High-Performance Implementation Techniques For Next-Generation Database Systems

by

Eugene Shekita

Computer Sciences Technical Report #1026
May 1991

HIGH-PERFORMANCE IMPLEMENTATION TECHNIQUES FOR NEXT-GENERATION DATABASE SYSTEMS

by

Eugene J. Shekita

A thesis submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
(Computer Sciences)

at the
UNIVERSITY OF WISCONSIN — MADISON
1990

ABSTRACT

Database systems have traditionally been designed for business applications. In the last decade, however, it has become increasingly clear that application areas such as computer-aided design and manufacturing (CAD/CAM), computer-aided software engineering (CASE), image processing, etc., can also benefit from database technology. Unfortunately, the relational database systems that are used for business applications are often ill suited for these sorts of applications. More recently, several "next-generation" database systems, which are often classified as extended relational database systems or object-oriented database systems, have appeared to address the demands of these emerging application areas. Of course, whether these next-generation database systems actually succeed will largely depend on their performance.

This thesis describes and analyzes three different implementation techniques for improving the performance of next-generation database systems. A separate research chapter is devoted to each technique. In the first research chapter of this thesis, we describe a technique called *field replication*, which uses replicated data to eliminate some of the functional joins that would otherwise be required for query processing. We describe how field replication is specified at the data model level and present storage-level mechanisms to efficiently support it. An analytical cost model is developed to give some feel for how beneficial this technique can be and the circumstances under which it breaks down. While field replication is a relatively simple notion, the analysis shows that it can provide significant performance gains in many situations.

In the second research chapter of this thesis, we examine how physical pointers can be used effectively in join processing. We describe several pointer-based join algorithms that are simple variations on the well known nested-loops, sort-merge, hybrid-hash, and hash-loops join algorithms used in relational database systems. An analytical cost model is developed to compare the performance of the pointer-based join algorithms to their standard counter-parts. The results of the analysis show that the pointer-based algorithms can often provide significant performance gains over conventional, value-based join algorithms.

In the final research chapter of this thesis, we argue that traditional database storage systems are poorly suited for some emerging application areas such as CAD/CAM, and that an approach based on a single-level store may offer better performance. We describe a prototype storage system called Cricket that was developed to explore the feasibility of such an approach. Cricket uses the memory management primitives of the Mach operating system to provide the abstraction of a shared, transactional, single-level store. Performance results for the Sun Benchmarks indicate that the approach taken in Cricket has the potential to perform well on applications whose working sets fit in memory.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

Researchers and commercial developers of database management systems have historically focused their efforts on business applications. In recent years, however, it has become increasingly clear that there are a number of other application areas that can benefit from database technology. Examples of such application areas include computer-aided design and manufacturing (CAD/CAM), computer-aided software engineering (CASE), scientific data collection, computer-aided publishing (CAP), image processing, etc. Unfortunately, the relational database systems that are used for business applications are often ill suited for these sorts of applications. This is due in part to the fact that the data modeling requirements of such applications are often quite different from those of business applications. The kinds of entities and relationships that one finds in a CASE system, for example, are likely to be very different from those found in a typical banking application. The set of operations that must be efficiently supported is often quite different, too. For example, a VLSI design tool that uses standard relational joins and selections to check wire-routing would almost certainly be too slow to be of any value.

During the 1980's, a number of research projects were started to address the needs of these new application areas, and that work lead to the development of several prototype "next-generation" database systems. Examples of such systems include EXODUS at the University of Wisconsin [Care89], DASDBS at Darmstadt University [Sche90], POSTGRES at UC Berkeley [Ston90], Starburst at IBM's Almaden Research Center [Haas90], ORION at MCC [Kim90], O2 at Altair [Deux90], Iris at Hewlett-Packard [Wilk90a], and ODE at AT&T [Agra89]. A few next-generation database systems have also appeared commercially, including Objectivity's Objectivity/DB [Obje90], Object Design's ObjectStore [Atwo90], Servio Logic's Gemstone System [Maie86a], and Versant's Object Manager [Vers90]. Among other things, these systems distinguish themselves from their relational predecessors by providing support for database programming languages, versions, object-oriented data models, rules, triggers, complex objects, etc. These next-generation database systems are typically labeled as either being extended relational database systems or object-oriented database systems, but the distinction is blurred at best, and becoming more so all the time. POSTGRES, for example, has been characterized as both an extended relational database system and an object-oriented database system [Rowe87].

Of course, whether these next-generation database systems will actually succeed in application areas such as CAD/CAM will largely depend on how well they perform [Maie89]. In some respects, the situation is similar to what happened with relational database systems. Although relational systems provide more functionality and programming ease than the network and hierarchical database systems that preceded them, their widespread acceptance did not come until they offered reasonable levels of performance. Much the same can be expected for next-generation database systems, and consequently, one of the important research areas in the early 1990's will be finding ways to improve their performance.

This thesis describes and analyzes three implementation techniques for improving the performance of next-generation database systems. The first of these techniques is called *field replication*. Many of the newer data models that have been proposed include the notion of reference attributes or object-valued attributes (e.g., [Ship81, Zani83, Cope84, Bane87, Fish87, Care88]). These attributes are often implemented with object identifiers (OIDs), which act like pointers, and functional or implicit joins are usually supported as a dereferencing mechanism. In a data model that supports reference attributes, for example, one could create a set *Emp* of employee objects in which each employee object referenced a department object via the reference attribute *dept*. A functional join of the form *Emp.dept.name* could then be used to determine the department name of an employee. The general idea behind field replication is to use replicated data to eliminate some of the functional joins that would otherwise be required for query processing. This technique gets its name from the fact that data fields that would normally be accessed through a functional join are replicated, allowing the join to be eliminated or at least made less costly. For example, a replication path could be specified on *Emp.dept.name*, causing department names to be replicated and stored in employee objects. This would in turn allow functional joins on *Emp.dept.name* to be eliminated.

The second implementation technique described in this thesis involves a collection of algorithms called *pointer-based joins*. Unfortunately, field replication will not always be appropriate, either because the cost of updating replicated data would be too high or because of the storage overhead associated with replicated data. Under such circumstances, more conventional join techniques are necessary to process functional joins. In the chapter on pointer-based joins, we describe how physical OIDs (i.e., pointers) can be used effectively in processing functional joins. We describe several pointer-based join algorithms that are simple variations on well-known relational join algorithms, and show how they can often provide better performance than their standard counterparts. The pointer-based algorithms that we describe are actually quite general, and they can be used in any database system where

physical pointers are used to link records or objects together.

The final implementation technique described in this thesis has to do with low-level storage management. Field replication and pointer-based joins are both techniques for improving the performance of next-generation database systems on fairly traditional data access patterns, where large sets of related data items are accessed via a non-procedural query language. These access patterns tend to be very regular, with each data item being operated upon with the same methods. The computational requirements and access patterns of emerging applications such as CAD/CAM, however, are quite different from those of traditional database applications [Katz87, Chan89a, Chan89b, Catt90]. Non-procedural query languages appear inappropriate for such applications, and fast, low-level navigation is required to obtain adequate levels of performance [Catt90, Maie89]. Unfortunately, traditional database storage systems appear poorly suited for these sorts of applications. Among other things, the procedure-based interface that must typically be used to traverse and update data is too slow [Moss90]. Also, as noted in [Maie89], traditional database recovery protocols are often inappropriate, requiring a log record to be generated on every update. In the final research chapter of this thesis, we argue that a storage system based on a single-level store may offer better performance for some emerging application areas such as CAD/CAM. We then describe a prototype database storage system called Cricket that was developed to explore the feasibility of such an approach. Cricket uses the memory management primitives of the Mach operating system [Acce86] to provide the abstraction of a shared, transactional, single-level store.

1.1. THESIS OUTLINE

The remainder of this thesis is organized as follows: In Chapter 2, a brief survey of related work is presented to give the reader an idea of how the research in this thesis compares to previous work. The chapter on related work is followed by the three research chapters of this thesis. For the most part, these are presented as independent research topics, and each chapter includes its own introduction, conclusions, and directions for future work. Note that much of the work presented in this thesis has been described elsewhere [Shek89, Shek90a, Shek90b].

The first of the research chapters, Chapter 3, is on field replication. In this chapter, we describe how field replication is specified at the data model level and we present storage-level mechanisms to efficiently support it. An analytical cost model is also developed to examine how beneficial this technique can be and the circumstances under which it breaks down.

Chapter 4 is on pointer-based joins. In this chapter, we describe several pointer-based join algorithms that are simple variations on the well known nested-loops, sort-merge, hybrid-hash, and hash-loops join algorithms used in relational database systems. Each join algorithm is described and an analysis is carried out to compare the performance of the pointer-based algorithms to their standard counterparts.

The last of the research chapters, Chapter 5, describes Cricket. In this chapter, we describe the implementation of the Cricket prototype, and then present the results of a performance study comparing Cricket to the EXODUS Storage Manager [Care89] on the Sun Benchmarks [Catt90]. The chapter on Cricket is essentially a feasibility study to examine whether a storage system built around a single-level store can provide better performance than a conventional database storage system on emerging applications such as CAD/CAM.

Finally, a thesis summary is presented in Chapter 6. This chapter contains a brief recap of each research chapter's main results and directions for future work.

CHAPTER 2

RELATED WORK

In this chapter, we review some of the more significant work that is related to the research chapters of this thesis. A separate related work section is devoted to field replication, pointer-based joins, and Cricket. While the survey in each section is by no means exhaustive, it should give the reader an idea of how the work presented in this thesis compares to previous work.

2.1. RELATED WORK FOR FIELD REPLICATION

2.1.1. Caching in Postgres

The work most closely related to field replication is that done in POSTGRES on caching the results of procedural fields [Hans87, Sell87, Ston87, Jhin88]. Such fields have their contents defined by POSTQUEL queries. Three basic caching strategies have been analyzed in the context of POSTGRES [Hans87, Hans88, Jhin88]:

cache-in-tuple and invalidate

In this strategy, cached query results are stored directly in tuples. Cached results are marked as invalid when they become out-of-date due to updates. Special locking mechanisms are used to detect when cached results become out-of-date.

cache-separately and invalidate

This is essentially the same as the cache-in-tuple strategy, except that cached results are stored in a separate *Cache* relation and shared when possible. Hashing is used to access the Cache relation.

update cache

This is a variant of the other two strategies in which cached results are kept up-to-date rather than being invalidated. Algebraic view maintenance algorithms [Blak86, Hans87, Hans88] are used to update cached results.

Implementation results in [Jhin88] showed that the cache-in-tuple and cache-separately strategies can improve performance on many types of join queries. Neither caching strategy dominated on all the workloads studied, however. Analytical cost models were used in [Hans87] and [Hans88] to show the viability of the update cache strategy

on select-project-join queries.

As the reader shall see, field replication and caching in POSTGRES have many things in common. In fact, the chapter on field replication will introduce two replication strategies, the *in-place replication* strategy and the *separate replication* strategy, that were directly motivated by the cache-in-tuple and cache-separately strategies of POSTGRES. To some extent, field replication can be viewed as a primitive form of caching, where only equi-joins with projection are permitted in procedural fields and where cached data is always kept up-to-date. On closer inspection, though, field replication and caching are found to differ in three major ways.

The first major difference between field replication and POSTGRES-style caching is that the mechanisms described in this thesis for keeping replicated data consistent are very different from those used in POSTGRES. Second, because field replication is more primitive than caching, it should prove easier to implement and also more efficient. Field replication should prove more efficient than caching because, among other things, special locks do not have to be maintained to invalidate replicated data, nor do general view maintenance algorithms have to be used to keep replicated data up-to-date. Finally, query optimization should be significantly easier with field replication. In caching, there is always the possibility that cached results may either be invalid or not present when needed. Query optimization would appear to be difficult in such a dynamic environment. With field replication, on the other hand, replicated values are always guaranteed to exist and, moreover, are guaranteed to be up-to-date. As a result, traditional optimization techniques that use static cost analysis can be applied.

Another example of related work in POSTGRES is that of [Jhin90], where POSTGRES-style caching was used to cluster sub-objects in complex objects. If, for example, objects A and B both shared a common sub-object C, then a cached copy of C would be clustered with A and another cached copy would be clustered with B. An analytical cost model was used to show that caching sub-objects in this manner can result in better performance than simply clustering C near either A or B. In general, all the disadvantages of caching that were mentioned above hold for this work as well. An examination of how field replication — actually full object replication — could be used in the context described in [Jhin90] is left for future work.

2.1.2. Path Indexes

The work on field replication is also related to that of [Maie86b], which described the design and implementation of *path indexes* in the Gemstone object-oriented database system. A path index is basically the same as a normal database index, except that it is defined on a reference path. Returning to the employee example in the

introduction, a path index on *Emp.dept.name* would map department names to employee objects in *Emp*, making it possible to efficiently find all the employees that belong to a given department. Path indexes in Gemstone are implemented using B+ trees. A path index on *Emp.dept.name*, for example, would consist of two B+ tree components. One B+ tree component would map each department name to the OIDs of the department objects with that name, while the other B+ tree component would map the OID of each department object D to the OIDs of the employee objects that reference D. In essence, a path index on *Emp.dept.name* implements the *inverted path* $Emp.dept.name^{-1}$, mapping department names to employee objects in *Emp*.

As the reader shall see, inverted paths are also used in field replication, except that there they are used to propagate updates to replicated values. If a replication path were defined on *Emp.dept.name*, for example, then the inverted path $Emp.dept.name^{-1}$ would be used to propagate department name updates back to objects in *Emp*. In the chapter on field replication, we describe how inverted paths can be implemented using direct object-to-object mappings, rather than with B+ trees as in Gemstone. The advantages of such an approach include lower traversal costs (by avoiding index I/O), and more clustering options. More will be said about this in the chapter on field replication.

Two other techniques for implementing path indexes were recently described in [Bert89] and [Kemp90]. The authors in [Bert89] basically suggested storing full reference paths in one B+ tree, rather than splitting up them up into separate B+ tree components, as in Gemstone. Using an analytical model, this approach was shown to be generally superior to the approach taken in Gemstone when updates were infrequent. The techniques described in [Bert89] were then combined with those used in Gemstone to arrive at the more generalized notion of *access relations* in [Kemp90]. Among other things, the authors in [Kemp90] showed how a path index could effectively be treated as a relation itself and used to optimize several types of queries. Although it has not been done in this thesis, an interesting research problem would be to examine how access relations could be used to simultaneously support both path indexes and field replication.

2.2. RELATED WORK FOR POINTER-BASED JOINS

2.2.1. Join Algorithms in Relational Database Systems

As mentioned in the introduction, the chapter on pointer-based joins describes and analyzes several join algorithms that are simple variations on the well-known nested-loops, sort-merge, hash-loops, and hybrid-hash join algorithms used in relational database systems. The original versions of these algorithms were analyzed in [Blas77],

[Dewi84], [Dewi85], and [Shap86]. In [Blas77], the nested-loops and sort-merge algorithms were described and analyzed, while in [Dewi84] and [Shap86], the hybrid-hash algorithm was analyzed and shown to be generally superior to the sort-merge as well as the Grace and simple-hash join algorithms. The hash-loops algorithm was described in [Dewi85], where a simulation study showed it to be inferior to the hybrid-hash algorithm in a shared-nothing multi-processor. All of these papers were concerned with joins in a purely relational context, however, and consequently none of them considered pointer-based joins.

2.2.2. Join Indexes

The work of [Vald87] is also related to pointer-based joins. There, auxiliary data structures called *join indexes* were described as a way to speedup join processing, The author of [Vald87] showed that a join index for R and S could often improve performance if their join was frequently needed. The join index would consist of pairs of pointers, matching each record in R with the record(s) it joins with in S, and it would effectively implement the pre-computed join of R and S.

In [Vald87], joins using a join index were shown to be superior to hybrid-hash joins in many situations, particularly when only a portion of R or S participated in the join (i.e., when the *join selectivity* was less than one). For the types of joins that are analyzed in this thesis, however, pointer-based joins would always outperform a join index. This is because we will be assuming the existence of embedded pointers, which effectively implement a join index without the overhead of accessing an auxiliary data structure. The chapter on pointer-based joins will also show that the pointer-based hash-loops algorithm, which is the essentially the same join algorithm used with a join index, generally performs quite poorly. Regrettably, the analysis in [Vald87] did not include the effects of using *bit-filtering* [Dewi85] in hybrid-hash joins, and this biased the results in favor of join indexes when the join selectivity was less than one. The chapter on pointer-based joins will include the effects of bit-filtering.

2.2.3. Starburst's Incremental Join Facility

The final work related to pointer-based joins is that of [Care90], which described the *incremental join facility* that was implemented in Starburst. That facility, which the author of this thesis helped design and implement, was used to study the potential benefits of inter-relation clustering and pointer-based joins. In [Care90], an empirical performance study was carried out to compare a variety of pointer-based join algorithms to their standard, value-based counterparts under three different clustering strategies. Among other things, the study provided empirical evidence favoring the use of pointers for join computations. In contrast to [Care90], the chapter on pointer-based joins

takes more of an analytical perspective. It provides the sort of analytical cost models that are required for query optimization but were missing in [Care90]. It also includes the analysis and comparison of several pointer-based join algorithms that were not considered in [Care90].

2.3. RELATED WORK FOR CRICKET

2.3.1. Bubba

As mentioned in the introduction, Cricket uses the memory management primitives of the Mach operating system [Acce86] to provide the abstraction of a shared, transactional, single-level store that can be directly accessed by user applications. Cricket runs as a user-level process and provides automatic page-level concurrency control by default. The implementors of the Bubba database system at MCC [Bora90, Cope90] also built a storage system based on a single-level store. In Bubba, the kernel of an AT&T UNIX System V was modified to provide a single-level store with automatic page-level locking.

Although Cricket borrows a number of ideas from Bubba, several differences distinguish it from the approach taken in Bubba. One of the key differences is that Bubba's implementors had to modify the operating system kernel, since they did not have the luxury of using Mach. Unfortunately, this led to problems with portability. Furthermore, Bubba's recovery algorithms relied on battery-backed RAM, which again led to problems with portability. Finally, the focus in Bubba was on building a highly parallel database system, whereas the focus in Cricket is to examine the feasibility of using a single-level store in non-traditional application areas. Although the implementors of Bubba claimed that their storage system was a success [Bora90], they provided no detailed performance results to back their claim. In contrast, a large portion of the chapter on Cricket is devoted to a performance study comparing the Cricket prototype to the EXODUS Storage Manager [Care89].

2.3.2. IBM's 801 Prototype

The storage system in IBM's 801 prototype hardware architecture, which was described in [Chan88], is another project related to Cricket. In the 801 prototype, the operating system essentially provided mapped files with automatic concurrency control and recovery. Special hardware was added for both locking and logging. While this is an interesting approach, it is generally viewed as being too inflexible [Lind88]. In particular, no support was given for anything other than two-phase locking and value-based logging, which causes problems for indexes and other meta-data structures where two-phase locking is often inappropriate. In fact, a simulation study by Kumar

[Kuma87] compared the approach taken in the 801 to to a more conventional database storage system. The results of that simulation study showed that the 801 approach would result in poor performance on transaction processing workloads, mainly as a result of using two-phase locking on index pages and value-based logging on index updates. Another problem with the 801 prototype was that it could not support distributed computing environments. It also required special hardware support, which clearly causes problems with portability.

2.3.3. Camelot

The Camelot Distributed Transaction System [Spec88] is yet another project related to Cricket. Like Cricket, Camelot used the memory management primitives of Mach to provide a single-level store [Eppi89]. In contrast to Cricket, however, the single-level store that Camelot provided was not meant to be directly accessed by client applications. Instead, it was intended to be accessed only within a "data server" that stores and encapsulates all the persistent data and meta-data managed by that server. Camelot also provided fairly conventional locking and recovery services that must be explicitly invoked by the implementor of a data server. The only comprehensive study of Camelot's performance [Duch89] showed it to perform rather poorly on transaction processing workloads.

CHAPTER 3

FIELD REPLICATION

As mentioned in the introduction of this thesis, many of the data models that have been proposed recently include the notion of reference attributes or object-valued attributes. In this chapter, we describe how such attributes can be used to specify data replication. The motivation for replicating data in this case is to speedup query processing. Data values that would normally be accessed through a functional join are replicated so the join can be eliminated or at least made less costly. We refer to our technique for replicating data as *field replication* because it allows individual data fields to be selectively replicated. In this chapter, we describe how field replication is specified at the data model level and we present storage-level mechanisms to efficiently support it. An analytical cost model is also developed to examine how beneficial field replication can be and the circumstances under which it breaks down.

The remainder of this chapter provides a detailed description of field replication. Although much of the discussion is based on the EXTRA data model of EXODUS [Care88], the ideas presented here can be extended to other data models that support reference attributes or referential integrity facilities of the type discussed in [Date87]. The rest of the chapter is organized as follows: In Section 3.1 we present a simple employee database that is used in examples throughout the chapter. Using the employee database, Section 3.2 provides several examples where replication is useful. Section 3.3 then describes the first of our two basic replication strategies, in-place replication, while Section 3.4 describes the second of our two basic replication strategies, separate replication. This is followed by Section 3.5, where an analytical cost model is developed and used to compare no replication, in-place replication, and separate replication. Finally, conclusions and directions for future work are presented in Section 3.6.

3.1. EMPLOYEE DATABASE EXAMPLE

In order to make the discussion that follows concrete, we will often refer to the employee database pictured in Figure 3.1. As shown, the database is modeling the organization of a company. Each *organization* is made up of one or more *departments*, and each department is made up of one or more *employees*. The database is obviously too simple to be considered realistic, but it will serve the purposes of this discussion. The schema of our database has been described in the syntax of the EXTRA data model [Care88]. We will not assume that the reader is intimately

familiar with the EXTRA data model. We will assume, however, that the reader is familiar with the notion of reference attributes [Zani83].

In the schema, three types have been defined: the type ORG, which defines the structure of organization objects, the type DEPT, which defines the structure of department objects, and the type EMP, which defines the structure of employee objects. Note that ORG, DEPT, and EMP have been capitalized to distinguish them as type definitions. Using the type definitions, four sets have been created: the set *Org*, the set *Dept*, and the sets *Empl* and *Emp2*. Two sets of employees have been included for reasons that will become clear later. In the EXTRA data model, the notation **own ref** indicates ownership or an existence dependency. Therefore, if *Empl* is deleted, all the EMP objects in *Empl* are also deleted. Note, however, that the DEPT objects referenced by *Empl* via the reference attribute *dept* are not deleted when *Empl* is deleted.

3.1.1. The Physical Representation of Sets and Objects

Before continuing, it is important to clearly state our assumptions about the way in which sets and objects are represented on disk. Our assumptions are relatively straightforward and coincide with what is being done in the EXODUS project [Care89]. First, we will assume that top-level (or named) sets are stored as disk files. Second, we will assume that objects with a simple, unnested structure are stored as single, contiguous objects on disk. Third, we will assume that every object contains a *type-tag*, which identifies the object's type. Finally, we will assume that OIDs, which act like pointers, are used to implement reference attributes. In our employee database, for example, the set *Empl* would be stored as a disk file, and the pages in that disk file would contain only the EMP objects belonging to *Empl*. Each object E in *Empl* would have its type-tag set to EMP and would be stored as a single,

---

define type ORG (name: char[ ], budget: int)
define type DEPT (name: char[ ], budget: int, org: **ref** ORG)
define type EMP (name: char[ ], age: int, salary: int, dept: **ref** DEPT)

create Org: {**own ref** ORG}
create Dept: {**own ref** DEPT}
create Empl: {**own ref** EMP}
create Emp2: {**own ref** EMP}

Figure 3.1: The Employee Database

contiguous object. The values of E's attributes would be stored in E itself — not as separate objects — and the reference attribute *E.dept* would contain the OID of a DEPT object.

Note that our assumptions do not say anything about the physical representation of more complicated objects, such as objects with nested sets. The physical representation of such objects is likely to be less straightforward. We sidestep these issues in this thesis by considering only objects that have a simple physical representation.

3.2. INTRODUCTION TO FIELD REPLICATION

3.2.1. A Simple Example

Although functional joins are generally much cheaper than value-based joins, they can still be expensive to execute, even when physical OIDs are used to implement reference attributes. For example, if E is an EMP object, then dereferencing *E.dept.name* will generally cause two I/Os, one to retrieve E and another to retrieve the DEPT object that is referenced by E. With field replication, the second I/O can often be eliminated. As its name implies, this is accomplished by replicating data. If it is known that a particular reference path will be frequently accessed, then the data at the end of that path is replicated so that a separate I/O does not have to be performed to retrieve it. For example, the following item could be added to our schema:

replicate Empl.dept.name

Here, replication is being specified along the reference path *Empl.dept.name*. The replicate statement specifies that the values for *dept.name* will be replicated in objects belonging to *Empl*. In other words, objects in *Empl* can be thought of as having a "hidden" field in which a replicated value for *dept.name* is stored[1]. By hidden, we mean that the replicated value will not be visible to users at the query language level for either updates or retrievals. Query processing, on the other hand, *will* know about field replication and will exploit it whenever possible to avoid functional joins. For example, consider the following query, which retrieves the name, salary, and department of each employee who makes more than $100,000:

retrieve (Empl.name, Empl.salary, Empl.dept.name)
where Empl.salary > 100000

[1]This is slightly misleading because later on we will describe a replication strategy in which values for dept.name are stored in separate objects. For now, however, it is easiest to think of objects in the set *Empl* as having a hidden *dept.name* field.

With *Empl.dept.name* replicated as above, the query can be executed without performing a functional join. Clearly, field replication will help speed up queries that would otherwise require a functional join. The only question is at what cost. First, there is the extra disk space that is required to store replicated values, and second, there is the cost of propagating updates to replicated values.

As far as disk space goes, we assume that the speed-up in query processing is considered worth the extra space. With the cost of disk space decreasing, and with the gap between disk speed and memory speed growing all the time, this is probably a reasonable assumption to make. Propagating updates presents more of a problem. If there are a large number of replicated values, and updates to replicated values are frequent, then obviously, field replication will not work. Propagating updates would be so time consuming that query processing as a whole would slow down. Our assumption here is that the database administrator (DBA) who defines the data model is knowledgeable enough to realize that replication should only be specified on reference paths that are frequently accessed and, at the same time, infrequently updated. Under these circumstances, field replication will generally be beneficial. In many respects, the decision making that goes into deciding whether to replicate a given field is similar to the decision making that goes into deciding whether to build an index on a given field. Later on, an analytical cost model will be developed to give some feel for the circumstances under which field replication is beneficial and when it breaks down.

3.2.2. Field Replication is Associated with Instance

One thing to notice about the example given earlier (where *Empl.dept.name* was replicated) is that field replication is associated with instance (the set *Empl*) rather than type (the type EMP). In general, associating field replication with instance provides more modeling power.[2] For example, it allows one to replicate *Empl.dept.name* and, at the same time, *not* replicate *Emp2.dept.name*. This would be impossible if replication was associated with type. Associating field replication with instance rather than type also allows us to avoid certain issues related to type inheritance, such as whether field replication should be an inherited property. In this thesis, we will assume that field replication is associated *only* with instance and *not* with type.

---

[2] Note that this is only an issue in data models that separate type definition from type instantiation.

3.2.3. More Examples of Field Replication

3.2.3.1. Full Object Replication

In addition to allowing individual fields to be selectively replicated, field replication can also be used to specify full object replication. For example, a replication path could be defined on *Empl.dept.all*. This would cause all the information about an employee's department to be replicated and would allow any information about an employee's department to be obtained without a functional join.

3.2.3.2. Field Replication on N-Level Reference Paths

Until now, all our examples have involved replication on *1-level paths*; that is, reference paths that require only one functional join. One of the important uses of field replication is in reference paths of two or more levels because it allows more than one functional join to be eliminated. For example, a replication path could be defined on *Empl.dept.org.name*, which is an example of *2-level replication* because it specifies replication on a 2-level path.

3.2.3.3. Using Field Replication to Collapse N-Level Paths

Another use of field replication is in collapsing n-level paths to $n - 1$ levels or less. For example, a replication path could be defined on *Empl.dept.org*. This effectively allows any information about an employee's organization to be obtained with just one functional join rather than two. The 2-level path from objects in *Empl* to objects in *Org* has been effectively collapsed into a 1-level path.

Of course, the same thing can be accomplished without replication by adding an *org* field to the type definition for EMP, but this could lead to problems with referential integrity. For example, if the *org* field of a DEPT object D is changed from X to Y, then all the EMP objects that reference D would have to have their *org* field explicitly changed from X to Y. In contrast, with replication, the required changes would take place automatically, and referential integrity would never be violated.

3.2.3.4. Indexing on an N-Level Path

Our final example shows how field replication can be used in indexing. There is basically no reason why an index cannot be built on replicated data, and by allowing indexes to be built on replicated data, new indexing opportunities are created. For example, suppose we had:

replicate Emp1.dept.org.name
build btree on Emp1.dept.org.name

Because of replication, an index for the path *Emp1.dept.org.name* can be built on the replicated values that are stored in *Emp1*. The index would map organization names directly to objects in *Emp1*, and could efficiently support queries that require an associative lookup on the path *Emp1.dept.org.name*.

3.3. THE IN-PLACE REPLICATION STRATEGY

This section describes the first of our two basic field replication strategies, namely, *in-place replication*. In-place replication is analogous to the *cache-in-tuple* strategy of POSTGRES [Ston87, Jhin88], which was briefly described in the chapter on related work. In-place replication gets its name from the fact that replicated values are stored directly in the objects that cause replication to take place. For example, if a replication path was defined on *Emp1.dept.name*, then an extra field would be added to the objects in *Emp1* for storing the replicated value *dept.name*. Of course, this means that structural changes will have to be made to the objects in *Emp1*, but such changes are easily handled through subtyping. For the remainder of this chapter, it can be assumed that all the structural changes that are required by replication are handled through subtyping.

3.3.1. Propagating Updates in In-Place Replication

Replicated values are kept consistent using what are referred to as an *inverted path*. To demonstrate how inverted paths are used, suppose a replication path was defined on *Emp1.dept.name*. In order to keep replicated values consistent, we create the inverted path $Emp1.dept^{-1}$, which maps DEPT objects to the objects in *Emp1* that reference them. If the *name* field in a DEPT object D is updated, the inverted path $Emp1.dept^{-1}$ is traversed to propagate that update to the objects in *Emp1* that reference D. Note that the inverted path for *Emp1.dept.name* is $Emp1.dept^{-1}$ rather than $Emp1.dept.name^{-1}$. This reflects the fact that in our physical representation of objects, the *name* field of a DEPT object is stored in the object itself, not as a separate object. The *name* field does not play a part in the mapping of DEPT objects to *Emp1* objects, so it can be dropped in the inverted path. It does play a part in determining what updates need to be propagated, but that issue will be addressed later.

Figure 3.2 illustrates what the inverted path $Emp1.dept^{-1}$ looks like. Basically, inverted paths are broken down into a series of *links*. For example, the inverted path $P_1.P_2.P_3 \cdots P_n^{-1}$ would be broken down into the links $P_1.P_2^{-1}$, $P_2.P_3^{-1}$, $P_3.P_4^{-1}$, etc. *Link objects*, which implement an inverse mapping, are then created for each link in an

inverted path. Strung together end-to-end, these link objects form the inverted path. In our example, the link objects for the *Dept* set map each DEPT object D to the objects in *Empl* that reference D. Collectively, the link objects for *Dept* implement the link $Empl.dept^{-1}$.

As Figure 3.2 suggests, each link object contains little more than a collection of OIDs. The OIDs that appear in a link object are kept in sorted order so that, if necessary, a particular OID can be found and deleted using a binary search. Keeping OIDs in sorted order also allows us to propagate updates in clustered order if OIDs are physically based, as they are in EXODUS.

Before continuing, several comments about Figure 3.2 are in order. First, it is important to notice that the link objects for *Dept* have been stored in a separate set. In general, each link object can contain a large number of OIDs, and can be quite large as a result. Consequently, the link objects are stored in a separate set so that the clustering of objects in *Dept* is not disrupted. Second, notice that the link objects for *Dept* are stored in the same physical order as the objects in *Dept* which reference them. This is important because in propagating updates it is desirable to access link objects in clustered order to ensure that as little I/O as possible is generated. Third, notice that only two objects in *Dept* have link objects, namely, D1 and D2. This is due to the fact that only D1 and D2 are actually referenced by *Empl*. If *D3.name* is updated, that update does not have to be propagated. The way we determine when an update needs to be propagated will be discussed shortly. Finally, it should be noted that each object D in *Dept* can lie on more than one replication path, although this is not shown in Figure 3.2. If that were the case, then more

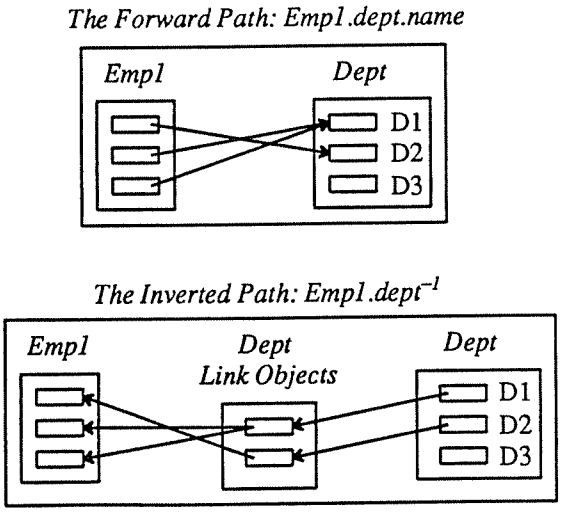


Figure 3.2: The Inverted Path $Empl.dept^{-1}$

than one link object would be generated for D. The way multiple paths are handled will also be discussed shortly.

3.3.2. Maintenance of a 1-Level Path

Once it is created, maintenance of an inverted path is straightforward. Continuing our example, let E be an object in *Empl* and let *E.dept* = *D*. The operations that affect the inverted path are: insert E (inserting E into *Empl*), delete E (deleting E from *Empl*), and update *E.dept* (updating the reference attribute *E.dept*). The following paragraphs summarize the modifications to the inverted path that take place as a result of these operations:

insert E

A link object is created for D if there is none. E's OID is then added to D's link object, after which *E.dept.name* is retrieved and stored in E.

delete E

E's OID is deleted from D's link object. If there are no longer any OIDs in the link object, it is deleted.

update E.dept:

The actions under *delete E* are executed with D set to the old value of *E.dept*, and then the actions under *insert E* are executed with D set to the new value of *E.dept*.

Note that deleting D does not affect the inverted path. The assumption here is that D can be deleted only when it is not referenced by any object in *Empl*. This implies that D cannot be part of the inverted path when it is deleted, and therefore its deletion cannot affect the inverted path.

3.3.3. Handling N-Level Replication Paths

Replication paths with two or more levels are handled in much the same way that 1-level paths are handled. It is mostly a matter of adding more links to the inverted path. For example, suppose a 2-level replication path was defined on *Empl.dept.org.name*. Figure 3.3 illustrates what the inverted path $Empl.dept.org^{-1}$ looks like. As shown, the inverted path has been broken down into two links, $Empl.dept^{-1}$ and $dept.org^{-1}$. The link objects for *Dept* implement $Empl.dept^{-1}$, while the link objects for *Org* implement $dept.org^{-1}$. From the figure, it should be clear how updates are propagated. If the *name* field of an ORG object O is updated, then $dept.org^{-1}$ and $Empl.dept^{-1}$ are traversed to propagate that update to the objects in *Empl* that reference O.

Maintenance of an inverted path with *n* levels is mostly just an extension of maintenance on a 1-level inverted path. The main difference is that the effects of insertions and deletions can ripple through *n* levels of the inverted

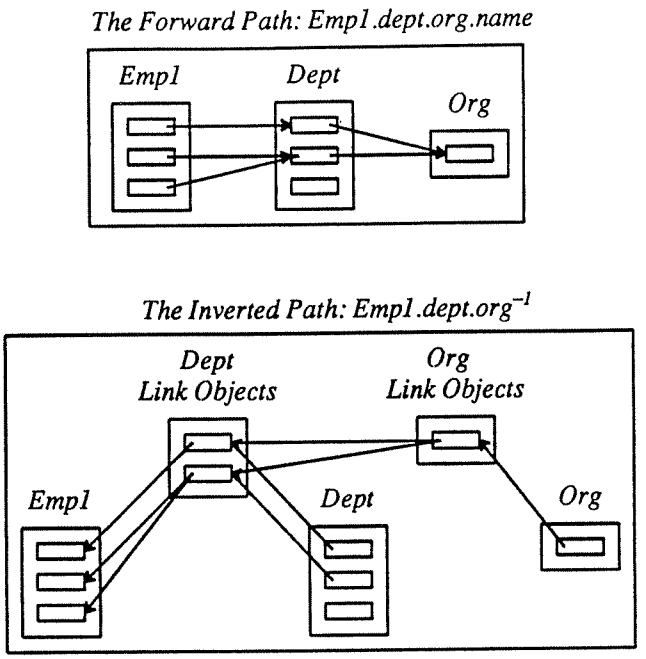


Figure 3.3: The Inverted Path *Empl.dept.org*$^{-1}$

path rather than just one. For example, suppose that we have the replication path defined above and that E is an object in *Empl* such that *E.dept* = *D*, and *D.org* = *O*. If E is inserted into *Empl*, then a link object object may have to be created for not just D, but O, too. Also, if E is deleted from *Empl*, then both D's link object and O's link object may end up being deleted if they become empty. The only other real difference is that updates to reference attributes of intermediate objects such as D can cause more than one update to be propagated. For example, if *D.org* is changed from O to X, then *X.name* will have to replace *O.name* in all of the objects in *Empl* that reference D.

As the above paragraphs demonstrate, maintenance of an inverted path can be costly. In general, though, replication is intended to be used only on paths in which object-to-object references are fairly static. Consequently, the cost of maintaining an inverted path consists primarily of the one-time cost to build it.

3.3.4. Determining How and When to Propagate an Update

To determine how and when to propagate an update, something called *link identifiers* (link IDs) are stored in each object O on a replication path. The link ID(s) stored in O identify the link(s) of the replication path to which O belongs. As such, they can be used to determine which updates to O need to be propagated and also how to propagate those updates. The association between link IDs and links is obtained from *link sequences*, which are actu-

ally generated by the database system. A link sequence is essentially just a sequence of link IDs that identify the links in a replication path, as illustrated by the following example:

replicate Emp1.dept.org.name link sequence = (1,2)

As shown, the replication path Emp1.dept.org.name has been assigned the link sequence (1,2).[3] In this example, there are two links in the replication path, and consequently there are two link IDs in the path's link sequence. Link ID 1 corresponds to the link *Emp1.dept*, and link ID 2 corresponds to the link *dept.org*. The association between link IDs, links, and replication paths would presumably be stored in the system catalog.

Figure 3.4 illustrates how link IDs would be used in this example. (Note that link-OID denotes the OID of a link object.) The fact that link ID 2 appears in O indicates that O belongs to the replication path(s) that contain the digit 2 in their link sequence, which in this case is the replication path *Emp1.dept.org.name*. Link ID 2 also indicates that O lies at the end of the link associated with link ID 2, namely, *dept.org* of *Emp1.dept.org.name*. Based on this information, it is possible for the query optimizer to tell that updates to *O.name* need to be propagated and also how to propagate those updates. Similarly, the presence of link ID 1 in D indicates that updates to *D.org* need to be propagated.

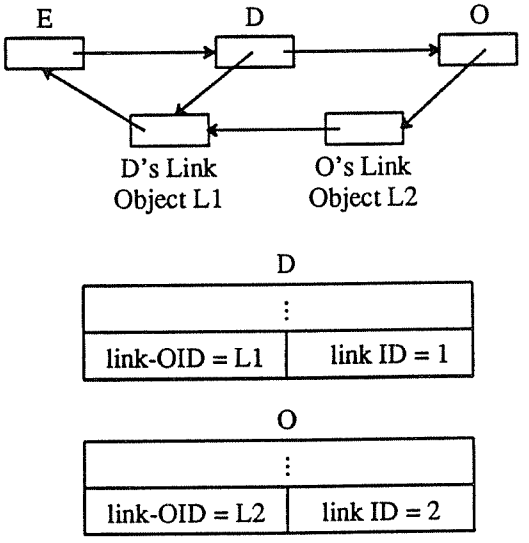


Figure 3.4: Determining when to Propagate an Update

[3] Note that the syntax "link sequence (1,2)" would *not* appear in the schema definition and has only been shown here for illustrative purposes.

In the above example, it is important to recognize that simple bit-masks identifying the fields whose updates need to be propagated cannot be used in place of link IDs. For updates to reference attributes such as *D.org*, more information is needed than a bit-mask can provide, and that is why link IDs are used. For example, if *D.org* is changed, then in order to propagate updates correctly and to carry out the necessary changes to the inverted path $Empl.dept^{-1}$, it is necessary to know that D appears in the replication path *Empl.dept.org.name* and also that D lies at the end of the first link in that replication path. The link ID stored in D provides this information, whereas a simple bit-mask would not provide enough information.

