must be high enough to ensure that the queue for the retriever tasks is kept non-empty, which in turn implies that α must be high enough to ensure that the LRA's input queue is non-empty. Thus, in order for the algorithm to benefit optimally from an increase in N, the representative segment must have matching resources. Finally, if the matching is not perfect, an increase in total CPU time is expected from the increase in the number of CPUs employed due to spinning.

Effect of changing P, the number of CPUs on the representative segment: The goal of increasing resources on the representative segment is to increase γ, which can be exploited by the retriever tasks assuming they are not blocked on I/O. Since the LRA does not do much work, γ can only be increased by ensuring that the LRA's input queue is non-empty, i.e., by increasing α. With one FST, increasing the number of CPUs does not provide any benefits because the FST blocks after generating a request if the LRA is busy because there is no actual buffer between them, i.e., the FST makes synchronous calls to the LRA to deliver a request. In fact, performance may deteriorate a little due to the extra scheduling contention of multiple CPUs trying to execute a single task. With multiple FSTs, increasing the number of CPUs helps because the FSTs can exploit the extra processing power to increase α, so a reduction in elapsed time is expected. In all cases, increasing the number of CPUs on the representative segment should result in increased CPU times caused by spinning.

Effect of changing *M*, the number of FSTs: For elapsed time, there are two distinct phenomena that need consideration. As stated earlier, increasing the number of FSTs increases α, which should result in reduced elapsed time as long as the LRA is not blocked waiting for retriever tasks. On the other hand, the extra FSTs increase the average length of the ready queue for the representative segment cluster, and therefore, the iterator task runs a little slower because it has to contend with an increased number of tasks for the available CPUs. A slowing down of the iterator task has the effect of increasing elapsed time. In order for the overall elapsed time for the algorithm to decrease, the increased elapsed time of the iterator task must be offset by a reduction in elapsed time caused by an increased γ.

With one worker segment, there is a bottleneck at the disks (i.e., no data parallelism), and as such, no gain can be made by increasing γ. Consequently, the increased elapsed time due to the iterator is not offset and there is an overall increase in elapsed time. With multiple worker segments, an increase in γ causes an improvement until the retriever CPUs once again become the bottleneck. Thus, when *M* is increased the elapsed time reduces initially, levels off when the retriever tasks become the bottleneck, and eventually increases due to a slowing of the iterator task (as mentioned above). The leveling off point depends upon the number of worker segments – with more worker segments the bottleneck is achieved at a higher number of FSTs.

For CPU time, there are two distinct components: time during which computation is done and time when the CPU(s) spin waiting for work before going to sleep. Because there is a fixed amount of work to be performed by the FST(s), when M is increased, the total amount of computation actually increases slightly because of the overhead of the extra task(s). However, the spinning time is affected by the total elapsed time – in general, an increase in elapsed time (such as the ones described above) results in an increase in the CPU time because of the extra spinning in between computation. Thus, with an increase in M, CPU time follows the same pattern as elapsed time. An exception occurs when M becomes more than P and there is an extra increase in CPU time that I am unable to explain.

As with the B$^+$-Tree, it was found that the effect on page faults of using the worker segments is only marginal and does not improve performance to any significant degree. The reduction in page faults is not significant enough to offset the extra overhead of using additional segments as evidenced by the large increase in CPU and elapsed times.

4.7 Summary

This chapter demonstrated the feasibility and viability of the EPD approach to memory mapping by comparing EPD file structures with their traditional counterparts. The experiments were conducted on a custom designed testbed and clearly showed that, for a variety of access patterns, the EPD environment provides performance that is comparable to that of a traditional LRU buffer manager. Subsequent chapters will show that the

EPD approach requires special page replacement support to be competitive in certain situations, but in most cases this is unnecessary. In addition, it was demonstrated that the EPD approach works particularly well when EPD file structures compete with other applications at execution time because all memory management is supervised by the operating system. This chapter further investigated the issue of parallelism in an EPD system by conducting experiments with parallel access methods.

Chapter 5

Application and Validation of the Analytical Model

This chapter presents the design and analysis of three new parallel join algorithms in the EPD system. The analysis is done according to the general procedure described in section 3.5.3 and validated by conducting experiments described later in this chapter.

The *validation* of the model, as with other cost models, is an attempt to establish confidence in the accuracy of the model and is done by the following method. An algorithm is chosen for validation purposes and an analysis of the algorithm is performed within the context of the analytical model of the system. The analysis of the algorithm, the values of the parameters of the model and a description of a chosen data set is used to predict performance behaviour of the algorithm on the specified physical machine. The results of the analysis are compared against the performance measurements obtained by running experiments with the algorithm on the specified machine and with the chosen data set. A close match between prediction and actual behaviour establishes the accuracy of the model for that particular environment and the model can be used with a degree of confidence for predicting the behaviour of the algorithm under varying circumstances.

Further, it needs to be emphasized that the goal of this work is to develop a new model for the EPD system because none of the existing models apply, and to use the new model to study new algorithms in the EPD system by being able to predict their performance on a physical machine. It is *not* the goal of this work to contrast the performance of algorithms in the EPD to other systems.

5.1 Parallel Pointer-Based Join Algorithms

In order to validate the analytical model of the EPD system, parallel pointer-based versions of three join algorithms were designed, implemented and analyzed: nested loops, sort-merge, and a variation of Grace [KTMo83]. "Because any data model supporting sets and lists requires at least intersection, union, and difference operations for large sets, I believe that [the discussion of join algorithms] is relevant to relational, extensible, and object-oriented database systems alike." [Gra94] In each case, a complete join of one relation, *R*, with another, *S*, is considered. The prediction from the analysis was compared against results obtained through experiments conducted with the parallel join algorithms.

The use of (location) pointers in an EPD environment provides a unique advantage with respect to joins and other algorithms. To demonstrate this advantage, the traditional join algorithms were modified so that the join attribute embedded in an object stored in the *R* relation is a pointer to an object in the *S* relation. Such algorithms, called pointer-based join algorithms, are ideal for an EPD environment and result in significant performance advantages; the most important being that a pointer provides the ordering of objects in *S*, which can be exploited to eliminate the usual sorting or hashing of *S* in sort-merge and hash-based joins, respectively. Note that in the conventional join algorithms the ordering of the two joining relations is not important, i.e., either of the two relations can be joined with the other producing with the same result. This feature is no longer available for pointer-based join algorithms, unless the *S* relation contains back-pointers to objects in the *R* relation.

Further, each join algorithm is parallelized so that the data is partitioned across several disks and the join performed in parallel on individual disks. It is assumed that *S* is initially partitioned on *D* disks into equal-sized partitions $S_1, \ldots, S_D$ and that the partition in which a particular object of *S* resides can be computed from a pointer to that object. The time for computing this mapping is denoted by *map*[1]. In addition, *R* is also assumed to be divided into equal-sized partitions $R_1, \ldots, R_D$. Excellent partitioning algorithms exist for different kinds of data structures (see sections 4.5 and 4.6). It is assumed that join attributes are randomly distributed in *R*. Finally, each relation is managed by a process (*Rproc* and *Sproc*, respectively), which is aware of the structure of the relations, and in particular, *Rproc* is capable of carrying out the join itself.

The following parameters are defined for various relations and their subsets. $|X|$ denotes the number of objects in X, P_X is the number of pages in X, and x denotes the size of a single object in relation X.

For the algorithms, private memory is viewed as being divided into *D* pieces, where the *i*-th piece is associated with partition R_i. An algorithm is described as it progresses on the *i*-th piece, with the understanding that work on the remaining $D-1$ pieces is progressing in an analogous fashion in parallel. Each $Rproc_i$ is a lightweight task; the number of real processors available for these tasks is a control parameter; *D* processors each for *R* and *S* are usually employed to achieve maximum parallelism (see figure 5.1). The partitions of *R* are conceptually divided into sub-partitions based on the partitions of *S* to which the join attributes refer; the subset of R_i with join attributes referring to objects in partition S_j is called $R_{i,j}$. R_{S_j} denotes the set of all objects in *R* that have pointers

[1]Such a computation is feasible in the EPD environment where multiple file structure partitions are mapped into a single segment and the mapping of each partition consists of a distinct range of virtual addresses. For other schemes, the join attribute could be made a composite field with an embedded partition number.

to objects in S_j, i.e., $R_{S_j} = \bigcup_{i=1}^{D} R_{i,j}$. This substructure is illustrated in the figures for subsequent algorithms. For a given i, the $R_{i,j}$ sub-partitions may have some skew in size because objects in R_i may contain more references to some S_j and fewer to others; the amount of skew is defined as $skew = \max_{j=1}^{D}\left(\frac{|R_{i,j}|}{|R_i|/D}\right)$. Skew is important as it affects the performance of certain algorithms.

All the parameters introduced in this section are tabulated in table 5.1, which also contains two parameters, *sptr* and *G*, to be described later. Additional parameters are defined for each specific algorithm as needed. Finally, because every algorithm forms and outputs the same join, the analysis does not count the time to perform this step, nor does it assume that the join results are generated in any particular order.

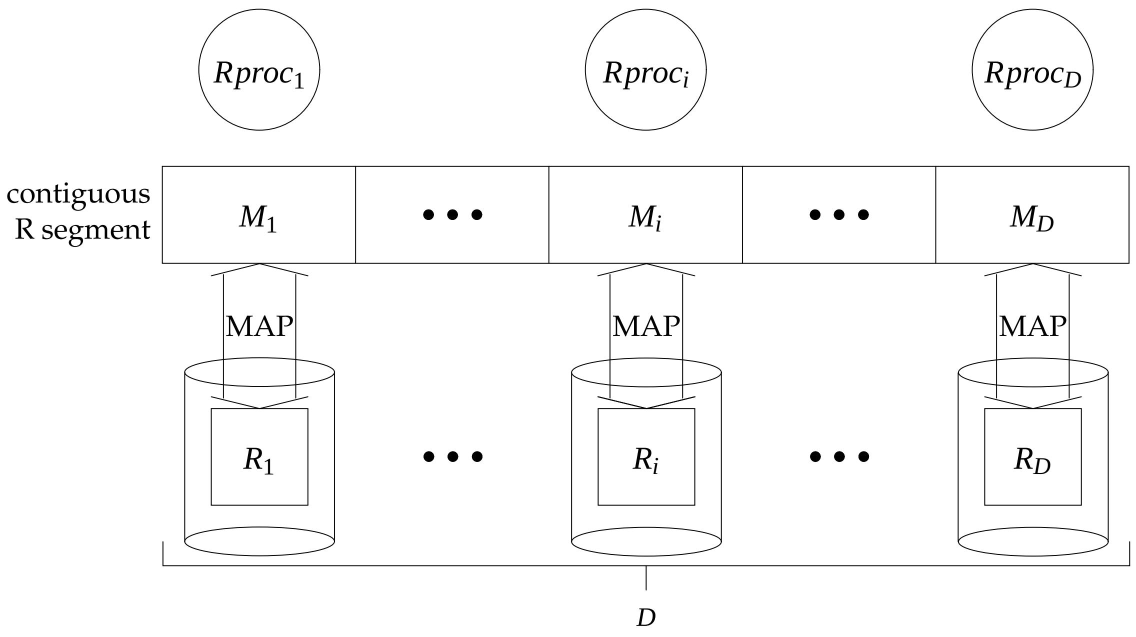


Figure 5.1: Segment Partitioning Structure

The analyses of join algorithms computes quantities of time that can be summed to give the total elapsed time for $Rproc_i$. Because there is little or no contention during the

Variable	Description
R, S	two joining relations
$R_1, \cdots, R_D$	partitions of R across D disks
$S_1, \cdots, S_D$	partitions of S across D disks
$Rproc$	process to manage R
$Sproc$	process to manage S
$Rproc_i$	light-weight task to manage R_i
$Sproc_i$	light-weight task to manage S_i
$R_{i,j}$	subset of R_i with join attributes pointing to S_j
R_{S_j}	subset of R with join attributes pointing to S_j

Parameter	Description
map	time to map a join attribute to the referent S partition
$\|X\|$	number of objects in relation X
P_X	number of pages in relation X
x	size of a single object in relation X
$skew$	skew in relative sizes of R_i sub-partitions
$sptr$	the size of a pointer to an S-object
G	size of the shared buffer used for transferring data out of S

Table 5.1: Variables and Parameters Used for all Joins

D-fold parallelism[2], the total elapsed time for $Rproc_i$ also represents the total time for the entire join. To account for the effect of skew, the maximum of the elapsed time for the various $Rproc_i$ is taken to be the time for the join.

While it is convenient to speak of data being read or written in the algorithms, input and output is not explicitly requested by any of the algorithms. When speaking of reading a block of data, the implementation actually accesses a location in virtual memory mapped to that block. If the block is not in primary memory, it is read in by means of

[2]The contention for S is eliminated by the scheduled reading of objects from S, as explained later in this chapter.

a page fault; otherwise, no disk access takes place. Similarly, when speaking of writing a file, no explicit action, other than to write cells of virtual memory, occurs in the implementation; the writing of a (dirty) block of data takes place when the corresponding page is replaced by the operating system. These actions are similar to what occurs in an explicitly managed buffer pool, where objects are fetched from already read buffers and written only when the buffer is written, albeit with more user control than in memory mapping.

5.2 Parallel Pointer-Based Nested Loops

Nested loops performs a join by sequentially traversing R. For each object in R, the algorithm accesses the S-object pointed to by the embedded join attribute. R is called the outer relation and S the inner. The resulting random accesses to S significantly slow nested loops. A naive parallel version may partition R and S so that each R_i can perform its join in parallel with other R partitions, accessing different S_j partitions simultaneously. However, parallelism in this case is severely inhibited by contention when several R_i reference the same S_j; this contention can be reduced or eliminated by careful algorithm design.

In the traditional nested loops algorithm, it is usually the smaller of the two relations that is used as the inner relation so that it can be kept in the buffer pool. In the EPD algorithm, S is always the inner relation unless back pointers from S to R are available in S objects.

5.2.1 Algorithm

For each partition R_i in parallel, the algorithm operates in two passes. In pass 0 (see figure 5.2), R_i is read, one object at a time, into the private memory of $Rproc_i$, which translates, in terms of actual I/O, to reading R_i in chunks of the virtual memory page size, B.

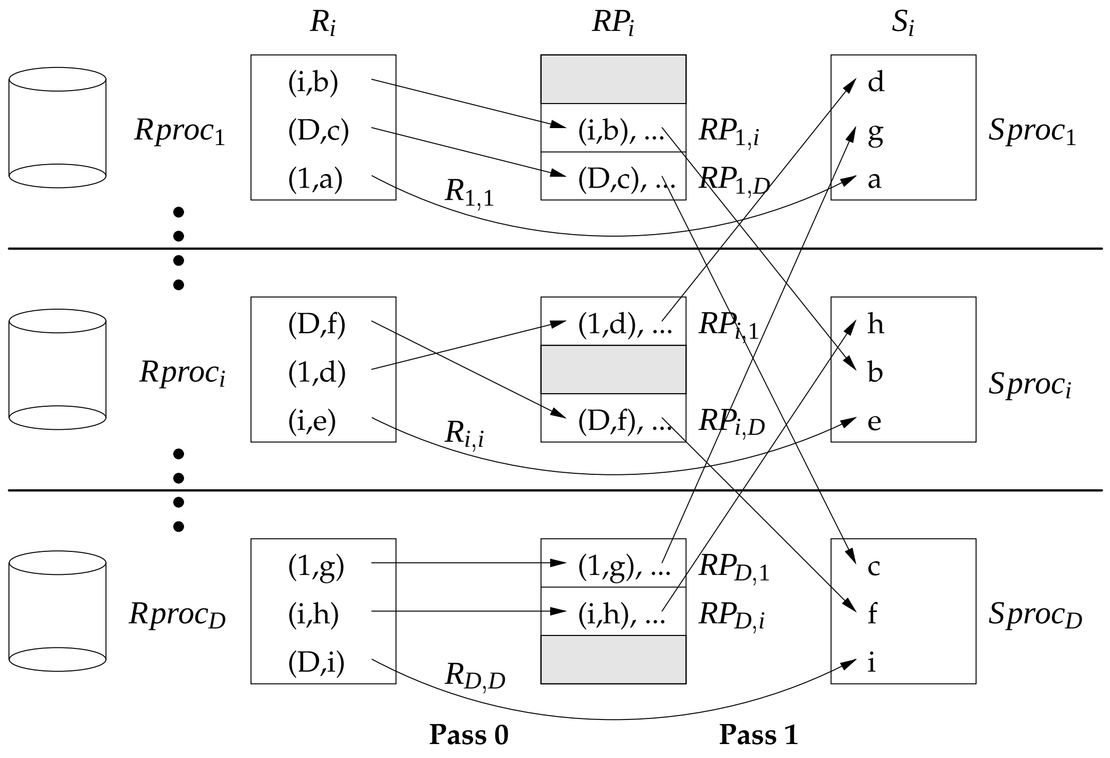


Figure 5.2: Parallel Pointer-Based Nested Loops

In figure 5.2, an R-object is represented by a tuple $(MAP(sptr), sptr)$, where $sptr$ is the join attribute and $MAP(sptr)$ is the number of the S partition containing the object pointed to by $sptr$. For each object in R_i, the S partition is computed from the join attribute and the object is copied (written) to a sub-partition inside of a temporary area RP_i, which is mapped onto the same disk as partition R_i; all the R-objects in R_i that point to an object in S_j are grouped together in sub-partition $RP_{i,j}$. This sub-partitioning largely eliminates disk contention in the next pass.