3.3.5. Handling Multiple Paths

To handle multiple replication paths, two cases need to be considered. The first case that needs to be considered is when replication paths share a common prefix. For example, suppose we had:

replicate Empl.dept.budget	link sequence = (1)
replicate Empl.dept.name	link sequence = (1)
replicate Empl.dept.org.name	link sequence = (1,2)

In this example, the fact that each path emanates from *Empl* means that the mapping defined by *Empl.dept* (and $Empl.dept^{-1}$) is the same in each path. As a result, link ID 1 appears in all three link sequences, indicating that not only is the first link of each path the same, but also that each path can share the link $Empl.dept^{-1}$. In terms of updates, the presence of link ID 1 in a DEPT object D indicates that D lies on all three replication paths, and therefore if either *D.budget*, *D.name*, or *D.org* is updated, that update has to be propagated. In general, if two replication paths A and B share a common prefix $P_1.P_2.P_3 \cdots P_n$, then A and B would share the links $P_1.P_2^{-1}$, $P_2.P_3^{-1}$,..., and $P_{n-1}.P_n^{-1}$ in their inverted paths, and link IDs would be assigned in a manner that reflects this sharing.

The other case that needs to be considered is when replication paths do not share a common prefix. For example, suppose we have:

replicate Empl.dept.budget	link sequence = (1)
replicate Empl.dept.name	link sequence = (1)
replicate Empl.dept.org.name	link sequence = (1,2)
replicate Emp2.dept.org	link sequence = (3)

In this example, the first three paths share a common prefix, which is different from the fourth path's prefix. Therefore, a new link ID is generated for the fourth path to reflect the fact that its first link is different from the first link

of the other three paths and also to reflect the fact that the link $Emp2.dept^{-1}$ cannot be shared by the other three paths.

Figure 3.5 illustrates the way the above replication paths are handled. Based on the preceding discussion, most of Figure 3.5 should be clear. The key thing to observe about Figure 3.5 is that only one link object (L1) is used to propagate updates in the first three replication paths. L1 and $Emp1.dept^{-1}$ are one and the same in this example. Another thing to observe about Figure 3.5 is the way two link objects have been generated for D. L1 is needed to propagate updates in the first three replication paths, while L2 is needed to propagate updates in the fourth replication path.

Based on Figure 3.5, it should be clear how multiple replication paths are handled in the general case. It basically comes down to sharing links in inverted paths whenever possible and generating a link object for an object O whenever O enters a link in which it does not already appear.

3.3.6. The Space Overhead of In-Place Replication

One of the concerns with in-place replication is the amount of extra space that is required to support it. The very fact that a replication path has been created indicates that the extra space required to store replicated data can

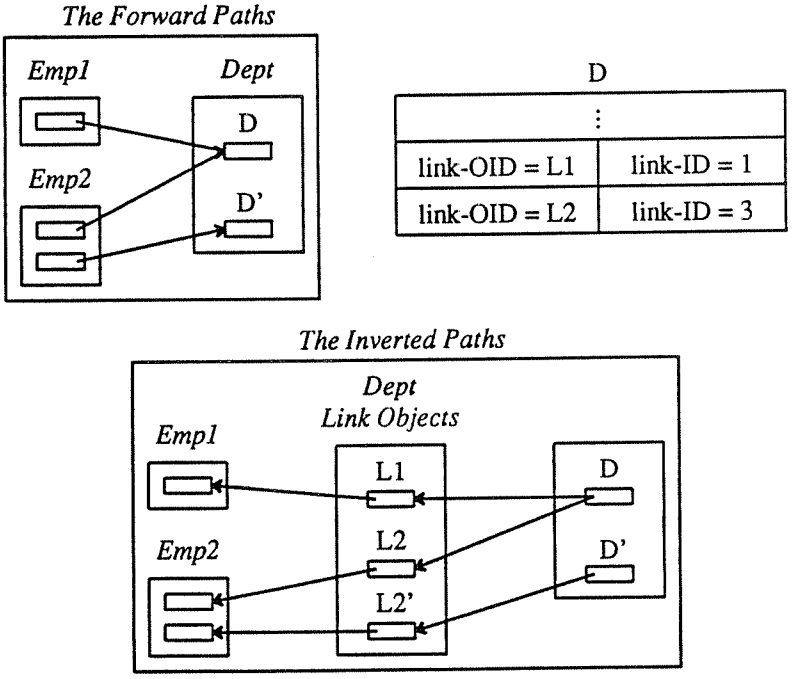


Figure 3.5: Determining When to Propagate an Update

be tolerated. However, for objects along the replication path (as opposed to the objects in which replicated values are stored), we still need to worry about the space required for (link–OID, link–ID) pairs.

Fortunately, there are three mitigating factors that come into play here. First, the size of (link–OID, link–ID) pairs does not have to be large. This is mainly due to the fact that link IDs which are not in use can be re-used and can probably be kept as small as 8 bits. Second, in a typical environment, we would expect only a few replication paths per set (much like there are typically only a few indexes built per set), and therefore it is unlikely that an object will lie on several replication paths at once. Finally, it can be assumed that the DBA will make reasonable modeling decisions. Creating several replication paths on very small objects would obviously be a poor modeling decision unless the performance gains made the storage overhead worthwhile.

3.3.7. Optimizing Inverted Paths

Before moving on to our next replication strategy, it is important to note that our storage structures for inverted paths can be optimized in a number of ways. This section takes a brief look at some of the optimizations that are possible.

3.3.7.1. Eliminating Link Objects when Possible

One way our inverted path structure can be optimized is to eliminate link objects when they contain a small number of OIDs. For example, suppose a link object L has only the OID X stored in it. If that is the case, then L can be eliminated, and X can be stored directly in the object(s) that reference L. The result is that L would no longer need to be retrieved to propagate an update. The space required to store L's OID is the same as the space required to store X, so there is no reason not to make this optimization. In general, it may even be worthwhile to eliminate L if it has less than n OIDs stored it, where n is on the order of two or three, or perhaps even larger. Using physical schema information about object size, it should be possible to determine a reasonable value for n on a type-by-type basis.

3.3.7.2. Clustering Related Link Objects in N-Level Paths

Another way our inverted path structure can be optimized is to cluster related link objects in an n-level path. For example, suppose a 2-level replication path was defined on *Empl.dept.org.name*. Further, suppose that E is an object in *Empl*, $E.dept = D$, $D.org = O$, L_D is D's link object in this path, and L_O is O's link object in this path. The problem with our current scheme is that an update to *O.name* will require both L_O and L_D to be retrieved. Since L_O

and L_D are stored in different sets, two I/Os would result.

One way to avoid two I/Os is to cluster L_O and L_D together on the same page, and in general, the same idea could be applied to replication paths with three or more levels. The only drawback of this scheme is that clustering goals can conflict. For example, supposed the replication path *Empl.dept.name* was added. For this replication path, L_D should be clustered with the other link objects that make up $Empl.dept^{-1}$ so that updates to department names can be propagated with a minimal amount of I/O. Unfortunately, this means that clustering L_D with L_O would conflict with the clustering of $Empl.dept^{-1}$. The way such conflicts might be resolved is left for future study.

3.3.7.3. Collapsing N-Level Inverted Paths to 1-Level Inverted Paths

The final optimization considered is collapsing inverted paths with two or more levels into inverted paths with just 1-level. To demonstrate how this is done, suppose we take our previous example, where *Empl.dept.org.name* is replicated. To collapse the inverted path, the links $Empl.dept^{-1}$ and $dept.org^{-1}$ would essentially be combined to form the collapsed link $Empl.org^{-1}$.

Figure 3.6 illustrates the way this would work. In the collapsed version of the inverted path, updates to O can be propagated directly to *Empl* via the link $Empl.org^{-1}$. It is important to note that the OIDs for E1, E2, and E3 that appear in O's link object would have to be tagged in some way to indicate their association with D. The tags would be needed to handle updates to *D.org*. For example, if *D.org* is set to some other object in *Org*, say X, then the OIDs of E1, E2, and E3 will have to be moved from O's link object to X's link object. Without the tags, there would be no way to know that the OIDs of E1, E2, and E3 need to be moved.

Although it would appear that it is always a good idea to collapse inverted paths, there are two reasons why this is not necessarily so. First, a collapsed path is more costly to maintain. For example, when *D.org* is updated, the OIDs of E1, E2, and E3 have to be moved. In contrast, in the uncollapsed version, only the OID of D would have to be moved. Second, collapsed paths prohibit the sharing of some links. For example, in the collapsed version, the link object of D cannot be shared due to the fact that it no longer exists. Even with these problems, though, collapsed paths may still prove useful in many situations, particularly when reference paths are static and when there is little opportunity for sharing links.

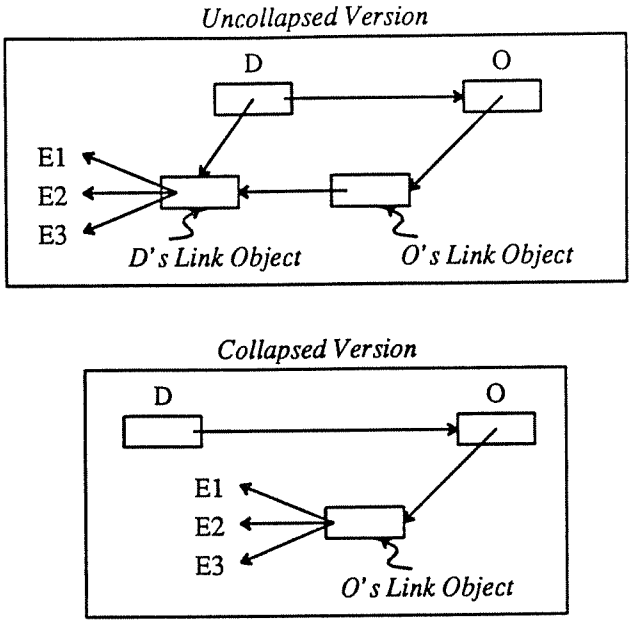


Figure 3.6: Collapsing a 2-Level Inverted Path

3.4. THE SEPARATE REPLICATION STRATEGY

In situations where sharing is heavy and where there is a moderate to high probability that replicated data will be updated, in-place replication is likely to perform poorly. For example, suppose replication paths were defined on *Empl.dept.name* and *Empl.dept.budget*, and suppose that departments tend to be large, consisting of, say, a thousand employees. If the name or budget of a DEPT object D is updated, then with in-place replication that update would have to be propagated to every object in *Empl* that references D. Clearly, if *D.name* or *D.budget* is updated with even moderate frequency, the cost of propagating updates will be so great that field replication would provide little benefit, if any.

Separate replication eliminates this problem by reducing the cost of propagating updates. This is achieved by storing replicated values in separate objects which are shared within a set. Separate replication is analogous to the *cache separately* strategy of POSTGRES [Jhin88], which was described in the chapter on related work. Figure 3.7 illustrates the way separate replication would work for our example. As shown, the replicated values for *Empl.dept.name* and *Empl.dept.budget* are stored in separate objects, which are shared. (Note that the replicated values for *D1.name* and *D1.budget* are stored together in *one* object, and similarly for D2.) Because replicated values are shared, propagating updates is less costly. For example, if *D1.name* is updated, then that update only has

to be propagated to the object associated with *D1.name*. In contrast, with in-place replication, the update would have to be propagated to E1 and E2. The way updates are propagated with separate replication will be described shortly. In Figure 3.7, it is important to notice that the objects in which replicated data is stored are kept in the same order as the corresponding objects in *Dept*. This is done to ensure that propagating updates generates as little I/O as possible.

One thing that needs to be emphasized is that replicated values are *not* shared between sets with separate replication. For example, suppose replication paths were defined on *Emp1.dept.name* and *Emp2.dept.name*. In this case, two sets would be generated for replicated values. One set would be used to store the replicated values for *Emp1.dept.name*, while the other set would be used to store the replicated values for *Emp2.dept.name*; the two sets would not be shared. For reasons that will become clear shortly, it is desirable to cluster the replicated values of a particular set as tightly as possible. By maintaining separate sets, we ensure that the clustering of one set's replicated values does not interfere with the clustering of another set's replicated values.

Separate replication can also be used for replication paths with two or more levels. Replicated values are stored the same way in *n*-level paths that they are in 1-level paths, as illustrated in Figure 3.8. This will be discussed in more detail shortly.

3.4.1. Why Separate Replication will do Better than No Replication

At first glance, it may appear that separate replication provides no cost benefit for retrievals (at least in 1-level paths). After all, a functional join still has to be performed to access replicated values. While this may be true for single-object retrievals, for retrievals that access many objects in the same set (a set scan, for example) it should be possible to physically cluster replicated values so tightly that relatively few I/Os would be required to retrieve all replicated values, particularly if replicated values are small. As a result, for retrievals such as scans, separate

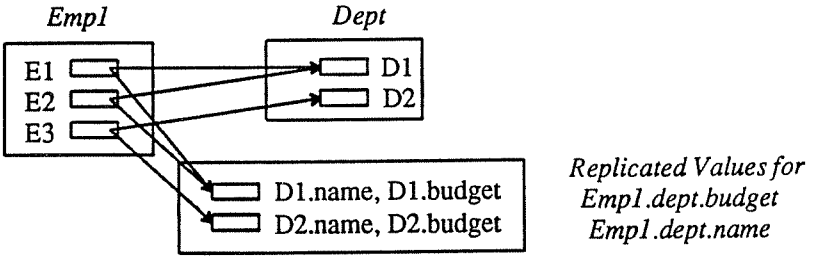


Figure 3.7: Separate Replication for *Emp1.dept.name*, *Emp1.dept.budget*

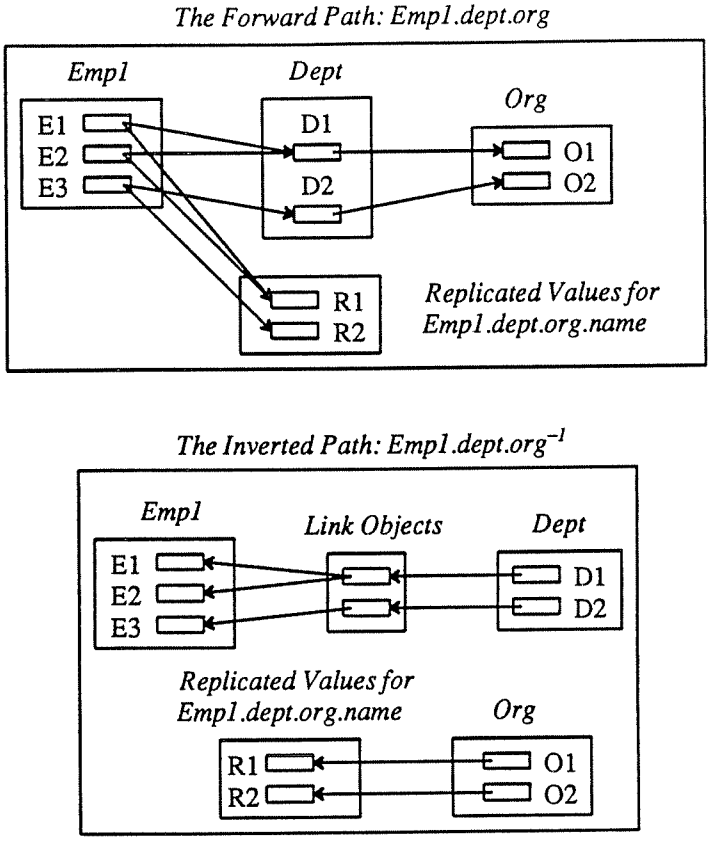


Figure 3.8: Inverted Paths in Separate Replication

replication should outperform no replication. Later in this chapter results are presented which suggest that this is indeed the case. Another situation where separate replication should outperform no replication is in *n*-level references. This follows because separate replication effectively reduces an *n*-level reference to a 1-level reference.

3.4.2. Propagating Updates in Separate Replication

With separate replication, updates to data fields (as opposed to reference attributes) are always handled in the same manner, regardless of whether a 1-level or an *n*-level replication path has been defined. This is illustrated in the right hand side of Figure 3.8. As shown, the objects in *Org* point directly to the objects containing replicated values (R1 and R2). Taking the object O1 as an example, updates to *O1.name* are propagated by simply retrieving the object R1 and updating it. Although it is not shown, O1 contains R1's OID, a reference count for R1, and a tag of some sort to indicate which fields have been replicated in R1 so we know which updates to propagate to R1.

When reference attributes such as *E1.dept* are updated, things get slightly more complicated. As Figure 3.8 indicates, for replication paths with two or more levels, an inverted path needs to be maintained. The inverted path

is the same in all respects as the one used with in-place replication, except that there is one less level. That is, with in-place replication, an *n*-level replication path requires an *n*-level inverted path to be maintained. In contrast, with separate replication, an *n*-level replication path requires an *(n-1)*-level inverted path to be maintained. The reason why an inverted path still needs to be kept is to propagate updates to reference attributes. In the replication path *Empl.dept.org.name*, for example, the link $Empl.dept^{-1}$ is needed to handle updates to the reference attribute *org*. If *D2.org* is changed from O2 to O1, then E3 must be updated so that it references R1, rather than R2. The link $Empl.dept^{-1}$ is needed to propagate the update to E3.

In general, everything that was said earlier about inverted paths in the section on in-place replication, including space issues, links IDs, and optimizations, applies to separate replication as well, so we will not discuss these matters further. The maintenance of an inverted path is the same in almost all respects. The only exception is that updates to reference attributes are handled somewhat differently with separate replication. Although the previous example only demonstrated how they are handled in a 2-level path, it should be clear how the general case is handled.

3.4.3. Supporting Both In-place and Separate Replication

Before leaving this section, it should be noted that in-place and separate replication can both be supported at the same time. The two strategies do not conflict with each other, and in fact, links can even be shared by the two strategies. This is viewed as important because there will undoubtedly be applications where a mix of in-place and separate replication provides the best performance.

3.5. COMPARING THE REPLICATION STRATEGIES

In this section, an analytical cost model is developed to compare no replication, in-place replication, and separate replication. Only queries with 1-level functional joins are considered. The cost model is based on the following schema:

```
define type RTYPE  (sref: ref STYPE, ...)
define type STYPE  (repfield: SCALAR-TYPE, ...)

create R:  {own ref RTYPE}
create S:  {own ref STYPE}
replicate  R.sref.repfield
```

As shown, there are two sets in the model, R and S, with objects in R referencing objects in S. For both in-place and

separate replication, a replication path has been defined on *R.sref.repfield*, where *repfield* is a scalar field that appears in members of S. Two types of queries are considered in the cost model:

Read Query:
retrieve (R.fields, R.sref.repfield)
where... some clause on a scalar field $R.field_r$

Update Query:
replace (S.fields = newvalues, S.repfield = newvalue)
where... some clause on a scalar field $S.field_s$

Read queries read data from R and also along the path *R.sref.repfield*, while update queries modify data in S, including the replicated field, *repfield*. Thus, read queries model those queries that read replicated data, while update queries model those queries that update replicated data. Note that the clause in read queries is on the scalar field $field_r$. We will assume that this field is indexed by a B+ tree, and likewise for the field $field_s$, which appears in update queries.

In order to compare replication strategies, an expected I/O cost function, C_{total}, is computed for each strategy. Let:

C_{read} = the expected I/O cost of processing a read query
C_{update} = the expected I/O cost of processing update query
P_{update} = the probability that an update query will be executed

For a given query mix, C_{total} is then defined as:

$$C_{total} = (1 - P_{update}) \cdot C_{read} + P_{update} \cdot C_{update}$$

In the analysis that follows, P_{update} is varied from 0 to 1, and the resulting values for C_{total} are used to compare the different replication strategies. Note that C_{total} does not include any CPU costs. It turns out that I/O costs dominate the small joins that shall be considered here. Therefore, including CPU costs in the cost model would not have significantly changed the relative results that were obtained — it would have only added unnecessary complexity.

3.5.1. File Structures in the Model

Figure 3.9 shows the file structures used in the cost model. Each object set is assumed to be stored as a single disk file. As shown, the number of files differs with each replication strategy. With no replication, there are only the files for storing R and S, while with in-place replication, there is R, S, and the file L, which is used to store link

objects for the inverted path $R.sref^{-1}$. With separate replication, there is R, S, and the file T, which is used to store replicated values for the path *R.sref.repfield.* The arcs in the figure denote the reference structure that exists between files, while the triangular shapes alongside R and S denote the B+ trees on $field_r$ and $field_s$, respectively. Note that no output file is included in our model. We have excluded output to keep things as simple as possible, and also because the result of a join often forms the input of another database operation without ever being completely written to disk.

One thing to note about Figure 3.9 is that the the objects in L and T are stored in the same order as the objects in S which reference them. By ordering L and T this way, updates are less expensive to propagate, as explained in the sections that introduced in-place and separate replication.

3.5.2. Assumptions in the Model

To make the analysis tractable, several assumptions are required in the cost model. The first and most important assumption we make is that R and S are *relatively unclustered*; that is, we assume that objects in R are *not* ordered by their references to S. (With separate replication, T is kept in the same order as S, so R and T are also assumed to be relatively unclustered.) This assumption is made because we feel it represents the most typical case. Objects in R would typically be ordered by the value of some data field, not by their references to S. Bear in mind that this is a key assumption and has a considerable impact on the analysis. Although the analysis is not carried out here, it should be clear that replication will be less beneficial when R and S are relatively clustered. This is because

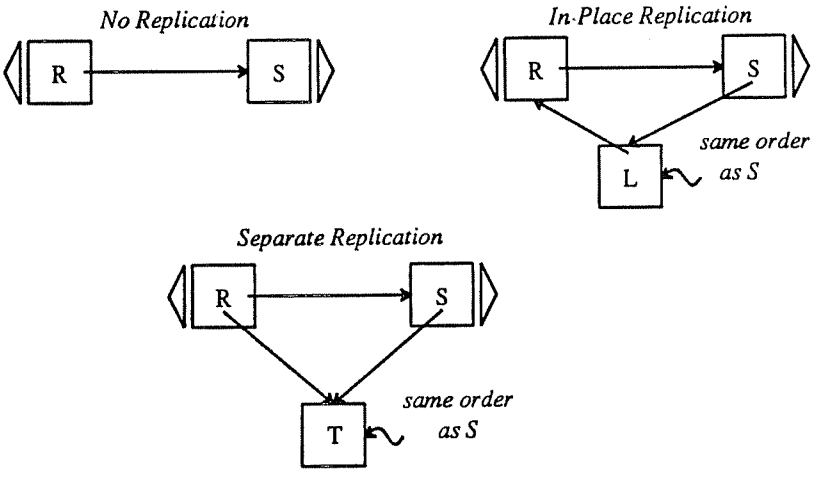


Figure 3.9: The File Structure with Each Replication Strategy

replication works by lowering or eliminating the cost of performing the functional join between R and S. When R and S are relatively clustered, the cost of performing the functional join will be lower, and replication cannot offer as much savings.

The next assumption we make is that functional joins are always performed in an optimal way in the sense that if a page is required in a functional join, then that page will be read only once in performing the join. This assumption allows us to ignore buffering and the exact details of how an efficient join algorithm might operate. The next chapter of this thesis examines those issues in more detail. For the small joins that shall be considered here, it turns out that buffering requirements are small enough to be a non-issue.

The last assumption we make is that read and update queries always access R and S through the indexes on $field_r$ and $field_s$, respectively. This assumption is made because we feel that it most accurately models the "typical" database query, as most queries will make use of some index.

3.5.3. The Parameters of the Cost Model

The parameters of the cost model are listed in Table 3.1. Although there are a large number of parameters, only a few of them are actually varied here. Moreover, most of the parameters are not really parameters per se, but rather functions of a small set of "core" parameters, which consist of the parameters in the top half of the first table. Default values for the core parameters are listed in Table 3.2.

The meaning of most parameters should be clear from Table 3.1. The parameters k, sel_r and sel_s require further explanation, however. The parameter k denotes the average sharing level of objects in S. In the cost model, it is assumed that each object in S is, on the average, referenced or shared by k objects in R. The parameters sel_r and sel_s denote the selectivity of read and update queries, respectively. Each read query reads $sel_r \cdot |R|$ objects in R, and each update query updates $sel_s \cdot |S|$ objects in S.

One thing to note about the parameter values is that the values for r and s represent the size of objects in R and S with no replication. Consequently, with in-place or separate replication, r and s need to be adjusted. For example, with in-place replication, r must be increased by h to account for the replicated data. Rather than introduce more notation, the cost equations that follow will tacitly assume that r and s (and the parameters that depend on r and s) reflect these adjustments.

Parameter	Definition
P	size of a disk page
b	average B+ tree fanout
$\|R\|$	number of objects in R
k	average sharing level
sel_r	selectivity of read queries
sel_s	selectivity of update queries
r	size of objects in R (varies with strategy)
s	size of objects in S (varies with strategy)
h	size of replicated field, *repfield*
$size(OID)$	size of OIDs
$size(link\text{-}ID)$	size of link IDs
$size(type\text{-}tag)$	size of type-tags
$\|S\|$	number of objects in S: $\|S\| = \|R\| / k$
t	size of objects in T: $t = h + size(type\text{-}tag)$
l	size of objects in L: $l = 1 + size(type\text{-}tag) + k \cdot size(OID)$
O_r	objects per page in R: $O_r = \lfloor P / r \rfloor$
O_s	objects per page in S: $O_s = \lfloor P / s \rfloor$
O_t	objects per page in T: $O_t = \lfloor P / t \rfloor$
O_l	objects per page in L: $O_l = \lfloor P / l \rfloor$
P_r	pages in R: $P_r = \lceil \|R\| / O_r \rceil$
P_s	pages in S: $P_s = \lceil \|S\| / O_s \rceil$
P_t	pages in T: $P_t = \lceil \|S\| / O_t \rceil$
P_l	pages in L: $P_l = \lceil \|S\| / O_l \rceil$

Table 3.1: The Parameters of the Cost Model

3.5.4. The Setting for the Analysis

The model that has been described will be analyzed in two settings. In the first setting, we will assume that the B+ trees on $field_r$ and $field_s$ are both unclustered indexes. In the second setting, we will assume that both indexes are clustered indexes. The reason for considering these particular settings is because they represent opposite ends of the performance spectrum. When both indexes are unclustered, more overall I/O is generated and, as a result, the savings in I/O due to replication will be smaller as a percentage of the total I/O. Conversely, when both indexes are clustered, less overall I/O will be generated and therefore the savings in I/O due to replication will be larger on a percentage basis. Note that, although it is not always stated as such, the analysis that follows is in terms of expected costs.

Defaults for Core Parameters	
P	4096 bytes
b	350
$\|R\|$	100,000
k	1 (varied)
sel_r	0.001 (varied)
sel_s	10 / $\|S\|$
r	100 bytes
s	200 bytes
h	20 bytes
size (*OID*)	8 bytes
size (*link–ID*)	1 byte
size (*type–tag*)	2 bytes

Table 3.2: The Defaults for the Core Parameters

3.5.5. Cost Analysis for Unclustered Indexes

In this section, cost equations are derived for read and update queries under the assumption that unclustered indexes are used. Throughout this section, C_{read} shall denote the expected cost of a read query and C_{update} shall denote the expected cost of an update query. For brevity, we let $|R'| = sel_r \cdot |R|$ and let $|S'| = sel_s \cdot |S|$

In the cost equations that follow, it should be noted that no distinction is made between sequential I/O and random I/O. We initially distinguished between the two, but found that it had little effect on the resulting graphs. This is due to the fact that our results are presented in terms of relative cost rather than absolute cost, and all the strategies that are examined receive roughly the same benefit from sequential I/O.

3.5.5.1. Read Queries with No Replication

In terms of I/O, processing a read query with no replication consists of reading the index on $field_r$, reading R, and reading S (to join R and S). Reading the index on $field_r$ consists of descending the B+ tree to a leaf, then scanning across the leaves to obtain pointers to the $sel_r \cdot |R|$ (i.e., $|R'|$) objects in R that satisfy the clause of the read query. For the purposes of this discussion, we can simply assume that the height of all indexes is always 2. Therefore the cost of reading the index on $field_r$ is:

$$C_{read\ index\ on\ R} = 2 + \lceil |R'| \ / \ b - 1 \rceil$$

To calculate the cost to read R, we first consider the probability that a given page P_i in R is *not* read by a read query. Since our access to R is unclustered, we can assume that any given subset of $|R'|$ objects is just as likely to be accessed by a read query as any other subset. Therefore, the probability that page P_i is not read is equal to the probability of choosing a subset of $|R'|$ objects from R such that the chosen subset contains no object from page P_i. This is equal to:

$$Prob(page\ P_i\ in\ R\ is\ not\ read) = \frac{\binom{|R| - O_r}{|R'|}}{\binom{|R|}{|R'|}}$$

The expected number of pages that are read in R is therefore:

$$expected\ pages\ read\ in\ R = P_r \cdot \left[1 - \frac{\binom{|R| - O_r}{|R'|}}{\binom{|R|}{|R'|}}\right]$$

This same quantity was derived in another context [Yao77]. If we let:

$$Y(u, v, w) = 1 - \frac{\binom{u - v}{w}}{\binom{u}{w}}$$

then the cost to read R is:

$$C_{read\ R} = P_r \cdot Y(|R|, O_r, |R'|)$$

The so-called Yao function $Y()$ has been introduced because it will be needed again shortly. To calculate the cost to read S, we use more or less the same technique that we used with R. In this case, however, the probability that a given page P_i in S is not read is equal to the probability of choosing a subset of $|R'|$ objects from R such that the chosen subset contains no object that references an object on page P_i. On the average, there are $k \cdot O_s$ objects in R that reference page P_i. Consequently, the probability that page P_i is not read is:

$$Prob(page\ P_i\ in\ S\ is\ not\ read) = \frac{\binom{|R| - k \cdot O_s}{|R'|}}{\binom{|R|}{|R'|}}$$

Therefore, the cost to read S is:

$$C_{read\ S} = P_s \cdot Y(|R|, k \cdot O_s, |R'|)$$

Summing it up, the net cost to process a read query with no replication is:

$$\begin{aligned} C_{read} &= C_{read\ index\ on\ R} + C_{read\ R} + C_{read\ S} \\ &= 2 + \lceil |R'| / b - 1 \rceil + P_r \cdot Y(|R|, O_r, |R'|) + P_s \cdot Y(|R|, k \cdot O_s, |R'|) \end{aligned}$$

3.5.5.2. Update Queries with No Replication

Processing an update query with no replication consists of reading the index on $field_s$ and updating S. (We are assuming that update queries do not cause the index on $field_s$ to be modified.) The cost equation for reading the index on $field_s$ is similar to the equation for $C_{read\ index\ on\ R}$. Updating S consists of reading pages in S, updating them, and then writing them back to disk. The expected number of pages read and written in S can be calculated in the same manner that was used to calculate the expected number of pages in R that are read by a read query. Based on this observation, the cost to update S is:

$$C_{update\ S} = 2 \cdot P_s \cdot Y(|S|, O_s, |S'|)$$

The net cost to process an update query with no replication is therefore:

$$\begin{aligned} C_{update} &= C_{read\ index\ on\ S} + C_{update\ S} \\ &= 2 + \lceil |S'| / b - 1 \rceil + 2 \cdot P_s \cdot Y(|S|, O_s, |S'|) \end{aligned}$$

3.5.5.3. Read Queries with In-Place Replication

Processing a read query with in-place replication consists of reading the index on $field_r$ and reading R. No join between R and S is required because *sref.repfield* is replicated in R. The cost equations for processing a read query

with in-place replication are basically the same[4] as with no replication except that the cost to read S is no longer included. Consequently, the net cost to process a read query with in-place replication is:

$$\begin{aligned} C_{read} &= C_{read\ index\ on\ R} + C_{read\ R} \\ &= 2 + \lceil |R'| \ / \ b - 1 \rceil + P_r \cdot Y(|R|, O_r, |R'|) \end{aligned}$$

3.5.5.4. Update Queries with In-Place Replication

Processing an update query with in-place replication consists of reading the index on $field_s$, updating S, reading L to propagate the updates in S to R, and updating R. The cost equations for reading the index on $field_s$ and for updating S are the same as with no replication. The cost equation for reading L can be derived in the same manner that $C_{read\ R}$ was derived with no replication.

The cost equation for updating R is similar to the equation for $C_{update\ S}$ with no replication. To derive the equation, first observe that, because of replicated values, each update to an object in S has to be propagated to an average of k objects in R. On the average, a total of $k \cdot |S'|$ objects in R are therefore modified by an update query. Since R and S are relatively unclustered, we can assume that any given subset of $k \cdot |S'|$ objects in R is just as likely to be updated as any other subset. Based on this observation, and following the reasoning that was used to derive $C_{update\ S}$ with no replication, the cost of updating R is:

$$C_{update\ R} = 2 \cdot P_r \cdot Y(|R|, O_r, k \cdot |S'|)$$

Adding in the other terms, the net cost to process an update query with in-place replication is therefore:

$$\begin{aligned} C_{update} &= C_{read\ index\ on\ S} + C_{update\ S} + C_{read\ L} + C_{update\ R} \\ &= 2 + \lceil |S'| \ / \ b - 1 \rceil + 2 \cdot P_s \cdot Y(|S|, O_s, |S'|) + P_l \cdot Y(|S|, O_l, |S'|) \\ &\quad + 2 \cdot P_r \cdot Y(|R|, O_r, k \cdot |S'|) \end{aligned}$$

3.5.5.5. Read Queries with Separate Replication

Processing a read query with separate replication is the same as with no replication except that R is joined with T rather than S. Consequently, the cost equations for separate replication are obtained by simply substituting T for S in the cost equations for no replication. Doing this, the net cost to process a read with separate replication is:

[4] In the equations, it is important to remember that the values for parameters such as O_r and O_s will differ from strategy to strategy, even though the same symbols are used. For example, O_r is smaller here than with no replication because of replicated data.

$$C_{read} = C_{read\ index\ on\ R} + C_{read\ R} + C_{read\ T}$$
$$= 2 + \lceil |R'| \ / \ b - 1 \rceil + P_r \cdot Y(|R|, O_r, |R'|) + P_t \cdot Y(|R|, k \cdot O_t, |R'|)$$

3.5.5.6. Update Queries with Separate Replication

Processing an update query with separate replication consists of reading the index on $field_s$, updating S, and updating T. The cost equations for reading the index on $field_s$ and for updating S are the same as the equations for $C_{read\ index\ on\ S}$ and $C_{update\ S}$ with no replication, respectively. The cost equation for updating T follows directly from the equation for $C_{update\ S}$. The net cost to process an update query with separate replication is therefore:

$$C_{update} = C_{read\ index\ on\ S} + C_{update\ S} + C_{update\ T}$$
$$= 2 + \lceil |S'| \ / \ b - 1 \rceil + 2 \cdot P_s \cdot Y(|S|, O_s, |S'|) + 2 \cdot P_t \cdot Y(|S|, O_t, |S'|)$$

3.5.6. Performance Results for Unclustered Indexes

Graphs 3.1-3.4 present the results for unclustered indexes. The graphs were obtained by computing C_{total}, which was described earlier, for each replication strategy while P_{update} was varied from 0 to 1. For in-place and separate replication, the values for C_{total} were compared to the corresponding values for C_{total} with no replication, and the percentage difference in C_{total} was plotted. The horizontal line in the graphs therefore represents no replication. Note that the vertical axes of the graphs were arbitrarily cutoff at 50% so that all the graphs could be placed on one page.

Each graph shown is for a different sharing level k. In all the graphs, $|R|$ was fixed at 100,000 objects, and an update query always updated 10 objects. The value of k was set at 1, 10, 20, and 50. Consequently, $|S|$ varied from 2,000 to 100,000 objects. In each graph, three lines have been drawn for both in-place and separate replication, corresponding to the read selectivity sel_r being set at 0.001, 0.002, and 0.005. The size of objects in R was fixed at 100 bytes, those in S at 200 bytes, and the replicated field at 20 bytes.

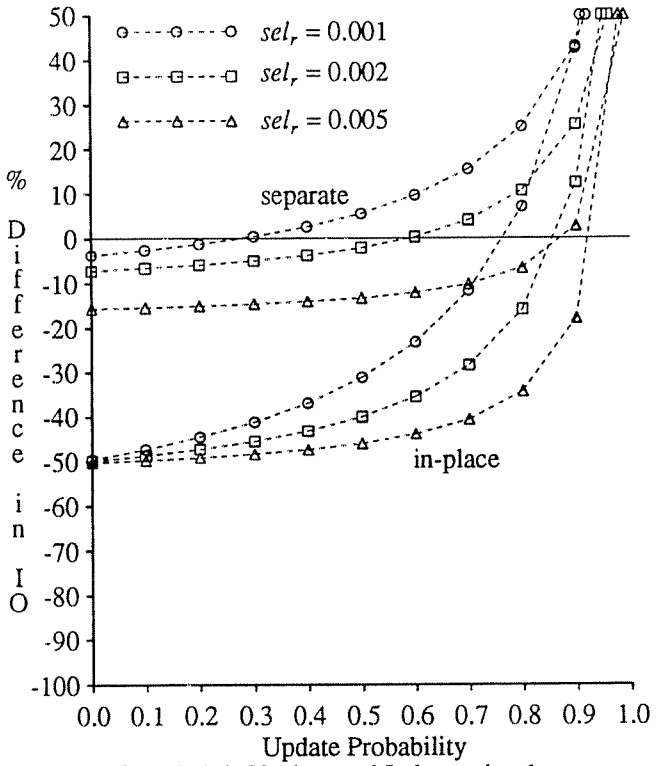
Looking at the graphs, it is clear that replication can be beneficial. As expected, replication is particularly useful when the probability of an update query is low. The graphs show that in-place replication performs its best for small update probabilities and for small values of k. Its performance decreases for large values of k because the cost to propagate updates is higher in that case. (Recall that with in-place replication, each update to an object in S has to be propagated to k objects in R.) In contrast to in-place replication, separate replication performs its best for large

values of k. This is because the size advantage provided by T in joins becomes more pronounced as k increases. By size advantage we are referring to the fact that there are more disk pages in S than there are in T, which means that it costs more to join R with S than it does to join R with T — especially if the join is large. The graphs also show that, compared to in-place replication, separate replication breaks down less rapidly as the update probability is increased. This is because the cost of propagating updates is much higher with in-place replication.

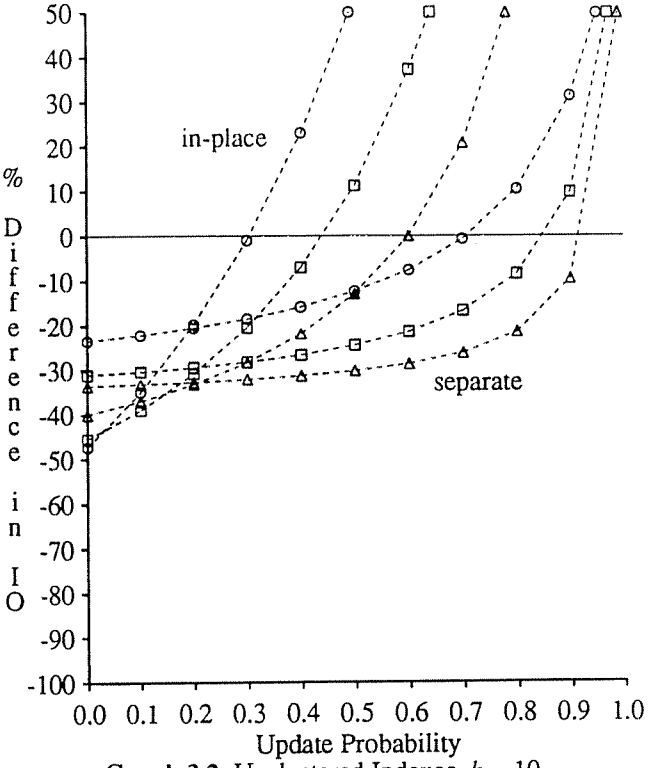
Excluding the case where $k = 50$, Graphs 3.1-3.3 show that in-place replication always outperforms separate replication when the probability of an update query is less than roughly 0.10. For update probabilities in that range, in-place replication reduces I/O costs by approximately 10% to 50%. In contrast, if the case where $k = 1$ is excluded, then separate replication always outperforms in-place replication when the probability of an update query exceeds roughly 0.25. For a wide range of update probabilities, separate replication reduces I/O costs by approximately 15% to 30%. What may be surprising is the large range of update probabilities for which separate replication outperforms no replication. The reason this happens is because update queries only update 10 objects in S; since each update to an object in S only has to be propagated to one object in T, the cost of propagating updates never becomes much of a factor with separate replication.

One aspect of the graphs worth noting is that, beyond a certain point, reading more data only reduces the relative effectiveness of replication. When $k = 10$, for example, separate replication performs the best when $sel_r = 0.005$, which corresponds to the case where read queries access the most data. By the time $k = 50$, however, $sel_r = 0.005$ usually results in the worst performance for separate replication, and instead $sel_r = 0.001$, which corresponds to the case where read queries access the least data, usually results in the best performance. This effect can be observed by noting the way the lines for $sel_r = 0.001$ and $sel_r = 0.005$ "flip" as f is increased from 10 to 50 in the graphs for separate replication. The reason why this occurs is fairly simple. As sel_r increases, read queries read more and more data in R. At some point, so much data is accessed in R that the cost of reading R overwhelms all other costs. Since replication does not reduce the cost of reading R, the savings in I/O due to replication start to decrease at that point on a percentage basis.

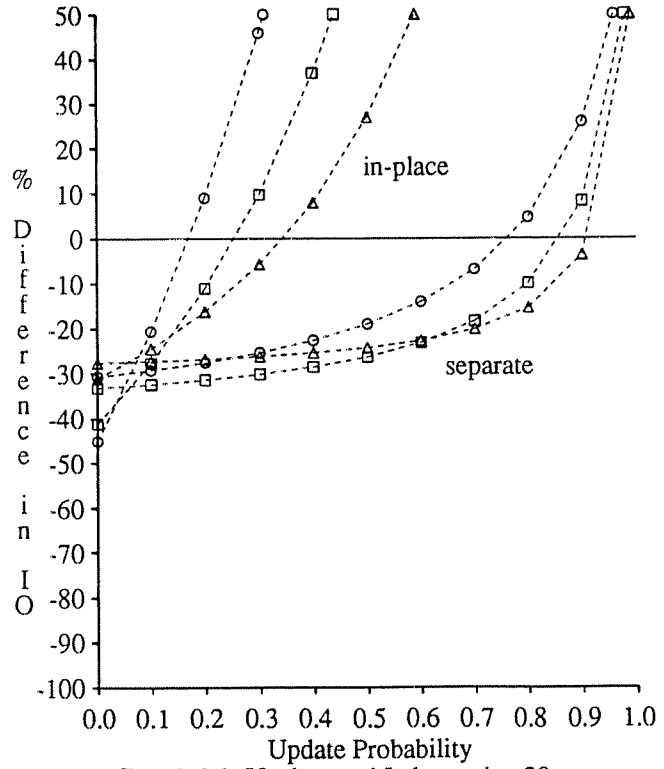
To provide further insight into why Graphs 3.1-3.4 appear as they do, selected values for C_{read} and C_{update} are presented in Table 3.3. (Recall that C_{read} is the cost of a read query and C_{update} is the cost of an update query.) Rather than present values for each combination of k and sel_r, only two sets of values are shown in Table 3.3. The left-hand side of the table lists C_{read} and C_{update} for $k = 1$ and $sel_r = 0.002$ and is representative of how C_{read} and



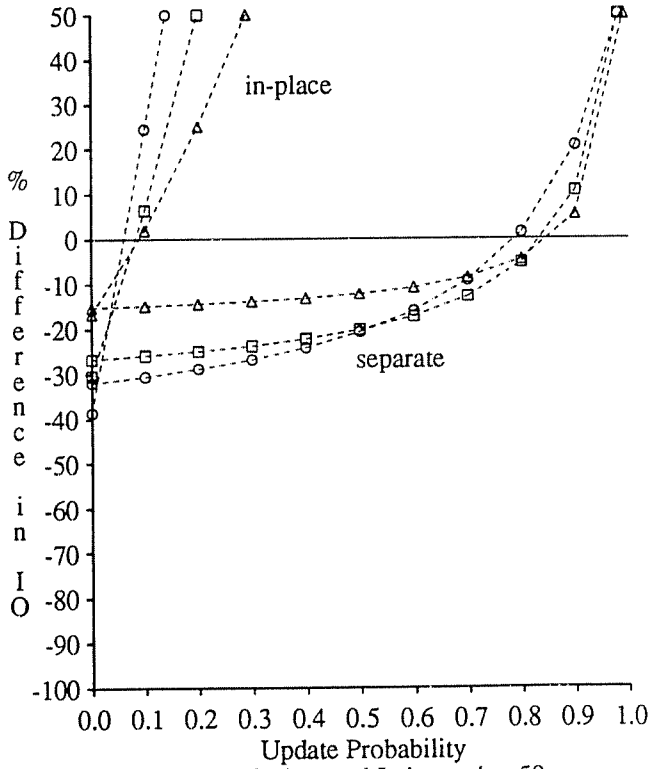
Graph 3.1: Unclustered Indexes $k = 1$
$|R| = 100{,}000$ $|S| = 100{,}000$



Graph 3.2: Unclustered Indexes $k = 10$
$|R| = 100{,}000$ $|S| = 10{,}000$



Graph 3.3: Unclustered Indexes $k = 20$
$|R| = 100{,}000$ $|S| = 5{,}000$



Graph 3.4: Unclustered Indexes $k = 50$
$|R| = 100{,}000$ $|S| = 2{,}000$

C_{update} differ from strategy to strategy when $k = 1$. The right-hand side of the table lists C_{read} and C_{update} for $k = 20$ and $sel_r = 0.002$ and is representative of how C_{read} and C_{update} differ from strategy to strategy when $k > 1$. In the table, fractional values were rounded up to the nearest unit.

Looking at Table 3.3, the reason for the shape of Graphs 3.1-3.4 should be clearer. By eliminating the join between R and S, in-place replication significantly reduces the cost of read queries for both $k = 1$ and $k = 20$. The cost of an update query, however, increases by roughly $20 \cdot k$ compared to no replication. This is because, with in-place replication, each update to an object in S (of which there 10 per update query) has to be propagated to k objects in R. Like in-place replication, separate replication also reduces the cost of read queries, but only slightly when $k = 1$. In contrast to in-place replication, the cost of an update query in separate replication is less affected by the value of k, and is roughly double the cost of an update query with no replication. The factor of two arises because, with separate replication, each update to an object in S has to propagated to an object in T.

3.5.7. Cost Analysis for Clustered Indexes

The cost equations for clustered indexes are similar to the ones for unclustered indexes. The only differences are that R is accessed in clustered order in read queries, and S and T are accessed in clustered order in update queries. Based on this observation, and referring to the cost equations for unclustered indexes, the cost equations for clustered indexes are relatively straightforward to derive. Without further explanation, they are:

No Replication:

$$\begin{aligned} C_{read} &= C_{read\ index\ on\ R} + C_{read\ R} + C_{read\ S} \\ &= 2 + \lceil |R'| \ / \ b - 1 \rceil + sel_r \cdot P_r + P_s \cdot Y(|R|, k \cdot O_s, |R'|) \\ C_{update} &= C_{read\ index\ on\ S} + C_{update\ S} \\ &= 2 + \lceil |S'| \ / \ b - 1 \rceil + 2 \cdot sel_s \cdot P_s \end{aligned}$$

	$k = 1, sel_r = 0.002$		$k = 20, sel_r = 0.002$	
Strategy	C_{read}	C_{update}	C_{read}	C_{update}
no replication	391	22	333	22
in-place replication	196	52	196	419
separate replication	363	42	222	39

Table 3.3: Selected Values for C_{read} and C_{update} (Unclustered Indexes)

In-Place Replication:

$$C_{read} = C_{read\ index\ on\ R} + C_{read\ R}$$
$$= 2 + \lceil |R'| \ / \ b - 1 \rceil + sel_r \cdot P_r$$
$$C_{update} = C_{read\ index\ on\ S} + C_{update\ S} + C_{read\ L} + C_{update\ R}$$
$$= 2 + \lceil |S'| \ / \ b - 1 \rceil + 2 \cdot sel_s \cdot P_s + + sel_s \cdot P_l + 2 \cdot P_r \cdot Y(|R|, O_r, k \cdot |S'|)$$

Separate Replication:

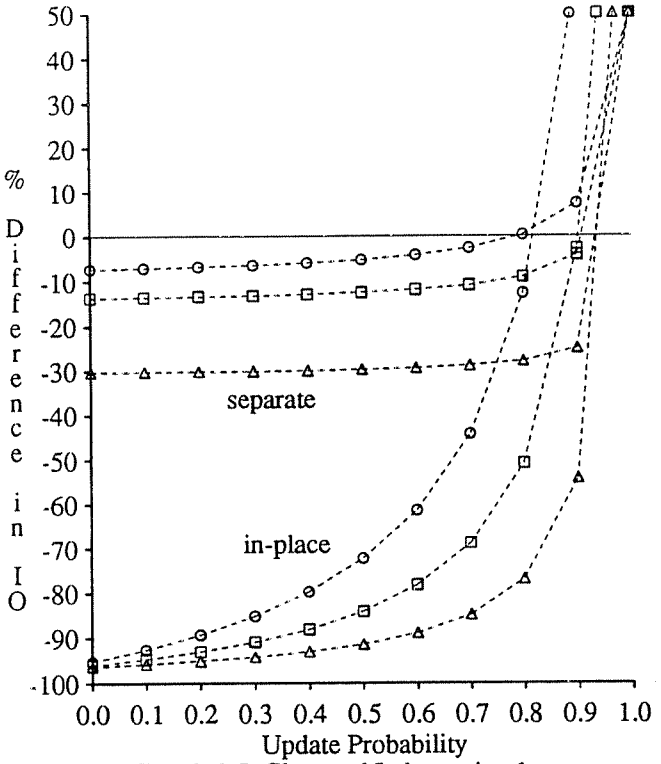
$$C_{read} = C_{read\ index\ on\ R} + C_{read\ R} + C_{read\ T}$$
$$= 2 + \lceil |R'| \ / \ b - 1 \rceil + sel_r \cdot P_r + P_t \cdot Y(|R|, k \cdot O_t, |R'|)$$
$$C_{update} = C_{read\ index\ on\ S} + C_{update\ S} + C_{update\ T}$$
$$= 2 + \lceil |S'| \ / \ b - 1 \rceil + 2 \cdot sel_s \cdot P_s + 2 \cdot sel_s \cdot P_t$$

3.5.8. Performance Results for Clustered Indexes

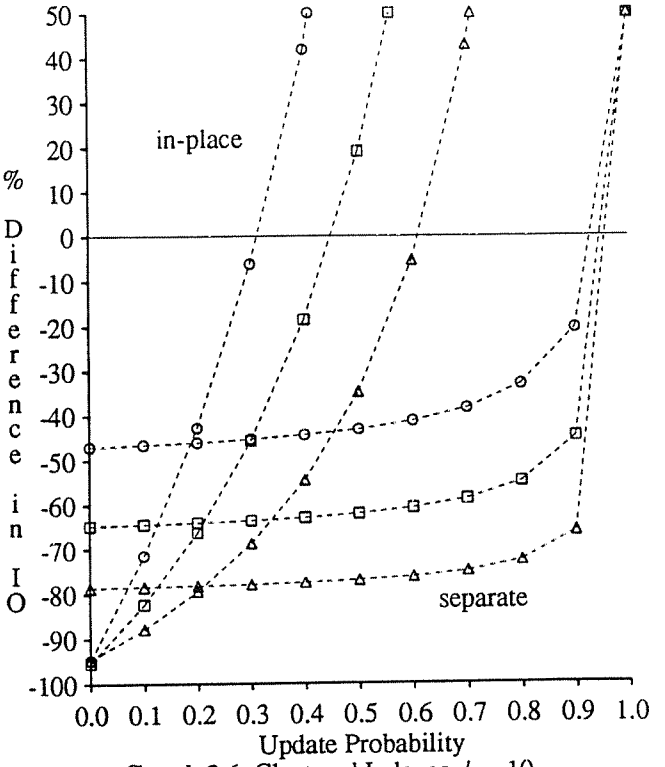
Graphs 3.5-3.8 present the results for clustered indexes. As mentioned earlier, when both indexes are clustered, less overall I/O is generated and therefore the savings in I/O due to replication is significantly larger on a percentage basis. As expected, Graphs 3.5-3.8 follow the same general patterns that were seen earlier. Once again, in-place replication performs its best for small values of k, while separate replication performs its best for large values of k. In-place replication is shown to be particularly effective when $k = 1$.

The surprising thing about these graphs is just how much replication reduces I/O costs when clustered indexes are used. For example, excluding the case when $k = 50$, the graphs show that in-place replication again outperforms separate replication when the probability of an update query is less than roughly 0.10. In this case, however, in-place replication reduces I/O costs by approximately 45% to 95% over that range of update probabilities, whereas it only reduced I/O costs by approximately 10% to 50% before. Similarly, if the case where $k = 1$ is excluded, then separate replication again outperforms in-place replication when the probability of an update query exceeds 0.25. In this case, separate replication reduces I/O costs by approximately 45% to 80% over a wide range of update probabilities, whereas it only reduced I/O costs by approximately 15% to 30% before.

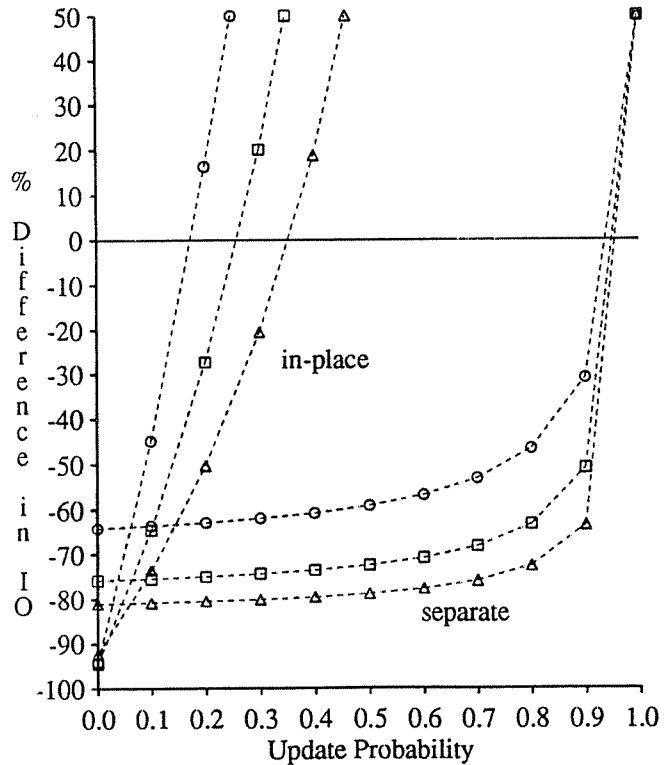
In Table 3.4, selected values of C_{read} and C_{update} are presented to provide further insight into why Graphs 3.5-3.9 take on the shape that they do. As expected, the values in Table 3.4 follow roughly the same patterns that were observed in Table 3.3. In general, the costs of read and update queries are much smaller here because R and S are accessed in clustered order. The one exception is the cost of an update query with in-place replication, which remains large. It remains large because the cost of propagating updates from S to R with in-place replication does



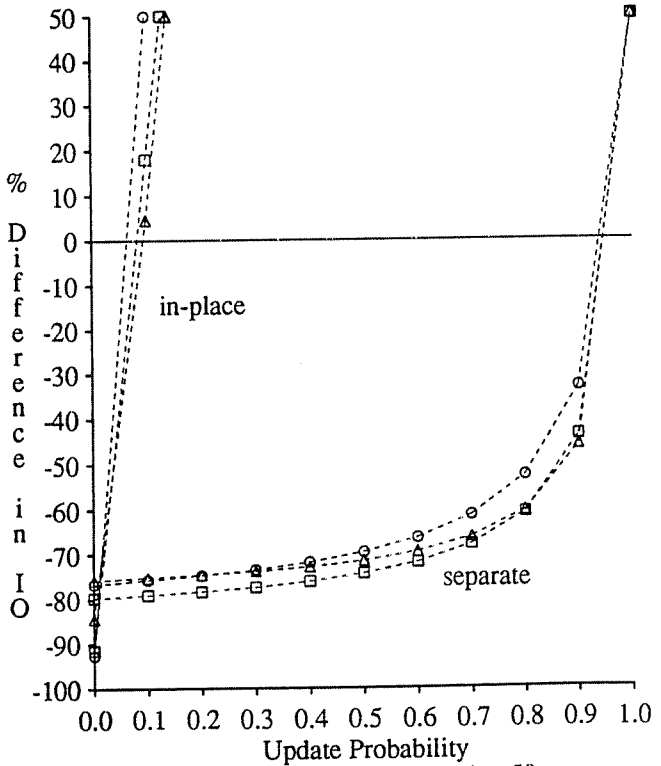
Graph 3.5: Clustered Indexes $k = 1$
$|R| = 100{,}000$ $|S| = 100{,}000$



Graph 3.6: Clustered Indexes $k = 10$
$|R| = 100{,}000$ $|S| = 10{,}000$



Graph 3.7: Clustered Indexes $k = 20$
$|R| = 100{,}000$ $|S| = 5{,}000$



Graph 3.8: Clustered Indexes $k = 50$
$|R| = 100{,}000$ $|S| = 2{,}000$

not change when clustered indexes are used.

3.6. CONCLUSIONS

This chapter introduced the notion of field replication and then described various ways to implement it. Two basic replication strategies were discussed, namely, in-place and separate replication. For both of these strategies, we showed how inverted paths can be used to keep replicated data consistent. A significant part of the chapter was devoted to describing how inverted paths can be efficiently implemented. Although much of the discussion was based on the EXTRA data model, the ideas presented here can also be extended to other data models that support reference attributes or referential integrity facilities of the type discussed in [Date87]. Finally, an analytical cost model was developed to give some feel for how beneficial field replication can be in executing simple queries.

Using the analytical cost model that was developed, we compared the I/O costs of executing simple queries with no replication, in-place replication, and separate replication. An analysis was carried out in two settings, one with unclustered indexes another with clustered indexes. For unclustered indexes, if the case where the sharing level was equal to one is excluded, we found that in-place replication reduced I/O costs by 10% to 50% when the update probability was small (less than 0.1). Under the same conditions, but for a much wider range of update probabilities, we found that separate replication reduced I/O costs by 15% to 30%. For clustered indexes, the improvement was even more dramatic. There, in-place replication reduced I/O costs by 45% to 95% when the update probability was small (less than 0.1), while for separate replication, I/O costs were reduced by 45% to 80% over a wide range of update probabilities. Thus, while field replication is a relatively simple notion, the analysis showed that it can provide significant performance gains in many situations.

	$k = 1, sel_r = 0.002$		$k = 20, sel_r = 0.002$	
Strategy	C_{read}	C_{update}	C_{read}	C_{update}
no replication	204	4	145	4
in-place replication	8	25	8	393
separate replication	176	6	35	6

Table 3.4: Selected Values for C_{read} and C_{update} (Clustered Indexes)

3.6.1. Directions for Future Work

As mentioned in the chapter on related work, an interesting direction for future work would be to examine how the access relations described in [Kemp90] could be used to simultaneously support both replication and path indexes. In the case of in-place replication, the reference paths that are stored in the access relation could be used to replace the link objects that were described in this chapter. In the case of separate replication, replicated data could be stored in the access relation itself. The advantage of such an approach is that one uniform mechanism would then be used to support both replication and path indexes.

Another possibility for future work would be to examine how full-object replication could be used to cluster shared sub-objects of complex objects. As mentioned in the chapter on related work, this has already been studied in [Jhin90], but only in the context of POSTGRES. Using full-object replication for clustering was also suggested in the context of the ObServer storage system [Horn87], but no performance study of its benefit was ever carried out.

Finally, it would be interesting to actually implement field replication in a real database system. It turns out that the inverted path structures described in this chapter were borrowed for use in Object Design's ObjectStore [Oren90], but there they are used to implement inverse relationships. Based on the author's implementation experiences with the incremental join facility of Starburst [Care90], it should also be fairly easy to add field replication to a database system that already supports referential integrity.

CHAPTER 4

POINTER-BASED JOINS

The previous chapter described how field replication can be used to eliminate or at least lower the cost of some functional joins. It should be clear that field replication will not always be appropriate, however. In many cases, the cost of updating replicated data would be too high or the storage overhead associated with replicated data would be undesirable. Under such circumstances, more conventional join techniques are necessary to process functional joins. In this chapter, we describe how physical OIDs (i.e., pointers) can be used effectively in processing functional joins. We describe several pointer-based join algorithms that are simple variations on the well known nested-loops, sort-merge, hybrid-hash, and hash-loops join algorithms [Blas77, Dewi84, Dewi85, Shap86], and then an analytical cost model is developed to compare the performance of the pointer-based algorithms to their standard, value-based counterparts. Results are presented for large, full-relation joins and small to medium-sized joins with a selection predicate. The pointer-based algorithms that are described in this chapter are actually quite general, and they can be used in any database system where physical pointers are used to link records or objects together.

The rest of the chapter is organized as follows: Section 4.1 provides simple examples of the type of joins that will be analyzed in this chapter. In Section 4.2, descriptions of the join algorithms that will be analyzed are given. The pointer-based join algorithms that are described in Section 4.2 assume a simple many-to-one pointer structure. In Section 4.3, these join algorithms are analyzed and compared to their standard, value-based counterparts. This is followed by Section 4.4, where pointer-based join algorithms that can make use of bidirectional pointer structures are described and analyzed. Finally, conclusions and directions for future work are presented in Section 4.5.

4.1. THE TYPES OF JOINS THAT ARE INITIALLY ANALYZED

The first part of this chapter will be concerned with the join of two sets, denoted R and S, that stand in a many-to-one relationship with each other. We will assume a simple pointer structure, where each object in R contains a physical OID (i.e., pointer) to its related object in S. Although pointer-based join algorithms can be developed for other relationships with different pointer structures, this is perhaps the simplest one to consider initially. Using the syntax of the EXTRA data model again, the following schema and query illustrate the type of joins that will initially be analyzed:

```
define type EMP (name: char[ ],  age: int,  job: ref JOB)
define type JOB (name: char[ ],  wage: int)

create Emp:  {own ref EMP}
create Job:  {own ref JOB}


retrieve  (Emp.name, Emp.age, Emp.job.name)
where     Emp.age < 30
```

Here we are assuming that the sets *Emp* and *Job* stand in a many-to-one relationship to each other. These sorts of functional joins are analogous to the foreign-key joins that are commonly executed in relational database systems. At the implementation level, the reference attribute *job* could be implemented as a physical OID, in which case it would effectively act like a physical pointer. There are currently at least two next-generation database systems that take this approach [Care89, Deux90]. Alternatively, *job* could be implemented as a foreign key or surrogate for the *Job* set, as described in [Zani83]. The pointer-based join algorithms that are described in this chapter will assume that *job* is implemented as physical OID. They will be compared against value-based join algorithms in which *job* is assumed to be implemented as a foreign key.

It is important to note that the pointer-based join algorithms which are analyzed in the first part of this chapter can be used on either (i) full joins or (ii) select-project-join queries in which there is a selection predicate on *Emp* that is more restrictive than any predicate on *Job*. The pointer-based algorithms that are analyzed initially cannot, however, be used effectively on select-project-join queries in which the selection predicate on *Job* is the more restrictive one. This is because the most efficient way to process such joins is in a direction that is opposite to the pointers that link *Emp* and *Job*. For example, in a join with a selection predicate on *Job* and no selection predicate on *Emp*, the best execution plan is likely to be one in which the selection on *Job* is evaluated first and then joined with *Emp*.

To process joins such as the one mentioned above with pointers, a simple many-to-one pointer structure (with pointers going from *Job* to *Emp*) is inadequate. What is needed for such joins is some sort of structure that links each object in *Job* to the objects in *Emp* that are related to it. One possibility is the kind of Codasyl-like pointer structure used in [Care90]. Another possibility is a one-to-many or bidirectional pointer structure, where each object in *Job* contains a set of embedded pointers linking that object to its related objects in *Emp*. In the latter part of this chapter, we describe and analyze pointer-based join algorithms that can be used on such pointer structures.

4.1.1. Pointer-Based Joins in Other Contexts

Before continuing, it is useful to note that the pointer-based join algorithms that are described in this chapter are actually quite general, and they can be used in other contexts besides just functional joins. For example, in a relational system, the schema presented earlier would appear as:

```
Emp  (name: char[ ],  age: int,  jobid: int)
Job  (jobid: int,  name: char[ ],  wage: int)
```

Here, the reference attribute *job* has been replaced with the attribute *jobid*, which serves as a foreign key for the *Job* relation. The join given earlier would then be rewritten as:

```
retrieve  (Emp.name, Emp.age, Job.name)
where     Emp.jobid = Job.jobid
and       Emp.age < 30
```

These sorts of foreign-key joins are probably the most common type of join in a relational database system. They are also a natural candidate for using pointer-based joins. For example, it is fairly easy to imagine a situation where the database system initializes and maintains hidden pointers in each *Emp* record, linking it to its associated *Job* record. This is in fact what the incremental join facility in Starburst does, as described in [Care90]. In general, the pointer-based join algorithms that are described in this chapter can be used in any database system where physical pointers are used to link records or objects together.

4.2. DESCRIPTIONS OF THE JOIN ALGORITHMS

This section describes the pointer-based join algorithms that will initially be analyzed. The description and analysis of the algorithms is based on the join of two object sets, denoted R and S, which stand in a many-to-one relationship with each other. Both R and S are assumed to be stored as separate disk files. In the standard, value-based join algorithms, we will assume that each object in R contains a foreign key for the object it references in S, while in the pointer-based join algorithms, we will assume that each object in R contains a physical pointer to the object it references in S.

As indicated above, our pointer-based join algorithms assume that physical pointers are used. More specifically, we will assume that each pointer includes a page identifier or PID that points directly to the disk page of the object that the pointer references. Depending on translation costs, our pointer-based join algorithms may or

may not be worthwhile in a system where pointers are logical surrogates that have to be translated via a system-wide table, as in [Horn87]. As indicated in [Shek88], however, physical pointers like those assumed here can often provide most of the functionality that logical surrogates provide without the overhead of a system-wide translation table.

The remainder of this section describes our pointer-based join algorithms. First, however, the standard, value-based counterparts against which they will be compared are reviewed. Note that only high-level descriptions of the algorithms are given here. Details such as how main memory is partitioned are addressed in the analysis. In addition, selections and projections are generally ignored for now. They will also be treated more thoroughly in the analysis.

4.2.1. Standard, Value-Based Join Algorithms

4.2.1.1. Standard Nested-Loops/Index-Nested-Loops

The nested-loops join algorithm usually performs poorly on full joins, and consequently it is not considered for such joins in the analysis. It will be used, however, in the analysis of small to medium-sized joins with a selection predicate on R. For those types of joins, a version of the nested-loops algorithm commonly referred to as index-nested-loops is used [Blas77]. Index-nested-loops is an option only when there is an index on S. It executes as follows:

(1) Let R′ be the result of the selection on R. The objects in R′ are first sorted by their join attribute (i.e., foreign key).

(2) Each object in R′ is then examined and its join attribute is used to probe the index on S to find the object that it joins with in S.

Note that this is slightly different than the conventional index-nested-loops algorithm in that R′ is sorted here. We have chosen this variation because when the index on S is a B+ tree, and when R′ is small, as in the examples that are analyzed, sorting R′ in this manner reduces the cost of probing the index enough to lower the overall cost of the join. More will be said about this in the analysis.

4.2.1.2. Standard Sort-Merge

Although sort-merge has been shown to be generally inferior to hybrid-hash [Dewi85, Shap86], it is still considered in the analysis because it is used by many relational database systems. In the analysis, we will assume that

memory is large enough so that R and S can be sorted and merged in two passes, as described in [Shap86]. Under that assumption, the algorithm executes as follows:

(1) R is read into memory and sorted output runs are generated using a heap or some other priority queue structure. The output runs are sorted by the value of the join attribute. The same is done for S.

(2) The output runs of R and S are then concurrently merged in memory. As objects from R and S are produced in sorted order by these merges, they are checked to see if their join attributes match. When an object from R matches one from S, the two are joined and the result is output.

4.2.1.3. Standard Hybrid-Hash

Hybrid-hash is usually considered to be the best-performing join algorithm when a full join is performed [Shap86]. Assuming that the size of S is bigger than R (otherwise exchange R and S), it executes as follows:

(1) R is read into memory[1] and divided into $B + 1$ partitions $R_0, R_1, \ldots R_B$ on the basis of some hash function that is applied to the join attribute. The same is done for S; thus, R_i will only join with S_i for $0 \leq i \leq B$. The value of B is chosen such that S_0 can be joined in memory with R_0 as S is being partitioned and such that a hash table for each R_i can later fit in memory for $1 \leq i \leq B$.

(2) After S has been partitioned and S_0 has been joined with R_0, each R_i is joined with S_i for $1 \leq i \leq B$. The joins between R_i and S_i for $0 \leq i \leq B$ are all performed with hashing. R_i is read into memory, and each object in R_i is hashed on its join attribute and inserted into a hash table. S_i is then read into memory, and the join attribute of each object in S_i is used to probe the hash table to find the object that it joins with in R_i.

In the above description, R is said to play the role of the building or *inner set*, while S is said to play the role of the probing or *outer set*. To minimize the number of I/Os, hybrid-hash always chooses the smaller of R and S to play the role of the inner set [Shap86].

4.2.1.4. Using Bit Filters in Standard Sort-Merge and Hybrid-Hash

As mentioned earlier, we shall consider small to medium-sized joins with a selection predicate on R in the analysis. For such joins, a technique call *bit filtering* [Dewi85, Mack86] can be employed to reduce the cost of the

[1] When we say that "R is read into memory", we mean that R is incrementally read into memory page by page; that is, page P_{i+1} is read only after all the objects in page P_i have been processed. This terminology shall be used throughout the description of the hybrid-hash algorithm.

sort-merge and hybrid-hash algorithms. In both algorithms, bit filtering works as follows:

(1) Let R′ be the result of the selection on R. Prior to executing the join, a bit vector in memory is initialized by setting all of its bits to zero. Then, as R′ is read, the join attribute of each object in R′ is hashed some number of times, and each resulting hash value is used to set a bit in the bit vector.

(2) Later, when S is read, the join attribute of each object in S is hashed in the same manner as in step (1), and the resulting hash values are used to check the bit vector. If one or more of the bits are not set, then that object can be safely discarded since it cannot possibly join with any object in R′.

The net effect of bit filtering is to filter out most of the objects in S that do not participate in the join at an early processing stage. As demonstrated in [Dewi85], this can result in significant savings due to the fact that these non-participating objects do not have to be stored on disk between the various phases of the sort-merge and hybrid-hash algorithms. In the analysis of small to medium-sized joins with a selection predicate on R, the analysis of both the sort-merge and hybrid-hash algorithms will include the effects of using a one-page bit filter.

4.2.2. Pointer-Based Join Algorithms

4.2.2.1. Pointer-Based Nested-Loops

The pointer-based nested-loops algorithm is the algorithm that results when naive pointer traversal is used to compute the join. In the algorithm, only one page of memory is allocated to read R, and the rest are allocated to read S. R is read into memory page by page. When a page of R is read into memory, the objects on that page are scanned one by one, and the pointer in each object r that is scanned is used to identify the object s that it joins with in S. The page containing s is read into memory (if it is not already there) and r is joined with s. In the above description, R is said to play the role of the outer set, while S is said to play the role of the inner set.

4.2.2.2. Pointer-Based Sort-Merge

As the analysis will clearly show, one of the problems with the pointer-based nested-loops algorithm is that it makes no attempt to optimize disk reads. As a result, a particular disk page in S can end up being read more than once. For example, suppose that two objects r_1 and r_2 reference the same object s in S. Depending on how R is organized, r_1 and r_2 may not be physically clustered on the same disk page in R. If that is the case, then between the time when r_1 is joined to s and the time when r_2 is joined to s, the page P containing s may be paged out of memory by the buffer replacement algorithm. In that event, P would have to be read twice, once to join r_1 with s,

and a second time to join r_2 with s.

The pointer-based sort-merge algorithm avoids this problem by first sorting all of the objects in R by the value of the page identifiers (PIDs) stored in their pointers. The effect of sorting R in this manner is to group all of the objects in R that reference the same page in S. Doing so guarantees that each page in S will be read only once. The algorithm executes as follows:

(1) R is read into memory and sorted much like it is in the standard sort-merge algorithm, except that here the output runs are sorted by PID values[2] rather than by the join attribute. S is *not* sorted.

(2) The output runs of R are then merged in memory. Each object r produced by the merge is examined and its pointer is used to identify the object s that it joins with in S. The page containing s is read into memory and r is joined with s.

4.2.2.3. Pointer-Based Hybrid-Hash

The pointer-based hybrid-hash algorithm groups the objects in R by PID values much like the pointer-based sort-merge algorithm. Instead of sorting R, however, hashing is used to group the objects that reference the same page in S. The algorithm executes as follows:

(1) R is partitioned much like it is in the standard hybrid-hash algorithm, except that here it is partitioned by PID values rather than by the join attribute. S is *not* partitioned.

(2) Each partition R_i of R is joined with S by taking R_i and building a hash table for it in memory. The hash table is built by hashing each object r in R_i by the value of its pointer's PID. The hash table is built in such a way that all objects which reference the same page in S are grouped together in the same hash entry (see Figure 4.1).

(3) Once the hash table for R_i has been built, each of its hash chains is scanned. Each time a new hash entry H is encountered on a chain, the page in S associated with H is read, and all of the objects in R that reference that page, which have been grouped in H, are joined with their corresponding objects in S.

---

[2] In the descriptions of the pointer-based sort-merge and hybrid-hash algorithms, "PID value" is meant to be understood as the value of the page identifier that is stored in a pointer.

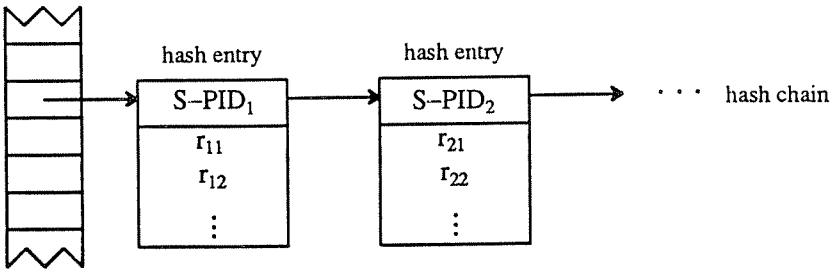


Figure 4.1: The Structure of the Hash Table for R_i

Note that one of the key differences between this algorithm and the standard hybrid-hash algorithm is that R is the only set that is partitioned, and as such it always plays the role of the inner set. This is necessary because the direction of the pointers is from R to S. As the reader shall see, when R is significantly larger than S, this can cause the standard hybrid-hash algorithm to outperform the pointer-based hybrid-hash algorithm.

4.2.2.4. Pointer-Based Hash-Loops

The pointer-based hash-loops join algorithm is really a cross between the standard simple-hash algorithm described in [Dewi84, Shap86] and the standard hash-loops algorithm described in [Dewi85]. Its performance characteristics are more like the standard hash-loops algorithm, however, and that is the reason for its name. The pointer-based hash-loops algorithm repeatedly executes the following two steps until all of R has been joined with S:

(1) A memory-sized chunk R_i of R (or whatever unprocessed portion of R that remains) is read into memory, and a hash table is built for it. The hash table for R_i has the same format as in the pointer-based hybrid-hash algorithm (see Figure 4.1).

(2) Once the hash table for R_i has been built, each of its hash chains is scanned, and R_i is joined with S just as in the pointer-based hybrid-hash algorithm.

Note that the pointer-based hash-loops algorithm is only slightly more sophisticated than the pointer-based nested loops algorithm, and as the analysis will show, it generally performs poorly, too. We still wanted to consider the pointer-based hash-loops algorithm here, though, because, with some minor changes, it can be used on bi-directional pointer structures. This is not true of all the other pointer-based join algorithms. Also note that the pointer-based hash-loops algorithm is essentially the same join algorithm that was described in [Vald87] for join

indexes. The main difference here, of course, is that no auxiliary data structure needs to be accessed to obtain pointer values. Another reason for considering the pointer-based hash-loops algorithm here is to demonstrate some of the algorithm's shortcomings that were not made clear in [Vald87]. In the above description, R is said to play the role of the outer set, while S is said to play the role of the inner set.

4.2.2.5. Pointer-Based PID-Partitioning

The final pointer-based join algorithm that we shall describe, which for lack of a better name is called PID-partitioning, has no standard counterpart. Unfortunately, the PID-partitioning algorithm is not as general as the other pointer-based algorithms that have been mentioned so far. It is generally impractical for anything other than full joins, and it also assumes that some sort of *page index* is maintained for each file F to keep track of all the page PIDs that make up F. Although such indexes are maintained by many storage systems (e.g., the EXODUS Storage Manager [Care89]), they may not always be present. Despite its limitations, however, the PID-partitioning algorithm performs very well in some situations and that is the reason for considering it here. The algorithm executes as follows:

(1) Using the page index for S, the PIDs of all the pages that make up S are read into memory and sorted. The resulting list of PIDs is then divided into $B+1$ partitions $L_0, L_1, \ldots L_B$. The range of pages in S that correspond to the PIDs in L_i will be denoted as S_i and they play much the same role as S_i does in the standard hybrid-hash algorithm when S forms the inner set. Similarly, the pages in R that join with S_i are denoted as R_i. As in the standard hybrid-hash algorithm, the value of B is chosen so that S_0 can be joined in memory with R_0 as R is being partitioned into $R_0, R_1, \ldots R_B$.

(2) After $L_0, L_1, \ldots L_B$ have been created, S_0 is read into memory. R is then read into memory page by page. When a page of R is read into memory, the objects on that page are scanned one by one and the pointer in each object r that is scanned is examined. By comparing the pointer in r to the first and last PID in L_i, it is possible to determine if r belongs to R_i. If r belongs to R_0, it is immediately joined with the object s in S_0 to which it refers. Otherwise, r is written to an output buffer associated with the R_i to which it belongs.

(3) After R has been partitioned and R_0 has been joined with S_0, each R_i is joined with S_i for $1 \leq i \leq B$. The joins between R_i and S_i are performed by first reading all of S_i into memory, after which R_i is read into memory page by page and joined with S_i. When a page of R_i is read into memory, the objects on that page are scanned one by one, and the pointer in each object r that is scanned is used to find the object s that it joins with in S_i.

As our performance results will demonstrate, the PID-partitioning algorithm often performs as well as or better than the standard hybrid-hash algorithm and the pointer-based hybrid-hash algorithm. Moreover, due to the nature of step (1), it is immune from the sort of skew problems that can degrade performance in the hash-based algorithms.

4.3. ANALYZING THE JOIN ALGORITHMS

In this section, an analysis is performed to quantitatively compare the first set of pointer-based join algorithms to their standard, non-pointer-based counterparts. The net CPU and I/O cost of executing each join algorithm is derived and used as the basis for comparison. Two types of joins are analyzed: full joins, where all of R is joined to all of S, and small to medium-sized joins, where the result of a selection on R is joined to S. For the small to medium-sized joins, the selectivity of the predicate on R will be varied to control the size of the joins. Projections are not considered in the analysis, since they do not change the general results in any significant way.

4.3.1. Assumptions in the Analysis

Figure 4.2 depicts the file structure that is assumed in the analysis. The object sets R and S are assumed to be stored as separate disk files, with each object in R referencing its related object in S via an embedded pointer or foreign key.

As in the previous chapter, we shall again assume that R and S are *relatively unclustered* (i.e., the objects in R are *not* ordered by their references to S). Once again, this assumption is made because we feel it represents the most typical case, as the objects in R would typically be ordered by the value of some data field, not by their references to S. Another reason for this assumption is because it is the most difficult case to analyze. Using the techniques developed in this section, the analysis for the cases where R and S are relatively clustered or clustered together is straightforward.

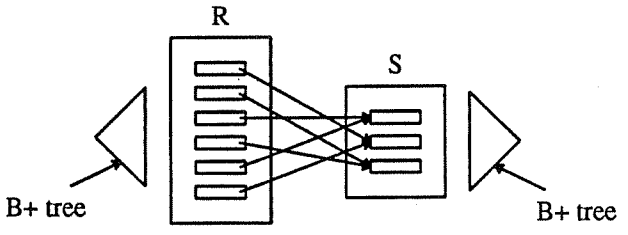


Figure 4.2: The Reference Structure of R and S

For uniformity, and to make the analysis tractable, we will assume that each object in S is, on the average, referenced by k objects in R. As in the previous chapter, the value k is referred to as the *sharing level* and it is varied in the analysis. The result of this assumption is that the cardinality (although not necessarily the size) of S will be equal to approximately 1 / k times the cardinality of R.

As Figure 4.2 illustrates, we will assume that B+ trees exist on both R and S. These will be used in the small to medium-sized joins with a selection predicate on R. The index on R will be used to evaluate the selection predicate, while the index on S will be used by the index-nested-loops join algorithm to avoid a file scan of S. For the PID-partitioning join algorithm, we will also assume that a page index is maintained for both R and S to keep track of all the page PIDs that make up each file. We assume here that the page indexes for R and S are implemented with B+ trees, as described in [Care89].

Finally, we will ignore the minimal memory requirements of the different join algorithms in the analysis and simply assume that there is always enough memory for each of the algorithms to execute. In all the examples that are analyzed, the minimal memory requirements turn out to be quite modest, so this is a reasonable assumption to make.

4.3.2. Parameters Used in the Analysis

The parameters that are used in the analysis are listed in Table 4.1. Although there are a large number of parameters, only a few of them are actually varied. As in the previous chapter, most of the parameters are not really parameters per se, but rather functions of a small set of "core" parameters, which consist of the parameters in the top half of Table 4.1. Defaults for the core parameters are listed in Table 4.2.

The meaning of most parameters should be clear from Table 4.1. The parameters that are kept fixed in the analysis are noted as such in Table 4.2. As indicated, all times are given in terms of milliseconds. The CPU time to compare, hash, move, and swap values (in memory) has been expressed as a function of the number of instructions executed and the instruction rate of the machine. This has been done because it allows us to look at how different CPU execution rates affect the results. Of course, instruction counts are highly implementation-dependent, so the goal here is mainly to come up with estimates that are reasonable in relative terms. Note that the values for $move_r$ and $move_s$ are based on the time to execute a small, 4-instruction loop that moves objects in word-size chunks.