Instead of putting RP_i in its own segment, managed by another process, the storage for the RP_i segment is made part of the storage for $Rproc_i$. That is, R_i is located at the lowest address of the $Rproc_i$ segment and storage for RP_i is located after the storage for R_i. Hence, both R_i and RP_i are mapped to the private memory of $Rproc_i$, which eliminates the costs of segment-to-segment transfer, namely copying data through shared memory. It also eliminates the cost of creating and managing an additional process for RP_i. The drawback of this optimization is that the maximum size of R_i is approximately half of the maximum available address space size.

As an optimization, the objects in R_i that point to objects in S_i are immediately joined, in pass 0, by extracting the join pointer, and having $Sproc_i$ read the corresponding S object. $Sproc_i$ dereferences the join attribute resulting in a loading of the page of S_i containing the referent object, if that page is not already in memory, and makes the S object available for the join by putting it into shared memory. $Rproc_i$ then does the join. As a further optimization, the requests for objects from S_i are grouped into a buffer of size G to reduce context switches between $Rproc_i$ and $Sproc_i$.

Pass 1 (see figure 5.2) eliminates disk contention by staggering access to S_i through a series of $D-1$ phases, without synchronizing the phases. In phase t ($t = 1, 2, \ldots D-1$), $RP_{i,offset(i,t)}$ is joined with $S_{offset(i,t)}$, where $offset(i,t) = ((i+t-1) \bmod D) + 1$. For example, a typical phase joins a sub-partition $RP_{i,j}$ with S_j; because of the offset, S_j is only accessed by one $Rproc_i$ in any one phase, assuming no skew. In the presence of skew, there are different numbers of objects in each $RP_{i,j}$, so there may be some contention when multiple $Rproc_i$ access the same S_j. $Rproc_i$ loops over objects in $RP_{i,offset(i,t)}$ in private memory; for each one, it extracts the join pointer and asks $Sproc_{offset(i,t)}$ for the corresponding S object.

Because a random distribution of join attributes in R is assumed and there is exactly one reference to each object of S, the references to S-objects in each R_i are uniformly distributed, and therefore, *skew* is very close to 1.0. As a result, no synchronization is used after each phase of pass 1 for all the $Rproc_i$; any contention that does occur is insignificant, as was verified by running experiments with synchronization after each phase of pass 1. In the best case, there was a 0.5% decrease in I/O and total time due to reduced contention. This saving was not considered significant enough to warrant complicating the algorithm and the analysis with synchronization.

5.2.2 Parameter Choices

M_{Rproc_i} should be large enough to hold, in pass 0, at least one block of the input R_i and at least one block for each $RP_{i,j}$. Since S_i is being read randomly, M_{Sproc_i} should be as large as possible. G should be large enough to avoid many context switches between $Rproc_i$ and $Sproc_i$, but small enough so that the volume of pending requests does not force important information out of memory. In an EPD environment, the value of G should be, but is not required to be, a multiple of the block size, B. The implementation used a value of B for G.

5.2.3 Analysis

Given $|R_i| = |R|/D$ and $|R_{i,i}| = (|R_i|/D) \cdot skew = (|R|/D^2) \cdot skew$, for the largest of $R_{i,i}$, then $|RP_i|$ is

$$|R_i| - |R_{i,i}| = \frac{|R|}{D} - \frac{|R|}{D^2} \cdot skew.$$

R_i is *not* adjusted by *skew* because there is no synchronization between phases in this algorithm; in essence, the skew in $RP_{i,j}$ is compensated for by the additional parallelism resulting from the lack of synchronization among the $Rproc_i$ between passes 0 and 1.

In pass 0, R_i is read sequentially, RP_i is written (mostly) randomly, and S_i is read randomly. Figure 5.3 shows the disk layout of the three partitions.

Since each partition is accessed, the band size of disk arm movement, in the worst

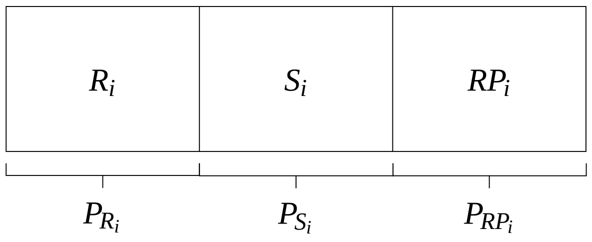


Figure 5.3: Disk Layout: Parallel Pointer-Based Nested Loops

case, is the total size of all partitions:

$$BandSize_{pass0} = P_{R_i} + P_{S_i} + P_{RP_i} = \frac{P_R}{D} + \frac{P_S}{D} + \left(\frac{P_R}{D} - \frac{P_R}{D^2} \cdot skew\right).$$

As well, because random reads and writes are interspersed on the same disk, all *dtt* formula are for random I/O (i.e., it does not matter that some objects are read sequentially). The disk transfer times for R_i and RP_i, then, are $P_{R_i} \cdot dtt_r(BandSize_{pass0})$ and $P_{RP_i} \cdot dtt_w(BandSize_{pass0})$, respectively.

$|R_{i,i}|$ S-objects are read randomly from S_i, one object at a time, during the join, but some of those objects may be in memory already when requested. The analysis uses a result of Mackert and Lohman [ML89] to approximate the number of page faults, which corresponds to disk transfers. [ML89] derives the following approximation: given a relation of N tuples over t pages, with i distinct key values and a b-page LRU buffer, if x key values are used to retrieve all matching tuples, then the number of page faults is

$$Y_{LRU}(N,t,i,b,x) = \begin{cases} t \cdot (1 - q^x) & \text{if } x \le n \\ t \cdot [(1 - q^n) + p \cdot (x - n) \cdot q^n] & \text{if } x > n \end{cases}$$

where

$$n = \max\{j : j \leq i, t \cdot (1 - q^j) \leq b\} \qquad \text{and} \qquad q = 1 - p = \begin{cases} (1 - 1/t)^{N/i} & \text{if } t \geq i \\ (1 - 1/i)^{N/t} & \text{if } t < i \end{cases}.$$

Assuming the references to S are randomly distributed in R, the disk transfer time for reading objects from S_i, in pass 0, is

$$Y_{LRU}\left(|R_{S_i}|, P_{S_i}, |R_{S_i}|, \frac{M_{Sproc_i}}{B}, |R_{i,i}|\right) \cdot dtt_r(BandSize_{pass0}).$$

In pass 1, RP_i is read sequentially, and S_i is read randomly. Since only the partitions S_i and RP_i are used, the band size of disk arm movement, in the worst case, is the total size of both partitions: $BandSize_{pass1} = P_{S_i} + P_{RP_i}$. As well, because random reads and writes are interspersed on the same disk, all dtt formulas are again for random I/O. The disk transfer times for RP_i and S_i are, therefore,

$$P_{RP_i} \cdot dtt_r(BandSize_{pass1}) \quad \text{and} \quad Y_{LRU}\left(|R_{S_i}|, P_{S_i}, |R_{S_i}|, \frac{M_{Sproc_i}}{B}, |RP_i|\right) \cdot dtt_r(BandSize_{pass1}),$$

respectively.

Furthermore, in pass 0, each object of R_i is moved once, either to RP_i or to shared memory for the join, and appropriate objects of S_i are moved to shared memory for the join. The transfers from R_i to RP_i are simple memory transfers among areas of $Rproc_i$'s memory because of the organization of $Rproc_i$'s memory (see section 5.2.1). The corresponding data transfer cost is $|RP_i| \cdot r \cdot MT_{pp} + |R_{i,i}| \cdot (r + sptr + s) \cdot MT_{ps}$.

The transfers from S_i require a data movement from $Sproc_i$'s private memory to shared memory so that an object can be accessed by $Rproc_i$; this requires two context switches, from $Rproc_i$ to $Sproc_i$ and back again so that $Sproc_i$ can perform the transfer. To optimize context switching, shared memory of size G is used (see section 5.2.1). During the sequential pass of R_i, objects for $R_{i,i}$ and their join attributes (i.e., the S-pointers) are placed into this buffer until there is only room for the corresponding S_i objects. While the S-pointer is embedded in the R object, it is copied out so that $Sproc_i$ does not have to know about the internal structure of R objects. The buffer is then given to $Sproc_i$ to copy the corresponding S-objects into the remaining portion of the buffer. The objects in the buffer can now be joined. The buffer reduces the number of context switches to $Sproc_i$. Also, copying the R_i object into the buffer prevents additional I/O in R_i during the join due to references back to previously read objects. The alternative is to join each individual R object when found during the sequential scan, which results in a context switch to $Sproc_i$ for each object.

In pass 1, each object of RP_i is moved once to shared memory, and the referent objects from S_i are moved to shared memory for the join in a total time of $|RP_i| \cdot (r + sptr + s) \cdot MT_{ps}$. The buffering technique employed in pass 0 is also used in pass 1 to retrieve S-objects. The context switching costs for pass 0 and 1 are

$$2 \cdot CS \cdot \left\lceil \frac{|R_{i,i}|}{\lfloor G/(r + sptr + s) \rfloor} \right\rceil \qquad \text{and} \qquad 2 \cdot CS \cdot \left\lceil \frac{|RP_i|}{\lfloor G/(r + sptr + s) \rfloor} \right\rceil,$$

respectively. The cost of mapping the join attributes to their S partitions in pass 0 is $|R_i| \cdot map$. Finally, the setup cost (see section 3.5.2) for mapping R_i, S_i and RP_i is

$$D \cdot (openMap(P_{R_i}) + openMap(P_{S_i}) + newMap(P_{RP_i})).$$

The setup time is multiplied by D because manipulating a mapping of a partitioned file structure is a serial operation.

5.3 Parallel Pointer-Based Sort-Merge

In nested loops, the random access of S slows down the join. Sort-merge changes the random access of nested loops to a single sequential scan of S, resulting in a significant performance gain. While Shapiro's sort-merge [Sha86] assumes only two passes, my algorithm permits multiple passes, writing out full records at each pass. Also, as noted earlier, the use of S-pointers as the join attribute makes sorting of S unnecessary.

5.3.1 Algorithm

The first two passes of the parallel sort-merge algorithm are the same as for parallel nested loops (see section 5.2.1) except for one difference: in nested loops, $R_{i,i}$ in pass 0 and $RP_{i,j}$ in pass 1 are joined with S_i, whereas in sort-merge, $R_{i,i}$ and $RP_{i,j}$ are written out to R_{S_j}. Fig. 5.4 shows the two passes for sort-merge.

Once the R_{S_i} partitions have been formed, the sequential sort-merge algorithm is executed on each partition in parallel. The algorithm proceeds by first sorting, in parallel, all R_{S_i} with respect to the join attributes to allow sequential processing of S_i. The sorting of R_{S_i} is done using multi-way merge sort, with the aid of a heap and with intermediate runs stored on disk. In the final pass, S_i is read in sequentially to perform the join.

As in nested loops, data movement is optimized by combining several partitions in $Rproc_i$'s segment. That is, R_i is located at the lowest address of the $Rproc_i$ segment, storage for RP_i is located after that, and then *all* partitions for the R_{S_i}. Hence, all these partitions are in the private memory of $Rproc_i$. The saving in data transfers through shared memory is significant and is possible because RP_i and the R_{S_j} are temporary areas where the data is manipulated as composite objects without the need to dereference embedded pointers. The drawback is that the maximum size of R_i is approximately $D+1$ times less than the maximum address space size. If this optimization poses a problem,

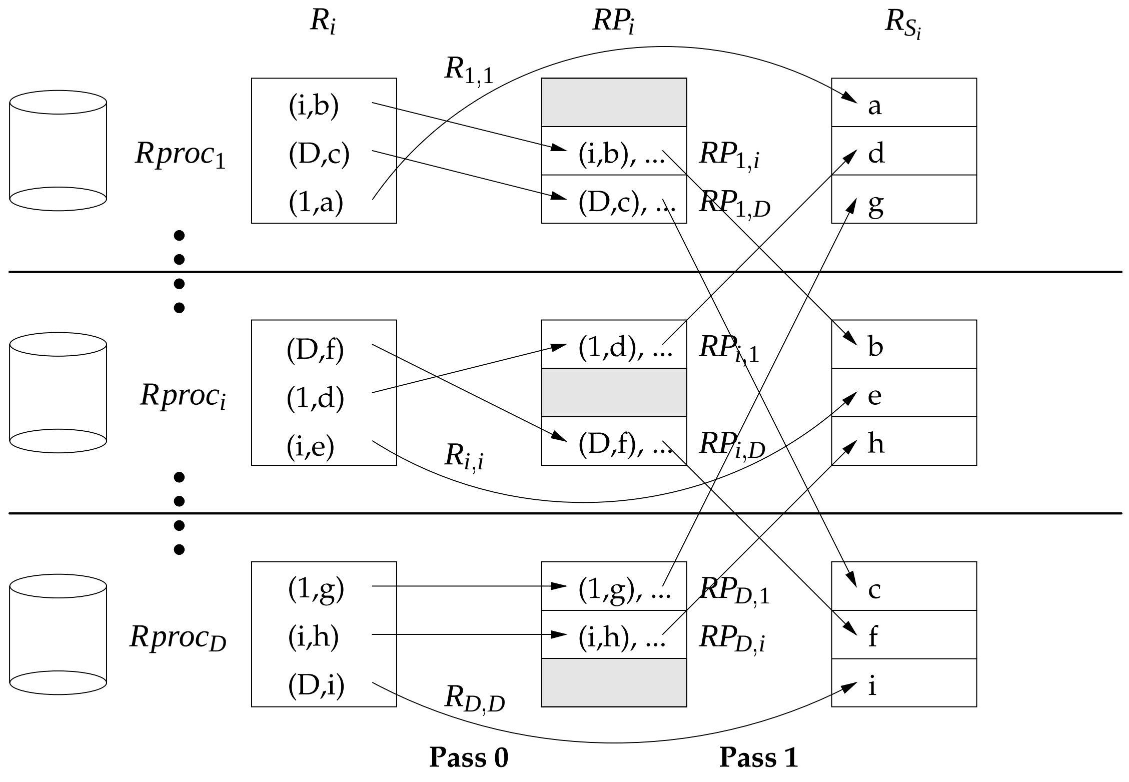


Figure 5.4: Parallel Pointer-Based Sort-Merge

the R_{S_j} can be separate segments and data can be copied to them through shared memory using a buffer.

The design and analysis of Sort-Merge introduce a number of additional parameters, tabulated in table 5.2, some chosen by the programmer, and some that are specified by the implementation. The programmer must choose $IRUN$, the length of a run created from unsorted data from pass 1, and $NRUN$, the number of runs to be merged in a given merging pass. In pass 2 of the sort-merge algorithm, $IRUN$ R-objects are read in from R_{S_i} and a heap of pointers to these memory-resident objects is created in memory. Heapsort is applied to the heap of pointers and the sorted list of pointers is used to sort, in place, the corresponding R-objects. The resulting sorted run of $IRUN$ R-objects is eventually written out to disk. These actions are repeated to sort successive runs until all of R_{S_i} has been processed.

Parameter	Description
$IRUN$	length of sort runs
$NRUN$	number of sort runs
hp	size of an element in a heap of pointers
$compare$	time to compare two heap elements
$swap$	time to exchange two heap elements
$transfer$	time to move a heap element

Table 5.2: Parameters of Sort-Merge Join

On subsequent merging passes, groups of $NRUN$ sorted runs are merged using delete-insert operations on a heap of $NRUN$ pointers. The heap always contains pointers to the next unprocessed element from each sorted run; when a pointer is deleted from the heap, the corresponding object is moved to the output run, and a pointer to the next object from the input run that contained the moved object is inserted into the heap. The merged run is written out to disk and becomes the input for the next merging pass. The process is repeated until all the remaining runs can be merged in a single pass.