Parameter	Definition
Mips	instruction execution rate
P	size of a disk page
IO_i	instructions to do an I/O
IO_{dev}	device time to do an I/O
M	number of memory buffer pages
b	average B+ tree fanout
|R|	number of objects in R
k	average sharing level
sel	selectivity of the predicate on R
r	size in bytes of objects in R
s	size in bytes of objects in S
$compare_i$	instructions to compare two values
$hash_{1_i}$	instructions for simple hash function
$hash_{2_i}$	instructions for complicated hash function
H	average length of hash chain [Shap86]
|S|	number of objects in S: $\|S\| = \|R\| / k$
O_r	objects per page in R: $O_r = \lfloor P / r \rfloor$
O_s	objects per page in S: $O_s = \lfloor P / s \rfloor$
P_r	pages in R: $P_r = \lceil \|R\| / O_r \rceil$
P_s	pages in S: $P_s = \lceil \|S\| / O_s \rceil$
IO_{cpu}	CPU time to do an I/O: $IO_{cpu} = IO_i / Mips$
IO	net time to do an I/O: $IO = IO_{cpu} + IO_{dev}$
compare	time to compare two values: $compare = compare_i / Mips$
$hash_1$	time to do simple hash: $hash_1 = hash_{1_i} / Mips$
$hash_2$	time to do complicated hash: $hash_2 = hash_{2_i} / Mips$
$move_r$	time to move an object in R: $move_r = 4 \cdot \lceil r / 4 \rceil / Mips$
$move_s$	time to move an object in S: $move_s = 4 \cdot \lceil s / 4 \rceil / Mips$
$swap_r$	time to swap two objects in R: $swap_r = 3 \cdot move_r$
$swap_s$	time to swap two objects in S: $swap_s = 3 \cdot move_s$

Table 4.1: The Parameters Used in the Analysis

Two hash functions have been introduced in Table 4.1 because in practice the hash function used in bit filtering ($hash_1$) would be simpler than the hash function used in the standard hybrid-hash algorithm ($hash_2$). Also note that the pointer-based join algorithms will use $hash_1$ instead of $hash_2$. This is because the PID values that are hashed in the pointer-based algorithms will typically have more uniformity than the join attributes that are hashed in the standard hybrid-hash algorithm, and a simpler hash function can be used as a result.

Defaults for Core Parameters	
Mips	30 (in millions of instructions per second)
IO_i	2,000 instructions
IO_{dev}	25 milliseconds
P	4096 bytes
M	256 pages (varied)
b	350
|R|	100,000 objects
k	1 (varied)
sel	0.01 (varied)
r	200 bytes
s	200 bytes (varied)
$compare_i$	10 instructions
$hash_{1_i}$	10 instructions
$hash_{2_i}$	20 instructions
H	1.4

Table 4.2: The Defaults for the Core Parameters

It is important to note that we do not distinguish between sequential I/O and random I/O in the analysis. This is reasonable for small to medium-sized joins, where the impact of sequential I/O is likely to be negligible. For full joins, however, the impact of sequential I/O can be significant if it is available. To gauge the impact of sequential I/O on full joins, results were also obtained for full joins with *P* set to 32 Kbytes (i.e., track-sized disk transfers). We shall comment on those results later in this chapter.

It should also be noted that CPU and I/O overlap is not considered in the analysis. We feel justified in doing this because all of the join algorithms would benefit from CPU and I/O overlap, and as a result, it should not significantly affect the relative performance of the algorithms. It is also questionable just how much CPU and I/O overlap is possible on a modern processor because of the large disparity between CPU and I/O speeds. For example, with the above parameter settings, nearly 38,000 objects can be hashed in the time it takes to read a 32 Kbyte track containing only 160 objects. In such an environment, very little CPU and I/O overlap is likely to be possible, even if multi-track read-ahead is supported.

4.3.3. Analysis of Full Joins

In this section, full joins between R and S are analyzed. As mentioned earlier, index-nested-loops is not considered for such joins. For brevity, we only touch on the analysis of the standard sort-merge and hybrid-hash algo-

rithms here. For more details, the reader should consult [Shap86]. Note that, although it is not always stated as such, the analysis that follows is in terms of expected costs. The same holds true for the analysis of small to medium-sized joins.

4.3.3.1. Standard Sort-Merge

Assuming enough memory is available, the cost of the two-pass sort-merge algorithm that was described earlier is [Shap86]:

$(P_r + P_s)\cdot IO$	read R and S
$+ \lvert R\rvert \cdot \log_2 \lvert R\rvert \cdot (compare + swap_r)$	manage the priority queues for R
$+ \lvert S\rvert \cdot \log_2 \lvert S\rvert \cdot (compare + swap_s)$	manage the priority queues for S
$+ (P_r + P_s)\cdot 2\cdot IO$	read and write the output runs for R and S
$+ (\lvert R\rvert + \lvert S\rvert)\cdot compare$	perform the final merge
$-min\,(P_s + P_r, M - \sqrt{(P_r + P_s) / 2})\cdot 2\cdot IO$	I/O savings if extra memory is available

In the final term, the value $\sqrt{(P_r + P_s) / 2}$ represents the minimal memory requirements of the algorithm. This is actually a more accurate estimate of the minimal memory requirements than the one presented in [Shap86]. To verify it, we have to show that approximately $\sqrt{(P_r + P_s) / 2}$ pages of memory are needed to simultaneously merge the sorted runs of R and S in memory, as required by the algorithm. This is verified by noting that each run of R will be on the average $2\cdot M$ pages in length [Knut73]. Consequently, there will be approximately $P_r / (2\cdot M)$ runs of R, and approximately $P_s / (2\cdot M)$ runs of S. To simultaneously merge the sorted runs of R and S in memory, M must therefore be large enough to satisfy:

$$M \geq \frac{P_r}{2\cdot M} + \frac{P_s}{2\cdot M}$$

This implies that:

$$M \geq \sqrt{(P_r + P_s) / 2}$$

Note that in the above analysis the cost to write the result of the join to disk has been excluded. This cost will be excluded in the analysis of all the algorithms. It is excluded because the result of a join often forms the input of another database operation without ever being completely written to disk; moreover, this cost would be the same for all the algorithms, anyway. Finally, note that the terms above group related costs in the algorithm, and their order

does not coincide with the actual step-by-step execution of the algorithm. This same approach will be used throughout the analysis.

4.3.3.2. Standard Hybrid-Hash

To analyze the cost of the hybrid-hash algorithm, recall that it begins by dividing R into $B + 1$ partitions $R_0, R_1, \ldots R_B$, and likewise for S. The proper value of B to make everything work is [Shap86]:

$$B = \left\lceil \frac{P_r - M}{M - 1} \right\rceil$$

When R is partitioned, $M - B$ pages of memory are allocated to build the hash table for R_0. The remainder of memory is allocated to serve as output buffers for $R_1, R_2, \ldots R_B$, with one page allocated per partition. S is then partitioned in a similar fashion, except that S_0 is joined with R_0 as it is being partitioned. Letting q equal the fraction of R represented by R_0:

$$q = \frac{M - B}{P_r}$$

the cost of the hybrid-hash algorithm is [Shap86]:

$(P_r + P_s) \cdot IO$	read R and S
$+ (\|R\| + \|S\|) \cdot hash_2$	partition R and S
$+ \|R\| \cdot q \cdot move_r$	extract R_0 from R to build the hash table for R_0
$+ \|R\| \cdot (1 - q) \cdot move_r$	move the objects in $R_1, R_2, \ldots R_B$ to their partition's output buffer
$+ \|S\| \cdot (1 - q) \cdot move_s$	move the objects in $S_1, S_2, \ldots S_B$ to their partition's output buffer
$+ P_r \cdot (1 - q) \cdot 2 \cdot IO$	read and write partitions $R_1, R_2, \ldots R_B$
$+ P_s \cdot (1 - q) \cdot 2 \cdot IO$	read and write partitions $S_1, S_2, \ldots S_B$
$+ (\|R\| + \|S\|) \cdot (1 - q) \cdot hash_2$	hash the objects in each R_i and probe with S_i, $1 \le i \le B$
$+ \|S\| \cdot compare \cdot H$	search a hash chain for each object in S to find its match

Note that here we have assumed that $P_r < P_s$, otherwise the roles of R and S should be exchanged in the above analysis. As shown in [Shap86], the minimal memory requirements of the hybrid-hash algorithm are $\sqrt{P_r}$ pages.

4.3.3.3. Pointer-Based Nested-Loops

In the pointer-based nested-loops algorithm, one page of memory is allocated to read R and the remainder are allocated to read S. Because of the way memory is allocated, and because R and S are relatively unclustered, this

causes S to be accessed in exactly the same manner as it would be in an unclustered index scan of S with a buffer of size $M - 1$. The I/O cost of performing an unclustered index scan was derived in [Mack89]. Making use of the equations in that paper, the function $U_s(x, B)$, which counts the expected number of I/Os that are generated when the pointers of x objects in R are dereferenced and a buffer of B pages is used, is defined as:

$$U_s(x, B) = \begin{cases} P_s \cdot (1 - q^x) & \text{if } x \le n \\ P_s \cdot (1 - q^n) + (x - n) \cdot P_s \cdot p \cdot q^n & \text{if } x > n \end{cases}$$

where

$$q \equiv 1 - p = \frac{|S| - O_s}{|S|}$$

$$n = max\left\{ j \text{ in } \{0, 1, ..., x\} \text{ such that } P_s \cdot (1 - q^j) \le B \right\}$$

Using $U_s(\,)$, the cost of accessing S is therefore $U_s(|R|, M - 1) \cdot IO$. We have defined $U_s(\,)$ in this manner because it will be needed again later.

In addition to the cost of accessing S, there is also the cost to read R, and the CPU cost to check whether the page referenced by a given pointer in R is in memory, which we assume is done with hashing. The net cost of the pointer-based nested-loops algorithm is therefore:

$P_r \cdot IO$	read R
$+ U_s(\|R\|, M - 1) \cdot IO$	access S
$+ \|R\| \cdot hash_1$	check the buffer for each pointer dereference

4.3.3.4. Pointer-Based Sort-Merge

The analysis of the pointer-based sort-merge algorithm is similar to the analysis of the standard sort-merge algorithm. The key difference here, of course, is that only R is sorted. Taking the analysis of the standard sort-merge algorithm and making the appropriate changes, the cost of the pointer-based sort-merge algorithm is:

$(P_r + P_s) \cdot IO$	read R and S
$+ \|R\| \cdot \log_2 \|R\| \cdot (compare + swap_r)$	manage the priority queues for R
$+ P_r \cdot 2 \cdot IO$	read and write the output runs for R
$- min\,(P_r, M - \sqrt{P_r / 2}) \cdot 2 \cdot IO$	I/O savings if extra memory is available

Note that here the I/O savings is a function of $\sqrt{P_r / 2}$, which corresponds to the minimal memory requirements of the algorithm. This is in contrast to the minimal memory requirements of the standard sort-merge algorithm, which as noted earlier are $\sqrt{(P_r + P_s) / 2}$ pages. Less memory is required here because only R is sorted.

4.3.3.5. Pointer-Based Hybrid-Hash

As one would expect, the analysis of the pointer-based hybrid-hash algorithm is similar to the analysis of the standard hybrid-hash algorithm. The main difference here is that only R is partitioned. Taking the analysis of the standard hybrid-hash algorithm and making the appropriate changes, the cost of the pointer-based hybrid-hash algorithm is:

$(P_r + P_s) \cdot IO$	read R and S
$+ \lvert R \rvert \cdot hash_1$	partition R
$+ \lvert R \rvert \cdot q \cdot move_r$	extract R_0 from R to build the hash table for R_0
$+ \lvert R \rvert \cdot (1 - q) \cdot move_r$	move the objects in $R_1, R_2, \ldots R_B$ to their partition's output buffer
$+ P_r \cdot (1 - q) \cdot 2 \cdot IO$	read and write partitions $R_1, R_2, \ldots R_B$
$+ \lvert R \rvert \cdot (1 - q) \cdot hash_1$	hash the objects in each R_i, $1 \le i \le B$

To determine the minimal memory requirements of the pointer-based hybrid-hash algorithm, it suffices to note that in [Shap86] the minimal memory requirements of the standard hybrid-hash algorithm were shown to be approximately equal to $\sqrt{P}$ pages, where P is the number of pages in the inner set. The minimal memory requirements were not affected by the size of the outer set. From this it follows that the minimal memory requirements of the pointer-based hybrid-hash algorithm are $\sqrt{P_r}$ pages, since R always plays the role of the inner set here.

4.3.3.6. Pointer-Based Hash-Loops

Recall that the pointer-based hash-loops algorithm executes by reading memory-sized chunks of R into memory and joining them with S. When a memory-sized chunk of R is read into memory, it is joined with S much like in the pointer-based hybrid-hash algorithm. Let n equal the number of times the algorithm iterates, and let R_i equal the portion of R read into memory and joined with S on the i^{th} iteration. With B buffers, the number of times the hash-loops algorithm iterates is simply:

$$n = \left\lceil \frac{P_r}{B} \right\rceil$$

Let $\lvert R_i \rvert$ equal the number of objects in R_i. For the first $n - 1$ iterations, each R_i will be as large as memory and

therefore:

$$|R_i| = B \cdot O_r \quad for\ 1 \le i \le n - 1.$$

The number of objects in R_n equals whatever unprocessed portion of R remains after the first $n - 1$ iterations, namely:

$$|R_n| = |\mathrm{R}| - (n-1) \cdot (B \cdot O_r)$$

To determine the number of pages in S that are accessed in the i^{th} iteration of the algorithm, the Yao function $Y(\,)$ that was introduced in Chapter 3 can be used again. The analysis proceeds by considering the probability that no object in R_i joins with an object on a given page of S. Since each page in S is referenced by $k \cdot O_s$ objects in R, a similar application of the previous analysis shows that the number of pages of S that are accessed on the i^{th} iteration equals:

$$P_s \cdot Y(|\mathrm{R}|, k \cdot O_s, |R_i|)$$

To summarize the above analysis, the function $T_s(B)$, which counts the expected number of I/Os generated for S by the hash-loops algorithm when a buffer of B pages is used, is defined as:

$$T_s(B) = (n-1) \cdot P_s \cdot Y(|\mathrm{R}|, k \cdot O_s, x) + P_s \cdot Y(|\mathrm{R}|, k \cdot O_s, y)$$

where

$$n = \lceil P_r \,/\, B \rceil$$
$$x = B \cdot O_r$$
$$y = |\mathrm{R}| - (n-1) \cdot x$$

Putting together all the terms, the cost of the pointer-based hash-loops algorithm is:

$P_r \cdot IO$	read R
$+ T_s(M-1) \cdot IO$	access S
$+ \|\mathrm{R}\| \cdot hash_1$	hash the objects in each R_i, $1 \le i \le n$

4.3.3.7. Pointer-Based PID-Partitioning

Using the page index for S, the PID-partitioning algorithm begins by reading the PIDs of all the pages that make up S. The PIDs are then sorted and divided into $B + 1$ partitions $L_0, L_1, \ldots L_B$. (We will describe how B is calculated shortly.) Since we are assuming that the page index on S is implemented with a B+ tree, the cost to sort the list of PIDs for S can be avoided. In all the examples that are analyzed, the height of the page index on S will be equal to 2. Consequently, the cost to read the page index[3] and form $L_0, L_1, \ldots L_B$ is:

$2 \cdot IO$	I/O cost to descend the page index (the CPU cost is negligible)
$+ \lceil P_s / b - 1 \rceil \cdot IO$	scan across the leaves of the page index
$+ P_s \cdot compare$	divide up the list of PIDs into $L_0, L_1, \ldots L_B$

As mentioned in the description of the algorithm, the range of pages in S that correspond to the PIDs in L_i are denoted as S_i and they play much the same role as S_i does in the standard hybrid-hash algorithm when S forms the inner set. Similarly, the pages in R that join with S_i are denoted as R_i. As in the standard hybrid-hash algorithm, the value of B is chosen so that S_0 can be joined in memory with R_0 as R is being partitioned into $R_0, R_1, \ldots R_B$:

$$B = \left\lceil \frac{P_s - M}{M - 1} \right\rceil$$

The fraction of S that is represented by S_0 is denoted as q, where:

$$q = \frac{M - B}{P_s}$$

$L_0, L_1, \ldots L_B$ effectively determine how S is partitioned and also how R is to be partitioned. A binary search is performed on $L_0, L_1, \ldots L_B$ for each object r in R to determine the R_i to which r belongs. In the final stages of the algorithm, each R_i is joined with S_i. This is accomplished by first reading S_i into memory. R_i is then read into memory page by page and joined with S_i. When a page of R_i is read into memory, the objects on that page are scanned one by one. The pointer in each object r that is scanned is hashed and the resulting valued is used to find the object s that it joins with in S_i. Based on the above discussion and the analysis for the standard hybrid-hash

[3] Note that in all the other join algorithms, we have tacitly assumed that the data pages of both R and S are linked together, thereby eliminating the need to access a page index in those algorithms.

algorithm, the cost of the PID-partitioning algorithm is:

$2 \cdot IO$	I/O cost to descend the page index
$+ \lceil P_s / b - 1 \rceil \cdot IO$	scan across the leaves of the page index
$+ P_s \cdot compare$	divide up the list of PIDs into $L_0, L_1, \ldots L_B$
$+ (P_r + P_s) \cdot IO$	read R and S
$+ \lvert R \rvert \cdot \log_2(B + 1) \cdot compare$	partition R
$+ \lvert R \rvert \cdot (1 - q) \cdot move_r$	move the objects in $R_1, R_2, \ldots R_B$ to their partition's output buffer
$+ P_r \cdot (1 - q) \cdot 2 \cdot IO$	read and write partitions $R_1, R_2, \ldots R_B$
$+ \lvert R \rvert \cdot hash_1$	hash the pointer in every object of R to find the object it joins with in S

Based on the above analysis, it should be clear that the minimal memory requirements of the PID-partitioning algorithm are the same as those of the standard hybrid-hash algorithm when S forms the inner set, namely, $\sqrt{P_s}$ pages.

4.3.4. Analysis of Small to Medium-Sized Joins

In this section, we analyze small to medium-sized joins, where the result of a selection on R is joined to S. As mentioned earlier, the analysis will assume that B+ tree indexes exist on both R and S. The index on R will be used to evaluate the selection predicate, while the index on S, which is assumed to be a unique primary-key index, will be used by the index-nested-loops algorithm to avoid a file scan of S. It should be clear that the pointer-based join algorithms will outperform their standard, value-based counterparts on most small to medium-sized joins by avoiding index lookups on S, and the performance results will bear this out. Consequently, the primary motivation for analyzing small to medium-sized joins is to derive cost equations that can be used in query optimization.

The only index clustering combination that will be analyzed in this section is the combination where the index on R is a clustered index and the index on S is an unclustered index. We will denote this combination as clustered/unclustered, and similarly for the other combinations. The clustered/unclustered index combination has been chosen here because it is perhaps the most difficult to analyze. Note that we will still present results for the unclustered/clustered index combination, as well as comment on how the results for the two remaining index combinations compare to the results that are presented. In addition, the analysis for the unclustered/clustered index combination is presented in the Appendix A. Using that analysis and the analysis of this section, the remaining index combinations are straightforward to analyze.

Note that, throughout this section, we will ignore the small amount of memory space that is required to read the indexes on R and S. A small, two-page MRU buffer group [Chou85] would be sufficient for reading either of these indexes in the examples that are analyzed, and accounting for that little space would not change the results in any significant way. In addition, we will assume that the selectivity of the predicate on R is such that R′, which is defined as the result of the selection on R, fits completely in memory. In contrast to the indexes on R and S, we will account for the memory space used by R′.

Another thing to note is that this section does not include an analysis for the pointer-based hash-loops join algorithm. Under our assumption that R′ fits completely in memory, the pointer-based hash-loops algorithm executes in the same manner as the pointer-based hybrid-hash algorithm, and therefore its analysis can be omitted. This section also does not include an analysis for the PID-partitioning join algorithm. As mentioned earlier, the PID-partitioning algorithm is generally impractical for anything other than full joins.

Finally, to simplify the equations of this section, let $|R'|$ denote the cardinality of R′ and let P_r' denote the number of pages in R′, that is, let $|R'| = sel \cdot |R|$ and $P_r' = sel \cdot P_r$.

4.3.4.1. Standard Index-Nested-Loops

The index-nested-loops algorithm begins by using the B+ tree index on R to obtain R′. This is done by descending the index to a leaf, and then scanning across the leaves to obtain the objects in R that satisfy the selection predicate. In all the examples that are analyzed, the height of the index on R is equal to 2, and similarly for S. The cost of reading the index on R is therefore:

$2 \cdot IO$	I/O cost to descend the index
$+ 2 \cdot \log_2 b \cdot compare$	CPU cost to descend the index
$+ \lceil \lvert R' \rvert \ / \ b - 1 \rceil \cdot IO$	scan across the leaves of the index

R′ is then read into memory using the index and sorted by foreign key (i.e., by join attribute). This costs:

$P_r' \cdot IO$	cost to read R′
$+ \lvert R' \rvert \cdot move_r$	extract R′ from R before sorting
$+ \lvert R' \rvert \cdot \log_2 \lvert R' \rvert \cdot (compare + swap_r)$	cost to sort R′

After R′ has been sorted, the foreign key in each object of R′ is used to probe the index on S. Assuming that the height of the index is 2, the I/O cost to probe the index consists of the cost to read the root page of the index plus

the cost of reading whatever leaf pages of the index are accessed by the index probes. Because R′ is sorted by foreign key, the leaf pages will be accessed in ascending key order, and consequently a simple two-page MRU buffer group will ensure that no leaf page is read more than once. All that remains, therefore, is to determine the number of leaf pages that are accessed by the index probes.

To determine the number of leaf pages that are accessed by the index probes on S, the Yao function $Y(\,)$ can be used again. The analysis proceeds by considering the probability that a particular leaf page L_i is not accessed. Since R and S are relatively unclustered, we can assume that any particular leaf page of the index is just as likely to be accessed as any other leaf page. Therefore, the probability that a leaf page L_i is not accessed is equal to the probability of choosing a subset of $|R'|$ objects from R such that the chosen subset contains no object with a foreign key in L_i. Since each leaf page contains b keys, and since there are $\lceil |S| \,/\, b \rceil$ leaf pages and k objects in R that share each key value, a similar application of the previous analysis yields:

$$\textit{expected leaf pages accessed} = \lceil |S| \,/\, b \rceil \cdot Y(|R|,\, b \cdot k,\, |R'|) \cdot IO$$

The total cost to probe the index on S is therefore:

IO	read the root page
$+\, 2 \cdot |R'| \cdot \log_2 b \cdot compare$	probe the index on S for each object in R′
$+ \lceil |S| \,/\, b \rceil \cdot Y(|R|,\, b \cdot k,\, |R'|) \cdot IO$	access the leaf pages

The final cost that needs to be included is the cost to read S. Here, we assume that S is read using all of the available memory that is not allocated to hold R′. In effect, this means that S ends up being read in the same manner as in the pointer-based nested-loops algorithm, except that here index pointers take the place of object pointers from R; in addition, S is referenced $|R'|$ times now and there are only $M - P_r'$ memory pages available to read S. Based on these observations and using the I/O function $U_s(\,)$ that was defined earlier, the cost to read S is simply:

$|R'| \cdot hash_1$	check memory for each index reference to S
$+\, U_s(|R'|,\, M - P_r') \cdot IO$	access S

Before going on to the analysis of the sort-merge algorithm, it is important to point out that we also analyzed the normal index-nested-loop algorithm where R′ is not sorted. In all of the examples that are considered, it performed considerably worse than the algorithm that we have described due to the added cost of accessing the leaf

pages of the index on S in a random fashion.

4.3.4.2. Standard Sort-Merge

The sort-merge algorithm begins by reading R′ into memory and sorting it, just as in the index-nested-loops algorithm. In addition, each object in R′ is also hashed to turn on a bit in the bit filter. Here we shall assume that only one hash function is used for bit filtering. In the examples that are analyzed, a simple page-sized bit filter using one hash function turns out to be sufficient to filter out virtually all of the non-participating objects in S. (Details on how to calculate the optimal number of hash functions to use, and what the effectiveness of the resulting bit filter is, can be found in [Seve76].)

After R′ has been sorted, S is read into memory, filtered, and the resulting objects are sorted. As mentioned, we are assuming that virtually all of the non-participating objects in S are screened out by the bit filter. Consequently, only the objects in S that join with R′ are sorted. Let S′ denote these objects, and let $|S'|$ and P_s' denote the cardinality and the number of pages in S′, respectively. To determine $|S'|$, the Yao function $Y(\,)$ can be used once again. Since each object in S joins with k objects in R, a similar application of the previous analysis yields:

$$|S'| = |S| \cdot Y(|R|, k, |R'|)$$

The corresponding value for P_s' is simply $P_s' = |S'| \,/\, O_s$. Based on the analysis of the standard sort-merge algorithm for full joins, and based on the analysis of the index-nested-loops algorithm on small to medium-sized joins, the cost of the sort-merge algorithm is:

$2 \cdot IO + 2 \cdot \log_2 b \cdot compare$	descend the index on R
$+ \lceil |R'| \,/\, b - 1 \rceil \cdot IO$	scan across the leaves of the index on R
$+ (P_r' + P_s) \cdot IO$	read R′ and S
$+ (|R'| + |S|) \cdot hash_1$	bit filter R′ and S
$+ |R'| \cdot move_r$	extract R′ from R before sorting
$+ |S'| \cdot move_s$	extract S′ from S before sorting
$+ |R'| \cdot \log_2 |R'| \cdot (compare + swap_r)$	manage the priority queues for R′
$+ |S'| \cdot \log_2 |S'| \cdot (compare + swap_s)$	manage the priority queues for S′
$+ (P_r' + P_s') \cdot 2 \cdot IO$	read and write the output runs for R′ and S′
$+ (|R'| + |S'|) \cdot compare$	perform the final merge
$-min\,(P_s' + P_r', M - \sqrt{(P_r' + P_s') \,/\, 2}) \cdot 2 \cdot IO$	I/O savings if extra memory is available

4.3.4.3. Standard Hybrid-Hash

The analysis of the hybrid-hash algorithm follows from the analysis that was used earlier on full joins and from the analysis of the standard sort-merge algorithm on small to medium-sized joins. With S′ defined the same way as it was in the analysis of the sort-merge algorithm, and with our assumption that R′ fits in memory, the cost of the hybrid-hash algorithm is:

$2 \cdot IO + 2 \cdot \log_2 b \cdot compare$	descend the index on R
$+ \lceil \lvert R' \rvert \ / \ b - 1 \rceil \cdot IO$	scan across the leaves of the index on R
$+ (P_r' + P_s) \cdot IO$	read R′ and S
$+ \lvert R' \rvert \cdot move_r$	extract R′ from R to build the hash table for R′
$+ (\lvert R' \rvert + \lvert S \rvert) \cdot hash_1$	bit filter R′ and S
$+ (\lvert R' \rvert + \lvert S' \rvert) \cdot hash_2$	partition R′ and S′
$+ \lvert S' \rvert \cdot compare \cdot H$	search a hash chain for each object in S′ to find its match

4.3.4.4. Pointer-Based Nested-Loops

The analysis of the pointer-based nested-loops algorithm follows almost directly from the analysis that was used earlier on full joins. Taking that analysis and making some straightforward changes, the cost of the pointer-based nested-loops algorithm is:

$2 \cdot IO + 2 \cdot \log_2 b \cdot compare$	descend the index on R
$+ \lceil \lvert R' \rvert \ / \ b - 1 \rceil \cdot IO$	scan across the leaves of the index on R
$+ P_r' \cdot IO$	read R′
$+ \lvert R' \rvert \cdot hash$	check memory for each pointer reference in R′
$+ U_s(\lvert R' \rvert, M - 1) \cdot IO$	access S

4.3.4.5. Pointer-Based Sort-Merge

In order to analyze the pointer-based sort-merge algorithm, we need to estimate how many pages in S participate in the join. Let P_s' denote number of pages in S that participate in the join. (Note that P_s' as defined here is not the same as the P_s' that was defined in the standard algorithms.) To determine P_s', we again turn to the Yao function $Y()$. The analysis proceeds by considering the probability that no object in R′ joins with a given page in S. Since each page in S joins with $k \cdot O_s$ objects in R, a similar application of the previous analysis yields:

$$P_s' = P_s \cdot Y(\lvert R \rvert, k \cdot O_s, \lvert R' \rvert)$$

With the exception of P_s', the rest of the analysis is straightforward and follows from the analysis that was used earlier on full joins. Under our assumption that R′ fits in memory, the cost of the pointer-based sort-merge algorithm is:

$2 \cdot IO + 2 \cdot \log_2 b \cdot compare$	descend the index on R
$+ \lceil \lvert R' \rvert \ / \ b - 1 \rceil \cdot IO$	scan across the leaves of the index on R
$+ (P_r' + P_s') \cdot IO$	read R′ and S′
$+ \lvert R' \rvert \cdot move_r$	extract R′ from R before sorting
$+ \lvert R' \rvert \cdot \log_2 \lvert R' \rvert \cdot (compare + swap_r)$	manage the priority queues for R′

4.3.4.6. Pointer-Based Hybrid-Hash

The analysis of the hybrid-hash algorithm follows directly from the analysis that was used earlier on full joins and also from the analysis of the pointer-based sort-merge algorithm on small to medium-sized joins. With P_s' defined as in the analysis of the pointer-based sort-merge algorithm, and with our assumption that R′ fits in memory, the cost of the pointer-based hybrid-hash algorithm is:

$2 \cdot IO + 2 \cdot \log_2 b \cdot compare$	descend the index on R
$+ \lceil \lvert R' \rvert \ / \ b - 1 \rceil \cdot IO$	scan across the leaves of the index on R
$+ (P_r' + P_s') \cdot IO$	read R′ and S′
$+ \lvert R' \rvert \cdot move_r$	extract R′ from R to build the hash table for R′
$+ \lvert R' \rvert \cdot hash_1$	hash R′

4.3.5. Performance Results for Full Joins

The results for full joins between R and S are presented in Graphs 4.1-4.4. The results presented in Graphs 4.1-4.4 are also representative of large select-project-join queries in which a selection on R is joined to S. In all of the graphs, the curves for the standard, value-based algorithms are labeled wo/ptr for "without pointer," while the curves for the pointer-based algorithms are labeled w/ptr for "with pointer." The graphs were obtained by using the equations from the analysis to compute the total time in seconds to run each join algorithm for a particular memory size. The size of memory was increased in 1/2 Mbyte increments and ranged from the minimal size required to execute all of the join algorithms, which was approximately 1/4 Mbytes, all the way up to 10 Mbytes. Of course, 10 Mybtes is a modest amount of memory by today's standards. In practice, however, the amount of memory allocated to a given join in a multi-user system will tend to be much less than the amount of physical memory that is available. Consequently, we feel that our choice of memory sizes is reasonable. Also note that similar results could have

been obtained for larger memory sizes by simply choosing larger values for the sizes of R and S.

We begin with Graph 4.1, which shows the performance of the join algorithms when R and S are the same size (in pages). In that graph, both R and S consist of 100,000 objects with 200 bytes per object. As shown, the pointer-based join algorithms can provide significant performance gains in this situation. Compared to the standard hybrid-hash algorithm, the pointer-based hybrid-hash algorithm and the PID-partitioning algorithm reduce the join time by approximately 30% over a large range of memory sizes. This, of course, is because S is not partitioned in either algorithm. A similar relationship is seen between the standard sort-merge algorithm and the pointer-based sort-merge algorithm.

One of the interesting things to notice about Graph 4.1 is just how poorly the pointer-based nested-loops algorithm performs. As mentioned earlier, this is because it does not try to optimize its disk reads of S like the other pointer-based algorithms. These results demonstrate that, even with pointers, something more intelligent than a nested-loops approach (or naive pointer traversal) is often needed for high performance.

As Graph 4.1 illustrates, the pointer-based hash-loops algorithm also performs very poorly, except when there is a large amount of memory available for the join. Like the pointer-based nested-loops algorithm, it performs poorly because a given page in S may end up being read several times. The curve for the pointer-based hash-loops algorithm takes on a "staircase" pattern due to the fact that its performance is largely a function of the number of iterations that it takes to read R; its performance improves as the number of iterations decreases. For small memory sizes, only a small increment in the size of memory is needed to reduce the number of times the algorithm iterates. As the size of memory increases, however, commensurately larger increments in the size of memory are needed to reduce the number of times the algorithm iterates.

Graph 4.2 shows the performance of the join algorithms when R is ten times the size of S. In this case, the pointer-based hybrid-hash algorithm can take up to three times as long to execute as the standard hybrid-hash algorithm. The large difference in performance is due to the fact that the pointer-based hybrid-hash algorithm always chooses R as its inner set, which turns out to be a poor choice in this case because R is so much bigger than S. Also note that the pointer-based nested-loops algorithm performs as well as the standard hybrid-hash algorithm when there are 2 Mbytes or more of memory. This is because all of S actually fits in memory in that case.

Another thing to note in Graph 4.2 is the similar performance of the standard hybrid-hash algorithm and the PID-partitioning algorithm. In both algorithms, the size of S determines how R is partitioned. Since the cost to

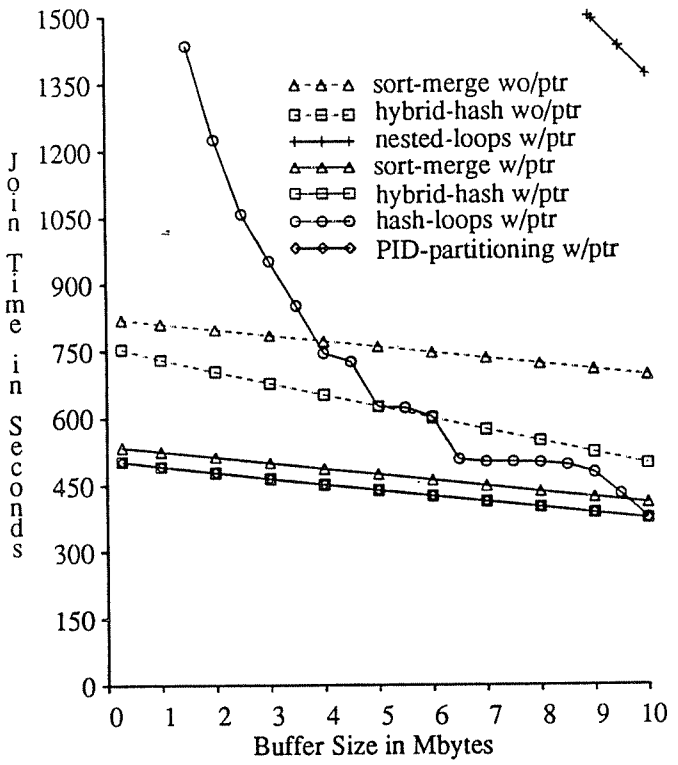
partition R dominates the cost of the join in this case, the performance of both algorithms is similar. The PID-partitioning performs slightly better when there are 2 Mbytes or less of memory because it does not partition S.

Turning to Graphs 4.3 and 4.4, we see many of the same trends that were seen in Graphs 4.1 and 4.2. In general, the relative performance of the algorithms depends on the size ratio of R and S. If R is roughly the same size as S or smaller, the pointer-based hybrid-hash algorithm performs the best. Otherwise, the standard hybrid-hash algorithm and the PID-partitioning algorithm perform the best. The pointer-based nested-loops algorithm and the pointer-based hash-loops algorithm perform poorly unless there is a large amount of memory available.

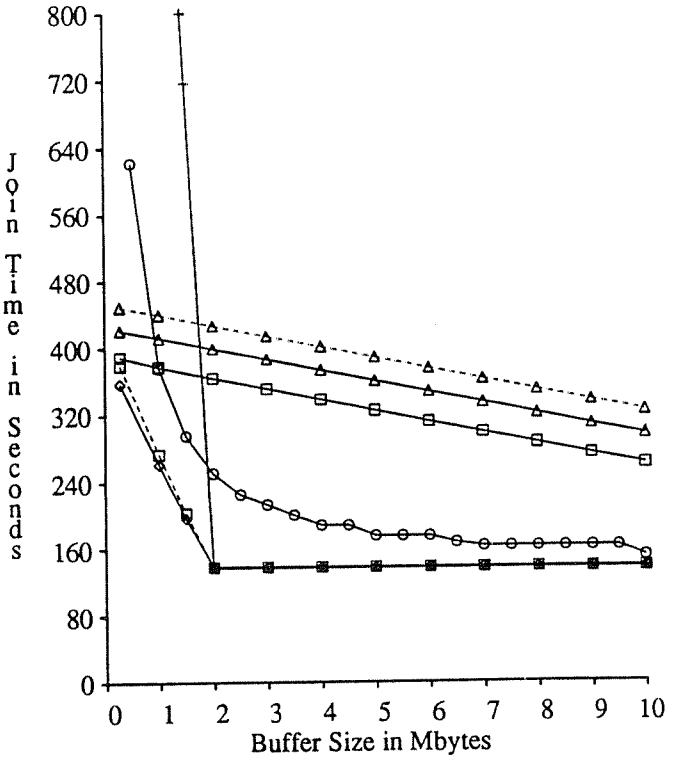
Graph 4.3 shows the performance of the algorithms when R is one fifth the size of S. In this case, the pointer-based hybrid-hash and sort-merge algorithms outperform their standard counterparts by up to 55%. This is because the relative cost to partition or sort S in the standard algorithms is larger in this case due to its increased size. The performance of the PID-partitioning algorithm is slightly worse than the pointer-based hybrid-hash algorithm because the size of S is used as the basis for partitioning. That turns out to be a poor choice in this case because R is smaller than S. The PID-partitioning algorithm still performs much better than the standard hybrid-hash algorithm, however, because it does not have to actually partition S.

Graph 4.4 shows the performance of the algorithms when R is twice the size of S. As shown, the PID-partitioning algorithm performs better than both the standard hybrid-hash algorithm and the pointer-based hybrid-hash algorithm in this case. It performs better than the standard hybrid-hash algorithm because it does not have to actually partition S, and it performs better than the pointer-based hybrid-hash algorithm because it uses the size of S (the smaller set) as the basis for partitioning R.

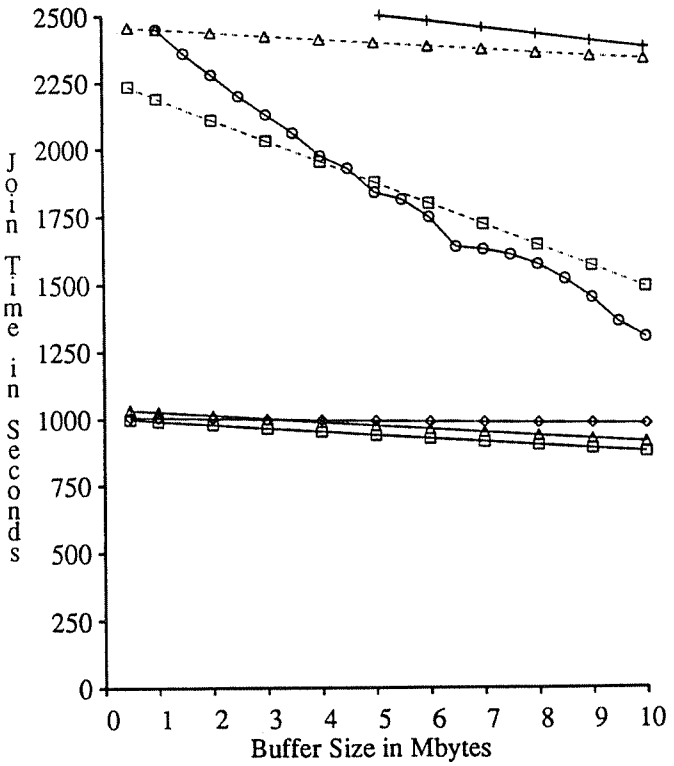
Perhaps the most interesting aspect of Graph 4.4 is the way the curves for the pointer-based and standard hybrid-hash algorithms cross each other. As shown, the pointer-based hybrid-hash algorithm performs better when there are 5 Mbytes of memory or less, but beyond that, the standard hybrid-hash algorithm performs better. The reason for the crossover is, again, because the pointer-based hybrid-hash algorithm always chooses R as its inner set, which is a poor choice in this case because R is larger than S. However, even with R as the inner set, the pointer-based hybrid-hash algorithm performs better initially because it does not have to partition S. With 5 Mbytes or more of memory, though, enough of S fits in memory so that the savings from not partitioning S are insufficient to offset the added costs of using R as the inner set; at that point, the standard hybrid-hash algorithm starts to perform better.



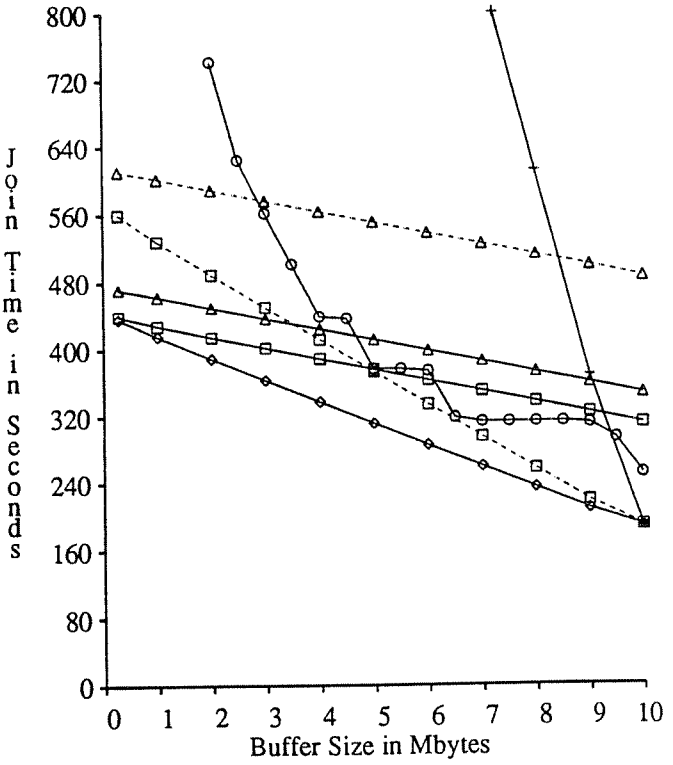
Graph 4.1: r = 200 s = 200 k = 1
|R| = 100,000 (~20 Mbytes) |S| = 100,000 (~20 Mbytes)



Graph 4.2: r = 200 s = 200 k = 10
|R| = 100,000 (~20 Mbytes) |S| = 10,000 (~2 Mbytes)



Graph 4.3: r = 200 s = 1,000 k = 1
|R| = 100,000 (~20 Mbytes) |S| = 100,000 (~100 Mbytes)



Graph 4.4: r = 200 s = 1,000 k = 10
|R| = 100,000 (~20 Mbytes) |S| = 10,000 (~10 Mbytes)

As noted earlier, we also obtained full-join results with the size of the disk transfer unit, P, set to 32 Kbytes in order to gauge the impact of sequential I/O. When this was done, the join times decreased by about 85% for the hash-based algorithms as well as for the PID-partitioning algorithm, and by about 80% for the sort-based algorithms (in all of the graphs). The join time for the pointer-based nested-loops algorithm decreased by about 10% except when S fit in memory, in which case its join time was again about the same as that of the standard hybrid-hash algorithm. These results demonstrate that the pointer-based algorithms benefit from sequential I/O as much as their standard counterparts do. The hash-based algorithms and the PID-partitioning algorithm benefit more than the sort-based algorithms because they are more I/O-bound. The pointer-based nested-loops algorithms benefits the least from sequential I/O because large disk transfers do not significantly lower its cost to read S, which is accessed in a random fashion.