On the last merging pass, instead of writing out the merged R-objects, the corresponding objects from S_i are read sequentially and the join computed. The reading of the objects from S_i is accomplished, as in nested loops, by means of a shared memory buffer of size G.

5.3.2 Parameter Choices

$IRUN$ is chosen to be the largest number such that an entire run, plus space for the heap of pointers, fits in available memory, i.e.,

$$IRUN = \left\lfloor \frac{M_{Rproc_i}}{r + hp} \right\rfloor,$$

where hp is the size, in bytes, of an element in the heap of pointers.

Ideally, merging of runs requires at least one page of memory for each run; otherwise excessive thrashing occurs because pages are replaced before they are completely processed. In reality, with this minimum memory, pages are replaced prematurely because the LRU paging scheme makes the wrong decisions when replacing a page during the merging passes. That is, when objects in an input page have been processed, the page is no longer needed, but it must age before it is finally removed; during the aging process, a page that is still being used for the output runs gets paged out, resulting in additional I/O. In the current implementation, the problem is avoided by reducing the value of $NRUN$, which is chosen to be $M_{Rproc_i}/(3 \cdot B)$ during all but the last pass (denoted $NRUN_{ABL}$), and $M_{Rproc_i}/(2 \cdot B)$ during the last pass. In other words, memory is underutilized to compensate for this anomaly so that the program behaves more consistently. The amount of underutilization is based on an approximation of the working set of the program during these passes. The same problem occurs in the Grace algorithm, discussed later. For the Grace algorithm, the processing is left unchanged and an analysis is done to quantify the amount of extra I/O that occurs due to premature replacement of pages. Thus, two alternative strategies of attacking the problem have been investigated.

5.3.3 Analysis

Given $|R_i| = |R|/D$ and $|R_{i,i}| = (|R_i|/D) \cdot skew = (|R|/D^2) \cdot skew$, for the largest of $R_{i,i}$, then $|RP_i|$ is

$$|R_i| \cdot skew - |R_{i,i}| = \frac{|R|}{D} \cdot skew - \frac{|R|}{D^2} \cdot skew.$$

R_i *is* adjusted by *skew* because there is synchronization between phases in this algorithm, therefore the worst case must be considered for each individual pass.

In pass 0, R_i is read sequentially, RP_i is written mostly randomly, and R_{S_i} is written sequentially. Figure 5.5 shows the disk layout, resulting in the band size of disk arm movement, in the worst case, of

$$BandSize_{pass0} = P_{R_i} + P_{S_i} + P_{R_{S_i}} + P_{RP_i} = \frac{P_R}{D} + \frac{P_S}{D} + \frac{P_R}{D} + \left(\frac{P_R}{D} - \frac{P_R}{D^2}\right) \cdot skew$$

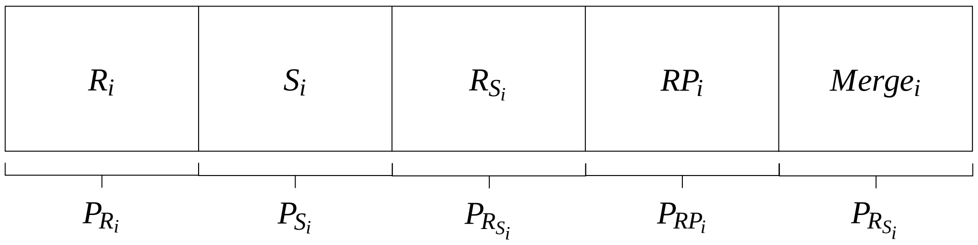


Figure 5.5: Disk Layout: Parallel Pointer-Based Sort Merge

The disk transfer times for R_i, R_{S_i} and RP_i are

$$P_{R_i} \cdot dtt_r(BandSize_{pass0}), \quad P_{R_{S_i}} \cdot dtt_w(BandSize_{pass0}) \quad \text{and} \quad P_{RP_i} \cdot dtt_w(BandSize_{pass0}),$$

respectively. In pass 1, RP_i is read sequentially, and R_{S_i} is written sequentially, giving

$BandSize_{pass1} = P_{R_{S_i}} + P_{RP_i}$. The disk transfer times for R_{S_i} and RP_i are

$$P_{R_{S_i}} \cdot dtt_w(BandSize_{pass1}) \quad \text{and} \quad P_{RP_i} \cdot dtt_r(BandSize_{pass1}),$$

respectively. All *dtt* formulas are for random I/O because of wide fluctuations in the disk arm between regions read or written sequentially.

In pass 0, each object of R_i is moved once within $Rproc_i$'s segment, either to RP_i or to R_{S_i}, at a cost of $|R_i| \cdot r \cdot MT_{pp}$. In pass 1, each object of RP_i is moved once within $Rproc_i$'s segment to the appropriate R_{S_i} at a cost of $|RP_i| \cdot r \cdot MT_{pp}$. Since all of the data movements are with $Rproc_i$'s segment, there are no context switch costs in passes 0 and 1. The mapping cost for pass 0, which generate a S partition from an S-pointer, is $|R_i| \cdot map$.

In pass 2 (the heap-sorting pass), runs of size $IRUN$ objects from R_{S_i} are sequentially read in and sorted in place. Since there is no explicit writing, the previous sorted run is written back by the operating system as the pages age with mostly random writes. This pattern results in a disk band size that is twice the size of a sort run: $2 \cdot (r \cdot IRUN/B)$. The disk transfer times for reading R_{S_i} and writing back the sorted runs are

$$P_{R_{S_i}} \cdot dtt_r\left(2 \cdot \frac{r \cdot IRUN}{B}\right) \qquad \text{and} \qquad P_{R_{S_i}} \cdot dtt_w\left(2 \cdot \frac{r \cdot IRUN}{B}\right),$$

respectively.

As shown in table 5.2, *compare* is the amount of time $Rproc_i$ requires to compare two elements in a heap of pointers to R-objects, stored in memory. Similarly, *swap* is the amount of time to swap two heap elements stored in memory, and *transfer* is the amount of time to move an element to or from the heap. These times do not count operations necessary to restore heap discipline after moving an element. The time required to restore heap discipline is computed separately.

In order to heap-sort each individual run, an array of pointers to the $IRUN$ R-objects in memory is converted into a heap using Floyd's heap construction algorithm (see [GM86, GBY91]). The heapsort method outlined in [SS93] is then used with a modification suggested by Munro [Mun95] that allows the heapsort to complete, in the average case, with approximately $N \log N$ comparisons and transfers. The creation of the heap takes time

$$1.77 \cdot |R_{S_i}| \cdot (compare + \frac{1}{2} \cdot swap) + |R_{S_i}| \cdot transfer$$

while the cost of heap-sorting the heap by repeated deletion of minima is

$$|R_{S_i}| \cdot \log IRUN \cdot (compare + transfer)^3.$$

A further cost of $|R_{S_i}| \cdot r \cdot MT_{pp}$ is required to permute the actual R-objects, in place, based on the sorted list of pointers.

The choice of $IRUN$ and $NRUN_{ABL}$ in turn determines $NPASS$, the number of merging passes, and $LRUN$, the number of runs on the last pass.

$$NPASS = \max\left\{ j : j \geq 1, \left\lceil \frac{|R_i|}{IRUN \cdot (NRUN_{ABL})^{j-1}} \right\rceil \leq NRUN \right\}$$

$$LRUN = \left\lceil \frac{|R_i|}{IRUN \cdot (NRUN_{ABL})^{NPASS-1}} \right\rceil$$

In the third and subsequent passes, groups of $NRUN_{ABL}$ (or $LRUN$ in the last pass) input runs are read in, merged into one, and written out. R_{S_i} and $Merge_i$ (see figure 5.5) alternate as source and destination of these runs. In the last pass, R_{S_i} (if $NPASS$ is odd) or

³Notice the omission of a ceiling on the value of log computations here and in subsequent formulae, which compensates for the fact that the heaps are not perfect and may have leaf nodes at two different levels.

$Merge_i$ (if $NPASS$ is even) contains all the objects that are merged into a single run, which is then joined with S_i. The disk band size during all but the last pass is

$$BandSize_{ABL} = P_{R_{S_i}} + P_{RP_i} + P_{Merge_i},$$

and during the last merging/joining pass is

$$BandSize_{Last} = P_{S_i} + P_{R_{S_i}} + (P_{RP_i} + P_{Merge_i}) \cdot ((NPASS + 1) \bmod 2).$$

The disk transfer time, except for the last pass, for reading and writing R_{S_i} and $Merge_i$, $NPASS - 1$ times are

$$P_{R_{S_i}} \cdot dtt_r(BandSize_{ABL}) \cdot (NPASS - 1), \quad \text{and} \quad P_{R_{S_i}} \cdot dtt_w(BandSize_{ABL}) \cdot (NPASS - 1),$$

respectively. During the last pass, I/O costs for R_{S_i} and S_i are

$$P_{R_{S_i}} \cdot dtt_r(BandSize_{Last}) \quad \text{and} \quad P_{S_i} \cdot dtt_r(BandSize_{Last})$$

respectively.

During the merge, except for the last pass, the delete-insert operation [GBY91, p. 214] is used on a heap of size $NRUN_{ABL}$ and the heap operations for each of the $(NPASS - 1)$ passes take time

$$\left((2 \cdot compare + swap) \cdot \frac{(NRUN_{ABL} + 1) \cdot k - \lfloor NRUN_{ABL}/2 \rfloor - 2^k}{NRUN_{ABL}} + 2 \cdot transfer\right) \cdot |R_{S_i}|$$

where $k = \lfloor \log NRUN_{ABL} \rfloor + 1$. The size of the heap used during the last merge pass is

$LRUN$ and the corresponding heap operations take time

$$\left((2 \cdot compare + swap) \cdot \frac{(LRUN+1) \cdot k - \lfloor LRUN/2 \rfloor - 2^k}{LRUN} + 2 \cdot transfer \right) \cdot |R_{S_i}|$$

where $k = \lfloor \log LRUN \rfloor + 1$. The data transfer cost during the $NPASS - 1$ merge passes and the last merge pass are $|R_{S_i}| \cdot r \cdot MT_{pp} \cdot (NPASS - 1)$ and $|R_{S_i}| \cdot (r + sptr + s) \cdot MT_{ps}$, respectively, with the corresponding context switching time of

$$2 \cdot CS \cdot \left\lceil \frac{|R_{S_i}|}{\lfloor G/(r + sptr + s) \rfloor} \right\rceil .$$

Finally, the setup cost for mapping R_i, S_i, R_{S_i}, RP_i and $Merge_i$ is

$$D \cdot (openMap(P_{R_i}) + openMap(P_{S_i}) + newMap(P_{R_{S_i}}) + newMap(P_{RP_i}) + newMap(P_{S_i})).$$

The setup time is multiplied by D because manipulating a mapping is a serial operation. An additional cost of $(deleteMap(P_{S_i}) + newMap(P_{S_i})) \cdot (NPASS - 1)$ is incurred in all merge passes but the last, to switch the source and destination areas for the merge.

Figure 5.6 illustrates the progress made by a particular run of the sort-merge join algorithm with actual times obtained from an experiment. The figure provides insights into the workings of the parallel sort-merge join algorithm because of its complexity. The term *staggered starts* used in the figure indicates where multiple threads perform a brief serial operation, e.g., initialization of shared structures, before proceeding with their individual parallel computation.

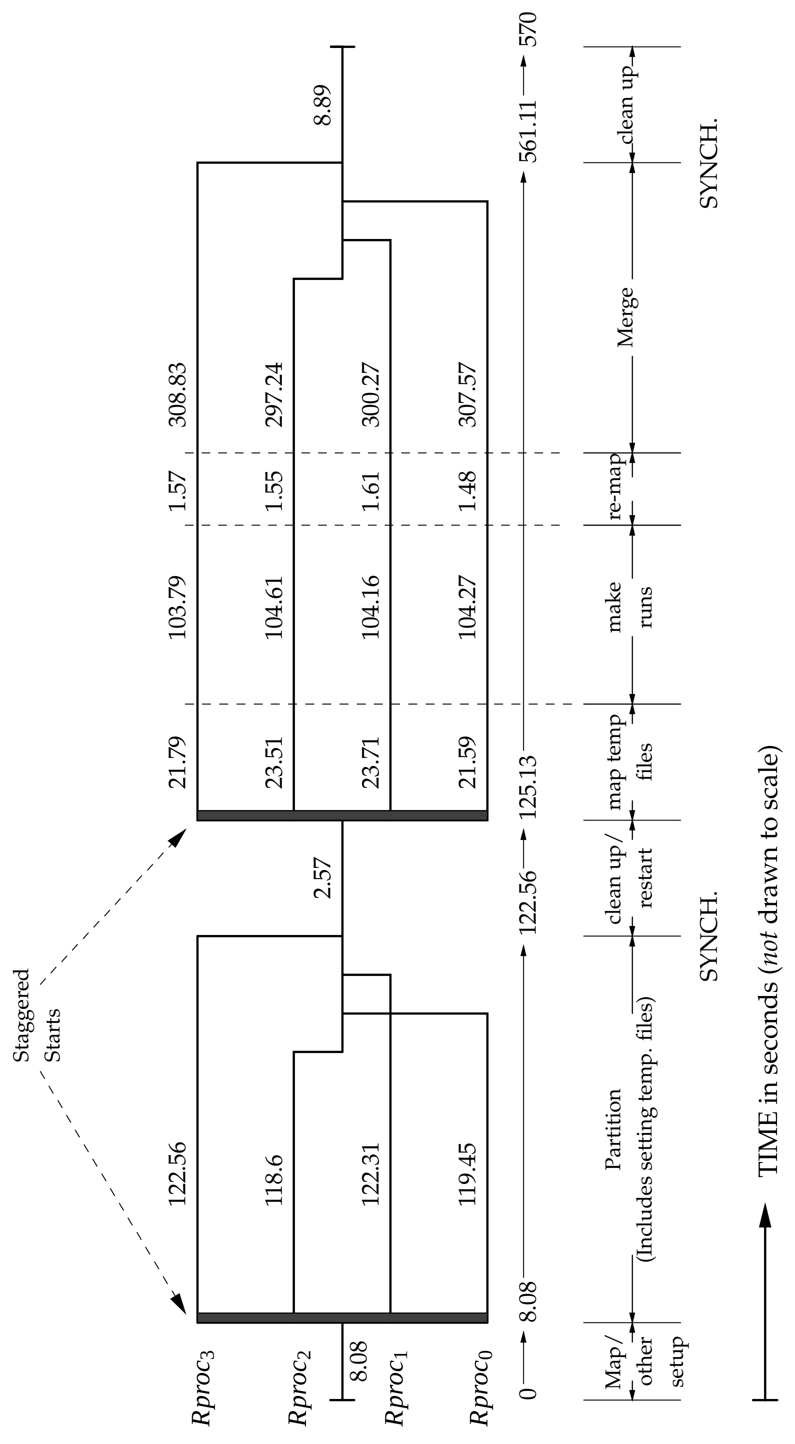


Figure 5.6: Time Line Progress of Parallel Sort-Merge

5.4 Parallel Pointer-Based Grace

Sort-merge improves the performance of the join by sorting R_i by the S-pointer, which allows sequential reading of S_i. However, sorting is an expensive operation. Hash-based join algorithms replace the sort with hashing to improve performance further. As an example of the hash-based join algorithms, I have chosen to model a parallelized pointer-based version of the Grace algorithm.

As with sort-merge, the spatial ordering property of the S-pointers makes it unnecessary to hash S_i. By carefully designing the hash algorithm, it can be ensured that each hash bucket contains monotonically increasing locations in S_i, so that S_i can be read sequentially.

5.4.1 Algorithm

The first two passes of the Grace algorithm, shown in fig. 5.7, are the same as in parallel nested loops, except for one difference; in nested loops, R-objects are joined with S_i, whereas in Grace, the join attributes (i.e., the S-pointers) from R-objects are hashed into one of K sub-partitions (or buckets) that make up R_{S_i}. The value of K is chosen by the programmer based on the amount of memory available. The jth sub-partition of R_{S_i} is referred to as $B_{S_i,j}$, i.e., $R_{S_i} = \bigcup_{j=1}^{K} B_{S_i,j}$. Figure 5.7 shows these two passes of the modified Grace algorithm and table 5.3 contains additional parameters for the algorithm.

Parameter	description
$TSIZE$	range of the hash function in pass 1
K	number of hash buckets formed
$fuzz$	hash table overhead factor

Table 5.3: Parameters for Grace Join

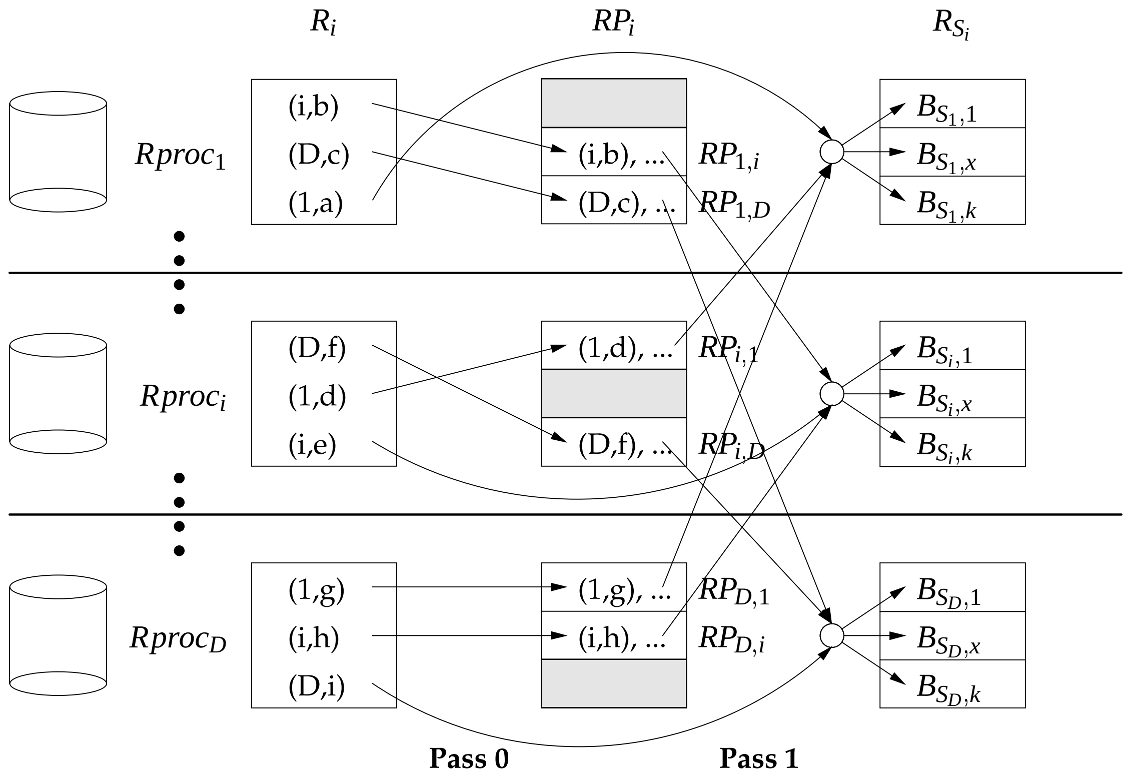


Figure 5.7: Parallel Pointer-Based Grace

As in the case of other parallel join algorithms described earlier, the parallel version of the Grace algorithm first repartitions each R_i into $R_{i,j}$ sub-partitions on the basis of the join attributes (which happen to be pointers), moves the objects of each $R_{i,j}$ to disk j to form R_{S_j}, and then, in parallel, executes a sequential algorithm on each (R_{S_j}, S_j) pair without disk contention.

In pass 0, $Rproc_i$ reads R_i, one object at a time, and depending upon the value of the join attribute, either moves the object into an $RP_{i,j}$ or hashes the object into one of the K buckets of R_{S_i}. In pass 1, $Rproc_i$ reads $R_{i,j}$ (for all $j \neq i$) one R-object at a time, and hashes each object into one of the K sub-partitions of R_{S_j}. As for pass 1 of the nested loops algorithm, the reading and hashing of $R_{i,j}$ in pass 1 takes place in phases to eliminate contention for the disks. At the end of pass 1, each R_{S_i} contains K sub-partitions that contain hashed R-objects. The hash function is chosen so as to cluster R-objects by the value of their join attributes. Therefore, $B_{S_i,j}$, $j = 1, \ldots K-1$, contains R-objects with join attributes smaller than that of any R-object in $B_{S_i,j+1}$.

In pass $1+j$ ($j = 1, 2, \ldots K$), for every i in parallel, $B_{S_i,j}$ is read in, and the value of the join attribute in each object is used as input to another hash function that further refines the partitioning given by the first hash function. The range of this hash function is $TSIZE$, a parameter chosen by the programmer. Once all of $B_{S_i,j}$ has been hashed into an in-memory hash table, the table is processed in order. Common references to objects in S_i (i.e., references that result in a collision when hashed) are in the same hash chain. If it is assumed that there are no more than M_{Sproc_i}/s different references to objects in S_i in any one hash chain during the processing of that hash chain, all objects from S_i needed during the processing of that hash chain can fit in memory; hence each object referenced from S need only be read once in order to perform the join. The reading of the objects from S_i is accomplished by means of a shared memory buffer of size G, as before.

5.4.2 Parameter Choices

During pass $1+j$, $j = 1, 2, \ldots K$, $Rproc_i$ reads each R-object in $B_{S_i,j}$ into a memory resident hash table. The value of K should be chosen such that each $B_{S_i,j}$ along with its associated hash table overhead fits entirely in memory.

$TSIZE$ should be small enough to not cause excessive hash-table overhead because of underutilization of memory and large enough so that the size of individual hash chains is low. Theoretically, the minimum amount of memory that needs to be made available to each $Rproc_i$, in pass 0, to avoid thrashing is $D + \lceil \sqrt{fuzz \cdot |R_i| \cdot r/B} \rceil$ blocks, where $fuzz$ makes room for the hash table overhead. In reality, even this threshold memory results in thrashing because the working set for the algorithm is greater than the theoretical threshold memory and the LRU paging scheme then makes the wrong decision, removing useful pages prematurely. See [Sha86, Sto81] for more discussion on this problem. The next section derives an approximation for the amount of extra I/O that takes place when memory is insufficient.

5.4.3 Analysis

The disk band sizes during pass 0 and pass 1 are $BandSize_{pass0} = P_{R_i} + P_{S_i} + P_{R_{S_i}} + P_{RP_i}$ and $BandSize_{pass1} = P_{R_{S_i}} + P_{RP_i}$, where P_{RP_i} is the same size as in sort-merge because there is synchronization between phases. Pass 0 involves reading objects from R_i, one object at a time, and writing each object to either RP_i or to one of the K buckets in R_{S_i}. The corresponding I/O costs are $P_{R_i} \cdot dtt_r(BandSize_0)$, $P_{RP_i} \cdot dtt_w(BandSize_0)$ and $(P_{R_{i,i}} + K) \cdot dtt_w(BandSize_0)$. The number of pages written to R_{S_i} has been increased by K to account for the fact that objects read from $R_{i,i}$ are hashed into K buckets in R_{S_i}. The additional costs incurred in pass 0 include $|R_i| \cdot map$ to map the join attributes to their corresponding S partition, $|R_{i,i}| \cdot hash$ to hash the $R_{i,i}$ objects into one of K R_{S_i}-buckets and $|R_i| \cdot MT_{PP}$ to move the R_i objects in private memory to either RP_i or R_{S_i}.

An urn model is used to derive an approximation for the amount of extra I/O that takes place due to lack of memory in pass 0. In pass 0, R-objects from R_i are placed in one of $RP_{i,j}$ or in one of the K buckets of R_{S_i}. The analysis computes the probability that, just after a page belonging to R_{S_i} is hit (either in memory or causing a page fault), that it gets hit again before it gets replaced. Once hit, a bucket page is replaced when there are M_{Rproc_i}/B references to newer pages before it is hit again; the probability of hashing t further objects without a second hit is $(1 - 1/K)^t$. At any given time, some of the pages in memory are partially filled or read pages (*current pages*) and some pages have been completely processed or filled (*fill events*), but which stay around because they are recently accessed and have not aged enough. It is assumed that the D current pages for R_i and $RP_{i,j}$ stay in memory until processed completely because these pages are processed at a much faster rate than the pages in R_{S_i}.

For convenience, divide the hashing of objects after a hit into epochs; the first α_0 objects, the next α_1, and so on. The number of fill events that have occurred at the beginning of epoch q is a random variable, which can be approximated as follows. Since the page replacement algorithm prefers clean pages over dirty pages, the fill events caused by the processing of R_i can be ignored. The fill rate for $RP_{i,j}$ is $\lceil (D-1)/|B| \rceil$, and for R_{S_i} is $\lceil 1/(K \cdot |B|) \rceil$, the latter being negligible. Therefore, the number of fill events is $\lceil H_j \cdot (D-1)/|B| \rceil$, where $H_j = \sum_{n=0}^{j-1} \alpha_n$ is the number of objects hashed at the beginning of epoch j.

The probability that at most $\lceil H_j \cdot (D-1)/|B| \rceil + D$ buckets are not hit by the beginning of the epoch, denoted p_j, multiplied by the probability that a page gets hit again during epoch j, denoted y_j, is the probability that the page is not present in memory during a second hit in epoch j. Summing over all epochs and multiplying by $|R_{i,i}|$ gives an approximation to the expected number of times a page of R_{S_i} gets replaced prematurely.

The probability p_j can be computed by reference to Johnson and Kotz [JK77, p.110], who show that the probability of exactly k urns being empty after n balls are randomly placed into m urns is

$$Pr[X = k] = \binom{m}{k} \cdot \left(1 - \frac{k}{m}\right)^n \cdot \sum_{j=0}^{m-k-1} \binom{m-k}{j} (-1)^j \left(1 - \frac{j}{m-k}\right)^n .$$

Every premature replacement necessitates one extra write (to replace the page) and one extra read (when the page is referenced again) for a total cost of reading and writing of $|R_{i,i}| \cdot \sum_{j \geq 1} (p_j \cdot y_j)$ blocks.

In pass 1, objects in $RP_{i,j}$ are read, one object at a time, and each object is hashed into one of the K buckets in R_{S_j}. The costs of reading RP_i and writing R_{S_i} are

$$P_{RP_i} \cdot dtt_r(BandSize_1) \quad \text{and} \quad (P_{RP_i} + K) \cdot dtt_w(BandSize_1),$$

respectively. Once again, the number of pages written to R_{S_j} has been increased by K. It takes a further time of $|RP_i| \cdot MT_{PP}$ to move the objects in private memory.

After pass 1, the subsequent reading of the partitioned R_{S_i}, one bucket at a time, and the corresponding S_i objects requires time

$$(P_{R_{S_i}} + P_{S_i}) \cdot dtt_r \left(\frac{1}{K/2} \cdot P_{R_{S_i}} \right).$$

The band size for dtt_r is chosen to be half the size, in blocks, of the objects that fit in the hash table in order to approximate the actual behaviour, which is to read sequentially objects from a sub-partition of R_{S_i} followed by the corresponding objects in S_i and so on.

Each object in R_{S_i} is hashed once during the processing of each bucket, for time $|R_{S_i}| \cdot$ *hash*. The cost of transferring objects to shared memory is $|R_{S_i}| \cdot MT_{PS} \cdot (r + sptr + s)$ with the corresponding context switching time of

$$2 \cdot CS \cdot \left\lceil \frac{|R_{S_i}|}{\lfloor G/(r + sptr + s) \rfloor} \right\rceil.$$

Finally, the setup costs for mapping R_i and S_i for reading, creating the new mappings for R_{S_i} and RP_i in pass 0 and setting up R_{S_i} for reading in pass 1 is

$$D \cdot (openMap(P_{R_i}) + openMap(P_{S_i}) + newMap(P_{R_{S_i}} + P_{RP_i}) + openMap(P_{R_{S_i}})).$$

5.5 Model Validation

In order to validate the model and the analysis presented earlier, experiments were run that performed full joins on two relations with 409,600 objects each. A generated data set was used to populate the two relations for all the experiments presented in this chapter. The join attributes in *R* were generated by means of a random number generator obtained from the standardized testbed described in section 4.3.2. The size of data set is the same as that used in [MLD94]. The objects in each relation were of size 128 bytes. *R* and *S* were partitioned across 4 disks with one *R* and one *S* partition on each disk. Table 5.4 contains the values used for various parameters of the model.

5.5.1 Experimental Testbed

The testbed described in section 4.1 was used to run validation experiments. The following extensions were made to the testbed for the validation experiments:

- the operating system kernel was rebuilt so that it put aside the minimum amount of memory for use as DYNIX buffers. This change was made to verify the earlier assumption that memory mapping a regular DYNIX file by-passes the file system buffering and does not benefit from DYNIX buffer memory.
- all the file systems used for storing data for the experiments were rebuilt with a file system block size of 4K, the size of the DYNIX virtual memory page. Thus, all I/O took place in 4K blocks, instead of 8K blocks that were used in experiments in chapter 4. This change in file system block size made the block size the same as the virtual memory system page size. A similar change could not be made for experiments conducted earlier for historical reasons.

Parameter	Measured Value
CS	145 μseconds
dtt	see figure 3.12(a)
MT_{sp}	0.31 μseconds
MT_{ss}	0.31 μseconds
MT_{ps}	0.31 μseconds
MT_{pp}	0.31 μseconds
$newMap$	see figure 3.12(b)
$openMap$	see figure 3.12(b)
$deleteMap$	see figure 3.12(b)
map	11μseconds
$skew$	0.98
$compare$	5.45 μseconds
$swap$	4.3 μseconds
$transfer$	2.1 μseconds
$hash$	2 μseconds

Parameter	Assumed Value
P	4
M	variable
B	4096
D	4
$\|R\|$	409600
$\|S\|$	409600
r	128
s	128
$sptr$	4
hp	8
G	4096

Table 5.4: Validation Values of Model Parameters

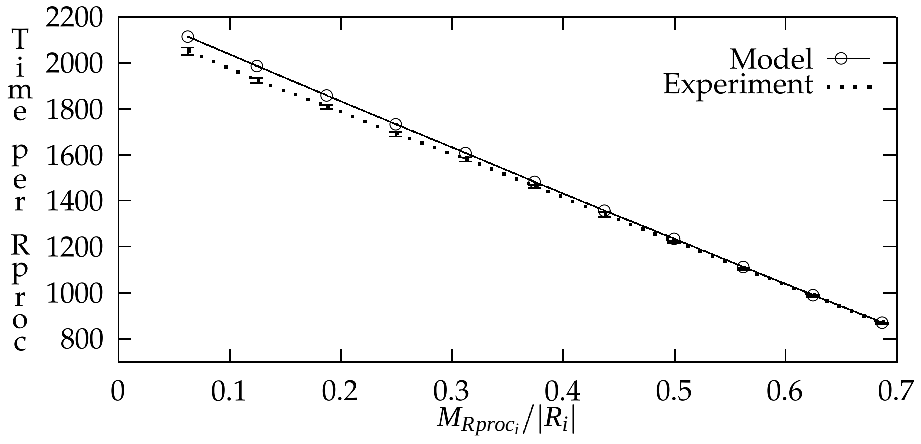
5.5.2 Results

Figure 5.8 shows the predicted and measured elapsed times for running the various join algorithms with varying amounts of memory available. The discontinuities in the sort-merge graph occur when additional merging phases are required. The curve in the Grace graph at low memory levels results from thrashing caused by the page replacement algorithm.

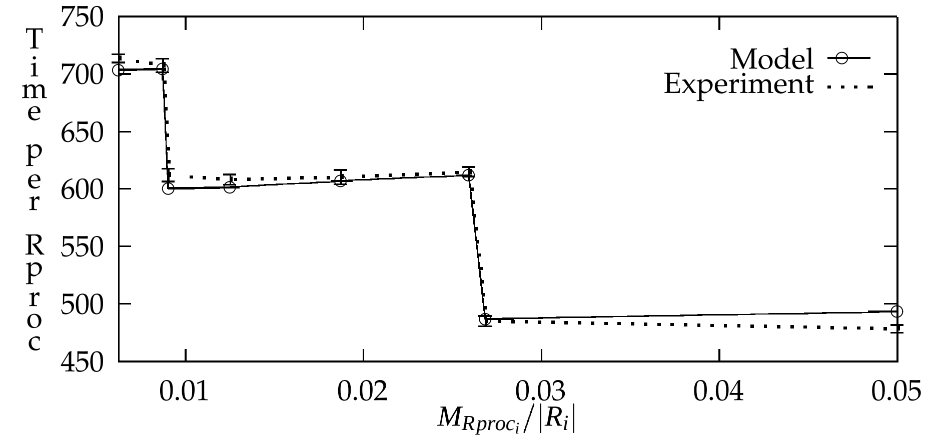
As is evident from the graphs, the model does an excellent job of predicting performance for the various join algorithms in almost all conditions. In particular, there is a close match between prediction and actual performance for nested loops and sort-merge. All the experiments were repeated several times in order to factor out any small deviations caused by the operating system (e.g., page replacement) behaviour and to make sure that the results were consistent, accurate and reproducible. For Grace, the approximation for I/O caused by thrashing at low memory levels is reasonably accurate; there is scope for further refinement of this approximation. A major part of the difference between prediction and actual behaviour at low memory levels comes from the overhead introduced by the particular replacement strategy used by the Dynix operating system. Further refinement of modelling this aspect of the page replacement scheme will be done in future work.