Also note that, on the joins in which they perform the best, the CPU times of the pointer-based join algorithms are significantly less than those of their standard counterparts. (This is not always true on the joins in which they perform worse, however.) Consequently, in a CPU-bound system, the pointer-based algorithms could improve performance by even larger margins than those suggested by Graphs 4.1-4.4. Table 4.3 gives a rough feeling for how widely the CPU times differ among the join algorithms. That table shows the CPU time of each algorithm for the join in Graph 4.3 with 1, 5, and 10 Mbytes of memory.

Before turning to the results for medium-sized joins, it is important to note that the kind of pointers that have been assumed can always be treated like surrogates and used to perform a value-based hybrid-hash join; that is, the

	Memory Size		
Algorithm	**1 Mbytes**	**5 Mbytes**	**10 Mbytes**
sort-merge wo/ptr	206.47	206.34	206.16
hybrid-hash wo/ptr	9.96	8.41	6.49
nested-loops w/ptr	6.97	6.69	6.36
sort-merge w/ptr	36.41	36.28	36.10
hybrid-hash w/ptr	3.37	3.22	3.04
hash-loops w/ptr	6.55	4.92	3.50
PID-partitioning w/ptr	3.59	3.46	3.35

Table 4.3: CPU Times for Graph 4.3 (in seconds)

pointer of each object in R can be treated like a foreign key for S. This becomes useful in situations where the PID-partitioning algorithm is not available, and where the standard hybrid-hash algorithm performs better than the pointer-based hybrid-hash algorithm (see Graphs 4.2 and 4.4). Since pointers can be treated like surrogates in those situations, foreign keys do not necessarily have to be stored in the objects of R in order to obtain optimal performance on full joins.

Finally, as Graphs 4.1-4.4 illustrate, the standard hybrid-hash algorithm and the PID-partitioning algorithm generally result in the best performance when S is smaller than R. To execute properly, these algorithms need to know about both of the sets that participate in the join. In some of the newer data models that have been proposed, however, the set that plays the role of S is not always known (e.g., this is true of the EXTRA data model [Care88]). For example, consider the following join, which uses the employee-job database presented earlier in this chapter:

retrieve (Emp.name, Emp.age, Emp.job.name)

In this join, Emp plays the role of R, but the set that plays the role of S is not explicitly identified. Therefore, unless there is additional information at the data model level that can be used to determine the set that plays the role of S, the query optimizer may have no choice but to use a join algorithm that results in sub-optimal performance. This is in contrast to a relational data model, where the sets that play the roles of R and S are identified in all queries.

4.3.6. Performance Results for Medium-Sized Joins

The results for medium-sized joins are presented Graphs 4.5-4.8. In all of these graphs, the size of R was set at 100,000 objects, and the selectivity of the predicate on R was kept fixed at 0.01. Consequently, each graph represents the join of 1,000 objects in R with S. As shown, the size of memory was increased in 1/8 Mbyte increments and ranged from the minimal size required to hold the selection on R in memory, which was approximately 1/4 Mbytes, all the way up to 2 Mbytes. As mentioned earlier, the PID-partitioning algorithm is generally impractical for small and medium-sized joins, so it has not been included here. The pointer-based hash-loops algorithm is not included here either, since its performance would be identical to that of the pointer-based hybrid-hash algorithm. This holds true for the small joins as well.

Graphs 4.5 and 4.6 are for the clustered/unclustered index combination, where the index on R is a clustered index and the index on S is an unclustered index. This is indicated by the label "clus/unclus." These graphs

represent the situation where the pointer-based algorithms perform their best in relation to the index-nested-loops algorithm, since the relative cost of accessing S via its index is high in this case.

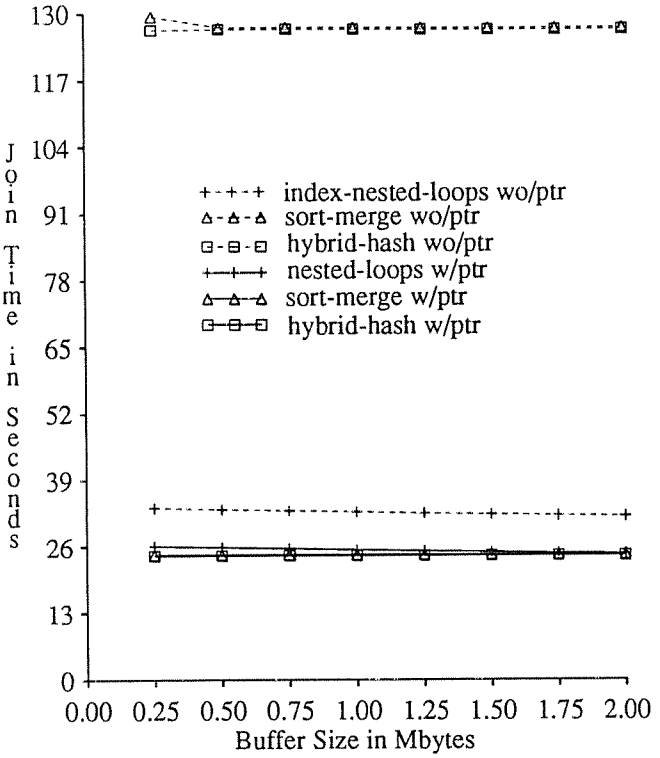
Graph 4.5 shows the performance of the join algorithms when R is the same size as S. Compared to the index-nested-loops algorithm, the pointer-based algorithms reduce the join time by approximately 25% over the whole range of memory sizes considered. This, of course, is because the index on S is not read by the pointer-based algorithms. Note that the standard sort-merge and hybrid-hash algorithms perform poorly here because, unlike their pointer-based counterparts, they read all of S, even though only a small fraction of S actually participates in the join.

Graph 4.6 shows the performance of the join algorithms when R is ten times the size of S. In this case, the fraction of S that participates in the join is large enough so that both the index-nested-loops algorithm and the pointer-based nested-loops algorithm perform poorly in relation to the other join algorithms. This is because neither of these algorithms tries to optimize its disk reads of S. Compared to the standard hybrid-hash algorithm, the pointer-based hybrid-hash algorithm reduces the join time by approximately 10% over the whole range of memory sizes considered. A similar relationship is seen between the standard sort-merge algorithm and the pointer-based sort-merge algorithm.

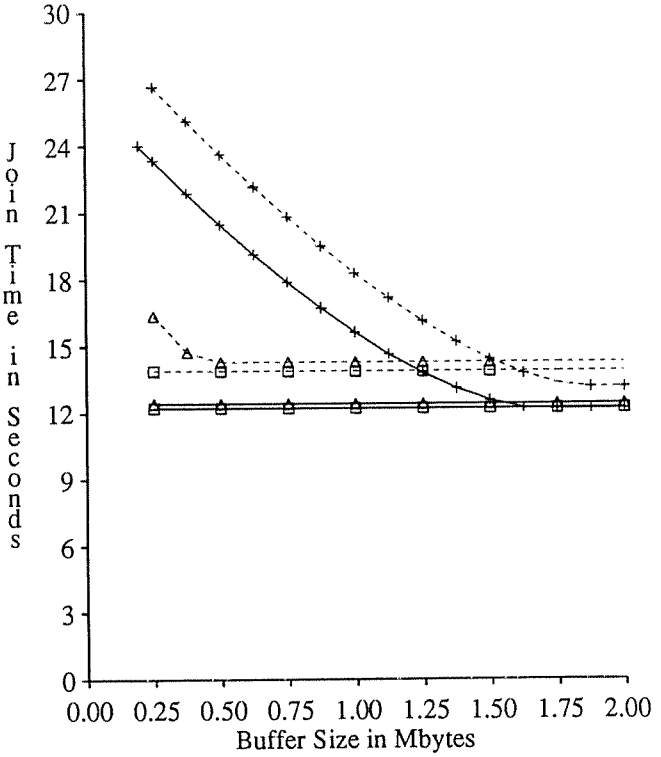
Graphs 4.7 and 4.8 are for the same cases as in the top half of the figure but with the unclustered/clustered index combination, which is denoted as "unclus/clus." It should be clear that these graphs represent the situation where the pointer-based algorithms perform their worst in relation to the index-nested-loops algorithm. This is because the relative cost of accessing S via its index is low in this case. As shown, the pointer-based algorithms perform approximately 15% better than the index-nested-loops algorithm in Graph 4.7 and approximately 5% better in Graph 4.8.

The graphs for the clustered/clustered and unclustered/unclustered index combinations have not been presented here. Although those graphs do not follow quite the same patterns as those in Graphs 4.5-4.8, similar benefits and tradeoffs were observed; the pointer-based algorithms always outperformed their standard counterparts.

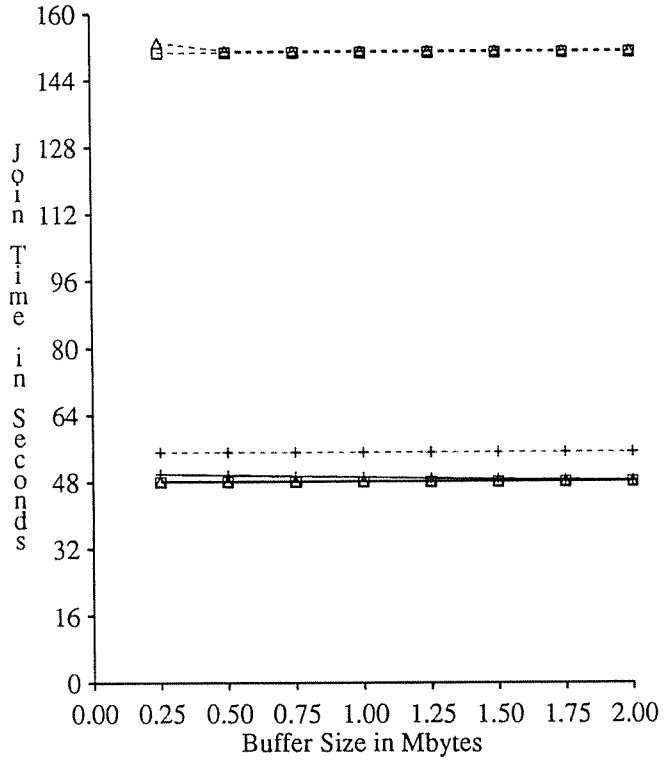
Before turning to the results for small joins, it is important to emphasize that the CPU times of the pointer-based algorithms are significantly less than their standard counterparts on all of the medium-sized joins. This holds true for the small joins as well. In Graph 4.5, for example, the CPU time for the index-nested-loops algorithm is 215.27 msec, whereas it is only 0.34 msec for the pointer-based nested-loops algorithm. In a CPU-bound system, such a large a difference in CPU times could have a significant impact on performance.



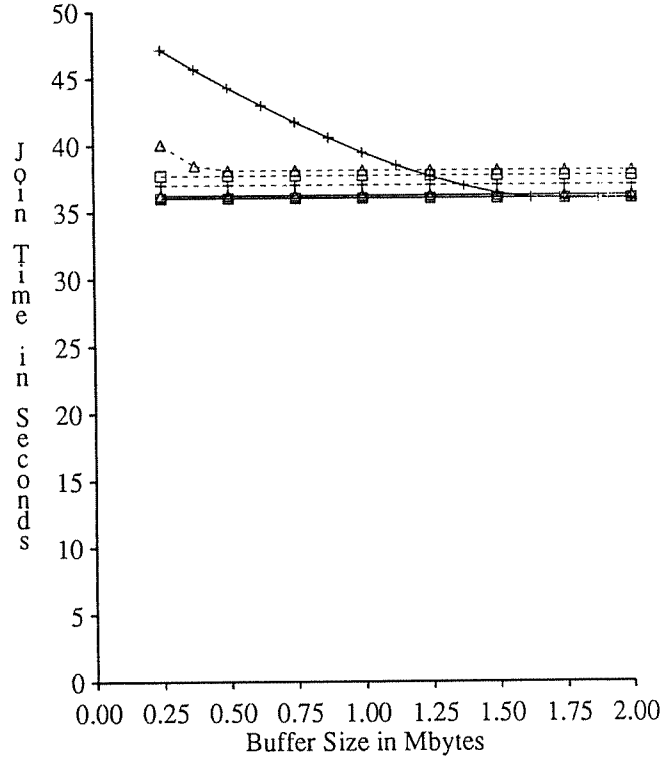
Graph 4.5: clus/unclus $r = 200$ $s = 200$ $k = 1$
|R| = 100,000 (~20 Mbytes) |S| = 100,000 (~20 Mbytes)



Graph 4.6: clus/unclus $r = 200$ $s = 200$ $k = 10$
|R| = 100,000 (~20 Mbytes) |S| = 10,000 (~2 Mbytes)



Graph 4.7: unclus/clus $r = 200$ $s = 200$ $k = 1$
|R| = 100,000 (~20 Mbytes) |S| = 100,000 (~20 Mbytes)



Graph 4.8: unclus/clus $r = 200$ $s = 200$ $k = 10$
|R| = 100,000 (~20 Mbytes) |S| = 10,000 (~2 Mbytes)

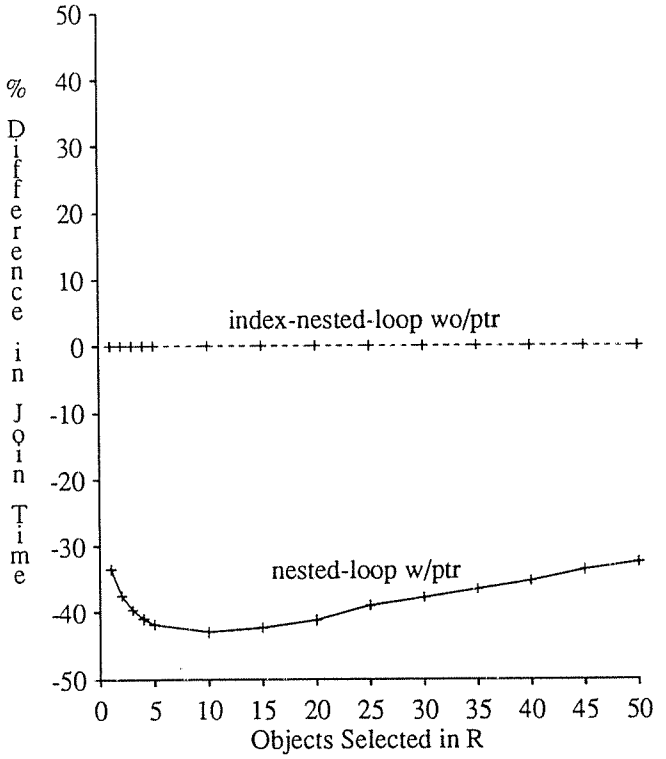
4.3.7. Performance Results for Small Joins

The results for small joins are presented in Graphs 4.9-4.12. Only the index-nested-loops algorithm and the pointer-based nested-loops algorithm are compared here because they perform as well as or better than the other algorithms in their respective classes on small joins. In all of the graphs, the size of memory was kept fixed at 1/4 Mbytes.

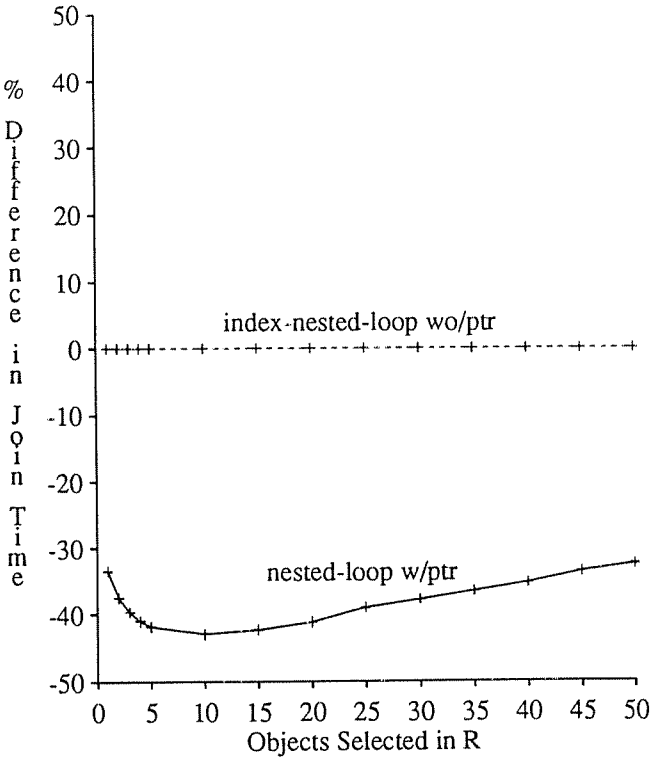
As indicated, the results in Graphs 4.9-4.12 are shown in percentage terms. The total time to execute the join using the index-nested-loops algorithm was computed and the percentage difference between that time and the time to execute the pointer-based nested-loops algorithm was then plotted. The reason for displaying the results in this manner is to make the performance differences between the two algorithms clearer. This was not done for the previous results because they were not uniform enough to make this a viable approach.

Graphs 4.9 and 4.10 are for the clustered/unclustered index combination. Graph 4.9 shows the performance of the join algorithms when R is the same size as S. As shown, the pointer-based nested-loops algorithm reduces the join time by almost 50% when more than 10 objects in R are joined with S. In this case, the cost to execute the index-nested-loops algorithm is dominated by the I/O cost to read the index on S and S itself, with roughly the same number of pages being read from the index and S. Therefore, the cost to read the index on S accounts for about one half of the cost to execute the index-nested-loops algorithm. Since the pointer-based nested-loops algorithm eliminates the cost to read the index on S, it effectively reduces the cost of the join by 50%.

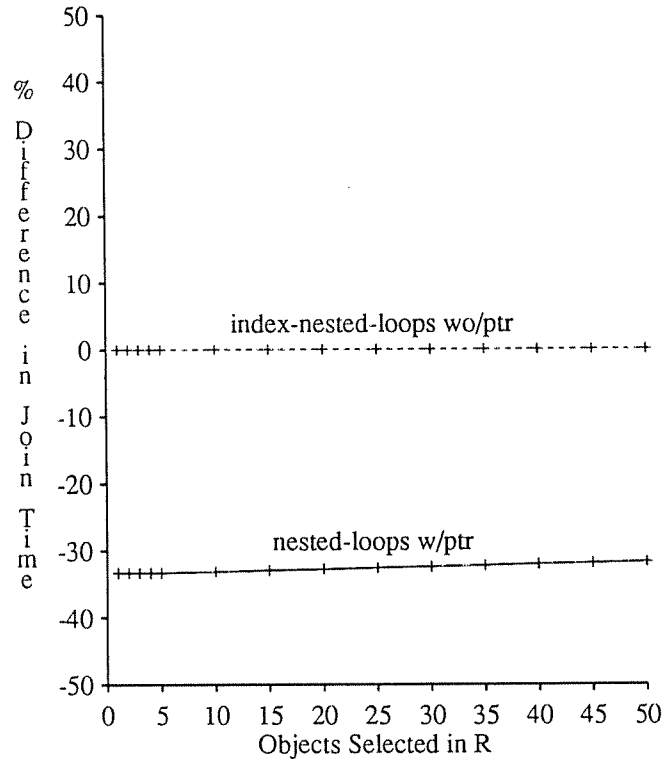
Graph 4.10 shows the performance of the join algorithms when R is ten times the size of S. As shown, the performance benefit of the pointer-based nested-loops algorithm decreases as the size of the join increases. In this case, the index on S is small enough so that some of its leaf pages end up being referenced more than once by the index probes of the index-nested-loops algorithm. (Recall that in the index-nested-loops algorithm, a given leaf page L_i may be referenced several times, but because the result of the selection on R is sorted by the join attribute, L_i is guaranteed to be read from disk only once even if it is referenced several times.) In contrast to its index, however, S is still large enough in this case so that each probe of S causes a different page in S to be accessed. The net effect is that as the size of the join increases, the proportional cost of the index probes in the index-nested-loops algorithm decreases. This in turn causes the performance benefit of the pointer-based nested-loops algorithm to decrease somewhat.



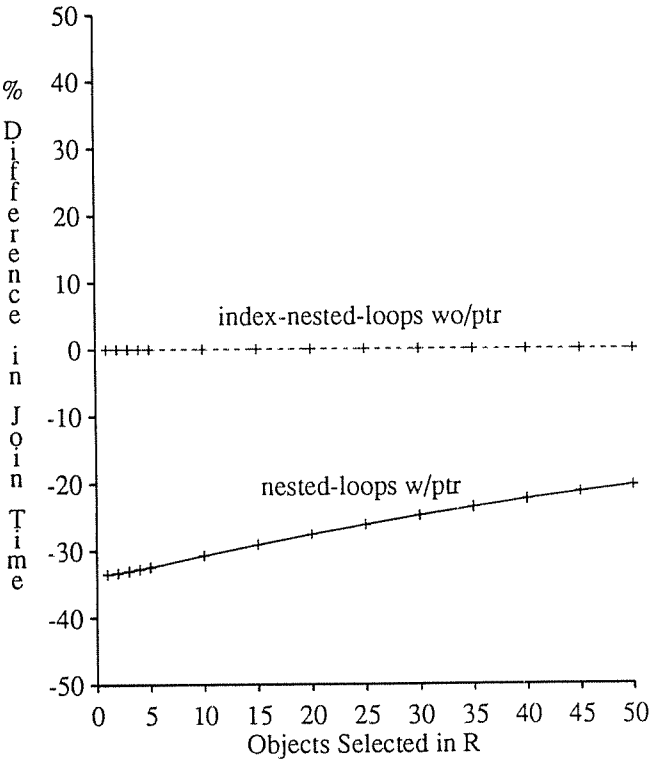
Graph 4.9: clus/unclus r = 200 s = 200 k = 1
|R| = 100,000 (~20 Mbytes) |S| = 100,000 (~20 Mbytes)



Graph 4.10: clus/unclus r = 200 s = 200 k = 10
|R| = 100,000 (~20 Mbytes) |S| = 10,000 (~2 Mbytes)



Graph 4.11: unclus/clus r = 200 s = 200 k = 1
|R| = 100,000 (~20 Mbytes) |S| = 100,000 (~20 Mbytes)



Graph 4.12: unclus/clus r = 200 s = 200 k = 10
|R| = 100,000 (~20 Mbytes) |S| = 10,000 (~2 Mbytes)

Finally, Graphs 4.11 and 4.12 are for the unclustered/clustered index combination. The differences between these graphs and Graphs 4.9 and 4.10 are due to the fact that an unclustered index is used to read R. In this case, the cost to execute the index-nested-loops algorithm is dominated by the I/O cost to read R, the index on S, and S itself, with roughly the same number of pages being read from the index and S. Therefore, the cost to read the index on S accounts for about one third of the cost to execute the index-nested-loops algorithm. Since the pointer-based nested-loops algorithm eliminates the cost to read the index on S, it effectively reduces the cost of the join by 33%. The curve for the pointer-based nested-loops algorithm has an upward slope in Graph 4.12 for the same reason it has an upward slope in Graph 4.10.

Once again, the graphs for the clustered/clustered and unclustered/unclustered index combinations have not been presented here. Although the graphs for those index combinations do not follow quite the same patterns as those in Graphs 4.9-4.12, similar benefits and tradeoffs were observed; the pointer-based algorithms always outperformed their standard counterparts by 30% or more.

4.4. USING BIDIRECTIONAL POINTER STRUCTURES

As mentioned earlier, the simple many-to-one pointer structure that has been assumed cannot be used effectively on select-project-join queries in which there is a selection predicate on S that is more restrictive than any predicate on R. To process such joins with pointers, a more complicated pointer structure is needed, where each object in S is somehow linked to the objects in R that are related to it. One possibility is the kind of Codasyl-like pointer structure described in [Care90]. With the pointer structure described there, each record *s* in S would point to the first of its related objects in R, while each related object in R would point to *s* as well as the next and previous objects in R that reference *s*.

It should be clear that our pointer-based join algorithms can make use of the pointers that run from R to S in a Codasyl-like pointer structure, and in fact the pointer-based sort-merge algorithm was one of the join algorithms implemented and studied in [Care90]. Unfortunately, our pointer-based algorithms cannot make use of the "pointer chains" that link each object in S to its related objects in R. There is actually little that can be done with such pointer chains other than to traverse them. For small to medium-sized joins that fit in memory, a join algorithm based on traversing pointer chains can result in good performance, as shown in [Care90]. For joins that do not fit in memory, however, such a join algorithm will obviously suffer from poor performance unless R and S are clustered in the same order or clustered together in the same file. To efficiently process joins that do not fit in memory, some other

pointer structure is required instead. One possibility is an embedded bidirectional pointer structure, where each object in R contains an pointer linking it to its related object in S, and where each object in S contains a set of pointers linking it to its related objects in R. In this section, we will analyze two pointer-based join algorithms that can make use of such a pointer structure.

It should be clear that all of the pointer-based join algorithms which have been described can make use of the pointers that run from R to S in a bidirectional pointer structure. Therefore, the question is how to make effective use of the pointers that run from S to R. Unfortunately, our pointer-based hybrid-hash, sort-merge, and PID-partitioning join algorithms cannot, in general, make use of the pointers from S to R. This is because those algorithms do their sorting or partitioning based on one pointer value per object, whereas here each object in S can have several pointers. The only pointer-based join algorithms that can make effective use of the pointers that run from S to R in a bidirectional pointer structure are the pointer-based nested-loops algorithm and the pointer-based hash-loops algorithm. The modifications that need to be made to these algorithms so that they can make use of a bidirectional pointer structure are described below.

4.4.1. Pointer-Based Nested-Loops

Only a slight modification needs to be made to the pointer-based nested-loops algorithm in order to make it work with a bidirectional pointer structure. In the modified version of the algorithm, the larger of R and S plays the role of the outer set to minimize the probability of a buffer miss on the inner set. This in turn minimizes the number of I/Os generated. When R (or the result of a selection on R) plays the role of the outer set, the algorithm executes in exactly the same manner that was described earlier, with the pointers from R to S being used. When S (or the result of a selection on S) plays the role of the outer set, the algorithm executes in much the same manner that was described earlier, but the roles of R and S are exchanged and the pointers from S to R are now used.

4.4.2. Pointer-Based Hash-Loops

Like the pointer-based nested-loops algorithm, only a slight modification needs to be made to the pointer-based hash-loops algorithm in order to make it work with a bidirectional pointer structure. In the modified version of the algorithm, the smaller of R and S plays the role of the outer set to minimize the number of times the algorithm iterates. This in turn minimizes the number of I/Os generated. When R (or the result of a selection on R) plays the role of the outer set, the algorithm executes in exactly the same manner that was described earlier, with the pointers from R to S being used. When S (or the result of a selection on S) plays the role of the outer set, the algorithm

executes in much the same manner that was described earlier, but the roles of R and S are exchanged and the pointers from S to R are now used.

One thing to note is that in the previous description of the pointer-based hash-loops algorithm we assumed that each object was hashed on exactly one pointer value, and the hash table was structured accordingly. Here, however, each object in S may be hashed on more than one pointer value. Consequently, the structure of the hash table that was described earlier has to be changed slightly to allow for this possibility. This is accomplished by using paired hash entries, as shown in Figure 4.3. When an object s_i of S is processed, each of the pointers stored in s_i is hashed on its PID values. Let $r_i.ptr$ be one of the pointers stored in s_i. When $r_i.ptr$ is hashed on its PID value, a paired entry of the form $s_i.ptr,r_i.ptr$ is put in the hash table, where $s_i.ptr$ is a pointer to s_i in memory and $r_i.ptr$ is a pointer to r_i on disk. Once the hash table has been built, it is processed in much the same manner that was described earlier. Note that, to conserve space, the version of s_i that is stored in memory can be stripped of its pointers to R.

4.4.3. Analysis with Bidirectional Pointer Structures

The results of the previous section showed that pointers can be used to improve the performance of many joins, and in cases where they offer no advantage they can still be treated like surrogates and used to perform a value-based join. In this section, we will assume that pointers are desired, and that a bidirectional pointer structure exists between R and S so that joins with either a selection on R or S can be handled efficiently. As in the previous section, an analysis is performed to quantitatively compare the different pointer-based join algorithms when a bidirectional pointer structure exists.

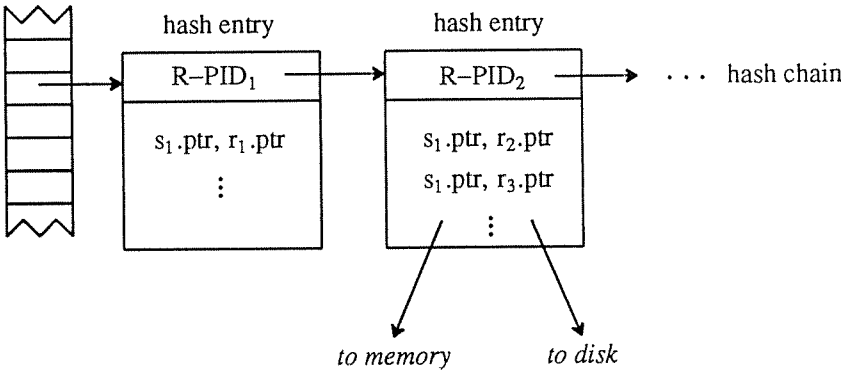


Figure 4.3: The Structure of the Hash Table

To avoid being tedious, the analysis and results for small to medium-sized joins are omitted in this section. Under our assumption that small to medium-sized joins fit in memory, it should be clear that the analysis and performance of the pointer-based nested-loops algorithm and the pointer-based hash-loops algorithm would be similar to that presented earlier for such joins. Consequently, only large, full-relation joins are considered in this section. We will also omit performance results for the standard, value-based join algorithms in this section, except when they perform the best. In those cases, we will assume that pointers are being used like surrogates. The analysis of the pointer-based nested-loops algorithm and the pointer-based hash-loops algorithm for bidirectional pointer structures follows. As one might expect, much of the analysis is similar to that presented earlier for each algorithm.

4.4.3.1. Pointer-Based Nested-Loops

Recall that the larger of R and S plays the role of the outer set in the pointer-based nested-loops algorithm when a bidirectional pointer structure is used. When R plays the role of the outer set, the analysis is the same as that presented earlier. When S plays the role of the outer set, however, we have to account for the fact that each object in S can now contain more than one pointer. Because of the way that memory is allocated, and because R and S are relatively unclustered, this causes R to be accessed in exactly the same manner as it would be in an unclustered index scan of R when there are duplicate index entries. The I/O cost of performing an unclustered index scan with duplicates was derived in [Mack89]. Making use of the equations in that paper, the function $U_r(x, B)$, which counts the expected number of I/Os that are generated when the pointers to x objects in S are dereferenced and a buffer of B pages is used, is defined as:

$$U_r(x, B) = \begin{cases} P_r \cdot (1 - q^x) & \text{if } x \le n \\ P_r \cdot (1 - q^n) + (x - n) \cdot P_r \cdot p \cdot q^n & \text{if } x > n \end{cases}$$

where

$$q \equiv 1 - p = \begin{cases} \left(\dfrac{|R| - O_r}{|R|} \right)^k & \text{if } k \le O_r \\ \left(\dfrac{|R| - k}{|R|} \right)^{O_r} & \text{if } k > O_r \end{cases}$$

$$n = max\left\{ j \text{ in } \{0, 1, ..., x\} \text{ such that } P_r \cdot (1 - q^j) \le B \right\}$$

The function $U_r( )$ is similar to the function $U_s( )$ that was defined earlier. The rest of the analysis follows directly from the analysis that was presented earlier. Taking that analysis and making some straightforward changes, the

cost of the pointer-based nested-loops algorithm with a bidirectional pointer structure is:

if $P_s \le P_r$ *then*	
$P_r \cdot IO$	read R
$+ U_s(\lvert R \rvert, M-1) \cdot IO$	access S
$+ \lvert R \rvert \cdot hash_1$	check the buffer for each pointer dereference
else	
$P_s \cdot IO$	read S
$+ U_r(\lvert S \rvert, M-1) \cdot IO$	access R
$+ k \cdot \lvert S \rvert \cdot hash_1$	check the buffer for each pointer dereference

4.4.3.2. Pointer-Based Hash-Loops

Recall that the smaller of R and S plays the role of the outer set in the pointer-based hash-loops algorithm when a bidirectional pointer structure is used. When R plays the role of the outer set, the analysis is the same as that presented earlier. When S plays the role of the outer set, however, we again have to account for the fact that each object in S can now contain more than one pointer. This is handled by a straightforward modification of the function $T_s(\,)$, which counts the expected number of I/Os generated for S by the hash-loops algorithm when R plays the role of the outer set. Taking the definition for $T_s(\,)$ and making the appropriate changes, we define the function $T_r(\,)$, which counts the expected number of I/Os generated for R by the hash-loops algorithm when a buffer of B pages is used and S plays the role of the outer set:

$$T_r(B) = (n-1) \cdot P_r \cdot Y(\lvert R \rvert, O_r, k \cdot x) + P_r \cdot Y(\lvert R \rvert, O_r, k \cdot y)$$

where

$$n = \lceil P_s \,/\, B \rceil$$
$$x = B \cdot O_s$$
$$y = \lvert S \rvert - (n-1) \cdot x$$

The rest of the analysis follows directly from the analysis that was presented earlier. Taking that analysis and making some straightforward changes, the cost of the pointer-based hash-loops algorithm with a bidirectional pointer structure is:

if $P_r \leq P_s$ *then*	
$P_r \cdot IO$	read R
$+ T_s(M-1) \cdot IO$	access S
$+ \lvert R \rvert \cdot hash_1$	hash the objects in each R_i, $1 \leq i \leq n$
else	
$P_s \cdot IO$	read S
$+ T_r(M-1) \cdot IO$	access R
$+ k \cdot \lvert S \rvert \cdot hash_1$	hash the objects in each S_i, $1 \leq i \leq n$

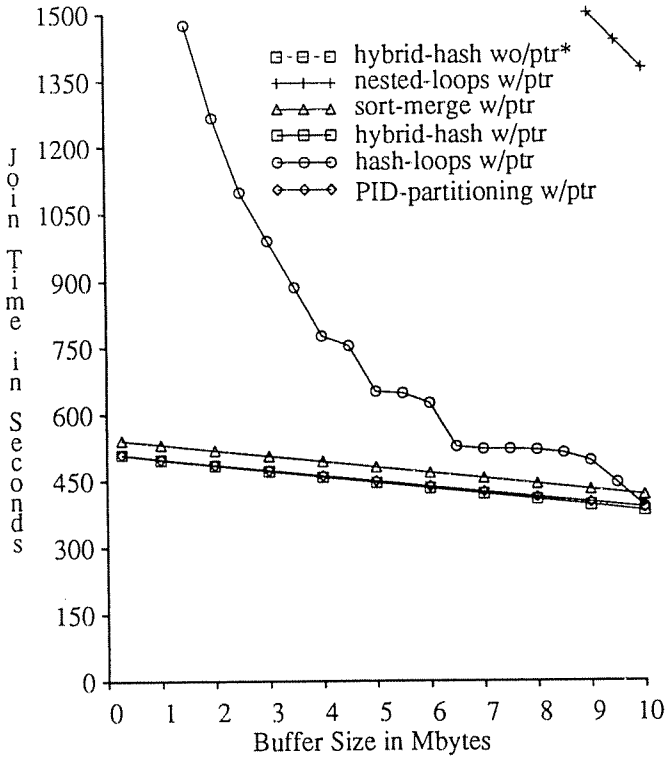
4.4.4. Performance Results for Bidirectional Pointer Structures

The results for full joins with a bidirectional pointer structure are presented in Graphs 4.13-4.16. Only results for pointer-based joins are presented, except in the cases where the standard hybrid-hash algorithm performs the best. In the results for the standard hybrid-hash algorithm, we are assuming that pointers are treated like surrogates; this is indicated by the asterisk that accompanies the label for the standard hybrid-hash algorithm. As noted above, only the pointer-based nested-loops algorithm and the pointer-based hash-loops algorithm can take advantage of the pointers from S to R.

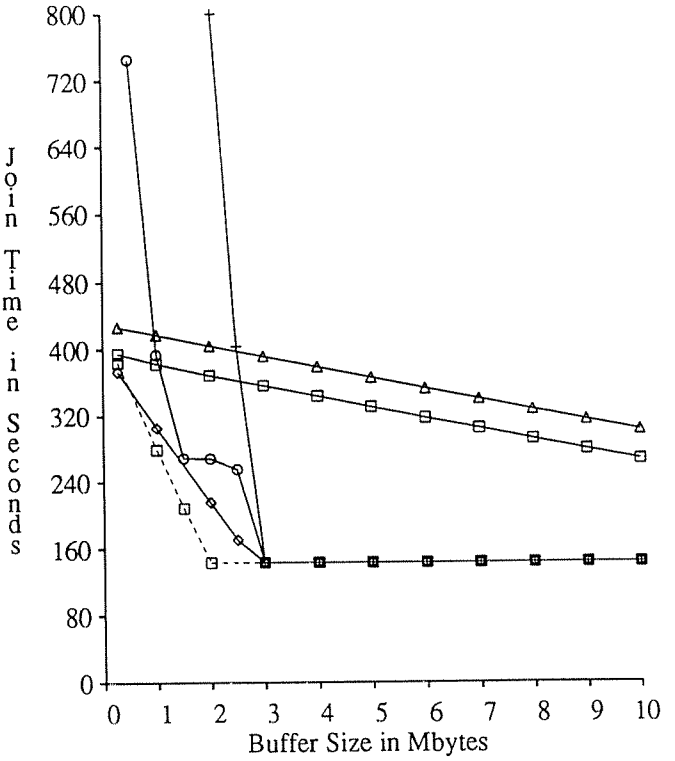
Graphs 4.13-4.16 cover the same cases as Graphs 4.1-4.4. Unlike Graphs 4.1-4.4, we have accounted for the extra space consumed by pointers in Graphs 4.13-4.16. This has been done because in the case where the sharing level k is equal to 10, the extra space consumed by pointers in S can make a difference in the performance of the algorithms. More will be said about this shortly. Here, we have assumed that each pointer is 8 bytes long.

Comparing Graphs 4.13-4.16 to Graphs 4.1-4.4, we see more or less the same trends that were seen earlier. As shown, the bidirectional pointer structure offers little improvement for the pointer-based nested-loops algorithm and the pointer-based hash-loops algorithm. Even with a bidirectional pointer structure, they perform very poorly unless there is a large amount of memory. This, of course, is because both algorithms still do a poor job of optimizing their disk reads.

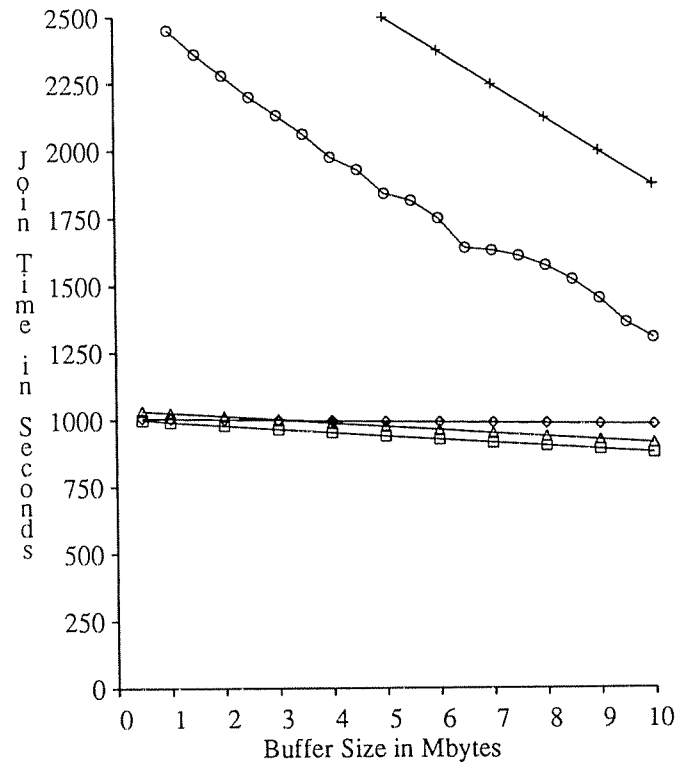
The results shown in Graphs 4.13 and 4.15 are quite similar to what was seen earlier and need no further comment. Graphs 4.14 and 4.16, on the other hand, reveal slightly different trends from what was seen before. Whereas the PID-partitioning algorithm performed the best in the earlier graphs when S was smaller than R, here we see that in some cases the standard hybrid-hash algorithm performs the best. This is because of the the extra space consumed by the pointers in S. In the standard hybrid-hash algorithm, these can be stripped from S as it is partitioned, whereas this cannot be done in the PID-partitioning algorithm. By stripping pointers in this manner, the standard



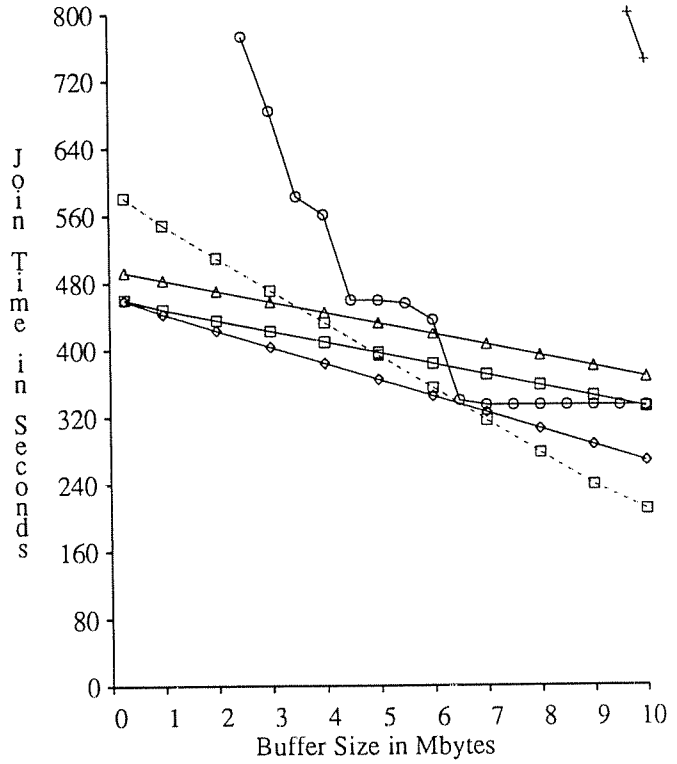
Graph 4.13: r = 208 s = 208 k = 1
|R| = 100,000 (~20.8 Mbytes) |S| = 100,000 (~20.8 Mbytes)



Graph 4.14: r = 208 s = 280 k = 10
|R| = 100,000 (~20.8 Mbytes) |S| = 10,000 (~2.8 Mbytes)



Graph 4.15: r = 208 s = 1,008 k = 1
|R| = 100,000 (~20.8 Mbytes) |S| = 100,000 (~100.8 Mbytes)



Graph 4.16: r = 208 s = 1,080 k = 10
|R| = 100,000 (~20.8 Mbytes) |S| = 10,000 (~10.8 Mbytes)

hybrid-hash algorithm ends up dividing R into fewer partitions, which in turn can drive down its I/O costs to the point where it starts performing better than the PID-partitioning algorithm.

The effect of stripping pointers is readily seen in Graph 4.14. Because of the space consumed by pointers in S, the curves for the pointer-based nested-loops, hash-loops, and PID-partitioning algorithms do not flatten out until there is 3 Mbytes of memory, at which point S fits in memory. In contrast, the curve for the standard hybrid-hash algorithm flattens out when there is 2 Mbytes of memory, as the the stripped version of S fits in 2 Mbytes.

Taking a step back, what Graphs 4.13-4.16 are saying, in effect, is that it may not be worthwhile to maintain a bidirectional pointer structure in some cases. In practice, a Codasyl-like pointer structure would probably be less costly to maintain. Moreover, a Codasyl-like pointer structure would be just as efficient on full joins or select-project-join queries in which there is a selection predicate on R that is more restrictive than any predicate on S. This, of course, is because the pointers from R to S are also maintained in a Codasyl-like pointer structure. The only case where a Codasyl-like pointer structure might not be as efficient as a bidirectional pointer structure is on select-project-join queries in which there is a a selection predicate on S that is more restrictive than any predicate on R. If the size of such queries tends to be small, however, a Codasyl-like pointer structure can still result in acceptable performance, as shown in [Care90].

4.5. CONCLUSIONS

In this chapter, we described how pointers can be used effectively in join processing. We described several pointer-based join algorithms that are simple variations on the well known nested-loops, sort-merge, hybrid-hash, and hash-loops join algorithms. We also described a pointer-based join algorithm called PID-partitioning, which has no standard counterpart and is somewhat less general than the other pointer-based join algorithms. In this chapter, we showed that, given the appropriate pointer structures, many common types of joins can be executed using our pointer-based algorithms.

For much of the chapter, an analysis was carried out to compare the pointer-based join algorithms to their standard, value-based counterparts. In addition to providing a basis for comparison, the cost equations that were derived in the analysis can also be used in query optimization and physical database design. Initially, the join of two sets R and S in a many-to-one relationship was studied. A simple pointer structure was assumed, where each object in R contained a pointer to its related object in S. Two types of joins were analyzed: full joins of R and S, and small to medium-sized joins with a selection predicate on R.

For full joins, the results of the analysis showed that the pointer-based sort-merge, hybrid-hash, and PID-partitioning algorithms can provide savings of 30% or more when the size of R (in pages) is roughly the same size as S or smaller. These results are due to the fact that S does not need to be sorted or partitioned in the pointer-based algorithms. When the size of R is larger than the size of S, we found that the standard hybrid-hash and the PID-partitioning algorithm performed the best. We also found that the pointer-based nested-loops algorithm and the pointer-based hash-loops algorithm perform very poorly in almost all cases. This indicates that, to make effective use of pointers, something more intelligent than a simple nested-loops approach (i.e., naive pointer traversal) or hash-loops approach is needed for high performance on large joins.

For medium-sized joins, where 1% of the objects in R were joined with S, the results showed that the pointer-based sort-merge and hybrid-hash algorithms always outperformed their standard counterparts, providing gains of up to 25% in many cases. The pointer-based nested-loops algorithm was again shown to perform poorly, although not in all cases. Finally, for small joins, where 0.01% of the objects in R were joined with S, all of the pointer-based join algorithms outperformed their standard counterparts by 30% or more. Both of these results are largely due to the fact that index lookups on S are eliminated in the pointer-based algorithms. The conclusion to be drawn here is that, for small to medium-sized joins, which are probably a very common type of join, pointer-based join algorithms can provide large performance gains. This conclusion was verified in [Care90], where experimental results showed that pointer-based joins can indeed provide significant performance gains on small to medium-sized joins.

In the latter part of the chapter, we examined whether a bidirectional pointer structure could be used effectively in join processing. There, we assumed that each object in R contained a pointer to its related object in S, and that each object in S contained pointers to its related objects in R. The advantage of a bidirectional pointer structure is that it can be used on select-project-join queries in which there is a selection predicate on S that is more restrictive than any predicate on R. This is in contrast to the simple many-to-one pointer structure we assumed earlier in the chapter, which cannot be used on such joins.

On small to medium-sized joins, it should be clear that, by eliminating index lookups, a bidirectional pointer structure will provide similar performance gains to those obtained with a many-to-one pointer structure. Consequently, only full joins with a bidirectional pointer structure were analyzed. Surprisingly, the results for full joins showed that a bidirectional pointer structure could not be used effectively on full joins. The only pointer-based join algorithms that can make full use of a bidirectional pointer structure are the pointer-based nested-loops algorithm

and the pointer-based hash-loops algorithm, and these were found to perform very poorly in almost all cases. The conclusion to be drawn here is that it may not always be worthwhile to maintain a bidirectional pointer structure. A Codasyl-like pointer structure such as the one described in [Care90] offers many of the same advantages and would probably be simpler to maintain and implement. Some combination of a many-to-one pointer structure and foreign keys may be another viable approach. The choice of which pointer structure to use will obviously be important in many of the emerging data models, which typically support references in one form or another. The results presented in this chapter should help in making the correct choice.

4.5.1. Directions for Future Work

As far as future work is concerned, it would obviously be interesting to obtain implementation results for the pointer-based joins described in this chapter. Although some implementation results were obtained in [Care90], only the pointer-based sort-merge and nested-loops algorithms were examined there. Since most of the pointer-based join algorithms that were described in this chapter are simple variations on well known value-based join algorithms, it should be fairly easy to start with the code for a value-based algorithm and convert it to a pointer-based algorithm.

Another possibility for future work would be to examine whether the index AND'ing techniques described in [Moha90] for optimizing selections with multiple predicates could be used with pointer-based joins. For example, the pointers in index entries could be AND'ed with the pointers in objects to filter out the objects that do not participate in the join at an early processing stage, much like in bit-filtering. This might be useful in joins where there is a selection predicate on both sets being joined.

Finally, it would be interesting to examine how to parallelize the pointer-based join algorithms described in this chapter. It would appear that little that can be done with them in a shared-nothing multiprocessor since inter-node pointers probably do not make much sense. In a shared-memory multiprocessor, however, many of the techniques described in [Grae90] may be applicable.

CHAPTER 5

THE CRICKET STORAGE SYSTEM

As mentioned in the introduction of this thesis, field replication and pointer-based joins are both techniques for improving the performance of database systems on fairly traditional access patterns, where large sets of related data items are accessed via a non-procedural query language. The computational requirements and access patterns of emerging applications such as CAD/CAM, however, can be quite different from those of traditional database applications [Katz87, Chan89a, Chan89b, Catt90]. Non-procedural query languages appear inappropriate for such applications, and fast, low-level navigation is required to obtain adequate levels of performance [Maie89]. For this reason, almost every one of the next-generation database systems that has been commercially marketed features a persistent programming language that supports low-level, procedural navigation (e.g., Objectivity's Objectivity/DB [Obje90], Object Design's ObjectStore [Atwo90], Servio Logic's Gemstone System [Maie86a], and Versant's Object Manager [Vers90]).

To efficiently support applications such as CAD/CAM, it may not be sufficient to simply put a persistent programming language on top of a traditional database storage system. Instead, a different approach to storage management may be required, as argued in [Maie89, Catt90]. In this chapter, we describe a prototype storage system called Cricket that was built to address and explore these issues. Cricket uses the memory management primitives of Mach [Acce86] to provide the abstraction of a shared, transactional, single-level store.

The main purpose in building the Cricket prototype was to explore the feasibility of using a single-level store in emerging application areas. We were particularly interested in investigating performance issues related to design environments (e.g., CAD/CAM systems) and persistent programming languages. Another reason for building Cricket was to explore whether Mach's memory management primitives can be used effectively by a database storage system. The implementors of database systems have historically shunned the buffering and storage facilities provided by operating systems [Ston81], and have "rolled their own" instead. Advanced operating systems like Mach may change this view, however, and allow operating systems and database systems to become more tightly integrated.

The remainder of this chapter provides a detailed description of the Cricket prototype. In Section 5.1, we argue that traditional database storage systems are poorly suited for some emerging application areas such as CAD/CAM. Section 5.2 then argues why a single-level store may be a better approach. This is followed by Section 5.3, where we review Mach's external pager facilities [Youn87], which play a central role in Cricket's design. Cricket's system architecture is then described in Section 5.4, and in Section 5.5, we present the results of a performance study that was carried out to compare Cricket to the EXODUS Storage Manager [Care89] on the Sun Benchmarks [Catt90]. Finally, conclusions and directions for future work are discussed in Section 5.6.

5.1. STORAGE SYSTEM DEMANDS FOR EMERGING APPLICATION AREAS

While traditional database storage systems are extremely good at retrieving large groups of related objects and performing the same operation on each object, they appear poorly suited for some emerging application areas such as CAD [Maie89]. Compared to traditional database applications, data accesses tend to be more unpredictable in CAD applications, and fast, interactive response time is usually given more weight as a performance criteria. Moreover, the data objects that make up a CAD design may be traversed and updated hundreds (or even thousands) of times before changes are committed [Maie89, Catt90]. This is in contrast to most traditional applications, where data objects are generally traversed and updated just once per transaction.

Unfortunately, traditional database storage systems are not geared for applications with this sort of behavior. Among other things, the procedure-based interface that must typically be used to traverse and update persistent objects is too slow [Moss90]. Also, as noted in [Maie89], traditional recovery protocols are often inappropriate. For example, generating a log record for each update in a CAD transaction would obviously have a negative impact on interactive response time, not to mention the volumes of log data that could be generated. For these reasons, CAD transactions often use a database system in more of a batch mode by loading a whole design into their virtual address space, converting it to an in-memory format, working on it, converting it back to a disk format, and then committing the entire design as changed at end-of-transaction [Maie89]. In general, one can argue that these problems are not just limited to design environments. Implementors of persistent languages have already run up against many of the same problems [Rich89, Atwo90, Rich90, Schu90].

Recently, a number of new database storage systems have been proposed to address some of these issues, e.g., [Horn87, Lind87, Moss88, Care89, Sche90, Ston90]. However, the feeling is that for some emerging application areas many of these systems will still fall short of the mark. This is due to the fact that many of them still use a

procedure-based interface to access persistent data. Moreover, many of them still use fairly traditional recovery techniques, e.g., write-ahead logging [Moh89a]. It was these observations and also the author's experiences with the EXODUS Storage Manager [Care89] and the persistent language E [Rich89, Schu90] that motivated Cricket's design.

5.2. THE ARGUMENT FOR A SINGLE-LEVEL STORE

As mentioned earlier, Cricket provides the abstraction of a single-level store to applications, and this is advertised as one of its key features. With a single-level store, the database itself is mapped into the virtual address space, allowing persistent data to be accessed in the same manner as non-persistent data. This is in contrast to a conventional two-level store, where access to persistent data is less direct and a user-level buffer pool is typically maintained to cache disk pages.

Single-level stores are nothing new, of course. Their origins can be traced back almost 20 years to Multics [Bens72], and many operating systems provide mapped file facilities that effectively implement a single-level store. More importantly, however, database implementors have repeatedly rejected the idea of using the mapped file facilities offered by operating systems and have instead chosen to manage buffering and disk storage themselves. There are a variety of reasons given why this is so [Ston81, Trai82, Ston84]. Among the most notable are:

- 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 [Moh89a] and sophisticated buffer management [Chou85].
- 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.

As pointed out in [Eppi89] and [Cope90], however, these criticisms may no longer be as valid as they once were. The above items can be countered by arguing that:

- With the right operating system hooks, it is possible to control when the data pages of a mapped file are written to disk. Mach, for example, provides many of the necessary hooks with its notion of *memory objects* [Youn87]. More will be said about this shortly.

- For many emerging database applications, a 32-bit address space is sufficient. Moreover, with the rapid increase in memory sizes — which are quadrupling in capacity almost every two years [Myer86] — and with shared-memory multiprocessors becoming more commonplace, processors with large virtual address spaces may soon become available. In fact, IBM's RS/6000 [Bako90] already supports a 52-bit address space, and HP's Precision Architecture [Maho86] supports a 64-bit address space, although strictly speaking, these are both segmented architectures. Some commercial vendors of microprocessors have publically predicted that microprocessors with a flat 64-bit addresses space will become available as early as 1993 [Mash90].
- As the cost of memory decreases, large page tables will become less of a concern, even more so in a distributed environment with a small number of users per workstation. Furthermore, inverted page tables such as those found in IBM's RS/6000 and HP's Precision Architecture may become more common with the increase in memory sizes. Inverted page tables exhibit the desirable property of growing in proportion to the size of physical memory rather than the size of virtual memory.

Despite these compelling arguments (see [Eppi89] for several more), the jury is still out on whether a single-level store offers any advantages for traditional database applications. In fact, the performance results for Camelot that were presented in Eppinger's Ph.D thesis [Eppi89] and also in [Duch89] seem to argue that it may not be a good idea for transaction processing. Interestingly enough, the real problem with a using single-level store for transaction processing appears to be the high CPU cost of handling page faults for persistent data rather than any of the criticisms mentioned above.

5.2.1. Why a Single-Level Store is Right for Cricket

Given the known problems with a single-level store, why consider it in Cricket? The answer is that, although a single-level store may be inappropriate for traditional database applications, it appears to offer several advantages for some emerging application areas that will outweigh its disadvantages. In the following paragraphs, we briefly mention some of these advantages.

One advantage is that a single-level store can eliminate the need for applications to distinguish and convert between non-persistent and persistent data formats. In many database storage systems, the format of persistent data and the process for gaining access to it usually differs from that of non-persistent data. Moreover, the cost of accessing persistent data is generally more expensive, even after it has been brought into memory [Atwo90, Moss90, Rich90, Schu90]. As a result, applications often convert persistent data to a more efficient, in-memory

format before operating on it. Unfortunately, this can involve copying costs, added buffering requirements, and format conversions. With a single-level store, non-persistent and persistent data can have a uniform representation and these costs can be reduced or eliminated [Cope90]. This has obvious benefits in applications like design environments, where fast interactive response time is a key concern.

For similar reasons, a single-level store should also simplify the job of implementing a persistent language. To reduce the cost of accessing persistent data, persistent languages often use "pointer swizzling" [Cock84, Moss90, Schu90]. In pointer swizzling, the embedded object identifiers (i.e., pointers) that are stored in persistent objects are typically converted to virtual addresses while they are in memory. This is done to reduce the cost of traversing objects. Unfortunately, swizzling is not as simple as it sounds. Issues arise regarding what identifiers to swizzle, when to swizzle them, and how to unswizzle them. Moreover, in persistent languages based on C [Agra89, Rich89], it is often difficult to know where identifiers are located, when they change, and when they need to be reswizzled [Schu90]. With a single-level store, object identifiers become virtual memory addresses, so all of these issues (and their associated costs) can be eliminated.

Yet another advantage of using a single-level store is that persistence and type can be kept orthogonal [Atki87]. That is, application code can be written without concern for whether it is operating on non-persistent or persistent data. This simplifies code development and also allows binaries that were originally designed to operate on non-persistent data to be used with persistent data -- which, of course, has obvious practical and commercial advantages. Persistent languages that are built on conventional two-level stores sometimes require programmers to distinguish between non-persistent and persistent data via a dual type system, e.g., O++ [Agra89] and E [Rich89]. Dual type systems are generally regarded as an unattractive solution [Atki87, Atwo90]. In addition to making a persistent programming language more complicated, a dual type system also prohibits standard libraries and debuggers from being used on persistent data.

Finally, a single-level store can simplify the management of persistent objects that span multiple disk pages. Because a single-level store makes use of MMU hardware, multi-page objects can be made to appear in memory as though they were contiguous without actually requiring physical contiguity. This is in contrast to the EXODUS Storage Manager [Care89], where considerable effort was required to implement contiguous buffering of multi-page objects.

5.3. EXTERNAL PAGERS IN MACH

In the next section, we describe the system architecture of the Cricket prototype. Before that can be done, however, we need to briefly review Mach's *external pager* interface [Youn87], since it plays a central role in Cricket's design.

Among other things, Mach provides a number of facilities that allow user-level tasks (i.e., processes) to exercise control over virtual memory management. Mach provides the notion of a *memory object*, which is simply a data repository that is managed by some server (in this case Cricket). Such a server is called an *external pager*. An external pager is in charge of paging the data of a memory object to and from disk.

In Mach, tasks can associate (i.e., map) a given region of their address space to a memory object using the *vm_map* kernel call. After doing so, the external pager for that memory object will be called by the Mach kernel when a page in the mapped region needs to be read or written to disk. Physical sharing of data occurs when more than one task maps the same memory object into its address space. The Mach kernel and external pagers coordinate their activity through a message-based interface, which is summarized in Table 5.1.

Mach Kernel to External Pager Interface	
memory_object_data_request( )	request for data page of a memory object
memory_object_data_write( )	request to write page of a memory object to disk
External Pager to Mach Kernel Interface	
memory_object_data_provided( )	supplies kernel with data page
memory_object_data_unavailable( )	tells kernel to use zero-filled page

Table 5.1: External Pager Interface

Figure 5.1 illustrates how the Mach kernel and an external pager coordinate their activity on a page fault for a memory object M. At startup, the external pager acquires a *port* (i.e., capability) from the Mach kernel and associates it with M. Through an exchange of messages, the capability for M is passed to the client task, which then calls *vm_map* to map M into its address space. (Alternatively, the external pager can call *vm_map* on behalf of the client if it has the right permissions.) When the client attempts to access a page P in the mapped region, a page fault is generated. The page fault is caught by the Mach kernel, which verifies the client's access permissions and then sends a *memory_object_data_request* message to the external pager, asking it to supply the data for P. The external

pager reads the data from disk and provides it to the kernel via *memory_object_data_provided*. The kernel then locates a free page frame, copies the data into the frame, and resumes the client. Subsequent accesses to P will not generate a page fault.

By default, Mach uses an LRU replacement algorithm to manage kernel memory. If its free-page list starts to run low and page P is at the top of the inactive list, then P will be replaced. If P is clean, its contents are simply discarded. Otherwise, the kernel sends a *memory_object_data_write* message to the external pager with a pointer to P, at which point the external pager is expected to write P to disk.

In addition to the interface calls mentioned above, Mach also provides the means for an external pager to force a page of a memory object to be cleaned or flushed. This effectively allows the external pager to control (to some extent) the replacement policy used for a memory object. Furthermore, *memory_object_data_provided* can be called asynchronously, so prefetching data for a memory object is also possible.

5.4. CRICKET'S SYSTEM ARCHITECTURE

This section describes the system architecture of the Cricket prototype. The section is broken into two parts. In the first part, we discuss Cricket's basic design, and in the second part we discuss more advanced design issues that are largely unresolved at this point in time and left for future work.

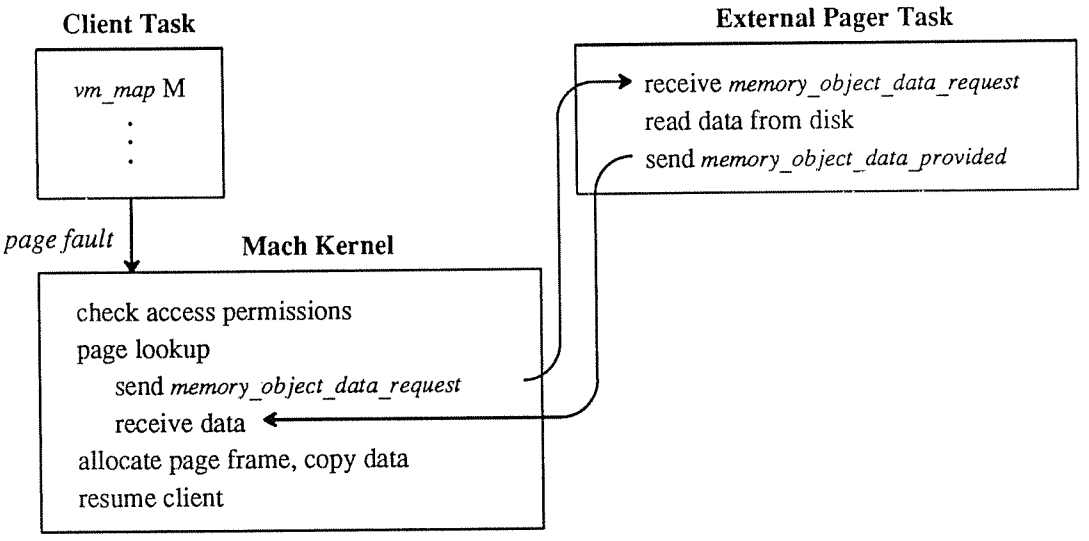


Figure 5.1: Handling a Page Fault on a Memory Object

5.4.1. Basic Design

5.4.1.1. Architecture Overview

Figure 5.2 illustrates the single-site architecture of Cricket. As shown, Cricket follows a client/server paradigm. Client applications run as separate tasks, each in their own protection domain, and they use an RPC interface to request basic services from Cricket. The RPC interface includes *connect* to establish a connection with Cricket, *disconnect* to break a connection, *begin_transaction* to begin a transaction, and *end_transaction* to end a transaction. For efficiency, some of Cricket's functionality is split between the Cricket server itself and a runtime library that gets linked with the application code at compile time. The runtime library includes RPC stubs as well as code for allocating persistent data.

The Cricket server is multi-threaded to permit true parallelism on multiprocessors and also to improve throughput by permitting threads to run even when others are blocked on synchronous events like I/O. The Mach C-Threads package [Drav88] is used to create and manage threads. When Cricket starts up, it creates a pool of threads which all line up on the same central message queue waiting to service client or kernel requests. A given thread is not tied to any particular function or transaction. When a thread finishes servicing a request, it puts itself on the central message queue again and waits for yet another request. Mach takes care of preemptively scheduling

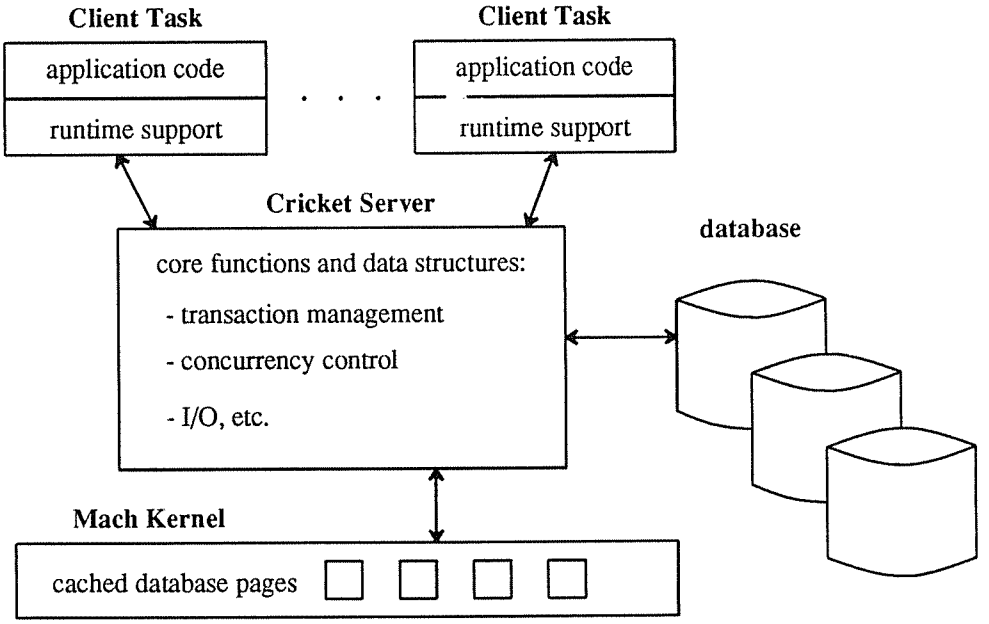


Figure 5.2: Single-Site Cricket

individual threads.

As mentioned earlier, client applications are allowed direct (shared) access to persistent data. This is accomplished using Mach's external pager facility. The database is simply treated as a memory object and the Cricket server plays the role of its external pager. When a client first *connects* to Cricket, a *vm_map* call is executed by Cricket on behalf of the client to map the database into the client's virtual address space. The *connect* call returns the virtual address that corresponds to the start of the database, as mapped in the client's address space. Using this address, the client can then access the database just as if it were in virtual memory — ala a single-level store. To ensure that pointers to persistent data remain valid over time, Cricket always maps the database to the same range of virtual addresses. It is important to note that database I/O is completely transparent to client applications as a result of using Mach's external pager facility. By default, the same holds true for concurrency control and recovery.

5.4.1.2. On Protection verses Performance

As shown in Figure 5.2, Cricket's core functions and their associated data structures are isolated in the Cricket server where they are protected from client applications. Because of its widespread use, nobody would seriously consider using a system like Cricket if it was unable to support applications written in C [Kern78] or its derivatives. Consequently, separate protection domains are a necessary evil. One can imagine the damage a buggy C application could inflict if it had access to the disk allocation bitmaps! In commercial database systems, the application code and the system software typically reside in separate address spaces for the same reason.

Where Cricket departs from a more traditional design is that it lets clients directly access regular data via Mach's external pager facilities. Bitmaps and other meta-data structures are still inaccessible, of course. Although this compromises protection somewhat, the view is that it can be managed and is worth the extra performance on Cricket's intended applications. Moreover, because all database accesses filter through Cricket's locking mechanism, which is discussed below, an application can only damage the data pages that it has gained access to anyway. Note that, without direct access to the database, a client application would have to make an explicit request to read data into its address space, and it would have to take analogous steps to have it written back. This would involve added complexity, copying costs, extra buffering (possibly leading to double-paging [Bric76]), and would also destroy the abstraction of a single-level store.

5.4.1.3. Concurrency Control

By default, Cricket provides transparent, two-phase, page-level locking for client access to the database. This is done using Mach's exception handling facility [Blac88], which allows the exceptions of one task to be caught and handled by another task. Currently, Cricket handles all the exceptions of its client tasks.

When a client first *connects* to Cricket, the client's exception handler is set to be the Cricket server. Later, when the client executes *begin_transaction*, all of the virtual addresses in the client that map to the database are protected against read and write access. Subsequent attempts by the client to access a page in the database triggers an address exception, causing Mach to block the client and send a message to Cricket. This message is received by a Cricket thread, which attempts to acquire the appropriate (read or write) lock for the client, blocking itself if necessary. Once the lock has been acquired, the thread fixes the client's access permissions for the page via a kernel call and then informs Mach that the exception has been successfully handled. Mach then resumes the client process.

The exchange of messages involved in catching an address exception and setting a lock is similar to that shown in Figure 5.1 for external pager fault-handling. As one can imagine, setting a lock is not cheap! Compared to a more traditional design, however, the scheme used in Cricket is not as bad as it may appear at first glance. In a more traditional design, an RPC would typically have to be sent from the client to the database system to acquire a lock. Moreover, as our performance results will show, exception handling in Mach is not drastically more expensive than sending an RPC.[1]

Clearly, it would be useful to support different concurrency control options other than simple two-phase, page-level locking. Other possibilities might include support for dirty reads [Moh89a] and design- or file-level locking. The latter would be used by design transactions, where coarse-grained locking may make more sense. Unfortunately, the smallest granularity of locking that can be transparently supported in Cricket is limited to a page, but that should be sufficient for Cricket's intended applications.

It is important to note that using address exceptions to trigger locking is not a new idea. For example, exceptions were also used in the Bubba database system [Bora90] to set locks. Our scheme differs from theirs in that lock management is performed by a user-level task in Cricket, whereas locking was performed by the operating system

[1] It is worth noting that we also experimented with an alternative locking scheme where the exception handler ran as a thread in client's runtime support code. When an address exception was caught, this thread would send an RPC to Cricket to acquire the appropriate lock. Because of RPC costs, this turned out to be more expensive than the design that has been chosen.

in Bubba. This required special modifications to the operating system. Address exceptions have also been used to implement memory coherency in a distributed virtual memory system [Li88], and in the language ML to trigger garbage collection [Appe86].

5.4.1.4. Buffer Management

At the moment, all page replacement decisions for regular data are delegated to Mach. Consequently, an LRU replacement policy is used by default. This is expected to be adequate for Cricket's intended applications. As noted in [Eppi89], the advantage of letting Mach buffer regular data is that it effectively provides a buffer pool that dynamically changes its size in response to other system activity.

Two alternatives were examined for managing system meta-data such as the page-allocation bitmaps. The first alternative was to maintain a small, wired-down virtual memory buffer pool in the Cricket server, while the second alternative was to map meta-data into the virtual address space of Cricket itself and treat it as yet another memory object. The first alternative was chosen because of the expense associated with using an external pager. Moreover, the abstraction of a single-level store does not appear to be particularly important for meta-data.

5.4.2. Unresolved Design Issues

5.4.2.1. Disk Allocation

The Cricket prototype currently supports only the crudest form of disk allocation — just enough to get the system up and running for benchmarks. A more elaborate scheme for disk allocation has been worked out, but has not actually been implemented yet. Eventually, the plan is to use an extent-based scheme for managing disk space. A disk would be partitioned into extents, with each extent containing the same number of pages, say, 16 Kbytes worth. Extents and the pages within an extent would be allocated in a lazy manner, much like in Camelot [Spec88]. Linear hashing [Litw80] would be used to map a virtual address to a physical extent on disk, allowing sparse databases to be efficiently handled. Because hashing would be done on an extent basis, the hash table would generally consume very little space.

For allocating persistent data, the plan is to provide what amounts to an object-based version of Camelot's recoverable virtual memory. The runtime support code would provide a *DBmalloc* function for allocating persistent "objects" and a corresponding *DBfree* function for deallocating them. *DBmalloc* would take *size* and *near_hint* parameters. The *size* parameter would specify how much space to allocate, while the *near_hint* parameter would be

a virtual memory address telling *DBmalloc* where it should try to allocate space. The *near_hint* would be used to simultaneously provide both virtual and physical clustering. That is, *DBmalloc* would try to allocate the new object on the same page as the *near_hint*. Failing that, it would send an RPC to Cricket, which would try to allocate the object either within the same extent as the *near_hint* or as physically close to it as possible.

As illustrated in Figure 5.3, individual disk pages would be formatted as slotted pages [Date81]. The slot information at the bottom of a page would be used to keep track of the objects and the free space on the page. When all the space on a page becomes free, it would be marked as such in page-allocation bitmaps maintained by the Cricket server. Large multi-page objects would be allocated as runs of pages that are virtually contiguous, but not necessarily physically contiguous. Only the first page of a large object would be formatted as a slotted page.

5.4.2.2. Dealing with External and Internal Fragmentation

Unfortunately, the disk allocation scheme described above can suffer from both external and internal fragmentation [Knut68]. External fragmentation can result from *DBmalloc* and *DBfree* operations on large multi-page objects of varying size, while internal fragmentation can result from *DBmalloc* and *DBfree* operations on small objects of varying size within the same page. Note that, since multi-page objects do not have to be physically contiguous on disk, external fragmentation can only occur within the virtual address space of the database.

One way to reduce external fragmentation would be to use a buddy system like the one described in [Lehm89]. The basic idea would be to simply treat the virtual address space like a page-based buddy system, and prohibit large, multi-page objects from sharing data pages with any other objects. Once again, multi-page objects would be

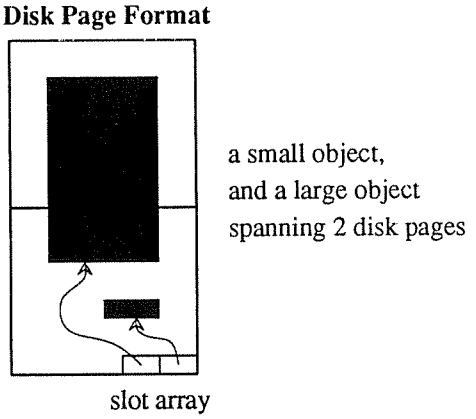


Figure 5.3: The Format of Disk Pages

allocated as runs of pages that are virtually contiguous, but not necessarily physically contiguous. If it was too costly to manage the whole virtual address space like a buddy system, then regions of it could be reserved for multi-page objects, and a separate buddy system would be used within each reserved region.

Internal fragmentation is a more onerous problem. One solution would be to simply let it occur. This may be a viable approach if the objects that are allocated on a given page tend to be roughly the same size. Another solution would be to force all object references to go through the extra level of indirection provided by the slot arrays on data pages. This is the approach taken in most database storage systems. If an extra level of indirection were always used, then it would be possible to compact the free space on pages at transaction commit time or during idle periods.

The main disadvantage of adding an extra level of indirection for persistent data is that it would force applications to distinguish between non-persistent and persistent data access, i.e., persistence and type would no longer be orthogonal [Atki87] at the storage-system level. This may not be as bad as it sounds, however, because in object-oriented languages that provide encapsulation or in persistent languages implemented on top of Cricket, it may be possible to hide the extra level of indirection with little loss in performance. Unfortunately, in languages like C [Kern78], which have a more physical notion of memory, there is probably little that can be done to truly hide the extra level of indirection.

5.4.2.3. Object Growth

Another unresolved design issue is whether to support object growth at the storage system level. While is it questionable whether it is worth supporting insert and delete operations on objects, the ability to append data to objects may be beneficial in many situations. Unfortunately, there is probably no way to support object growth at the storage system level other than to force all object references to go through an extra level of indirection. Once again, the slot arrays on data pages could provide this extra level of indirection. This is the approach taken in most database storage systems.

As noted above, the main disadvantage in adding an extra level of indirection for persistent data is that persistence and type would no longer be orthogonal at the storage-system level; one may then question whether a single-level store still offers any advantages. We would argue that it still does. Among other things, multi-page objects would still appear contiguously in virtual memory without actually requiring physical contiguity on disk or in memory. Moreover, the procedure-based interface that is used in most storage systems to access persistent data would still be eliminated. In the E persistent programming language [Rich89], a considerable amount of effort has

been spent on devising ways to reduce the number of procedure calls to the underlying storage system [Rich89, Rich90, Schu90]. With a single-level store — even one that required an extra level of indirection for persistent data — this effort would be unnecessary. We would expect similar benefits for other persistent programming languages.

5.4.2.4. Files

Files are another unresolved design issue. Given enough address bits (e.g., 64 bits), it may be sufficient in many cases to simply partition the virtual address space into large fixed-sized regions and treat each region as a different file. Another alternative is to view a file as a list of (not necessarily contiguous) extents. This would require that all the objects in an extent belong to the same file.

5.4.2.5. Index Management

Obviously, Cricket cannot be considered to be a complete storage system without support for indexes such as B+ trees. An index in Cricket would simply map from some user-defined key to the virtual address of an object in the database. In general, two-phase locking is inadequate for indexes [Kuma87], and obtaining acceptable system performance usually requires fairly complex concurrency control and recovery algorithms to be used [Moh89b]. The same holds true for all meta-data structures. Consequently, indexes should probably be managed by Cricket itself and not be treated as regular data.

Index management presents something of a dilemma because, while it would be useful to protect indexes from being damaged by client applications, the cost of sending an RPC to the Cricket server for each index access is likely to be too expensive, even if index accesses are batched. One way to get around this dilemma might be to give read-only access to index pages. In this scheme, the runtime support code would take care of read operations on indexes (including locking), but updates would be forwarded (perhaps in batch-mode) to the Cricket server.

5.4.2.6. Recovery

Recovery is yet another unresolved design issue. In discussing recovery algorithms, one of the key things to remember is that Cricket is intended for applications where interactive response time is a key concern. As a result, we are willing to accept a recovery algorithm that slows down transaction commit somewhat if it significantly improves response time during the execution of the transaction. Another thing to remember is that Cricket is intended to be used in applications where the same set of persistent objects may be updated thousands of times by the same transaction. In such an environment, traditional old-value/new-value logging is clearly inappropriate.

For disk allocation data, indexes, and all other meta-data, a conventional database recovery algorithm such as the ARIES algorithm [Moh89a] can be used. For regular data, a number of alternative have been identified, all of which require a *no-steal* buffer policy.[2] With a no-steal policy, steps must be taken to ensure that a dirty data page is not written to its home location on disk until the transaction that has modified the page commits. The advantages of using a no-steal policy are that old-values do not have to be logged and that repeated changes can be accumulated before being logged at commit time.

One alternative for regular data recovery in Cricket is to simply log full pages at commit time. Although this sounds like it could generate excessive amounts of log data, the applications that use Cricket may tend to update a large fraction of each page that they modify. If this turns out to be the case, then logging full pages at commit time will result in an efficient recovery algorithm. During idle periods, the on-line log can be compressed by purging all but the most recent copy of a given page. To further save on log space, a compression algorithm such as the one described in [Welc84] can be applied to pages before they are logged.

Another alternative is to use a copy-on-write mechanism. When a write lock for a page is granted, the page is copied to a temporary location in memory. Then, at transaction commit time, the new version of the page is compared to its original version and the changed portions are logged. If log space is still a concern, a compression algorithm can be applied to the log records that are generated.

One final alternative for recovery in Cricket is to require all updates to persistent data to filter through a run-time support function. The support function would record information that indicates which persistent objects have been modified. At commit time, the Cricket server would then use the recorded information and its knowledge of which pages were modified to generate new-value log records. The primary disadvantage of this approach is that persistence and type would no longer be orthogonal at the storage-system level.

5.4.2.7. Moving to a Distributed Environment

Since a client/server hardware configuration is expected to be the norm for design applications, it is obviously important to consider how Cricket should be restructured for a distributed environment. Because it has been built on top of Mach, client applications and the Cricket server can already run on separate machines. However, the

[2] Note that with a no-steal policy, logging is only needed to provide commit atomicity and to support recovery from media failures.

current design has not yet been optimized for the distributed case.

One possible distributed architecture is shown in Figure 5.4. In the architecture depicted in Figure 5.4, Cricket would be split into a front-end and a back-end. The front-end would take care of functions that can be handled more efficiently on the local machine, while the back-end would take care of global functions like cache coherency. Note that, because Cricket provides a single-level store to clients, this architecture would support what amounts to distributed, transactional, shared, persistent virtual memory. Although the algorithms described in [Li86] could be used to maintain memory coherency across machines, transaction semantics open up the possibility to use more efficient algorithms. There has been some work done in this area (e.g., [Wilk90b, Dewi90]), but not in the context of a single-level store.