5.6 Predictions

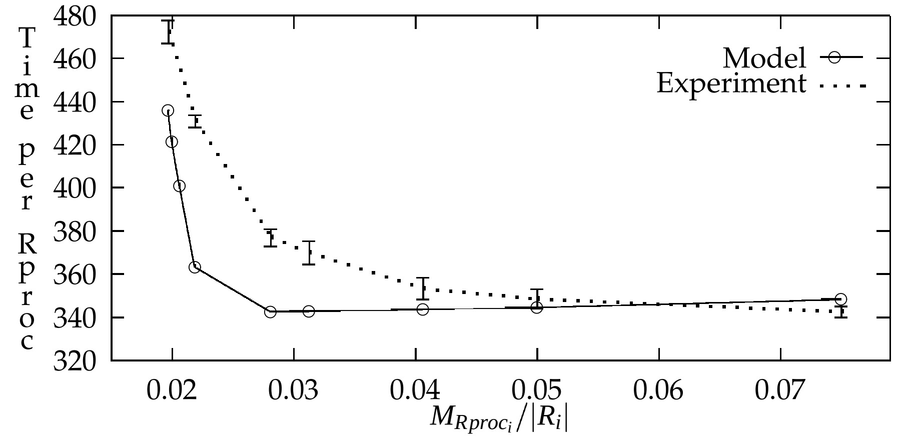
Once the model has been validated, it can be used to accurately predict the performance of a join algorithm for any given set of resources; Figure 5.9 depicts the predicted performance of the three join algorithms as a function of available memory with each of *R* and *S* partitioned across 4 disks. All the graphs follow the same pattern as the validated portion of the curves.



(a) Nested Loops

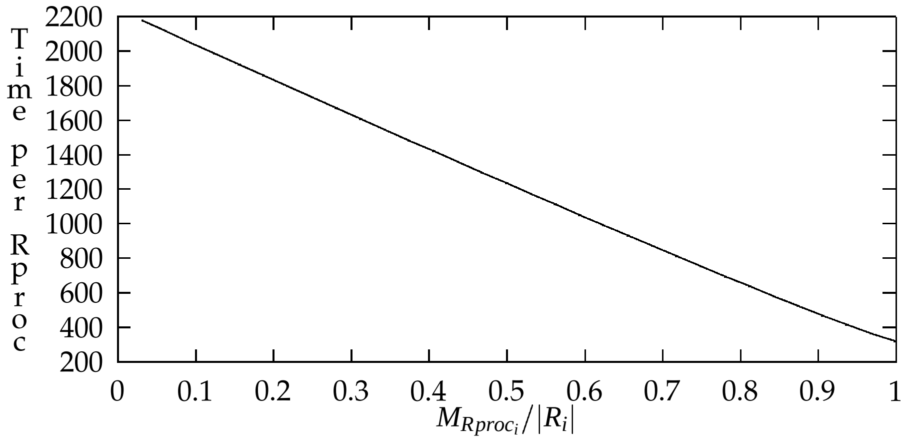


(b) Sort Merge

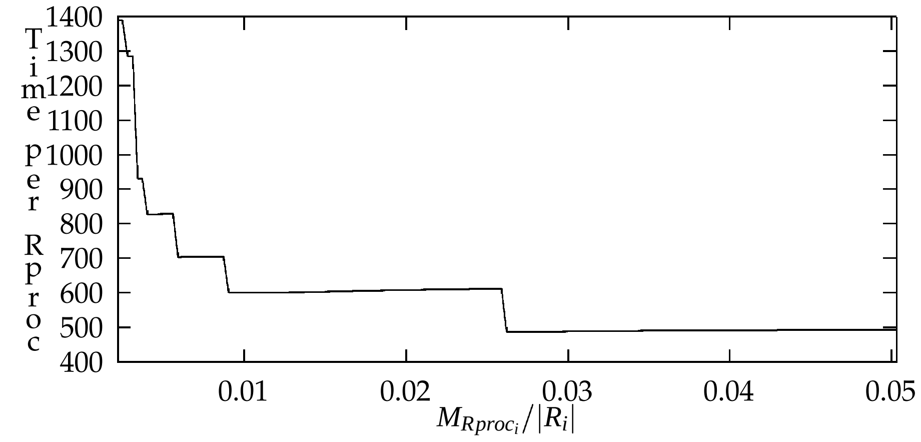


(c) Grace

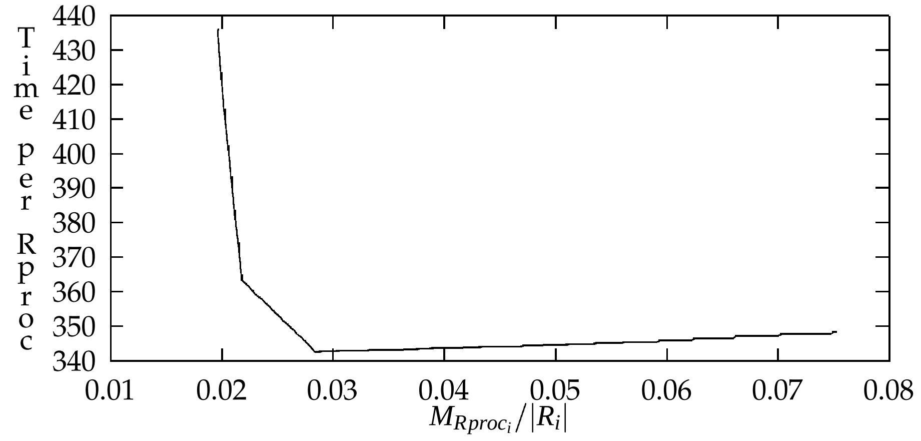
Figure 5.8: Model Validation



(a) Nested Loops



(b) Sort Merge



(c) Grace

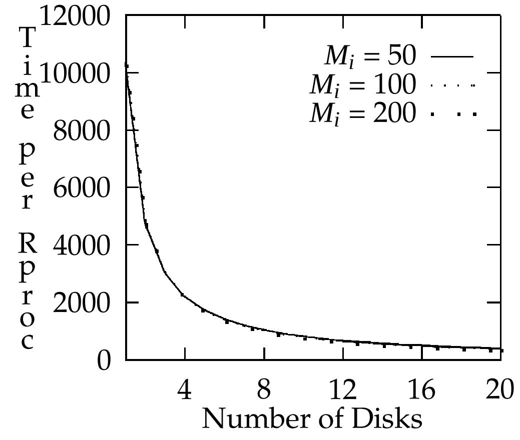
Figure 5.9: Model Predictions

5.6.1 Speedup and Scaleup

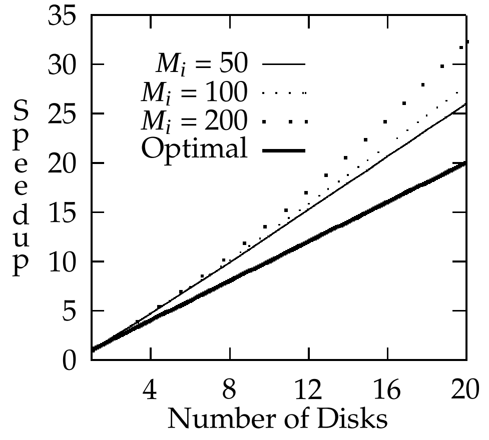
Further predictions with the model can be made to study the speedup and scaleup behaviours of the three parallel join algorithms. Speedup is an indication of performance improvement as available physical resources are increased while keeping the problem size constant. Scaleup also measures the effect on performance of increasing all available resources while at the same time increasing the problem size by the same proportion. In the case of joins, the problem size is indicated by the size, in blocks, of R and S and the relevant resources are CPUs, disks and physical memory. These predictions significantly stress the model and should illustrate any obvious anomalies.

Figure 5.10 presents the performance of the three join algorithms as the number of disks and other resources are increased; in each case, the size of R and S is kept fixed at 6400 blocks each. The number of CPUs is the same as the number of disks and absolute memory per disk is kept fixed, resulting in a corresponding increase in total memory as the number of disks is increased, i.e., M_{P_i} increases, which is the total memory available for processing of relation P. For each join algorithm, there are two graphs; one shows the actual reduction in time as resources are increased whereas the second plots the speedup factor. The speedup factor is simply the time spent with 1 disk divided by the time taken with D disks. The optimal speedup for an algorithm is linear speedup and is depicted in figure 5.10 for comparison.

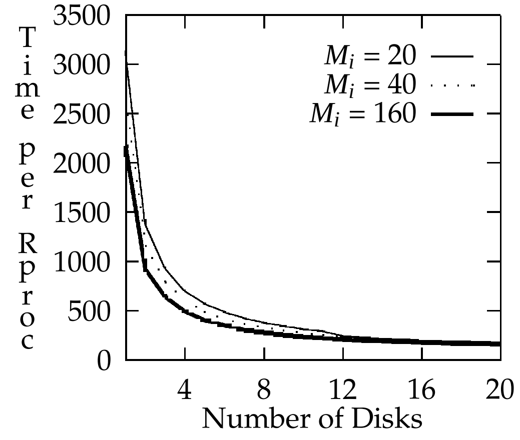
The nested-loops algorithm displays good speedup behaviour with some interesting features. For nested-loops, increasing resources results in a performance increase better than linear speedup. This behaviour is caused by the random reading of S objects in the two passes. When the number of disks is increased while keeping M_i constant, the relative amount of memory available for random I/O substantially increases, which means there is a significantly greater probability that a referenced page is already in memory.



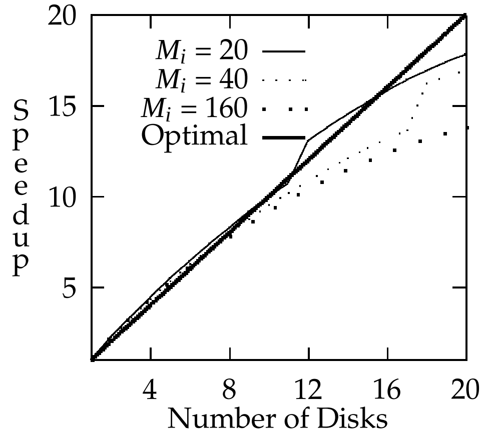
(a) Nested Loops – Time



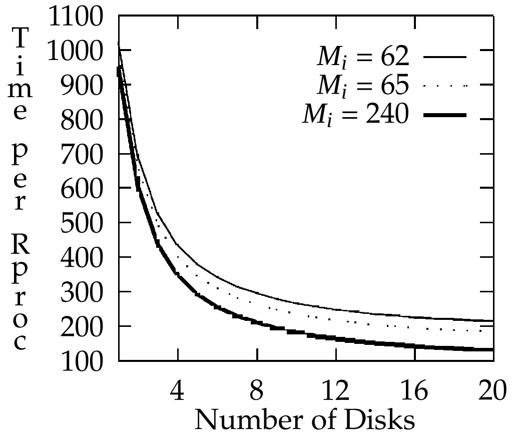
(b) Nested Loops – Speedup



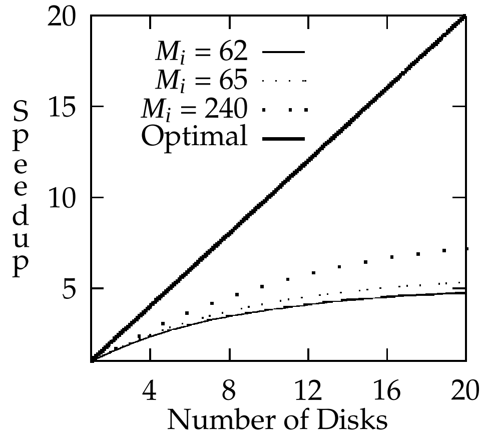
(c) Sort Merge – Time



(d) Sort Merge – Speedup



(e) Grace – Time



(f) Grace – Speedup

Figure 5.10: Speedup ($P = D$)

As an illustration, consider the following setup:

$$LR = LS = 409600,\ D = 1,\ B = 4096,\ M_i = 1600 \cdot B,\ r = s = 128,\ \text{and } skew = 1.0.$$

With one disk to work with, all the 409600 S objects are retrieved in a single pass and the total number of input operations is:

$$Y_{LRU}(409600, 409600 * 128/4096, 409600, 1600, 409600) = 358468.$$

Now, if the number of disks is increased to 2, the size of each S_i is reduced to 204800 records and each $Sproc_i$ is only responsible for retrieving 204800 S records. Half of the S_i records are read in pass 0 of the nested loops because of the direct join, without contention, on the same disk and the other half are read in pass 1 during the low contention staggered reading. Therefore, 102400 records are retrieved in each pass resulting in a total

$$2 * Y_{LRU}(204800, 204800 * 128/4096, 204800, 1600, 102400) = 154007$$

disk blocks read. Thus, increasing the number of disks from 1 to 2 results in a speedup factor of $\frac{358468}{154007} = 2.32$ as compared to a linear speedup factor of 2. Similarly, with 8 disks in the above example, total number of disk blocks read is

$$Y_{LRU}(51200, 51200 * 128/4096, 51200, 1600, 6400) = 1571 \quad \text{in pass 0, and}$$
$$Y_{LRU}(51200, 51200 * 128/4096, 51200, 1600, 44800) = 1600 \quad \text{in pass 1}$$

for a speedup factor of $\frac{358468}{1571+1600} = 113$ in the reading of S as opposed to the corresponding linear speedup factor of 8. This component of the total elapsed time causes the time to fall rapidly.

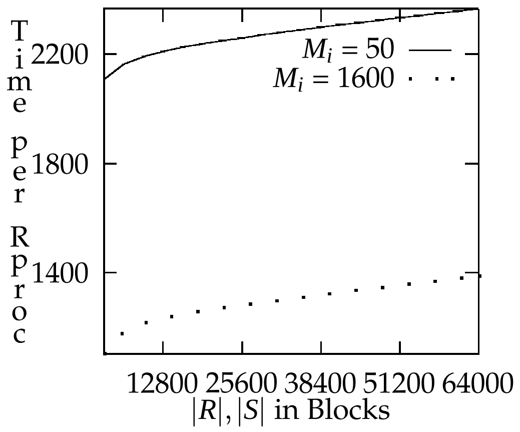
The sort-merge algorithm displays speedup behaviour that is closest to linear speedup. It starts with a linear speedup and stays that way until reaching its saturation point after which it begins to lag off. The Grace algorithm displays linear speedup for portions of its curve but the speedup factor is not close to optimal, i.e., the line has a slope much less than the desired value of 1.0. This behaviour is explained by the fact that the Grace algorithm does not make use of any extra memory that is made available, which implies that increasing resources does not succeed in improving the performance of the Grace algorithm by an equivalent proportion. Other more modern hash-based join algorithms, such as the hybrid-hash, make better use of available memory.

Figure 5.11 on page 211 presents similar results for the scaleup measurements. The sizes of R and S are increased in conjunction with a corresponding increase in available resources. In an ideal situation, this should result in time remaining constant; in practice, perfect scaleup is hard to achieve. In order to achieve perfect scaleup, the algorithms must employ perfect parallelism. As can be seen from figure 5.11, none of the algorithms displays behaviour close to the desired one. In each case, not only is the curve much lower than the desired value, but it also continues to a downward drop, which means that increasing parallelism achieves only marginal speedup. To understand the reason for this behavior, the total cost was broken down into its individual components and biggest cause of the problem is the memory mapping costs associated with each algorithm. The reason is that the memory mapping for the multiple partitions of a file structure is done in a serial manner. Therefore, as the number of partitions is increased, the memory mapping setup costs increase linearly and soon become quite significant. Clearly, this behaviour negatively impacts on performance and can be solved by parallelizing the initial setting up of the various partitions of the file structure. However, it is not obvious how this parallelization can be efficiently achieved.