Some of the interesting problems that surface in a distributed environment include index management, buffering, and the general question of what functionality belongs in the front-end and what functionality belongs in the

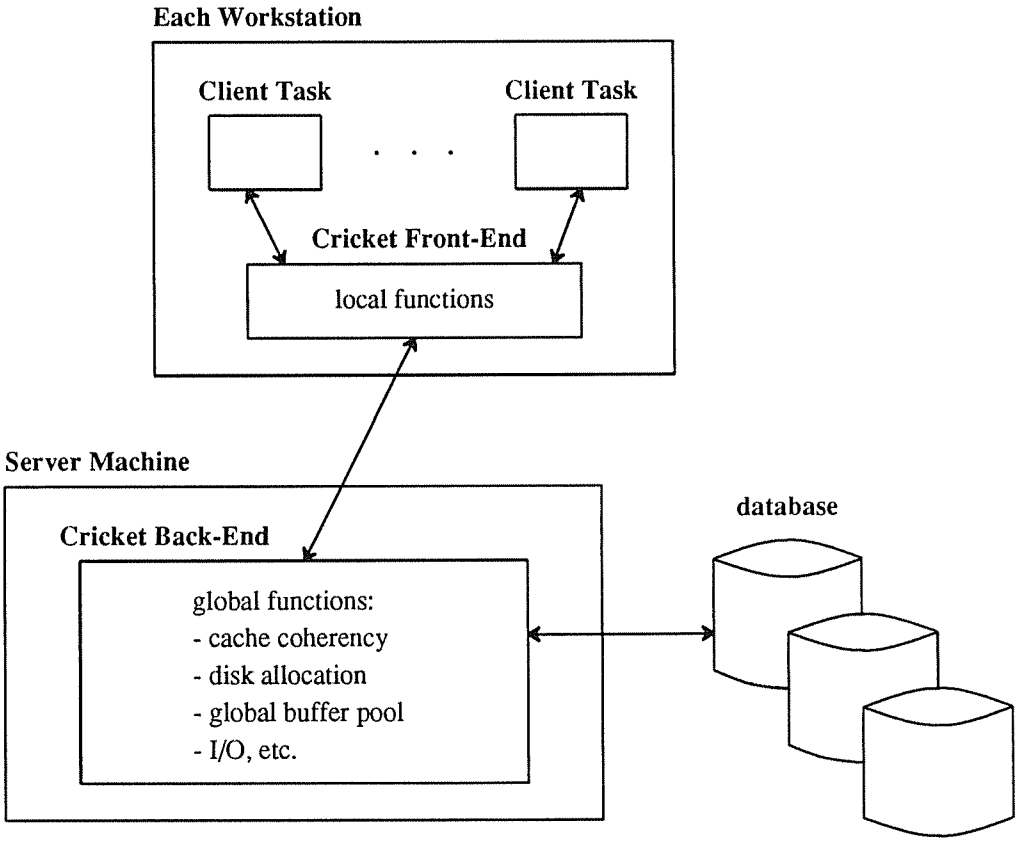


Figure 5.4: Distributed Cricket

back-end. Distribution would also affect the choice of recovery algorithms. For example, in a distributed environment, it probably makes sense to offload as much commit processing as possible to the front-end machine. Also note that there is no need to use Mach's external pager facilities in the back-end, as the data pages that are cached there are not directly accessible to clients.

5.5. PERFORMANCE RESULTS

In this section, we present the results of a performance study that was carried out to compare the EXODUS Storage Manager [Care89] and the Cricket prototype on the Sun Benchmarks [Catt90]. The purpose of this exercise was to further examine the feasibility of Cricket's approach. The Sun Benchmarks were chosen for our performance study because they attempt to measure system performance on design applications such as CAD/CAM, which are just the sort of applications for which Cricket is intended. The Sun Benchmark suite consists of three benchmarks that are run against a database made up of parts and connections between parts. The suite includes a *lookup* benchmark, which looks up 1,000 random parts, a *traversal* benchmark, which traverses 3,280 connected parts, and an *insert* benchmark, which inserts 100 new parts into the database. This section includes results for the lookup and traversal benchmarks on databases with 20,000 and 200,000 parts.

5.5.1. Brief Review of The Exodus Storage Manager

Before presenting a description of the Sun Benchmarks, it is necessary to briefly review the interface that the EXODUS Storage Manager (ESM) provides to clients, since an understanding of its interface will be needed in the discussion that follows. The basic abstraction provided by the ESM is a *storage object*, which is essentially an uninterpreted byte string. The size of an individual storage object is largely unconstrained, and clients of the ESM see the same uniform interface whether they are accessing a small object containing only a few bytes or a large object containing several megabytes. Storage objects are referenced by object identifiers (OIDs). Among other things, the OID of an object O contains the disk address of the page containing O and a 4-byte *unique field* that is used to provide object identity and protect clients against dangling OIDs.

Clients of the ESM gain access to storage objects via the *ReadObject* procedure call. When *ReadObject* is called for an object O, the ESM 1) locates the page P containing O in its buffer pool, 2) *pins* P making it ineligible for selection by the buffer replacement algorithm, and 3) sets up a *user descriptor* (i.e., a handle) for O that is returned to the client. Among other things, the user descriptor that is returned to the client includes an indirect pointer to O. This indirect pointer gives the client access to O in the ESM's buffer pool. After O is no longer

needed, the client calls *ReleaseObject*, passing in the user descriptor that was set up in *ReadObject*. *ReleaseObject* unpins O, at which point P becomes eligible for selection by the buffer replacement algorithm. The ESM also provides a *WriteObject* call for updating objects. In order for recovery to work properly, a client is not allowed to directly update a storage object O. Instead, *WriteObject* must be called every time some portion of O is updated. The parameters to *WriteObject* include a length and an offset to indicate which part of the object is being updated.

Figure 5.5 illustrates how a client interacts with the ESM. The C code segment shown there simply reads the storage object referenced by *oid* and makes sure its *x* and *y* fields are equal. Note that some liberties have been taken with the actual interface of the ESM to keep things as simple as possible. As Figure 5.5 suggests, updates tend to be the most inconvenient aspect of the ESM's interface. Calculating the correct update offset can be difficult when complex data structures are involved, and carrying around the context that *WriteObject* requires is often a nuisance when updates to a storage object are spread over different procedures. In contrast to updates, read access to storage objects in the ESM is somewhat cleaner. As shown in Figure 5.5, the use of type casting in C makes it fairly easy to access the fields of a storage object after the initial *ReadObject* call.

To highlight some of the differences between the ESM and Cricket, Figure 5.6 shows how the example in Figure 5.5 would be rewritten in Cricket. Because of its single-level store, persistent data is accessed just as if it were transient data in Cricket. This in turn can simplify code development, as a comparison of Figure 5.5 and Figure 5.6 demonstrates.

5.5.2. The Benchmark Database Structure

5.5.2.1. Part Format in Cricket

As mentioned in the introduction, the database used in the Sun Benchmarks consists of a collection of parts and connections between parts. Within certain limits, the implementor of the benchmarks has the freedom to choose whatever data structures he or she sees fit for parts and the connections between parts. We chose to use variable-length parts, where the connections between parts are stored in the parts themselves. The C data structures that were used for parts in Cricket are shown in Figure 5.7. Note that the 'from' array is actually of variable length, and the usual C trick of specifying just one array element in the type definition has been used.

As specified in the benchmark description, each part contained 3 out-going "to" connections, and a variable-length list of in-coming "from" connections. Thus, each part was connected to exactly 3 other parts in the "to"

```
struct OBJ {            /* structure of the object being accessed */
   int x;
   int y;
};
OID         oid;        /* the object's OID */
USERDESC*   userDesc;   /* user descriptor set up by ReadObject */
OBJ**       obj;        /* an indirect pointer to access the object */
int         newValue;   /* new x value */
int         offset;     /* offset of the update */

/*
 * Read and pin the object.
 */
ReadObject(oid, userDesc);

/*
 * Cast an indirect OBJ pointer onto the indirect pointer
 * returned in the user descriptor.
 */
obj = (OBJ**) userDesc->indirectPtr;

/*
 * See if the x and y fields are the same. If not, then
 * call WriteObject passing it the new x value (taken from y)
 * the offset of the update (the offset of x within the object),
 * and the size of the update (equal to the size of x).
 */
if ((*obj)->x != (*obj)->y) {
      newValue = (*obj)->y;
      offset = (char*) &(obj)->x - (char*) &obj;
      WriteObject(userDesc, newValue, offset, sizeof((*obj)->x));
}

/*
 * Release and unpin the object.
 */
ReleaseObject(userDesc);
```

Figure 5.5: Example of a Client Interacting with the ESM

direction and was referenced by a variable number of parts in the "from" direction. As shown, associated connection data only had to be kept for "to" connections. Because of the way the database was built, which is discussed in more detail below, parts generally had about the same number of "to" connections as "from" connections. We did find, however, that some parts had as many as 9 "from" connections.

```
struct OBJ {          /* structure of the object being accessed */
   int x;
   int y;
};
OBJ*        obj;      /* pointer to access the object */

/*
 * Make sure the x and y fields are the same.
 */
if (obj->x != obj->y) {
      obj->x = obj->y;
}
```

Figure 5.6: Example of a Client Interacting with Cricket

5.5.2.2. Part Format in the ESM

The C data structures that were used for parts in the ESM are shown in Figure 5.8. As shown, the data structures used in Cricket were identical to those used in the ESM except that OIDs were used to connect one part to another instead of standard C pointers. The places where OIDs are used in Figure 5.8 have been highlighted in boldface to make them stand out.

It is important to note that OIDs in the ESM are 12 bytes long. This is in contrast to disk pointers in Cricket, which are really virtual addresses and therefore only 4 bytes long. Consequently, the average part with 3 "to" and 3 "from" connections was 94 bytes long in Cricket and 142 bytes long in the ESM. This size difference resulted in better disk clustering in Cricket, which in turn allowed it to perform better on some of the benchmarks. However, this can be viewed as an unfair advantage for Cricket because with its "skinny" pointers it does not match the functionality that the ESM provides with its "fat" OIDs. For example, the 4-byte unique field in each OID provides object identity and a limited form of protection against stray disk pointers.[3] Moreover, the ESM can support databases with hundreds of terabytes. This is in contrast to Cricket, which provides no notion of object identity and can only support databases with up to 4 Gbytes on conventional hardware with 32-bit virtual addresses.

If clients of the ESM were willing to do away with the extra functionality that "fat" OIDs provide, then it might be possible shrink the size of an OID down to 32 bits in the ESM. Therefore, to be as fair as possible, two sets of

[3] Unique fields only provide a limited form of protection because clients of the ESM still have direct, unchecked access to the buffer pool and other critical data structures in the ESM.

```
/*
 * Data associated with "to" connections.
 */
struct ConnectionData  {
    char    type[10];              /* type information */
    int     length;                /* length information */
};

/*
 * Part definition.
 */
struct Part {
    int             partId;        /* the part's ID */
    int             x;             /* the part's x coordinate */
    int             y;             /* the part's y coordinate */
    long            date;          /* the part's creation date */
    char            type[10];      /* the part's type */
    Part*           to[3];         /* "to" connections */
    ConnectionData  toData[3];     /* "to" connection data */
    short           fromCnt;       /* count of "from" connections */
    Part*           from[1];       /* "from" connections */
};
```

Figure 5.7: The Structure of Parts in Cricket

```
/*
 * Part definition.
 */
struct Part {
    int             partId;        /* the part's ID */
    int             x;             /* the part's x coordinate */
    int             y;             /* the part's y coordinate */
    long            date;          /* the part's creation date */
    char            type[10];      /* the part's type */
    OID             to[3];         /* "to" connections */
    ConnectionData  toData[3];     /* "to" connection data */
    short           fromCnt;       /* count of "from" connections */
    OID             from[1];       /* "from" connections */
};
```

Figure 5.8: The Structure of Parts in the ESM

benchmarks were run in Cricket. In one set, parts were formatted as in Figure 5.7, while in the other set, parts and various other per-page and per-object data structures were padded in Cricket so that the resulting database was the same size in Cricket as in the ESM. It could be argued that the results for the padded database provide a less biased means for capturing the intrinsic differences between the ESM and Cricket. The results for the unpadded database

are still presented, of course, because they reflect Cricket's actual performance on the Sun Benchmarks. Moreover, the differences between the unpadded and padded results illustrate the added cost of using "fat" OIDs — a cost which many applications may not want to pay, despite the added functionality.

5.5.2.3. The Physical Layout of Parts on Disk

As specified in the Sun Benchmark, two databases were built in each system, one with 20,000 parts and one with 200,000 parts. The 20,000 part database is intended to measure system performance on applications whose working sets fit in memory, while the 200,000 part database is intended to measure system performance on applications whose working sets are larger than the size of memory. The benchmark optionally calls for a database with 2,000,000 parts, but in our view there is little reason to study that case, since the 200,000 part database was already much larger than the size of physical memory on our machine. Consequently, the results for the 2,000,000 part database would not have revealed anything more than the results for the 200,000 part database revealed — it would have just taken much longer to load the database and run the benchmarks.

When a part database was loaded in the ESM and Cricket, it was done in such a way that the *i'st* part generated had its ID set to *i*, and such that the *i+1'th* part directly followed the *i'th* part on disk. As specified in the benchmark, the "to" connections were chosen in such a way that there was a 90% probability that a "to" connection of a part P was to a part "close" to P; the remaining 10% of the time, a random part was chosen from among all the parts to form a "to" connection. By "close", we mean that the connected part was randomly chosen from those parts with IDs that were within ±1% of the ID space away from P. In the 20,000 part database, for example, there was a 90% probability that a "to" connection for the part with ID 1,000 was connected to a part with an ID in the range 800 to 1,200. The remaining 10% of the time a "to" connection was to a part with an ID in the range 1 to 20,000.

By connecting parts in this manner, and by physically ordering parts by their IDs, the resulting database captured the type of logical and physical reference locality that is expected to be common in design environments [Catt90]. This is precisely what the creators of the Sun Benchmark had in mind with their notion of "closeness." To ensure that the identical part connections were used in both systems, the same sequence of pseudo-random numbers was used in both the ESM and Cricket to load each database..

By generating all the part connections for a database beforehand, the length of a part could be calculated as the database was being loaded. Armed with this knowledge, we were able to pack parts end-to-end on disk pages in both the ESM and Cricket. Packing parts in this manner was "cheating" in some respects because object forwarding,

which is not currently supported in Cricket, would have been required to handle part growth. For reasons that will be discussed shortly, however, the portion of the Sun Benchmark that would have caused parts to grow was not run. As a result, parts were simply packed end-to-end. This probably throws off the absolute results that were obtained, but the goal here was only to make a relative comparison of Cricket and the ESM operating under the same conditions.

As a database was loaded, a main-memory index was constructed to keep track of the disk address or OID of each part in the database. This so-called *part index* was written to a UNIX file at the end of the build step and then re-read into memory before each benchmark was run. The main-memory index was necessary because neither Cricket nor the ESM currently provides support for indexes. The part index could have been stored in the database itself, of course, but a main-memory index was simpler to implement. Because of the size difference between OIDs in the ESM and disk pointers in Cricket, a straightforward implementation of the part index for the 200,000 part database ends up being considerably larger in the ESM than in Cricket. There was some concern that this might bias the results in favor of Cricket on the padded database, so index entries were padded in Cricket for those results in order to ensure that the part index was the same size in both systems (roughly 2.3 Mbytes).

5.5.3. The Sun Benchmarks

Three programs are specified in the Sun Benchmarks. They are:

Lookup

Generate 1,000 random part IDs and fetch the corresponding parts from the database. For each part looked up, call a null procedure, passing the x,y position and type of the part.

Traverse

Traverse all parts connected to a randomly selected part, the parts connected to it, and so on, up to 7 hops. Perform the traversal depth-first, following the "to" connections of each part. For each part traversed, call the null procedure mentioned above. A total of 3,280 parts are traversed in all, with possible duplicates. Also perform a reverse traversal, where the "from" connections of each part are used in the depth-first search instead of "to" connections.

Insert

Create and insert 100 new parts into the database. Connect each new part to other parts with the same notion

of "closeness" mentioned earlier. For each new part, call the null procedure mentioned above. Commit all changes to disk.

Only the results for the lookup and the traversal benchmarks are included here. Results for the insert benchmark are omitted for two reasons. First, there is currently no mechanism for supporting variable-length data in Cricket, and such support is necessary for the insert benchmark. It would be fairly simple to add support for variable-length data via an extra level of indirection or through linked lists, but that has not been done yet. Second, neither the ESM nor Cricket currently supports recovery. Since recovery support will significantly impact the cost of committing changes, it was felt that an accurate comparison of each system's performance on this benchmark could not be made at this time.

Also note that, although results were collected for reverse traversals, those results have been omitted here. With "to" and "from" connections stored together in parts, a reverse traversal executed in much the same manner as a forward traversal. Consequently, the results for reverse traversals provided little information beyond that already provided by forward traversals.

5.5.4. Benchmark Measures

The Sun Benchmark specification only calls for two measures to be presented for each benchmark. One is the "cold start" elapsed time, while the other is the "warm start" elapsed time. The "cold start" time reflects the elapsed time to run a given benchmark with a cold system; that is, when no part data has been cached in memory. The "warm start" time reflects the asymptotic best results when the cache is fully initialized. Our view was that these two simple measures do not provide enough information. As a result, we present timing information for each benchmark after 1 execution, 2 executions, 3 executions, and so on, with random parts chosen in each execution. By doing so, both the "cold start" and asymptotic "warm start" times are captured, as well as the transitional behavior of each system as part "re-use" becomes more probable. In addition to elapsed times, I/O statistics are also presented, as these had a major impact on the results.

The benchmark specification also calls for the benchmarks to be run in a client-server hardware configuration. Unfortunately, only the single-site prototype of Cricket exists at this time. Consequently, all benchmark measures were obtained on a single machine. The single-site version of the ESM, which has been reasonably tuned, was used to keep the comparison as fair as possible.

5.5.5. The Hardware and Software Environment

All of the benchmarks were run on a DEC MicroVax 3200 with 16 Mbytes of memory. The benchmarks were run on Mach 2.5, which includes an embedded version of BSD 4.3 UNIX. To eliminate as much noise as possible, the benchmarks were run in single-user mode with the machine disconnected from the network and all but the minimal set of tasks running. The page size was set to 4 Kbytes in both the ESM and Cricket. The benchmarks were written in C and compiled with the highest level of optimization. All data was stored on a single RD54 159 Mbyte disk, which has an average seek time of 30 msec and an average access time of 38.3 msec. Note that the observed access time was actually better than the nominal measure due to the fact that benchmark data was clustered on disk and spread across relatively few cylinders. A separate RD54 disk was used for swapping.

A single 90 Mbyte raw disk partition was used to store part data rather than go through the UNIX file system. This was done for several reasons. First, it was more realistic. Because UNIX buffers file data and file meta-data in uncontrolled ways, no storage system that uses the UNIX file system can provide true recovery. Second, it guaranteed that data was clustered on disk in exactly the way it was intended to be. Third, it ensured that no hidden buffering between benchmark runs occurred. Finally, it ensured that no read-ahead was taking place. Although the benchmark results would have improved from read-ahead, we did not want them to be affected by a UNIX mechanism that neither storage system had control over. In practice, the ESM and Cricket would both have their own read-ahead mechanism, although neither system has one at the moment.

5.5.6. CPU Costs in the ESM and Cricket

The performance of Cricket is largely dependent on the CPU cost associated with Mach's external pager and exception handling facilities. Consequently, it was important to measure these costs and see how they compare to CPU costs in the ESM before any of the Sun Benchmarks were run,

To measure the CPU cost of Mach's external pager and exception handling facilities, a simple scan program was used. In the scan program, a single Cricket client sequentially touched the first 1,280 pages (i.e., 5 Mbytes) of a dummy database. All I/O was short-circuited by having the Cricket server pass Mach a pointer to a dummy page in *memory_object_data_provide*. By using the scan program and by turning off all aspects of transaction management in Cricket other than exception handling and external pager requests, it was possible to obtain the results shown in Table 5.2. These results capture the per-page CPU cost of using Mach's external pager and exception handling facilities.

Event	Cost in usec
1) page fault that is handled completely in the kernel	420
2) handle address exception in Cricket	3,180
3) *memory_object_data_request* & *memory_object_data_provided*	3,221
4) page fault + address exception	3,605
5) page fault + address exception + *memory_object_data_request* & *memory_object_data_provided*	6,845

Table 5.2: The Per-Page CPU Cost of Mach's External Pager and Exception Handling Facilities

The costs listed in Table 5.2 are as follows: 1) is simply the CPU cost of handling a page fault for a page that is already cached in kernel memory. 2) is the CPU cost of handling an address exception in Cricket to trigger locking on a database page. 3) is the CPU cost of having the kernel send a *memory_object_data_request* message to Cricket, with Cricket responding via a *memory_object_data_provided* message. 4), which should approximately equal 1) + 2), is the CPU cost that a client incurs on the first access to a database page that is cached in kernel memory. Finally, 5), which should approximately equal 1) + 2) + 3), is the CPU cost that a client incurs on the first access to a database page that is not cached in kernel memory.

As Table 5.2 clearly indicates, the external pager and exception handling facilities of Mach are not exactly free. Most of the expense presumably comes from context switches, message costs, and management of the kernel data structures associated with memory objects. However, the reader should bear in mind that 4) or 5) will only be incurred on the first access to a page. Furthermore, there are a number of ways that these costs can be reduced. One way is to do multi-page I/O operations. We simulated the effect that this would have on the CPU costs in Table 5.2 by providing 4 pages of data to Mach in each *memory_object_data_provided* message. When this was done, the CPU cost of 5) dropped to 4,954 usec per 4 Kbyte page.[4] Another way to reduce costs is to "piggyback" I/O with exception handling. By this we mean that data is read from disk and a *memory_object_data_provided* message is asynchronously sent as soon an address exception for an uncached data page is caught. This is in contrast to waiting for an explicit *memory_object_data_request* message from the Mach kernel.[5] Finally, large-grained locks can be

[4] Unfortunately, a bug in Mach prevented us from doing this with real data. When we tried 4-page reads in the Sun Benchmarks, Mach overwrote database pages in memory with garbage.

[5] Piggybacking I/O in this manner is a rather obvious thing to do, but surprisingly Mach does not currently provide a way for an external pager to determine if a given page of a memory object is cached in kernel memory. Hopefully, this design flaw will be fixed in the near future.

used to cut down on the number of exceptions generated. To examine the effects of combining these methods, we simulated doing 4-page block I/O as soon as an address exception was generated for the first page in the block, and the granularity of locking was set to 4-page units. When this was done the CPU cost of 5) dropped further to 2,248 usec per 4 Kbyte page. In design environments, where file- or design-level locks may be common, CPU costs could obviously be reduced even further.

Unfortunately, the version of the ESM that was used lacked the instrumentation needed to collect the same sort of detailed cost statistics that were collected in Cricket. The only CPU cost that could be obtained for the ESM was the cost of a *ReadObject/ReleaseObject* pair for an object that was memory resident. This was measured to be roughly 330 usec. Using this value and cost 4) in Table 5.2, one sees that when there is no I/O activity, Cricket should outperform the ESM whenever there are 11 or more object accesses per page, even with straightforward page-level locking. Results in [Shek90] showed that this is indeed the case. Using the results in Table 5.2, it would be nice to make similar predictions about the performance of Cricket when there is I/O activity. Without more cost information for the ESM, however, such predictions are difficult to make. Due to the high cost of using Mach's external pager facilities, it would appear that the ESM should have a clear advantage over Cricket when there is a lot of I/O activity. This will be confirmed by the results that follow shortly.

5.5.7. Sun Benchmark Results

This section presents the results that were obtained for the Sun Benchmarks on the 20,000 and 200,000 part databases. These results should be viewed as complementing those presented in Section 5.5.6. In addition to comparing the performance of the ESM and Cricket on a well-known benchmark that includes I/O activity and actual data, the results in this section also capture the performance of each system on applications with large working sets.

5.5.7.1. Notation Used in the Results

As mentioned earlier, two sets of results are presented for Cricket on each part database. In one set, parts were padded in Cricket, resulting in a database that was the same size as the one used in the ESM. In the other set, parts were left unpadded in Cricket, resulting in a smaller database than in the ESM. Each set of results for Cricket was further split into three sub-cases, depending on whether exception handling was enabled and whether I/O was piggybacked with exception handling, as described earlier. The notation used to refer to each sub-case is summarized in Table 5.3.

Notation	Cricket Settings	
	Exceptions On	Piggybacked I/O
$CRK \cdot \overline{EXC} \cdot \overline{PB}$	NO	NO
$CRK \cdot EXC \cdot \overline{PB}$	YES	NO
$CRK \cdot EXC \cdot PB$	YES	YES

Table 5.3: The Notation Used to Refer to Cricket Results

The results with exception handling enabled (those with *EXC*) measure Cricket's performance with transparent, page-level locking. In generating the results with *EXC*, the address exceptions needed to do locking were caught and handled by Cricket, but the actual setting of locks was disabled. The results with *EXC* therefore measure the costs above and beyond setting a lock that clients of Cricket see when they elect to use transparent, page-level locking. Since the single-site version of the ESM does not support locking, the actual setting of locks was disabled in Cricket to keep the comparison as fair as possible. Note that if the ESM did support page-level locking, its relative performance would have been worse due to the fact that it would have to check for a page lock on every *ReadObject* call, regardless of whether the page containing the requested object had already been locked. In contrast, Cricket never has to check the status of a page's lock more than once.

The results with exception handling disabled (those with $\overline{EXC}$) measure Cricket's performance without transparent, page-level locking. In a design environment, file- or design-level locking may often be more appropriate than page-level locking. The results with $\overline{EXC}$ measure Cricket's performance in such an environment.

Finally, the results with piggybacking enabled (those with *PB*) measure the improvement in Cricket's performance when I/O was piggybacked with exception handling. In generating the results with *PB*, Cricket was set up so that it read a data page P from disk and sent a *memory_object_data_provided* message to Mach for P as soon as an address exception for P was caught. As noted earlier, Mach does not currently provide a system call to determine if a given page of a memory object is cached in kernel memory. Consequently, piggybacking I/O in this manner is not really possible at the moment. For example, a previous transaction could have already read P into kernel memory, in which case re-reading P from disk would be a mistake. The required system call could probably be easily added to Mach, however. Finally, note that $\overline{EXC}$ and *PB* do not appear together because that would make no sense.

5.5.7.2. Results for the 20,000 Part Database

The first set of results that are presented are for the 20,000 part database. The 20,000 part database was small enough so that the working set of each benchmark completely fit in the memory of our machine. Consequently, the results for the 20,000 part database are probably indicative of how each system can be expected to perform on applications whose working sets typically fit in memory.

The size of the database used in the ESM and in Cricket (with and without padding) is shown in Table 5.4. Note that the average part size includes the per-object overhead added by each system. As Table 5.4 shows, the size of the database in the ESM was considerably larger (by almost 50%) than the size of the unpadded database in Cricket. This was almost entirely due to the ESM's "fat" OIDs. Considering the large difference in the size of the databases, it should now be more clear why it was important to present both the padded and unpadded results for Cricket.

In generating the results for the 20,000 part benchmarks, an LRU replacement policy was used in the ESM, with its buffer size set to 9 Mbytes. The actual size of the buffer for the 20,000 part database was unimportant, so long as it was big enough to hold the working set of each benchmark. A 9 Mbyte buffer was more than sufficient. Since Cricket relied on Mach's page replacement policy, it also used an LRU replacement policy. The effective size of Cricket's buffer, which corresponds to the amount of physical memory that was available for paging on our machine, was approximately 12.5 Mbytes.

Results for the Lookup Benchmark on the 20,000 Part Database

The results for the lookup benchmark on the 20,000 part database are presented in Graphs 5.1-5.4. The unpadded Cricket results are presented in Graphs 5.1 and 5.2, while the padded Cricket results are presented in Graphs 5.3 and 5.4. A total of 20 lookups were run in each system. All lookups ran within the same transaction, and a different

Storage System	Database Size	Avg. Part Size	Avg. Parts Per Page
ESM	848 pages (3.3 Mbytes)	168 bytes	24
Cricket (unpadded)	575 pages (2.3 Mbytes)	116 bytes	35
Cricket (padded)	848 pages (3.3 Mbytes)	168 bytes	24

Table 5.4: The Size of the 20,000 Part Database

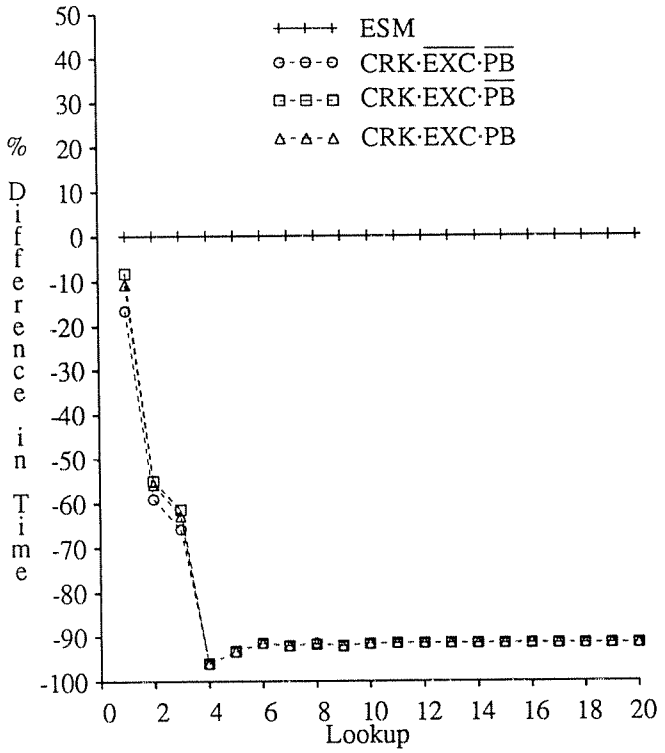
set of 1,000 random parts was chosen in each lookup.

Graphs 5.1 and 5.3 show the percentage difference in elapsed times between the ESM and Cricket. In those graphs, the elapsed time for lookup 1, lookup 2, and so on, was measured in the ESM. The corresponding times were then measured in Cricket and the percentage difference between those times and the ESM times was plotted, with the ESM times as the baseline. The results have been presented in this manner to make the differences between the two systems clearer. If absolute results had been displayed in Graphs 5.1 and 5.3, the differences between the two systems on lookups 4-20, which did not generate any I/O, would have been obscured by lookups 1-3, which took considerably longer because they generated I/O. Graphs 5.2 and 5.4 show the number of I/Os generated by the ESM and Cricket on each lookup.

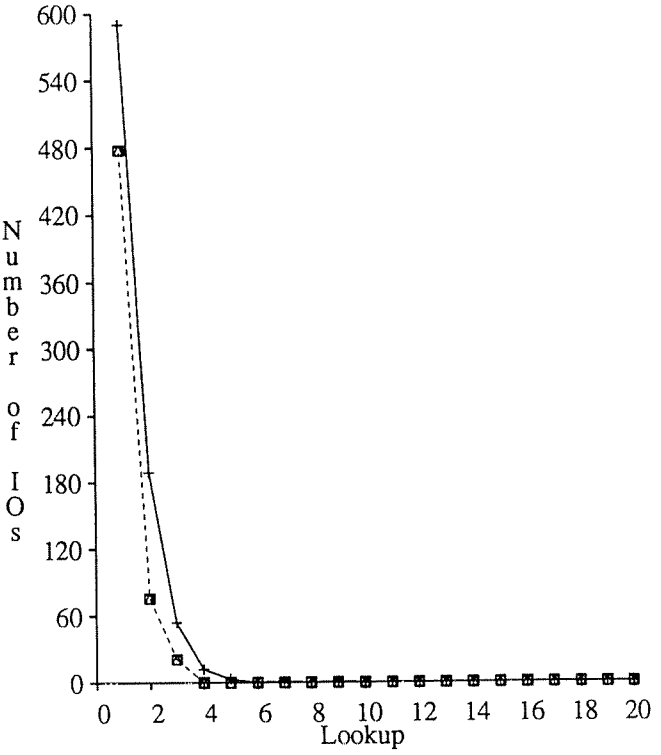
Beginning with Graphs 5.1 and 5.2, it is clear that Cricket was much faster than the ESM once all the data had been read into memory (see lookups 4-20). With all the data in memory, a lookup took about 399 msec on the ESM and about 34 msec on Cricket — a 91% reduction in time. Roughly 330 msec of the extra time spent in the ESM was due to *ReadObject* and *ReleaseObject* calls. We suspect that the remaining 69 msec was partly due to higher cache miss ratios in the ESM. Graph 5.1 also shows that Cricket was faster on lookups 1-3, which included I/O activity. As Graph 5.2 reveals, however, the reason that Cricket did better on lookups 1-3 was mainly because it did less I/O. It did less I/O on those lookups because its unpadded database was much smaller than the database used by the ESM. The smaller database had a compound effect on times: Not only did it result in less I/O for Cricket, but it also resulted in faster I/O because of reduced seek times. In this particular benchmark, each I/O took 33.11 msec in Cricket and 34.06 msec in the ESM. Both of these times include CPU overhead.

Graph 5.1 also illustrates the high cost of using address exceptions to trigger locking in Cricket and the effectiveness of piggybacking I/O with exception handling. As shown, the elapsed time for lookup 1 with exception handling disabled was about 10% faster than lookup 1 with exception handling enabled. Piggybacked I/O reduced this difference to about 7%. The difference in times are, of course, due to the CPU cost of using Mach's external pager and exception handling facilities.

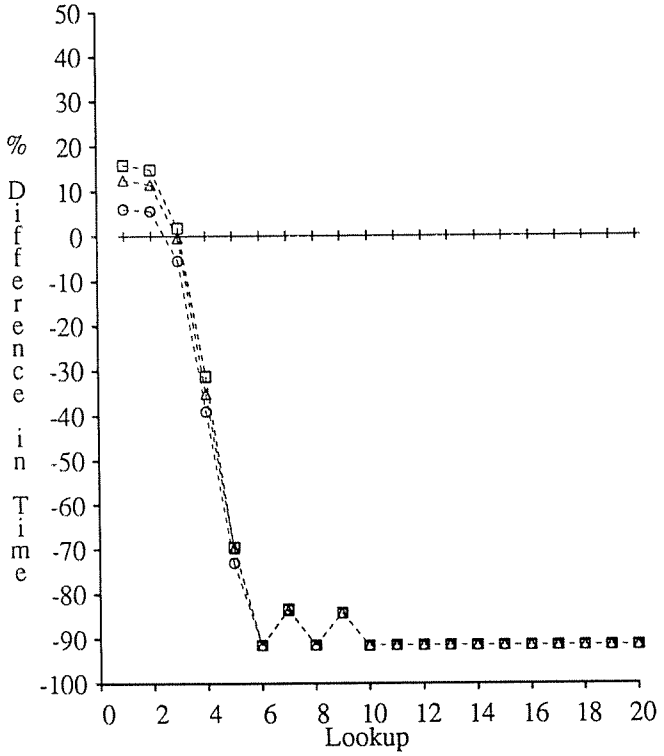
Turning to Graphs 5.3 and 5.4, which show the padded Cricket results, we see pretty much the same patterns that emerged in Graphs 5.1 and 5.2. In this case, however, Cricket did the same amount of I/O as the ESM and saw the same I/O times. Because of this, and because of the CPU cost associated with Mach's external pager and exception handling facilities, Cricket was slower than the ESM on the first two lookups by 5% to 15%. Note that the



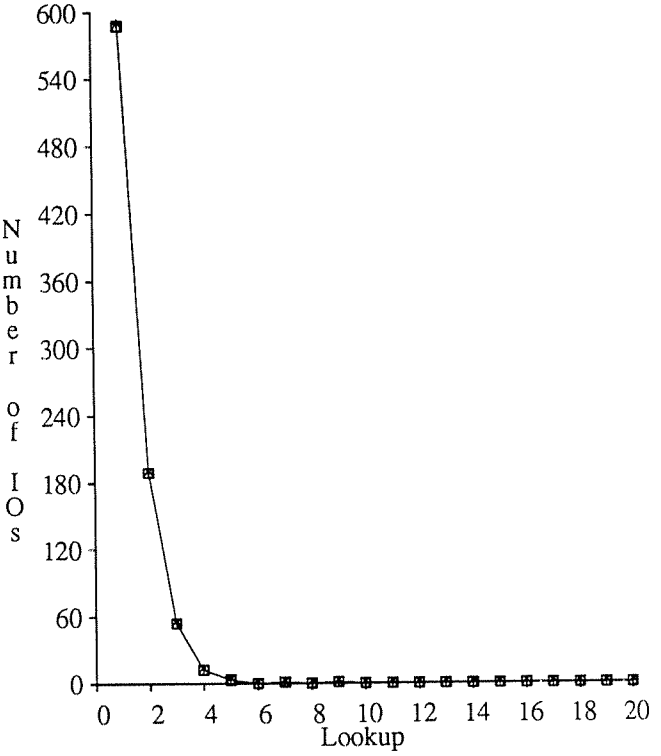
Graph 5.1: % Difference in Lookup Time
20,000 Parts, Unpadded Cricket Results



Graph 5.2: Number of I/Os Per Lookup
20,000 Parts, Unpadded Cricket Results



Graph 5.3: % Difference in Lookup Time
20,000 Parts, Padded Cricket Results



Graph 5.4: Number of I/Os Per Lookup
20,000 Parts, Padded Cricket Results

"bumps" in Graph 5.3 on lookups 7 and 9 are due to I/O activity that does not show up clearly in Graph 5.4. The random part selection generated by each lookup was the reason for the irregular I/O pattern. Cricket did worse on lookups 7 and 9 because I/O activity always hurts its relative performance.

Table 5.5 summarizes the absolute timing results for the lookup benchmark on the 20,000 part database. Each "cold-start" time is for lookup 1, while each "warm-start" time represents the average time for the last five lookups. It is interesting to note that using the CPU results in Table 5.2 and the I/O results in Table 5.5, one can estimate the elapsed time of a lookup on Cricket and get a value that is fairly close to the one that was measured. For example, one can estimate the time (in msec) on lookup 1 for $CRK \cdot EXC \cdot \overline{PB}$ (unpadded) to be the sum of:

- time to do 478 I/Os: 478 * 33.11
- CPU time to invoke external pager and exception handler for 478 pages, (see cost 5 in Table 5.2): 478 * 6.85
- CPU time to do the lookup: 34

This comes to 19,134 msec or 19.13 sec, which is close to the measured time of 19.45 sec. The difference between the estimated time and the measured one is probably due to timing noise, the effects of context switching on cache hit ratios, and the costs associated with page table management in the kernel. Since it did not use actual data, the simple program used to generate Table 5.2 probably did a poor job of measuring these last two effects.

Setting	Elapsed Time (in sec) Cold/Warm	Number of I/Os Cold/Warm	Avg. Time (in msec) Per I/O
ESM (9 Mbyte buffer)	21.19/0.40	588/0	34.06
$CRK \cdot \overline{EXC} \cdot \overline{PB}$ (unpadded)	17.68/0.03	478/0	33.11
$CRK \cdot EXC \cdot \overline{PB}$ (unpadded)	19.45/0.03	478/0	33.11
$CRK \cdot EXC \cdot PB$ (unpadded)	18.92/0.03	478/0	33.11
$CRK \cdot \overline{EXC} \cdot \overline{PB}$ (padded)	22.47/0.03	588/0	34.06
$CRK \cdot EXC \cdot \overline{PB}$ (padded)	24.55/0.03	588/0	34.06
$CRK \cdot EXC \cdot PB$ (padded)	23.86/0.03	588/0	34.06

Table 5.5: Summary of Lookup Results for the 20,000 Part Database

Results for the Traversal Benchmark on the 20,000 Part Database

The results for the traversal benchmark on the 20,000 part database are presented in Graphs 5.5-5.8. The graphs are displayed in the same manner as before, with the unpadded Cricket results in the top portion of the page and the padded results in the bottom portion of the page. All traversals ran within the same transaction, and a different random part was chosen as the starting point for each traversal.

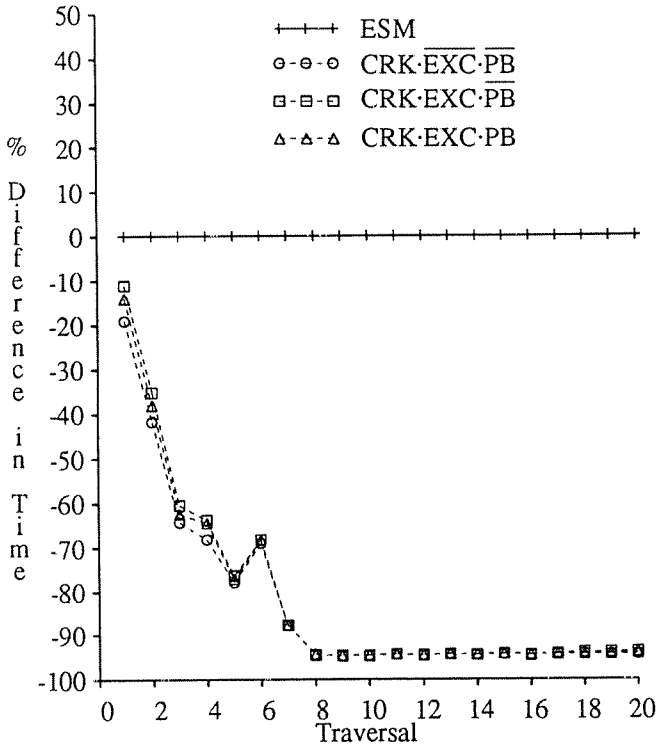
Graphs 5.5-5.8 follow the same general patterns that were observed in Graphs 5.1-5.4. With all the data in memory, a traversal took about 1,178 msec on the ESM and about 72 msec on Cricket — a 94% reduction in time. Once again, the cost of making *ReadObject* and *ReleaseObject* calls accounts for most of the extra time in the ESM. We also see the same sort of differences that were seen earlier between the unpadded and padded results. On the unpadded database, Cricket performed better than the ESM on the first few traversals because it did less I/O, while on the padded database it did worse on the first few traversals because of the CPU costs associated with Mach's external pager and exception handling facilities. The "bumpiness" in Graphs 5.5 and 5.6 is the again the result of an irregular I/O pattern. In this case, the irregular I/O pattern was due to chosing a random part to begin each traversal and also due to the randomness in part connections.