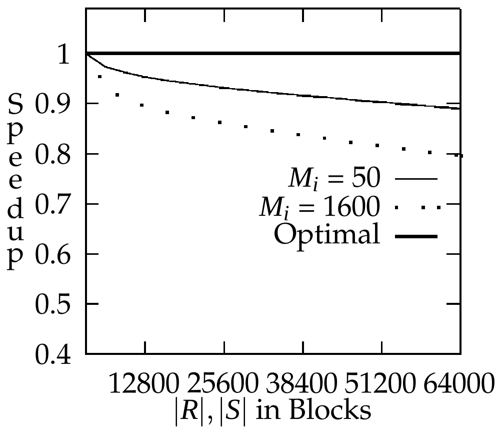
In order to see the impact of the serial mapping setup costs, the scaleup graphs were recomputed after subtracting the memory mapping setup costs from total costs. The resulting graphs are presented in figure 5.12 on page 212, and the results are clearly much better. All the algorithms display similar scaleup behaviour. Scaleup is not very good in the lowest part of the curve but after about 4 disks, the curves straighten out and stay horizontal indicating near-perfect scaleup after that point.

5.7 Summary

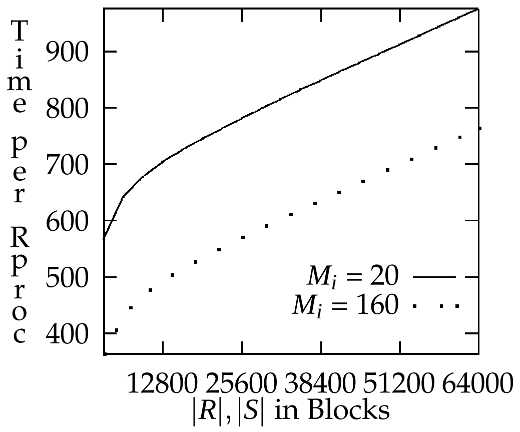
The analytical model developed in chapter 3 was used to predict the performance of the parallel multi-disk versions of three database join algorithms, namely, nested loops, sort-merge and Grace. The parallel versions were developed as part of this work and were especially tuned for performance in the EPD environment. A unique aspect of the algorithms is the use of a virtual pointer as the join attribute, which results in considerable time savings by eliminating the sorting or hashing of one of the two joining relations. Note however, that the use of pointer-based joins is not appropriate in all applications, e.g., the use of a pointer as the join attribute makes updates to the database much more expensive. The accuracy of the analytical model has been verified by conducting experiments on a controlled testbed. This chapter also highlighted a fundamental problem associated with abrogating control to the operating system for making page replacement decisions. In a rigid operating system, this lack of control can seriously hamper performance under specific scenarios. Therefore, to achieve the maximum performance out of the EPD approach, the use of an operating system system with flexible page replacement and related support is required.



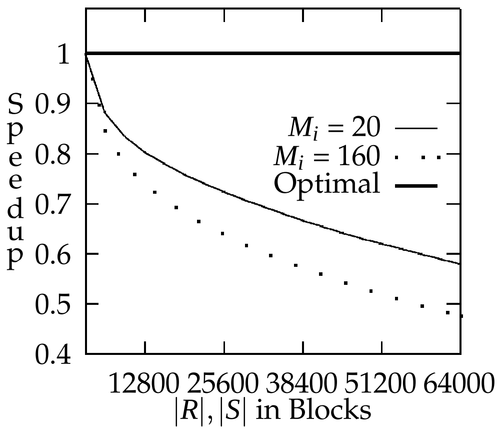
(a) Nested Loops – Time



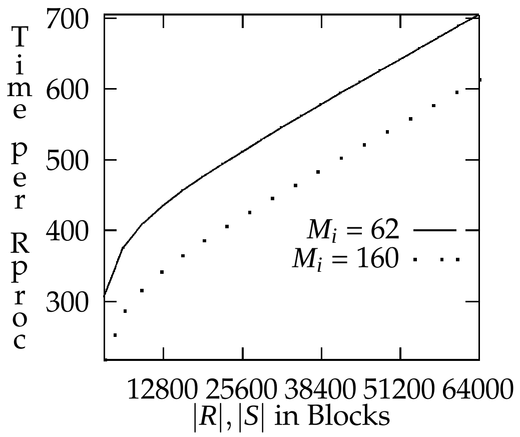
(b) Nested Loops – Speedup



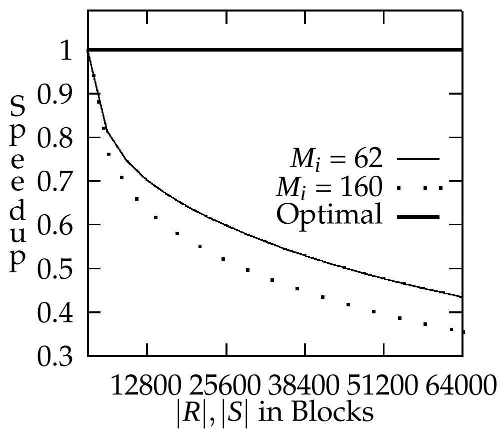
(c) Sort Merge – Time



(d) Sort Merge – Speedup

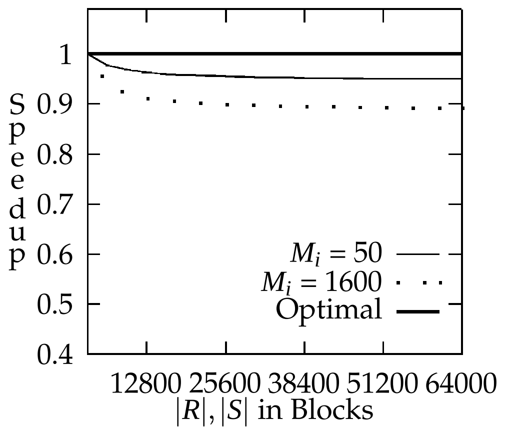


(e) Grace – Time

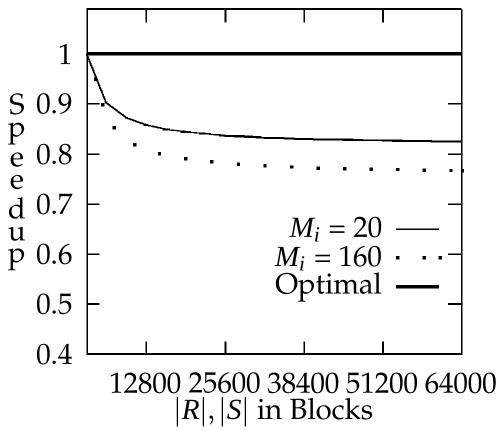


(f) Grace – Speedup

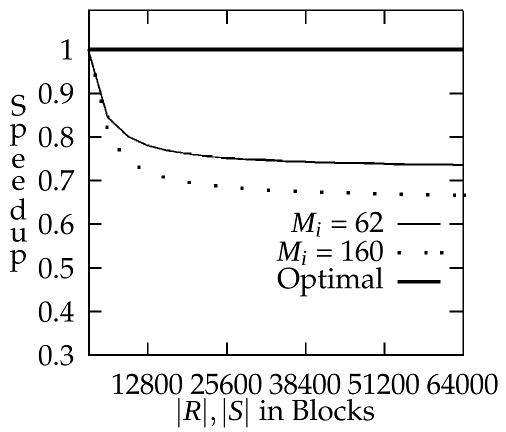
Figure 5.11: Scaleup ($P = D = |R|/3200$)



(a) Nested Loops



(b) Sort Merge



(c) Grace

Figure 5.12: Scaleup without Mapping Overhead ($P = D = |R|/3200$)

Chapter 6

Unresolved Issues and Future Work

As stated earlier, some aspects of constructing a persistent system based on the EPD approach to memory mapping have not been dealt with in this dissertation due to the size of the undertaking. A deliberate decision was made to concentrate efforts on designing and building an EPD based store that supported multiple simultaneously accessible persistent areas and on extensively measuring and analyzing the resulting store. Some of the unresolved issues have been dealt with by other researchers while some problems are still open research issues. This chapter describes the major unresolved problems, by means of a partial survey of related work, and future work.

6.1 Concurrency Control

Concurrency control deals with front-end concurrency mentioned in section 3.3, i.e., how to deal with multiple simultaneous reads and writes to a persistent store while maintaining the consistency of data. The concurrency control problem has been studied extensively by the database community and there are excellent solutions available, e.g., see [BK91] for a good survey of concurrency control techniques for advanced database systems. However, the traditional concurrency control solutions are often not directly applicable or are inefficient for persistent systems, especially page-based persistent systems.

Most of the persistent systems built so far have been single-user systems that ignore problems of concurrent accesses to a persistent store by multiple users and applications. In fact most existing persistent systems provide no support for concurrency control [RD95a]. The concurrency problem as it applies to persistent systems remains a largely open research issue, although some work dealing with the problem is beginning to appear (see [RD95b]). For example, Inohara, *et al* [ISU$^+$95] have proposed a versioned optimistic (VO) page based scheme for memory mapped persistent systems.

One of the major problems with using conventional concurrency control schemes is the granularity of locking. Conventional schemes work best for object-grain locking whereas persistent systems tend to be page-based; providing object level locking in a page-based persistent store is quite problematic. A central issue to be resolved is to determine what constitutes a unit of data for the purposes of locking, atomicity, etc. While it might be desirable to use the *individual objects* in a file structure as the units of data on which atomic operations are permitted, this is clearly not feasible in the proposed architecture of μDatabase with implicit concurrency (it is possible with explicit concurrency). This support requires the availability of an *object manager*, as in object-oriented database systems/servers such as ORION [KBC$^+$88] and GemStone [PSM87]. Furthermore, it must be possible to lock arbitrary collection of bytes in memory, a luxury not usually available to a mapped system.

The VO scheme developed by Inohara, *et al*, derives from conventional optimistic schemes while adding support for multi-version page images and a new validation algorithm. Optimistic schemes usually work by letting concurrent clients work on different versions of objects; each client updates its own copy of data independently of other clients. At transaction commit time, a validation algorithm attempts to determine if the changes made by the specific transaction are consistent with other committed transactions. If not, the transaction is aborted. Practical validation algorithms attempt to serialize as many as possible (instead of all) transactions that could, in theory, be serialized; doing otherwise has been claimed to be an NP-complete problem. Thus, some transactions that could be committed safely are aborted. The validation algorithm used by the VO scheme commits all read-only transactions and is claimed to perform better than the original optimistic schemes for other transactions.

6.1.1 Integration of Concurrency, Distribution and Persistence

As part of his doctoral work Munro [Mun93] investigated the integration of distribution and concurrency mechanisms into an existing orthogonal persistence system, Napier88 [Bro89]. Munro modified the Napier88 architecture and Brown's stable store, while maintaining upward compatibility to the extent possible, in the process of his investigation. One of the main contributions of Munro's work is the development of a new layered architecture, now called Flask [MCM$^+$94], whose principal aim is to provide support for building generic concurrency mechanisms for persistent stores. Like μDatabase, Flask rejects the notion of hard-wiring fixed concurrency schemes into the store itself. Instead, it provides a framework on top of the store that can be exploited to build whatever notions of concurrency are desirable for specific applications. Future work on concurrency in μDatabase can gain from an incorporation of ideas from Flask. For instance, [MCM$^+$94] includes a design of concurrent shadow paging mechanisms for providing stability (see section 6.2 for a discussion of stability in persistent stores). In Flask, stability is provided in a layer built on top of the concurrency layer.

6.1.2 Scalability

A related issue is that of scalability along two dimensions, namely, the number of simultaneously accessed databases and the number of concurrent applications accessing these databases. Increasing the number of databases accessed by a μDatabase application requires a linear increase in the number of representative segments (i.e., UNIX processes) and the accompanying resources such as physical memory for resident sets and swap space on disk. Currently, μDatabase does not support concurrent access to the same database by multiple applications. When such support is implemented in the future, an important consideration is to ensure that multiple applications share common resources allocated for a single database. In other words, there should not be a multiplicative effect on the amount of resources required as the number of applications and databases is increased.

6.2 Recovery Control

It is important for a persistent system to guarantee that all stored data is in a consistent state, i.e., the system must maintain the integrity of all data. This guarantee must be made in the face of system failure, such as system crashes and unsuccessful disk writes caused by disk failure. Recovery control is the mechanism to guarantee integrity of data by being able to recover from system failures. On restart after a failure, the recovery process returns the system to a previously recorded consistent state; the property of a system to recover from failures is also called stability. A number of proposals for stability in persistent systems have appeared in the literature (see [RHB$^+$90] for a list of references). Some of these proposals, especially the ones for page-based systems, can be adapted for the EPD system described in this dissertation. The rest of this section provides a general description of some of the proposed recovery schemes.

6.2.1 Shadow Paging

The earliest proposal for stability in persistent stores for database systems [Lor77] developed a new scheme called shadow paging, which has since been adapted by many other proposals.

The essential idea in a stable persistent system is to move the system from one consistent state to another as updates are made. If the system crashes before fully progressing to a new state, it must go back to a previous consistent state before applications are allowed to resume after restarting. Two basic operations required to implement this facility are the ability to perform an atomic update and the ability to distinguish the before and the after states of the system with respect to a commit or stabilize operation.

Challis's algorithm [Cha78] provides the underlying mechanism for implementing the atomic update operation. The basic idea is that a new copy of the data stored in the persistent system is made after each stabilize operation; each copy of the data is assigned a version number that distinguishes it from all other copies and can also be used to determine the temporal ordering of two copies. Each copy of the data also contains a mapping table that can be used to locate all the data components on disk. The location of the mapping table for a copy of the data on disk is maintained in a fixed location on disk called a root block; there are two root blocks with each describing a different consistent copy of data. The version number of the data pointed to by a root block is stored both at the beginning and the end of the root block; if the two copies of the version number stored in a root block match, the data copy pointed to by the root block is consistent. All changes to data are made to the current (or new) copy. In order to commit an update, the root block containing the oldest version number is updated to refer to the current updated copy of data. After a successful writing of the root block, the old copy of data can be removed. In the case of restarting after a failure, the status of the two root blocks is checked to find the one with the latest *consistent* version number, which is then used to revert the state of the system to the consistent state referred to by the root block. Should both root blocks be corrupted, it is called a catastrophic failure and not covered by the recovery mechanism.

The basic shadow paging scheme provides recovery control for a paged persistent store and is an adaptation of the expensive atomic update procedure described above. In order to implement shadow paging, a mapping between the virtual address space of the persistent store to the stored pages on disk is maintained. The mapping is called the disk page table and can be used to locate all pages of disk that make up a virtual address space. At the start of an update operation, a transient copy of the disk page table is made in primary memory, which is used to locate pages on disk to service page fault. When a location in a page in main memory needs to be changed, the paging system creates a copy of the page being updated on disk, if a copy does not already exist, and changes the transient disk page table to refer to the new copy. The copy of the page on disk is called a shadow page and ensures that a dirty page is never written back to the same location where it was read from. Note that the creation of a shadow page does *not* involve an actual copying of data from the original disk page to the new page. Instead, the image of the original page in memory is simply written back to a new location on disk.

Shadow paging makes copies of only those pages of data that are actually modified as opposed to copying all data as is done in the atomic update operation. In order to perform the stabilize operation, the system writes back all dirty pages in primary memory to the shadow disk pages, copies the transient disk table to the disk in a new location, and then updates the root block of the system in a manner similar to the atomic update operation. The above is a high-level description of the shadow paging scheme; see [RHB$^+$90] for more details.

It is non-trivial to implement a shadow paging scheme on top of an existing paging mechanism; some operating system support is necessary.

6.2.2 Write Ahead Logging

Conventional write ahead logging schemes (e.g., see [RM89]) can be used for paged persistent stores provided the paging mechanism can be modified to defer the writing back of dirty pages until explicitly requested. The essential idea is to let the persistent data be modified in place during updates and to make copies of all changes made to data in a separate persistent log. The actual persistent data is not written back to disk until after the changes have been safely written into the log. In case of failure, the system can be restored to a consistent state by means of the logs written since the last checkpoint. An earlier version of Texas [SKW92] used a write-ahead logging scheme. Again, operating system support is essential.

6.2.3 Page Diffing

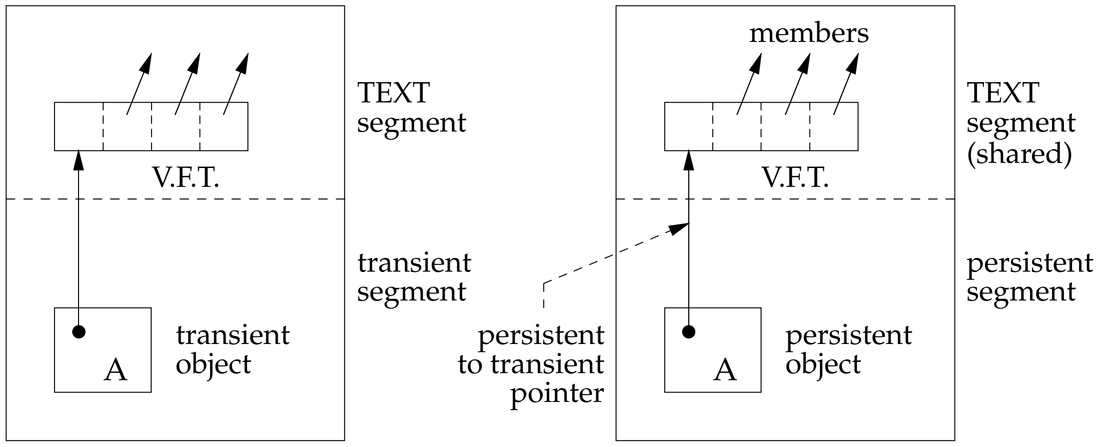
One problem with logging entire pages is the inefficiency caused by saving too much unchanged data, e.g., changing a single byte results in the entire containing page being logged. What is needed is the ability to checkpoint subsets of pages, or sub-pages. Dirty bits for sub-pages is one possible solution but it requires special hardware and/or software support. A simpler solution that requires virtually no hardware and software support is to save only the changed portions of a page by performing a word-by-word comparison of the modified page with a clean copy. This technique is called *page diffing* and has been use by Texas, QuickStore and others. Write protection traps are usually employed in page diffing; when the paging system detects a write fault exception, it copies the contents of the faulted page into a separate clean pages buffer, un-protects the original page and lets the execution continue. At the time of committing (or, when a dirty page needs to written back due to paging), the current dirty pages are compared against their respective clean copies and the differences are used to generate logs. The dirty pages themselves are not written back to the disk until after logging is complete. A space saving optimization used by Texas employs a bounded buffer for storing clean copies of dirty buffers; when the buffer fills up, some dirty pages are written out.

Finally, the problems of concurrency and recovery control become even more difficult when the persistent store is distributed; [RD95b] contains some early work to deal with the problem on distribution in persistent systems.

6.3 Support for Virtual Pointers

Recall, the current version of μDatabase does not support virtual pointers for persistent objects. Virtual pointers are used internally by C++ to implement virtual member functions and virtual base classes, two important reuse mechanisms in C++. This section assumes a basic understanding of the C++ virtual pointer implementation. Virtual pointers embedded in an object are stored as part of the object memory storage and are initialized by the compiler when the object is first created during program execution.

As shown in figure 6.1(a), a virtual pointer embedded in an object refers to the appropriate *virtual function table* (V.F.T.), which is stored in transient memory in the text segment. There is exactly one virtual function table created by the compiler for each *type* declared in an application. Since the exact location in memory of virtual function tables is determined only at the program linking/loading time, the values of virtual pointers stored in an object have a meaning only during the life of the program. This restriction does not cause any problems for transient objects because the objects also vanish when the program terminates, and are recreated and reinitialized when the program is run again.



(a) Object stored in transient area (b) Object stored in persistent area

Figure 6.1: Embedded Virtual Pointer Problem

However, as shown in figure 6.1(b), once an object with embedded virtual pointers is made persistent, a problem occurs because the object on disk now contains a reference to a transient object, and as such, the integrity of embedded virtual pointers is no longer guaranteed by the compiler across program invocations. It is up to the persistent system to ensure that the virtual pointers embedded in a persistent object are properly initialized at object loading time to refer to the current locations of the corresponding virtual function tables.

A solution adapted in O++ [BDG93] to support virtual pointers works by modifying all user-defined constructors to perform an initialization of embedded virtual pointers if a special global condition is true; otherwise, the normal constructor code is executed. When a persistent pointer to a non-memory-resident object is dereferenced, the system loads the disk page(s) containing the object into memory and invokes a constructor after asserting the special global condition mentioned earlier. This special invocation of the constructor results in the virtual pointer(s) embedded in the newly loaded object being initialized; the special global condition is reset after the invocation to allow subsequent calls to the same constructor to proceed normally.

The O++ solution can be easily implemented for μDatabase with the help of a front-end translator. However, this solution violates the μDatabase design objective of eliminating pointer modification every time an object is reloaded. While some pointer modification is unavoidable in order to support virtual pointers, the rest of this section outlines a possible solution that significantly reduces the extent of such modifications. The basic idea is illustrated in figure 6.2 and is based on the observation that in a persistent system the total number of different *types* of objects is much smaller than the total *number* of objects.

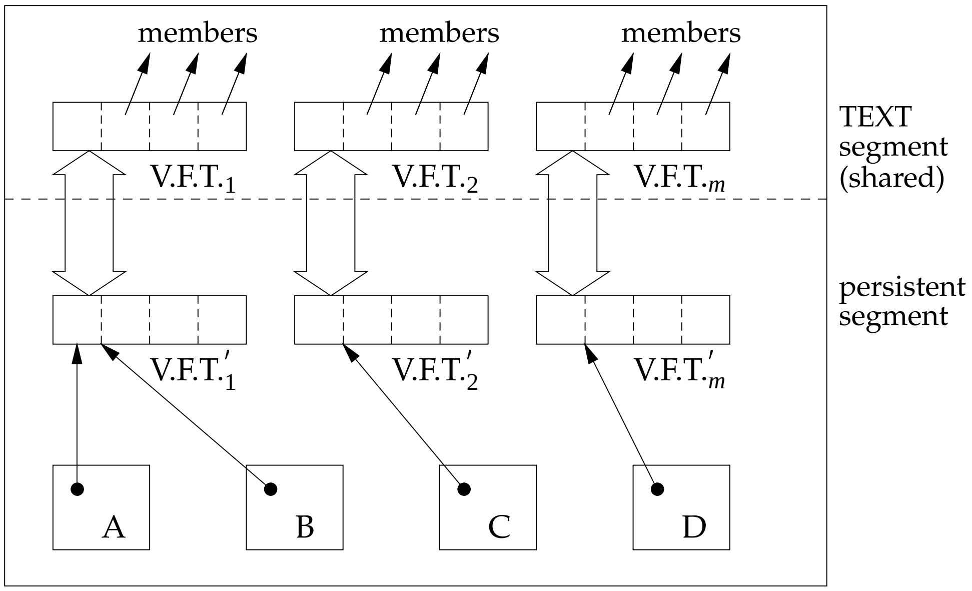


Figure 6.2: Efficiently Supporting Virtual Pointers

When the representative for a segment is first created, it queries the run-time system to locate all virtual function tables (VFTs) and copies them into fixed locations in the persistent area. It is important to note that the VFT for a given type is always copied to the same location in the persistent area so that any existing persistent pointers to the VFT stay valid. With the VFT copies in place, when a new persistent object of a type with virtual members is created, its embedded virtual pointers are initialized to refer to the persistent copies of the VFTs instead of the transient VFTs. When such an object is reused during a subsequent execution, the integrity of its embedded virtual pointers is guaranteed. This scheme, therefore, avoids the cost of pointer modification each time an object is reloaded. Instead, the scheme incurs a one time initialization cost of copying the VFTs when the persistent segment is first made accessible. This solution would require some compiler support to ensure the correct initialization of VFT pointer for an object, or some compiler-level modifications by μDatabase.

6.3.1 Persistent Code

μDatabase does not contain any mechanisms for storing compiled code in the persistent store because of the size of the undertaking and due to an inherent conflict with the μDatabase design goal to eliminate swizzling of pointers. As with virtual pointers, future versions of μDatabase may have to compromise this goal to make code persist but still retain the performance benefits of not having to swizzle pointers relating data. Some of the basic issues in persistent code are similar to persistent virtual pointers discussed in the previous section. In essence, loading and linking of code is just a complex form of pointer swizzling. A detailed discussion of these issues is beyond the scope of this work; see [BDBV94] for some relevant material.

6.4 Implementation of Inter-Database Pointers

As discussed earlier, a system that supports simultaneous access to multiple persistent areas has to deal with inter- as well as intra-database pointers. Providing a uniform interface for the two types of pointers, while desirable, results in very poor performance because of the extra cost involved with dereferencing inter-database pointers. Consequently, most systems settle on a different representation for the two types of pointers. Recall, in ObjectStore, a user has to explicitly declare inter-database pointers.

An inter-database pointer, by definition, has two logical components: a reference to a database or persistent area, and the location of the referent object within the persistent area. The current version of μDatabase does not support inter-database pointers but does allow inter-database pointers to be passed among segments, i.e., the entity dereferencing an intra-database pointer from another segment has to use its knowledge about the missing component and get the pointer dereferenced within the address space of the segment's representative. This lack of support is a clearly unacceptable situation and future work must concentrate on a suitable implementation of inter-database pointers. The rest of this section discusses some related issues.

Inter-database pointers can be implemented by means of *smart* or *long* pointers. A smart pointer is an abstract data type(ADT) that encapsulates the information needed to represent an inter-database pointer: the name of the containing database, the virtual address of the referent object within the corresponding persistent area and any other pertinent information (e.g., access information for the object). In addition, a smart pointer has a method that is invoked when an instance is *dereferenced*. For μDatabase, this method can create a representative for the database, establish a mapping, cause the virtual address of the object to be dereferenced within the address space, and the data copied out of the representative segment. After the dereferencing has been completed, the representative can be destroyed. The per dereference mapping costs in this procedure represent a high cost, and yet, it is the only probable solution if the semantics of the inter-database pointers have to be kept invisible at the user level. What is needed is a mechanism to save the context at the first dereference until after the last pointer that needs to be dereferenced in the same context has been processed. If the inter-database pointers can be made visible at the program level (e.g., as in the with clause in PASCAL), it is possible to implement the above solution at a reasonable cost.

6.5 Modelling

The analytical model developed as part of this work does an excellent job of predicting performance but there are several areas where new work or refinement of existing work can be done:

Modelling disk contention: When multiple disk requests arrive at a disk at the same time, the current model leaves the disk arbitration mechanism unspecified, which can result in some error, especially for algorithms that cause significant disk contention. It is possible to model the contention at the disk analytically or change the DTT measurements to include amortized disk contention costs.

More hash-based algorithms: There is scope to investigate more hash-based join algorithms given the importance of these algorithms; the more modern hash-based algorithms make much better use of available memory than the Grace algorithm. Also, there is a need to develop new pointer-based algorithms that further exploit the EPD environment.

Better modelling of the page replacement strategy: In the analysis of the Grace algorithm an attempt has been made to model the thrashing that occurs when the underlying page replacement algorithm makes non-optimal choices. While the attempted modelling produced acceptable results, there is scope for further refinement.

Chapter 7

Conclusions

Chapter 6 outlines some unresolved issues and much work that still needs to be done. In this chapter, I summarize what *has* been done.

7.1 Review of Work Done

A number of objectives that were set out have been achieved. The work done as part of this dissertation has been made available to the research community as three published articles, [BGW92], [BGNR96b] and [BGNR96a]. The achievements can be broadly classified into the following categories.

7.1.1 Static Type Safety

One of the fundamental motivations behind pursuing a single-level store is the desire to ensure type safety for accessing persistent data just like for transient programming language data. An attempt to achieve statically type checked access to a database has been partially successful. Currently, static type safety cannot be guaranteed for access to the UNIX file system and the storage management of a file structure's address space. However, once these two aspects of a file structure are specified correctly, all subsequent access to the database file structure are statically type-checked. The latter constitutes the majority of references to a typical file structure.

7.1.2 Development of the EPD Approach

The rationale for the EPD approach to memory mapping was developed based on an extensive survey of other related work. Using the EPD approach in conjunction with a world view, which is not flat and envisions persistent data objects being stored in collections of related objects, poses special challenges and problems. A working solution to these problems was investigated, developed and painstakingly measured. The solution has been shown to work remarkably well in spite of several outstanding problems some of which may indeed defy solution within the software non-architectural platform. The methodology that has been developed allows direct use of virtual memory pointers without any modification for referring to persistent and transient objects alike while allowing simultaneous access to multiple persistent areas or databases.

7.1.3 Experimental Work

Extensive experimentation is a novel and important part of this work. No other project has documented experimental results obtained from real or prototypical programs run on a memory mapped single-level store. Experiments were conducted to demonstrate the feasibility and viability of the EPD approach, to study the behaviour of parallel algorithms in an EPD environment, and for validating an analytical model of the system. In order to conduct all these experiments, a tightly controlled testbed was designed and developed. The testbed provided instrumentation support and allowed consistent experiments to be conducted with precision.

7.1.4 Feasibility Studies

I have shown that the EPD approach to memory mapping is an attractive alternative for implementing traditional file structures, both sequential and parallel, for databases. I have also presented a convincing case for using the EPD based databases in complex design environments such as CAD/CAM, text management and GIS. EPD based 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 file structures. Further, buffer management supplied through the page replacement scheme of the operating system seems to provide excellent performance for many, though not all, access patterns. Finally, these benefits can be made available in toolkit form on any UNIX system that supports the mmap system call.

Significant work was also done towards the study of parallel multi-disk file structures. Data partitioning is an important strategy essential for improving performance of persistent stores in the face of a primary to secondary storage speed disparity. Partitioned file structures and parallel access methods were found to work remarkably well in the EPD environment. The work on parallel structures included the design of a generic concurrent retrieval algorithm and related tools.

7.1.5 Analytical Modelling

I have designed and validated a quantitative analytical model for database computing in an EPD environment. The model is successfully used to make accurate predictions about the real time behaviour of three different parallel join algorithms, namely, nested-loops, sort-merge and a variation of Grace. The EPD methodology allows the use of virtual pointers as the join attributes, which introduces significant performance gain by eliminating the need to sort/hash one of the two relations. The analysis of the join algorithms also highlighted an inherent drawback in single-level stores: the lack of control over buffer management on the part of the database application results in incorrect decisions being made at times by the underlying page replacement strategy. While accepting this inefficiency, I have demonstrated two approaches to achieving predictable behaviour, an essential property in a database system. With single-level stores becoming more common, it is my hope that future research and development in operating system architecture will make it feasible for database applications to exercise more control over the replacement strategies used [AL91]. There is scope for further improvement in the design of the model, especially in the modelling of the underlying paging behaviour. Future work will involve extending the model to other memory mapped environments in order to perform comparative studies. It will also be an interesting exercise to explore the applicability of the model to traditional join algorithms.

Bibliography

[AACS87] Alok Aggarwal, Bowen Alpern, Ashok K. Chandra, and Marc Snir. A Model for Hierarchical Memory. In *ACM STOC*, pages 305–314, May 1987.

[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.

[AC88] Alok Aggarwal and Ashok K. Chandra. Communication Complexity of PRAMs. In *ICALP*, pages 1–17, 1988.

[ACFS94] B. Alpern, L. Carter, E. Feig, and T. Selker. Uniform Memory Hierarchies. *Algorithmica*, 12(2/3):72–109, August/September 1994.

[ACS87] Alok Aggarwal, Ashok K. Chandra, and Marc Snir. Hierarchical Memory with Block Transfer. In *IEEE FOCS*, pages 204–217, 1987.

[ACS89] Alok Aggarwal, Ashok K. Chandra, and Marc Snir. On Communication Latency in PRAM Computations. In *ACM SPAA*, pages 11–21, 1989.

[AHU83] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, Reading, Massachusetts, 1983.

[AL91] Andrew W. Appel and Kai Li. Virtual Memory Primitives for User Programs. In *4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 96–107, Santa Clara, California, U. S. A., April 1991.

[AM85] Malcolm P. Atkinson and Ronald Morrison. Procedures as Persistent Data Objects. *ACM Transactions on Programming Languages and Systems*, 7(4):539–559, October 1985.

[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.

[AV88] Alok Aggarwal and Jeffrey Scott Vitter. The Input/Output Complexity of Sorting and Related Problems. *Communications of the ACM*, 31(9):1116–1127, September 1988.

[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.

[BDBV94] S.J. Bushell, A. Dearle, A.L. Brown, and F.A. Vaughan. Using C as a Compiler Target Language for Native Code Generation in Persistent Systems. In Malcom Atkinson, David Maier, and Veronique Benzaken, editors, *Persistent Object Systems*, pages 16–42, Tarascon, France, September 1994. Springer-Verlag.

[BDG93] A. Biliris, S. Dar, and N. H. Gehani. Making C++ Objects Persistent: the Hidden Pointers. *Software – Practice and Experience*, 23(12):1285–1303, December 1993.

[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.

[BDZ89] P. A. Buhr, Glen Ditchfield, and C. R. Zarnke. Basic Abstractions for a Database Programming Language. In Richard Hull, Ron Morrison, and David Stemple, editors, *Database Programming Languages, 2nd International Workshop*, pages 422–437. Morgan Kaufmann, June 1989.

[BFC95] Peter A. Buhr, Michel Fortier, and Michael H. Coffin. Monitor Classification. *ACM Computing Surveys*, 27(1):63–107, March 1995.

[BGNR96a] Peter A. Buhr, Anil K. Goel, Naomi Nishimura, and Prabhakar Ragde. μDatabase: Parallelism in a Memory-Mapped Environment. To appear in the Proceedings of the ACM Symposium on Parallel Algorithms and Architectures, 1996.

[BGNR96b] Peter A. Buhr, Anil K. Goel, Naomi Nishimura, and Prabhakar Ragde. Parallel Pointer-Based Join Algorithms in Memory Mapped Environments. In *Proceedings of the 12th IEEE International Conference on Data Engineering*, pages 266–275, New Orleans, USA, February 1996. IEEE Computer Society Press.

[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.

[BK91] N. S. Barghouti and G. E. Kaiser. Concurrency Control in Advanced Database Applications. *ACM Computing Surveys*, 23(3):269–317, September 1991.

[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.

[BOS91] P. Butterworth, A. Otis, and J. Stein. The GemStone Object Database Management System. *Communications of the ACM, Special Section on Next-Generation Database Systems*, 34(10):64, October 1991.

[Bro89] A. L. Brown. *Persistent Object Stores*. PhD thesis, Universities of Glasgow and St. Andrews, Scotland, 1989. PPRR–71.

[BS90] Peter A. Buhr and Richard A. Stroobosscher. The μSystem: Providing Light-Weight Concurrency on Shared-Memory Multiprocessor Computers Running UNIX. *Software—Practice and Experience*, 20(9):929–963, September 1990.

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

[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.

[CFL93] Jeff Chase, Mike Feeley, and Hank Levy. Some Issues for Single Address Space System. *IEEE Workshop on Workstation Operating Systems*, October 1993.

[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.

[Cha78] M. F. Challis. Database Consistency and Integrity in a Multi-user Environment. In B. Schneiderman, editor, *Databases: Improving Usability and Responsiveness*, pages 245–270. Academic Press, 1978.

[CLBHL92] Jeff Chase, Hank Levy, Miche Baker-Harvey, and Ed Lazowska. Opal: A Single Address Space System for 64-bit Architectures. *IEEE Workshop on Workstation Operating Systems*, April 1992.

[CLFL94] Jeffrey S. Chase, Hank M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and Protection in a Single Address Space Operating System. *ACM Transactions on Computer Systems*, 12(4):271–307, May 1994.

[CLV91] Neil Coburn, Per-Ake Larson, and Surendra K. Verma. A Query Processing Architecture for Share-Memory Multiprocessors. Technical Report CS-91-48, University of Waterloo, Waterloo, Ontario, Canada, 1991.

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

[Coc85] W. P. Cockshott. Addressing Mechanisms and Persistent Programming. In *Workshop on Persistent Object Systems: their design, implementation and use*, volume PPRR 16, pages 369–389, Appin, Scotland, August 1985. Universities of Glasgow and St. Andrews, Scotland.

[CRJ87] R. Campbell, V. Russo, and G. Johnston. The Design of a Multiprocessor Operating System. *Proceedings of the USENIX C++ Workshop*, pages 109–125, November 1987.

[D$^+$91] O. Deux et al. The O2 System. *Communications of the ACM*, 34(10):34–49, October 1991.

[DAG93] S. Dar, R. Agrawal, and N. H. Gehani. The O++ database programming language: implementation and experience. In *Proc. IEEE International Conference on Data Engineering*, pages 61–70, Vienna, Austria, 1993.

[DC90] Partha Dasgupta and Raymond C. Chen. Memory Semantics in Large Grained Persistent Objects. *Implementing Persistent Object Bases: Principles and Practice,* Proceedings of the Fourth International Workshop on Persistent Object Systems, pages 226–238, September 1990.

[DdBF$^+$94] Alan Dearle, Rex di Bona, James Farrow, Frans Henskens, Andres Lindstrom, John Rosenberg, and Francis Vaughan. Grasshopper: An Orthogonally Persistent Operating System. *Computing Systems*, 7(3), 1994.

[Den70] P. J. Denning. Virtual Memory. *ACM Computing Surveys*, 2(3):153–189, September 1970.

[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.

[DRH^{+}92] Alan Dearle, John Rosenberg, Frans Henskens, Francis Vaughan, and Kevin Maciunas. An Examiniation of Operating System Support for Persistent Object Systems. In *Proceedings of the 25th Hawaii International Conference on System Sciences*, volume 1, pages 779–789, Hawaii, USA, 1992. IEEE Computer Society Press.

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison Wesley, first edition, 1990.

[FMP^{+}95] Michael J. Feeley, William E. Morgan, Frederic H. Pighin, Anna R. Karlin, Henry M. Levy, , and Chandramohan A. Thekkath. Implementing Global Memory Management in a Workstation Cluster. *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[FON90] Mary Fontana, LaMott Oren, and Martin Neath. A Portable Implementation of Parameterized Templates Using A Sophisticated C++ Macro Facility. In *Proceedings of the C++ at Work 1990 Conference*, NJ, USA, September 1990.

[FW78] Steven Fortune and James Wyllie. Parallelism in Random Access Machines. In *ACM STOC*, pages 114–118, 1978.

[GADV92] Olivier Gruber, Laurent Amsaleg, Laurent Daynès, and Patrick Valduriez. Support for Persistent Objects: Two Architectures. In *Proceedings of the 25th Hawaii International Conference on System Sciences*, volume 1, pages 757–768, Hawaii, USA, 1992. IEEE Computer Society Press.

[GBY91] G. H. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures*. Addison-Wesley, 1991.

[GM86] Gaston H. Gonnet and J. Ian Munro. Heaps on Heaps. *SIAM Journal on Computing*, 15(4):964–971, November 1986.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[Gra94] G. Graefe. Sort-Merge-Join: An Idea whose Time has(h) Passed? In *IEEE International Conference on Data Engineering*, page 406, Houston, TX, February 1994.

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

[HK81] Jia-Wei Hong and H. T. Kung. I/O Complexity: The Red-Blue Pebble Game. In *ACM STOC*, pages 326–333, 1981.

[IBM81] International Business Machines. *OS and DOS PL/I Reference Manual*, first edition, September 1981. Manual GC26-3977-0.

[ISU$^+$95] Shugekazu Inohara, Yoji Shigehata, Keitaro Uehara, Hajime Miyazawa, Kouhei Yamamoto, and Takashi Masuda. Page-Based Optimistic Concurrency Control for Memory Mapped Persistent Object Systems. In *Proceedings of the 28th Hawaii International Conference on System Sciences*, pages 645–654, Hawaii, USA, 1995. IEEE Computer Society Press.

[JK77] Norman L. Johnson and Samuel Kotz. *Urn Models and their Application: An approach to modern discrete probability theory.* John Wiley & Sons, 1977.

[KBC$^+$88] Won Kim, Nat Ballou, Hong-Tai Chou, Jorge F. Garza, and Darrell Woelk. Integrating an Object-Oriented Programming System with a Database System. *Proceedings of the OOPSLA'88 Conference*, pages 142–152, October 1988.

[Kol90] Elliot K. Kolodner. Automatic Incremental Garbage Collection and Recovery for a Large Stable Heap. In A. Dearle et al, editor, *Implementing Persistent Object Bases: Principles and Practise,* Proceedings of the Fourth International Workshop on Persistent Object Systems", pages 185–198. Morgan Kaufmann, 1990.

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

[KSU91] Orran Krieger, Michael Stumm, and Ron Unrau. Exploting the Advantages of Mapped Files for Stream I/O. *Proceedings of the USENIX C++ Workshop*, June 1991.

[KTMo83] M. Kitsuregawa, H. Tanaka, and T. Moto-oka. Application of Hash to Data Base Machine and Its Architecture. *New Generation Computing*, 1(1):63–74, 1983.

[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.

[Lar88] P. A. Larson. The Data Model and Query Language of LauRel. *Data Engineering Bulletin*, (3):23–30, 1988.

[LDM93] Daniel F. Lieuwen, David J. DeWitt, and Manish Mehta. Pointer-Based Join Techniques for Object-Oriented Databases. In *International Conference on Parallel and Distributed Information Systems*, San Diego, CA, USA, January 1993.

[LL82] Henry M. Levy and Peter H. Lipman. Virtual Memory Management in the VAX/VMS Operating System. *IEEE Computer*, 15(3):35–41, March 1982.

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

[Lor77] R. A. Lorie. Physical Integrity in a Large Segmented Database. *ACM Transactions on Database Systems*, 2(1):91–104, March 1977.

[MC92] Peter W. Madany and Roy H. Campbell. Organizing and Typing Persistent Objects within an Object-Oriented Framework. In *Proceedings of the 25th Hawaii International Conference on System Sciences*, volume 1, pages 800–809, Hawaii, USA, 1992. IEEE Computer Society Press.

[MCM$^+$94] D. S. Munro, R. C. H. Connor, E. Morrison, S. Scheuer, and D. W. Stemple. Concurrent Shadow Paging in the Flask Architecture. In Malcom Atkinson, David Maier, and Veronique Benzaken, editors, *Persistent Object Systems*, pages 16–42, Tarascon, France, September 1994. Springer-Verlag.

[MCM$^+$95] D. S. Munro, R. C. H. Connor, R. Morrison, J. E. B. Moss, and S. J. G. Scheuer. Validating the MaStA I/O Cost Model for DB Crash Recovery Mechanisms. *Proceedings of the OOPSLA Workshop on Object Database Behaviour, Benchmarks and Performance*, 1995.

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

[ML89] L. Mackert and G. Lohman. Index Scans Using a Finite LRU Buffer: A Validated I/O Model. *ACM Transactions on Database Systems*, 14(3):401–424, September 1989.

[MLD94] T. P. Martin, P.-A. Larson, and V. Deshpande. Parallel Hash-Based Join Algorithms for a Shared-Everything Environment. *IEEE Transactions on Knowledge and Data Engineering*, 6(5):750–763, October 1994.

[MM92] Ashok Malhotra and Steven J. Munroe. Support for Persistent Objects: Two Architectures. In *Proceedings of the 25th Hawaii International Conference on System Sciences*, volume 1, pages 737–746, Hawaii, USA, 1992. IEEE Computer Society Press.

[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.

[Mun93] D. S. Munro. *On the Integration of Concurrency, Distribution and Persistence*. PhD thesis, University of St. Andrews, Scotland, 1993.

[Mun95] Ian Munro. Private communication with Prof. Ian Munro, CS Dept., University of Waterloo, 1995.

[Obj93] *ObjectStore User Guide: DML, ObjectStore Release 3.0 for UNIX Systems*. Object Design, Inc., 25 Burlington Mall Road, Burlington, MA, U. S. A., 01803, December 1993.

[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.

[PSM87] Alan Purdy, Bruce Schuchardt, and David Maier. Integrating an Object Server with Other Worlds. *ACM Transactions on Office Information Systems*, 5(1):27–47, January 1987.

[PU87] Christos Papadimitriou and Jeffrey D. Ullman. A Communication-Time Tradeoff. *SIAM Journal on Computing*, 16(4):639–646, August 1987.

[RC89] Vincent F. Russo and Roy H. Campbell. Virtual Memory and Backing Storage Management in Multiprocessor Operating Systems Using Object-Oriented Design Techniques. Technical Report UIUCDCS-R-89-1509, University of Illinois at Urbana-Champaign, Urbana, Illinois, April 1989.

[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.

[RD95a] John Rosenberg and Alan Dearle. Distribution and Concurrency in Persistent Systems – Introduction to Minitrack. In *Proceedings of the 28th Hawaii International Conference on System Sciences*, pages 633–644, Hawaii, USA, 1995. IEEE Computer Society Press.

[RD95b] John Rosenberg and Alan Dearle, editors. *Mintrack on Distribution and Concurrency in Persistent Systems, Proceedings of the 28th Hawaii International Conference on System Sciences*. IEEE Computer Society Press, Hawaii, USA, 1995.

[Req80] Aristides A. G. Requicha. Representations for Rigid Solids: Theory, Methods, and Systems. *ACM Computing Surveys*, 12(4):437–464, December 1980.

[RHB$^+$90] John Rosenberg, Frans Henskens, Fred Brown, Ron Morrison, and David Munro. Stability in a Persistent Store Based on a Large Virtual Memory. In *Proceedings of the International Workshop on Computer Architectures to Support Security and Persistence of Information*, pages 229–245, Bremen, West Germany, May 1990. Springer-Verlag.

[RK87] J. Rosenberg and J. Keedy. Object Management and Addressing in the MONADS Architecture. *Workshop on Persistent Object Systems: their design, implementation and use*, pages 114–133, August 1987.

[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.

[Ros90] John Rosenberg. The MONADS Architecture: A Layered View. *Implementing Persistent Object Bases: Principles and Practice,* Proceedings of the Fourth International Workshop on Persistent Object Systems, pages 215–225, September 1990.

[SC90] Eugene J. Shekita and Michael J. Carey. A Performance Evaluation of Pointer-Based Joins. In *ACM SIGMOD*, pages 300–311, Atlantic City, NJ, June 1990.

[SD89] Donovan A. Schneider and David J. DeWitt. A Performance Evaluation of Four Parallel Joins Algorithms in a Shared-Nothing Multiprocessor Environment. In *ACM SIGMOD*, pages 110–121, June 1989.

[Sha81] Mary Shaw, editor. *ALPHARD: Form and Content*. Springer-Verlag, 1981.

[Sha86] Leonard D. Shapiro. Join Processing in Database Systems with Large Main Memories. *ACM Transactions on Database Systems*, 11(3):239–264, September 1986.

[Sit92] Richard L. Sites, editor. *Alpha Architecture Reference Manual*. Digital Press, One Burlington Woods Drive, Burlington, MA, U. S. A., 01803, 1992.

[SKW92] Vivek Singhal, Sheetal V. Kakkad, and Paul R. Wilson. Texas: An Efficient, Portable Persistent Store. In Antonio Albano and Ron Morrison, editors, *Persistent Object Systems*, pages 11–33, San Miniato, Italy, September 1992. Springer-Verlag. Workshops in Computing, Ed. by Professor C. J. van Rijsbergen, QA76.9.D3I59.

[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.

[SS93] Russel Schaffer and Robert Sedgewick. The Analysis of Heapsort. *Journal of Algorithms*, 15:76–100, 1993.

[Sto81] M. Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7):412–418, July 1981.

[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.

[SW92] Walter R. Smith and Robert V. Welland. A Model for Address-Oriented Software and Hardware. In *Proceedings of the 25th Hawaii International Conference on System Sciences*, volume 1, pages 720–729, Hawaii, USA, 1992. IEEE Computer Society Press.

[Sym87] *Symmetry Technical Summary*. Sequent Computer Systems, Inc., 1987.

[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.

[Tha86] Satish M. Thatte. Persistent Memory: A Storage Architecture for Object-Oriented Database Systems. In *Proceedings of the International Workshop on Object-Oriented Databases*, pages 148–159, Pacfic Grove, CA, September 1986.

[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.

[VD92] Francis Vaughan and Alan Dearle. Supporting Large Persistent Stores using Conventional Hardware. In Antonio Albano and Ron Morrison, editors, *Persistent Object Systems*, pages 34–53, San Miniato, Italy, September 1992. Springer-Verlag. Workshops in Computing, Ed. by Professor C. J. van Rijsbergen, QA76.9.D3I59.

[vdBL89] Jan van den Bos and Chris Laffra. PROCOL: A Parallel Object Language with Protocols. *SIGPLAN Notices*, 24(10):95–102, October 1989. Proceedings of the OOPSLA'89 Conference, Oct. 1–6, 1989, New Orleans, Lousiana.

[vDT72] Andries van Dam and Frank Wm. Tompa. Software Data Paging and Segmentation for Complex Systems. *Information Processing Letters*, 1:80–86, 1972.

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

[VS94a] Jeffrey S. Vitter and Elizabeth A. M. Shriver. Algorithms for Parallel Memory, I: Two-Level Memories. *Algorithmica*, 12(2/3):110–147, August/September 1994.

[VS94b] Jeffrey S. Vitter and Elizabeth A. M. Shriver. Algorithms for Parallel Memory, II: Hierarchical Multi-Level Memories. *Algorithmica*, 12(2/3):148–169, August/September 1994.

[VSWL91] Zvonko G. Vranesic, Michael Stumm, Ron White, and David Lewis. The Hector Multiprocessor. *Computer*, 24(1):x–x, January 1991.

[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.

[WD94] Seth J. White and David J. DeWitt. QuickStore: A High Performance Mapped Object Store. In *ACM SIGMOD*, pages 395–406, Minneapolis, MN, U.S.A., May 1994.

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

[Wil91a] 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.

[Wil91b] Paul R. Wilson. Some Issues and Strategies in Heap Management and Memory Hierarchies. *SIGPLAN Notices*, 26(3):45–52, March 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