Table 5.6 summarizes the absolute results for the traversal benchmark on the 20,000 part database. It is worth noting that the time per I/O is slightly lower in Table 5.6 than it was in Table 5.5. In contrast to the lookup benchmark, each traversal accessed connected parts that tended to be somewhat more closely clustered on disk, and this in turn resulted in smaller seeks and faster I/O.

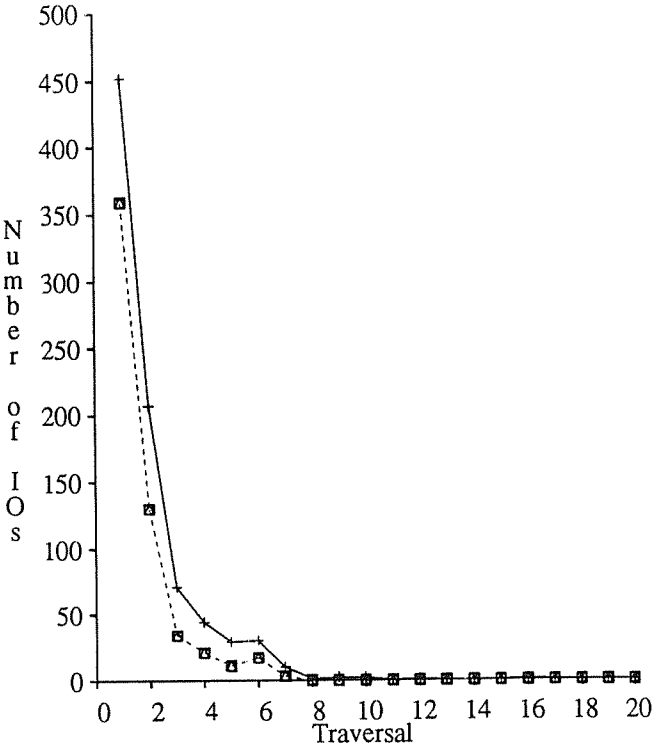
5.5.7.3. Results for the 200,000 Part Database

We now present the results for the 200,000 part database. The 200,000 part database was large enough so that the working set of each benchmark did not fit in the memory of our machine. The size of the database in the ESM and in Cricket (with and without padding) is shown in Table 5.7.

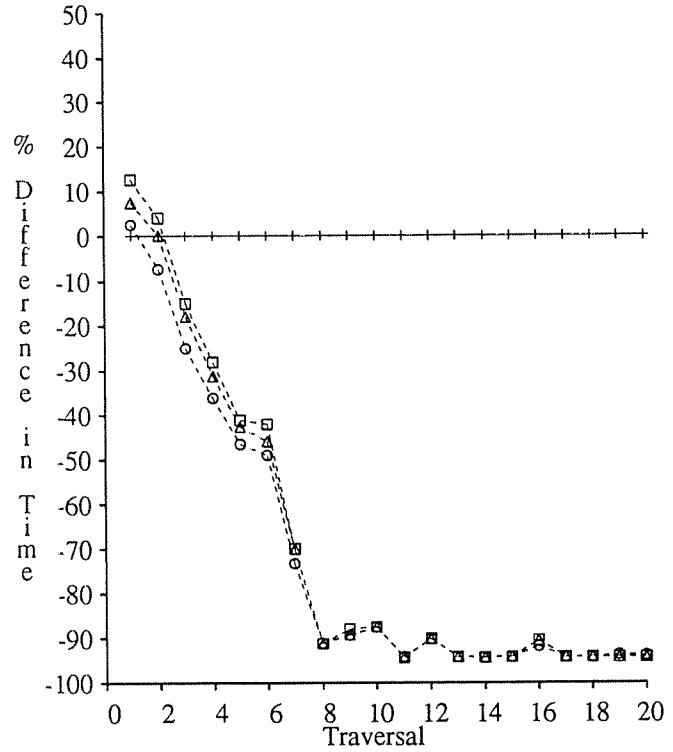
In generating the results for the 200,000 part database, we were faced with the problem of chosing a buffer size for the ESM. By default, Cricket will use as much physical memory as Mach will give it. Consequently, its effective buffer size dynamically adjusts itself to the system load. In contrast, the size of the ESM's buffer pool is fixed at startup, and its clients must statically choose the buffer size that they think is appropriate. In practice, this presents something of a dilemma, since choosing a buffer size that is too small will result in extra I/O and poor



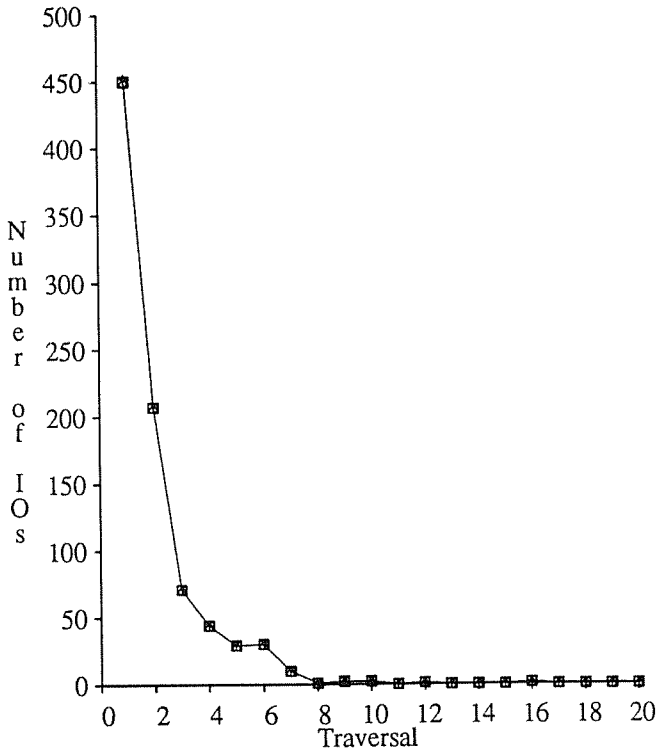
Graph 5.5: % Difference in Traversal Time
20,000 Parts, Unpadded Cricket Results



Graph 5.6: Number of I/Os Per Traversal
20,000 Parts, Unpadded Cricket Results



Graph 5.7: % Difference in Traversal Time
20,000 Parts, Padded Cricket Results



Graph 5.8: Number of I/Os Per Traversal
20,000 Parts, Padded Cricket Results

Setting	Elapsed Time (in sec) Cold/Warm	Number of I/Os Cold/Warm	Avg. Time (in msec) Per I/O
ESM (9 Mbyte buffer)	15.18/1.18	450/0	31.98
CRK·$\overline{EXC}$·$\overline{PB}$ (unpadded)	12.28/0.07	359/0	31.37
CRK·*EXC*·$\overline{PB}$ (unpadded)	13.49/0.07	359/0	31.37
CRK·*EXC*·*PB* (unpadded)	13.04/0.07	359/0	31.37
CRK·$\overline{EXC}$·$\overline{PB}$ (padded)	15.56/0.07	450/0	31.98
CRK·*EXC*·$\overline{PB}$ (padded)	17.10/0.07	450/0	31.98
CRK·*EXC*·*PB* (padded)	16.32/0.07	450/0	31.98

Table 5.6: Summary of Traversal Results for the 20,000 Part Database

Storage System	Database Size	Avg. Part Size	Avg. Parts Per Page
ESM	8,475 pages (33.1 Mbytes)	168 bytes	24
Cricket (unpadded)	5,756 pages (22.5 Mbytes)	116 bytes	35
Cricket (padded)	8,475 pages (33.1 Mbytes)	168 bytes	24

Table 5.7: The Size of the 200,000 Part Database

performance, while choosing a buffer size that is too big will also result in poor performance because of double paging [Bric76]. In a dynamic environment, with several applications running at once, the correct choice may be difficult to make.

Before the final benchmark runs, a binary search on buffer sizes was used to find the size resulting in the best performance for the ESM. It turned out that a 9 Mbyte buffer gave the best results for the lookup benchmark, while an 11.25 Mbyte buffer gave the best results for the traversal benchmark. The reason for these sizes can be understood by recalling that our machine had about 12.5 Mbytes[6] of physical memory available for paging. Since the lookup benchmark made heavy use of the 2.3 Mbyte main-memory part index, it needed a buffer that could hold a large working set but at the same time was small enough to ensure that the part index was never swapped out by Mach.[7] In contrast, the traversal benchmark made little use of the part index. Consequently, it needed a buffer that was as big as possible but not so big that it caused double paging. Of course, with a mixed workload like the one in

[6] 12.5 Mbytes appeared to be an upper bound. Because of kernel tasks and Mach's memory management policies, it was difficult to determine exactly how much physical memory was really available to a user task.

[7] Note that swapping is particularly expensive in Mach because it is done "outside" the kernel by a default external pager.

the Sun Benchmarks, a client would normally have to choose one buffer size. Therefore, we wanted to choose one buffer size for both benchmarks. It turned out that a 9 Mbyte buffer with an LRU replacement algorithm resulted in the best all-around performance for the ESM, so that was the size that was used to generate our results. Because of the high reference locality in part connections, traversal times were for the ESM only slightly worse with the 9 Mbyte buffer than they were with the 11.25 Mbyte buffer.

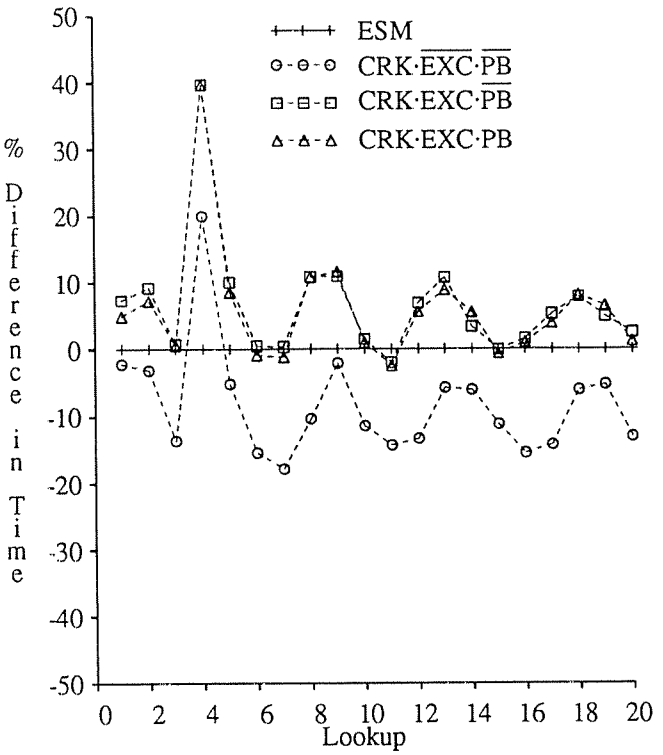
Results for the Lookup Benchmark on the 200,000 Part Database

The results for the lookup benchmark on the 200,000 part database are presented in Graphs 5.9-5.12. The results are displayed in the same manner as before. Once again, the "bumpiness" seen in all the graphs was caused by an irregular I/O pattern.

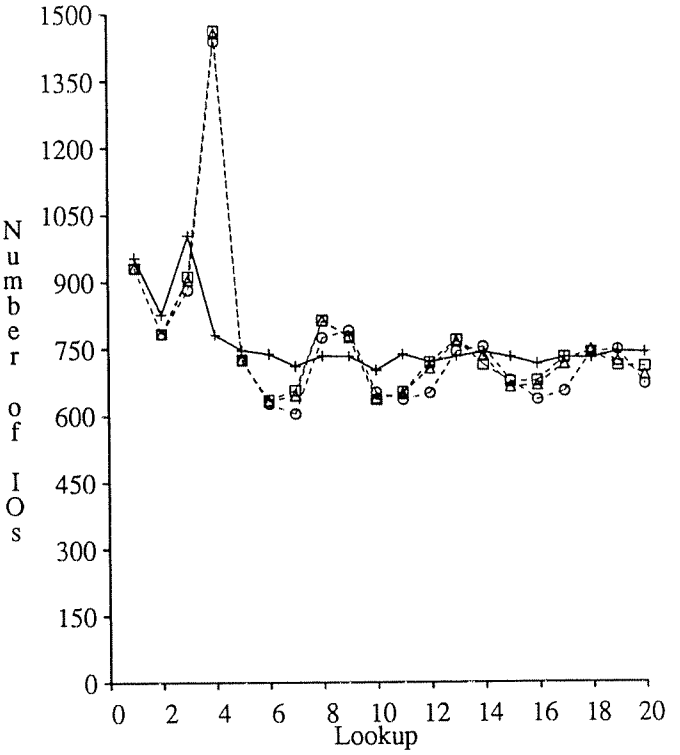
Beginning with Graph 5.9, we see that the relative performance of Cricket on this benchmark was mixed. With exception handling disabled, its performance ranged between 20% faster to 20% slower than the ESM. And with exception handling enabled, its performance ranged from just a trace faster to 40% slower than the ESM. The difference between the results with exception handling enabled and those with it disabled (about a 20% spread) again illustrates the high CPU cost of using Mach's exception handling facilities. That cost is more apparent here than it was in earlier graphs because more exceptions were generated per lookup in this benchmark.

The relative performance of Cricket in Graph 5.9 can be understood by looking at Graph 5.10. Once again, Cricket's relative performance was largely a function of how much I/O it did. It therefore comes as no surprise that Cricket's performance improved with less I/O.

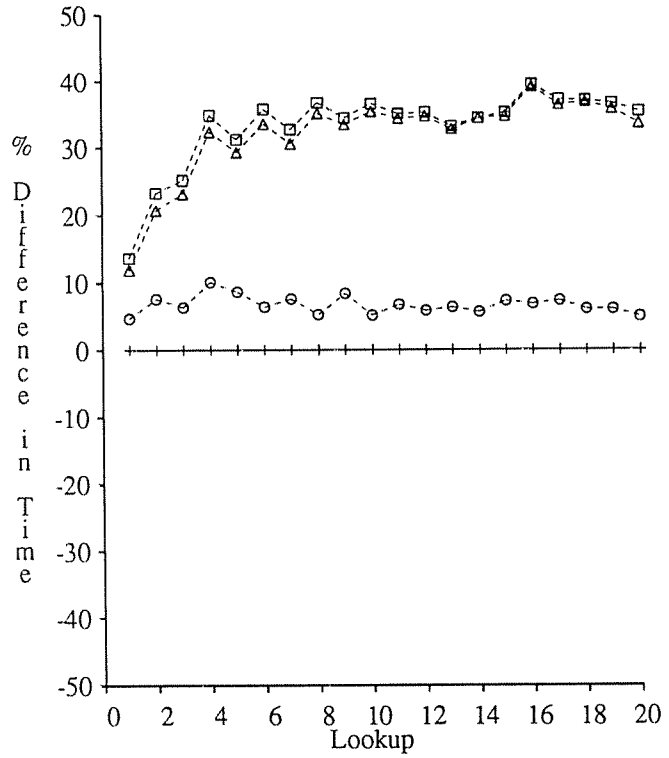
One interesting aspect of the results which is not directly apparent from Graph 5.10 is that the main-memory part index was being swapped in Cricket. More specifically, Mach was generating about 70 swapping I/Os for the part index on each lookup of 1,000 parts in Cricket. (Swapping I/O is included in Graph 5.10.) The spike in Graph 5.10 on lookup 4 was in fact caused by swapping activity. By lookup 4, memory had started to turn over, and since some of the part index's pages were further down on Mach's LRU stack, they started getting swapped out at that point. In general, the swapping activity that was observed in Cricket illustrates the problem with using a simple, global LRU replacement algorithm for all data. This is a well-known problem in the database literature [Stone81], and is one of the reasons why indexes and system meta-data should probably be treated differently than normal data in Cricket (although that was not done here).



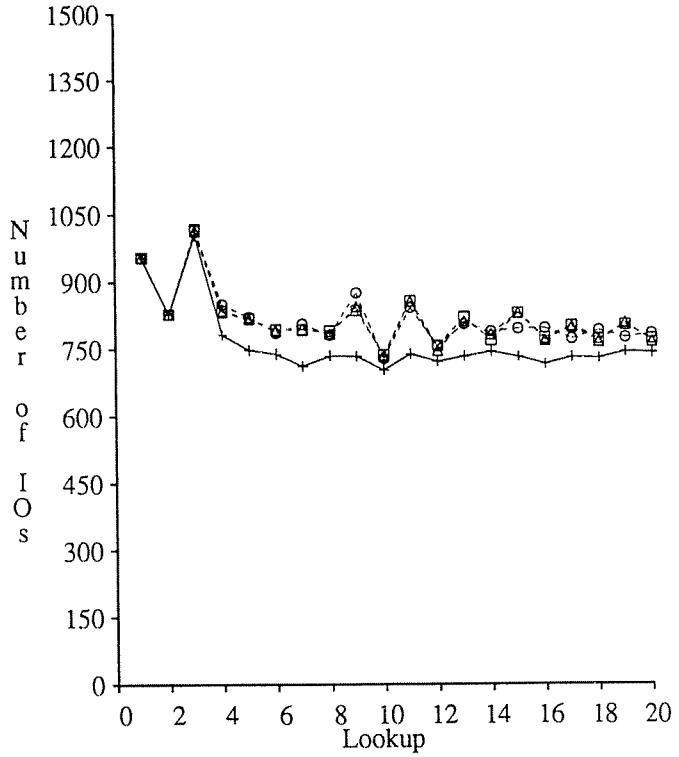
Graph 5.9: % Difference in Lookup Time
200,000 Parts, Unpadded Cricket Results



Graph 5.10: Number of I/Os Per Lookup
200,000 Parts, Unpadded Cricket Results



Graph 5.11: % Difference in Lookup Time
200,000 Parts, Padded Cricket Results



Graph 5.12: Number of I/Os Per Lookup
200,000 Parts, Padded Cricket Results

In contrast to Cricket, almost no swapping activity occurred in the ESM. This was because there were effectively two buffer groups [Chou85] in the ESM benchmarks. One buffer group, which existed in the stack of the benchmark program, contained the part index and was managed by Mach. The other buffer group, which existed in the ESM's buffer pool, contained part data and was managed by the ESM. By doing a binary search on buffer sizes for the ESM, we ended up chosing the optimal allocation for each (effective) buffer group.

Turning to Graphs 5.11 and 5.12, we see that the performance of the ESM was uniformly better than Cricket on the padded database. As Graph 5.12 shows, Cricket actually ended up doing more I/O than the ESM in that case, mainly because of swapping activity. This extra I/O and the high CPU cost of using Mach's external pager and exception handling facilities were the reasons for Cricket's poor performance. Note that the difference in Cricket's performance with exception handling enabled and with it disabled is much larger with the padded database than it was with the unpadded database (see Graphs 5.9 and 5.11). This is because the padded database was spread over a larger number of pages, which in turn decreased the likelihood of a page being re-visited. More exceptions were generated per lookup as a result, causing the relative performance of Cricket with exception handling enabled to worsen.

Table 5.8 summarizes the absolute results for the lookup benchmark on the 200,000 part database. It is worth noting that the time per I/O is higher here than it was in the previous lookup results. This is due to the larger database, which led to larger seeks and slower I/O. Note that the I/O counts in Table 5.8 include swapping I/O; the average time per I/O, however, is only for database I/Os. It was not possible to measure the time per swapping I/O, as swapping was out of our control.

Results for the Traversal Benchmark on the 200,000 Part Database

The final set of results are for the traversal benchmark on the 200,000 part database. Those results are presented in Graphs 5.13-5.16. By now, the graphs should almost be self-explanatory. In general, Cricket performed better when it did less I/O, and in the cases where it performed worse, it did about the same amount of I/O but suffered from the high CPU cost of using Mach's external pager and exception handling facilities. Note that, in contrast to the lookup benchmark, there were relatively few swapping I/Os in Cricket on this benchmark because the part index was only accessed once per traversal.

Setting	Elapsed Time (in sec) Cold/Warm	Number of I/Os Cold/Warm	Avg. Time (in msec) Per I/O
ESM (9 Mbyte buffer)	43.00/33.34	954/732	43.16
CRK·$\overline{EXC}$·$\overline{PB}$ (unpadded)	42.04/29.74	931/690	41.30
CRK·EXC·$\overline{PB}$ (unpadded)	46.15/34.81	931/714	41.30
CRK·EXC·PB (unpadded)	45.11/34.72	931/709	41.30
CRK·$\overline{EXC}$·$\overline{PB}$ (padded)	45.01/35.41	954/783	43.16
CRK·EXC·$\overline{PB}$ (padded)	48.88/45.70	954/784	43.16
CRK·EXC·PB (padded)	48.11/45.44	954/781	43.16

Table 5.8: Summary of Lookup Results for 200,000 Part Database

As mentioned earlier, a 9 Mbyte buffer pool was used in the ESM for this benchmark, which was sub-optimal. This shows up in Graph 5.16, where we see that the ESM did slightly more I/O than Cricket even when the padded database was used. The results for the ESM with its buffer size set to the optimal value of 11.25 Mbytes were about 5% to 10% better than those shown. Table 5.9 summarizes the absolute results for the traversal benchmark on the 200,000 part database.

Setting	Elapsed Time (in sec) Cold/Warm	Number of I/Os Cold/Warm	Avg. Time (in msec) Per I/O
ESM (9 Mbyte buffer)	54.66/44.59	1560/1225	33.55
CRK·$\overline{EXC}$·$\overline{PB}$ (unpadded)	48.54/27.48	1350/749	32.97
CRK·EXC·$\overline{PB}$ (unpadded)	54.02/32.17	1350/761	32.97
CRK·EXC·PB (unpadded)	52.97/32.12	1350/759	32.97
CRK·$\overline{EXC}$·$\overline{PB}$ (padded)	57.75/41.47	1558/1107	33.55
CRK·EXC·$\overline{PB}$ (padded)	64.19/51.56	1588/1114	33.55
CRK·EXC·PB (padded)	62.65/51.33	1558/1109	33.55

Table 5.9: Summary of Traversal Results for 200,000 Part Database

5.5.8. Some of the Shortcomings in Our Results

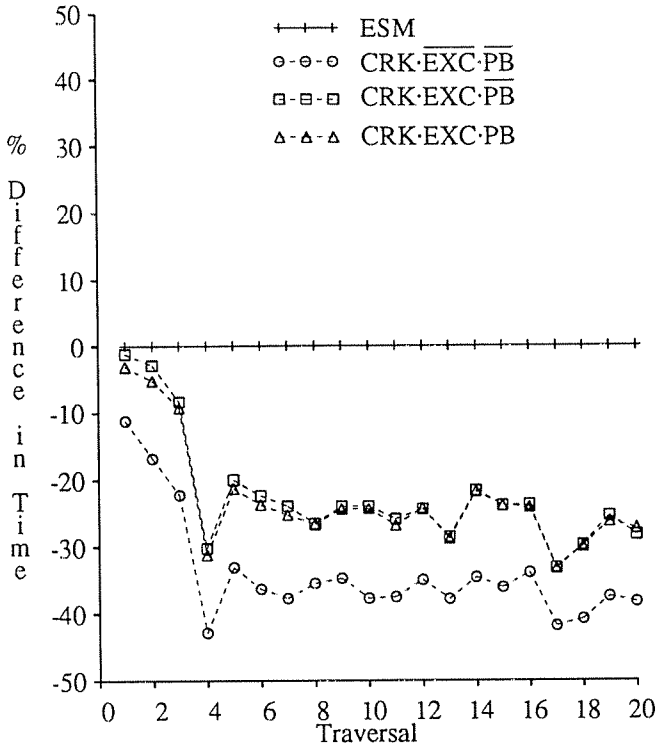
Before drawing any conclusions, the reader should be cautioned against reading too much into the results presented here. Although the results represent the best data that could be obtained at the present time, they suffer from a number of shortcomings. Some of these shortcomings are briefly discussed below.

5.5.8.1. Apples and Oranges?

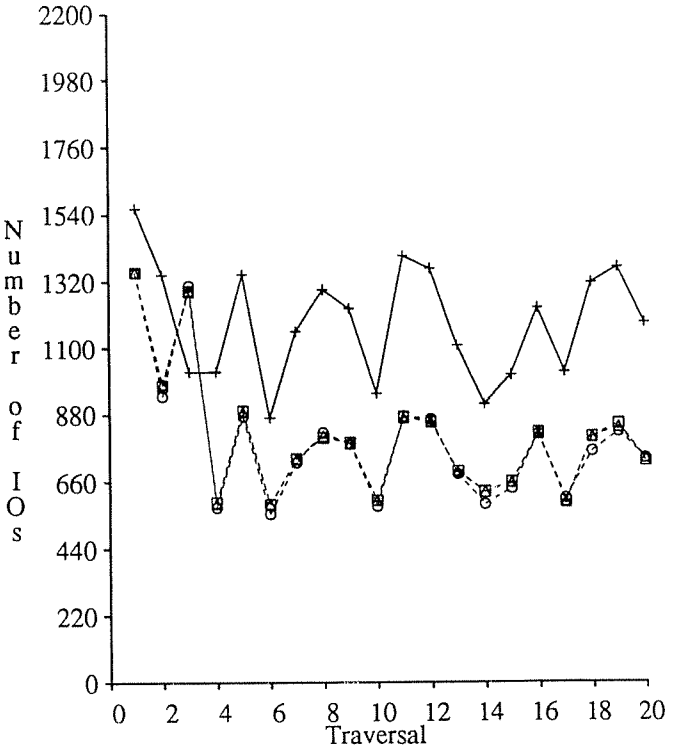
One shortcoming is that the functionality provided by the ESM and Cricket is substantially different, which in turn makes it hard to conduct a fair comparison of the two systems with a simple benchmark measure like the one used here. For example, Cricket is a multi-threaded, multi-user storage system that permits physical sharing of data between clients on the same machine. In contrast, the version of the ESM used here is strictly a single-user system. This difference undoubtedly biases the results in favor of the ESM. It is unclear how much slower a multi-user ESM would be, but there is little doubt that it would indeed be slower. Although it has not been done, the cost of using Mach's external pager and exception handling facilities could probably be reduced if a single-user version of Cricket was built. The reduced cost would result from eliminating the context switches that are required in Cricket for paging and exception handling.

Another area where the ESM and Cricket differ in terms of functionality is the form of protection that they provide. The philosophy in Cricket is that clients can do whatever they want to data for which they have gained access. However, system meta-data, such as page-allocation bitmaps, are protected against client access. The resulting protection mechanism is similar to that of the UNIX file system, where clients can do whatever they want to their own data, but the integrity of the file system itself (the inodes, the superblock, etc.), is protected. No notion of object identity is provided in Cricket, however, and this may be inadequate for many applications. In contrast, the ESM supports object identity and protects against dangling OIDs. However, its buffer pool and all system meta-data remains unprotected, which may make it difficult to guarantee certain kinds of safety.

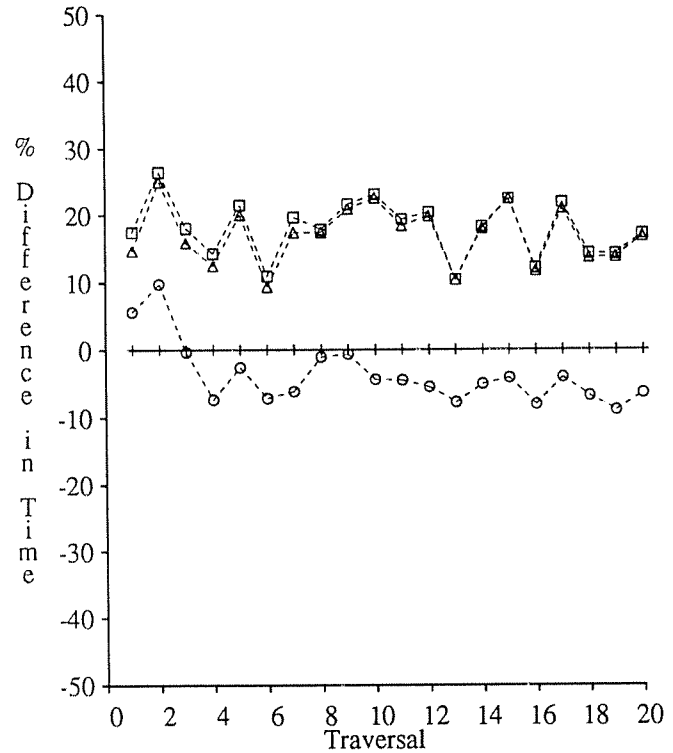
Finally, the ESM is able to support much larger databases than Cricket, at least on conventional hardware with 32-bit addresses. In fact, on the Vax, where 1/2 of the address space is reserved for the kernel, the effective size of a database in Cricket is limited to slightly less than 2 Gbytes after accounting for program stack and data space. For many applications, that may not be sufficient, and multiple databases would be required. Unfortunately, simultaneous access to multiple databases is not possible on the Vax because only one database can be mapped into the vir-



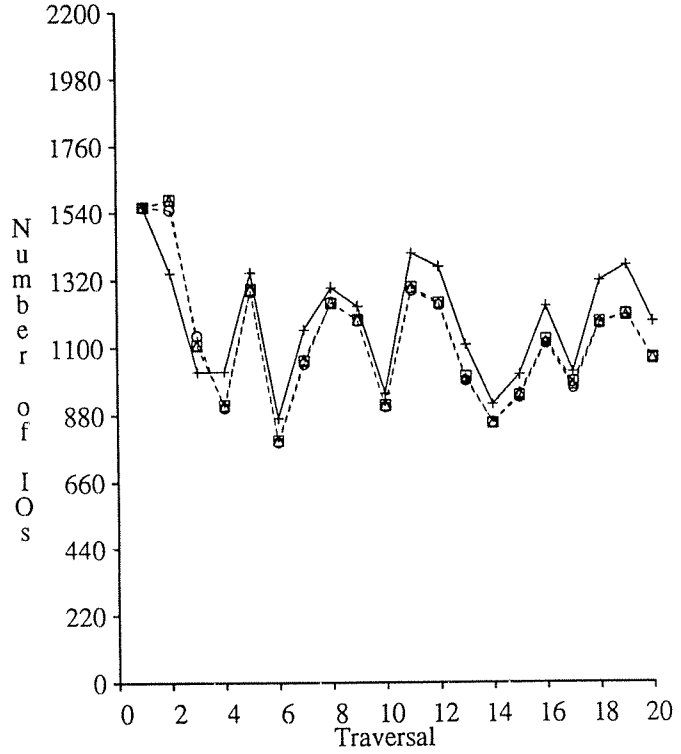
Graph 5.13: % Difference in Traversal Time
200,000 Parts, Unpadded Cricket Results



Graph 5.14: Number of I/Os Per Traversal
200,000 Parts, Unpadded Cricket Results



Graph 5.15: % Difference in Traversal Time
200,000 Parts, Padded Cricket Results



Graph 5.16: Number of I/Os Per Traversal
200,000 Parts, Padded Cricket Results

tual address space of a client at a time.[8]

5.5.8.2. Impact of the Vax Architecture

Another shortcoming of our results is that they were run on a Vax, which introduces a high procedure call overhead that would not be present on a modern RISC architecture. This overhead affected both systems, biasing the results in favor of Cricket on the 20,000 part database and in favor of the ESM on the 200,000 part database. On the 20,000 part database, most of the database had been read into memory after the first few benchmark executions. Once that happened, the ESM's relative performance was largely determined by how much time each *ReadObject* call took. Since each *ReadObject* call results in about 7 nested procedure calls when everything is in memory, the high procedure call overhead on the Vax probably hurt the relative performance of the ESM more on the 20,000 part database. On the other hand, the 200,000 part database never fit in memory, and each database page was referenced only once or twice while it was in memory. Since the first access to a non-resident page in Cricket goes through Mach's external pager mechanism, which involves several procedure calls (certainly more than 7), the high procedure call overhead probably hurt the relative performance of Cricket more on the 200,000 part database.

The memory management hardware of the Vax, which uses a two-level page table, also had an impact on the results. Although the Vax's page tables themselves are not paged in Mach, they may be discarded when the number of free page frames starts to run low [Rash87]. The discarded parts of a page table are reconstructed if needed again later. It is hard to say for certain, but the cost of managing page tables in this manner probably affected Cricket more than the ESM. On architectures that use an inverted page table, such as IBM's RS6000 architecture [Bako90] or Hewlett-Packard's Precision architecture [Maho86], it may be less costly to maintain the data structures needed for memory management. The relative performance of Cricket would probably improve on such architectures.

Finally, context switching is not particularly fast on the Vax. Since there is frequent context switching between clients and the Cricket server, the relative performance of Cricket would probably improve on an architecture with faster context switching. Of course, the flip side is that it would probably perform worse on architectures with slower context switching.

---

[8] Note that on segmented architectures such as the Hewlett-Packard's Precision architecture [Maho86], simultaneous access to multiple 4 Gbyte databases might be possible with a small amount of compiler support.

5.5.8.3. Impact of Slow Disks and No Read-Ahead

The fact that rather slow disks were used also affects the relative results that were obtained. As the results indicated, the ESM generally outperformed Cricket whenever there was a lot of disk I/O. This was mainly due to the added CPU cost of using Mach's external pager facilities. With slow disks, however, this cost is overshadowed by the high real-time cost of doing I/O. If faster disks had been used, this cost would have made up a larger percentage of the total cost, and the ESM would have performed better in relative terms as a result.

Another shortcoming related to disk I/O is that single-page reads with no read-ahead were performed in all of the benchmarks. If read-ahead or multi-page reads had been used, the number of *memory_object_data_request* and *memory_object_data_provided* messages that are exchanged between the Mach kernel and the Cricket server to read data from disk would have been reduced. Each *memory_object_data_provided* message from Cricket could have provided, say, a disk track's worth of data in reply to a *memory_object_data_request* message instead of providing just one page. This in turn would have significantly improved the relative performance of Cricket, as the cost of exchanging *memory_object_data_request* and *memory_object_data_provided* messages is quite high. Since many SCSI disk controllers do track-sized reads regardless of the number of pages requested, it may often be quite reasonable to replace single-page reads with sequential, multi-page reads of this sort in systems with large memories.

5.5.8.4. Beta Version of Mach

The final shortcoming that needs to be mentioned is that the benchmarks were run on what should really be considered a beta version of Mach. It is impossible to predict exactly how much the performance of Mach will improve in the future, but it should indeed improve. In fact, performance results for the most recent version of Mach (version 3.0) [Drav90] indicate that its IPC facilities are 30% faster than the version of Mach that was used here (version 2.5). Since Cricket relies extensively on Mach, it would obviously have more to gain from any improvements in Mach's performance than the ESM would.

5.6. CONCLUSIONS

In this chapter, we argued that traditional database storage systems are ill suited for emerging application areas like CAD/CAM, and that an approach based on a single-level store may offer better performance. We then described a prototype of a novel storage system called Cricket. Cricket uses the memory management primitives of

the Mach operating system to provide the abstraction of a shared, transactional, single-level store that can be directly accessed by user applications. The main purpose in building Cricket was to explore the feasibility of using a single-level store in emerging application areas such as CAD/CAM. We also wanted to explore whether Mach's memory management primitives can be used effectively by a database storage system.

Much of the chapter was devoted to a performance study in which Cricket was compared to the EXODUS Storage Manager (ESM) on the Sun Benchmarks. Results were presented for the Sun lookup and traversal benchmarks on databases with 20,000 and 200,000 parts. Although several graphs and tables of results were presented, in general, the relative performance of each system simply depended on how much I/O activity there was. When all of the data was in memory and there was little or no I/O activity at all, Cricket was 90% to 95% faster than the ESM on both benchmarks. Cricket was faster under those circumstances because its single-level store allowed it to access data directly instead of going through a per-object, procedure-based interface as in the ESM. Conversely, whenever there was I/O activity, the relative performance of Cricket suffered. In particular, for the same level of I/O activity, Cricket was roughly 10% to 40% slower than the ESM when exception handling was enabled to trigger transparent, page-level locking, and it was 5% to 10% slower than the ESM when exception handling was disabled. Cricket was slower than the ESM in both cases because of the high CPU cost associated with using Mach's external pager and exception handling facilities.

The results that were obtained for the Sun Benchmarks clearly illustrate some of the strengths and weaknesses of Cricket. The results for the 20,000 part database indicate that the approach taken in Cricket has the potential to perform well on applications whose working sets fit in memory. On the other hand, the results for the 200,000 part database indicate that the approach taken in Cricket may perform poorly on more traditional database applications. As we noted, our results suffer from a number of shortcomings, including the fact that they were obtained on a relatively slow CPU using a beta version of Mach. In general, though, the CPU costs associated with using Mach's external pager and exception handling facilities are so high (anywhere from 3 to 7 msec per 4 Kbyte page on our machine) that our conclusions are unlikely to drastically change even under different testing conditions.

5.6.1. Directions for Future Work

As mentioned in Section 5.4, there are a number of important design issues that remain largely unresolved in Cricket. These include how to manage external and internal fragmentation at the disk allocation level, how to handle object growth, files, index management, recovery, and how to optimize Cricket for a client/server computing

environment. Clearly, all of these design issues deserve further attention, as all must be resolved in order to build a full-function storage system. Once these issues are resolved, a more complete version of Cricket could be implemented and used as the basis for further performance studies. Other possibilities for future work include support for versions and implementation of persistent programming languages on top of Cricket.

CHAPTER 6

THESIS CONCLUSIONS

Whether or not the next-generation database systems being developed today will succeed in emerging application areas will largely depend on how well they perform. As a result, finding ways to improve their performance is an important research area. This thesis described and analyzed three different implementation techniques for improving the performance of next-generation database systems.

In Chapter 3, we introduced the notion of field replication and showed how it could be used to eliminate or reduce the cost of some functional joins. The in-place replication strategy and the separate replication strategy were discussed, and for both of these strategies, we showed how inverted paths can be used to keep replicated data consistent. An analytical cost model was developed to give some feel for how beneficial each replication strategy can be and the circumstances under which each strategy breaks down. While field replication is a relatively simple notion, the results of the analysis showed that it can provide significant performance gains in many situations.

In Chapter 4, we described how physical pointers can be used effectively in join processing. There, we described several pointer-based join algorithms that are simple variations on the well known nested-loops, sort-merge, hybrid-hash, and hash-loops join algorithms used in relational database systems. We also described a pointer-based join algorithm called PID-partitioning, which has no standard counterpart and is somewhat less general than the other pointer-based join algorithms. An analytical cost model was again developed to compare the performance of the pointer-based algorithms to their standard counterparts. The results of the analysis showed that the pointer-based algorithms can often provide significant performance gains over conventional, value-based join algorithms.

Finally, in Chapter 5, we argued that traditional database storage systems are poorly suited for some emerging application areas such as CAD/CAM, and that an approach based on a single-level store may offer better performance. We then described a prototype storage system called Cricket that was built to explore the feasibility of such an approach. Cricket uses the memory management primitives of the Mach operating system to provide the abstraction of a shared, transactional, single-level store. Performance results for the Sun Benchmarks indicated that the approach taken in Cricket has the potential to perform well on applications whose working sets fit in memory. Our

results also indicated that, due to the high CPU cost of using Mach's external pager and exception handling facilities, the approach taken in Cricket may perform poorly on more traditional database applications.

6.1. Directions for Future Work

In each research chapter, we discussed possible directions for future work. For field replication, the possibilities that were mentioned include an implementation study, an examination of how access relations [Kemp90] might be used to support both replication and path indexes, and an examination of how full-object replication can be used to cluster shared sub-objects of complex objects. For pointer-based joins, the possible directions for future work include an implementation study, an examination of how the index AND'ing techniques described in [Moha90] might be used in pointer-based joins, and work on parallelizing pointer-based joins. Finally, a long list of possibilities for future work was given for Cricket. These include how to manage external and internal fragmentation at the disk allocation level; how to handle object growth, files, index management, recovery; how to optimize for a client/server computing environment; how to support versions; and how best to implement persistent programming languages on top of Cricket.

REFERENCES

[Acce86] M. Accetta et al., "Mach: A New Kernel Foundation for UNIX Development," *Proc. of the Summer Usenix Conf.*, July 1986.

[Agra89] R. Agrawal and N. Gehani, "ODE (Object Database and Environment): The Language and the Data Model," *Proc. of the 1989 ACM SIGMOD Conf.*, Portland, OR, May 1989.

[Appe86] A. Appel et al., "Garbage Collection Can be Faster Than Stack Allocation," Computer Science Tech. Report 045-86, Princeton Univ., June 1986.

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

[Atwo90] T. Atwood and S. Hanna, "Two Approaches to Adding Persistence to C++," *Proc. of the 4th Intl. Workshop on Persistent Object Systems: Design, Implementation and Use*, Marthas Vineyard, MA, Sept. 1990.

[Bako90] H. Bakoglu et al., "The IBM RISC System/6000 Processor: Hardware Overview," *IBM Journal of Research and Development*, 34(1), 1990.

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

[Bens72] A. Bensoussan et al., "The Multics Virtual Memory: Concepts and Design," *CACM*, 15(5), May 1972.

[Bert89] E. Bertino and W. Kim, "Indexing Techniques for Queries on Nested Objects," *IEEE Trans. on Knowledge and Data Engineering*, June 1989.

[Blac88] D. Black et al., "The Mach Exception Handling Facility," Computer Science Tech. Report 88-129, Carnegie Mellon Univ., April 1988.

[Blak86] J. Blakeley et al., "Efficiently Updating Materialized Views," *Proc. of the 1986 ACM SIGMOD Conf.*, Washington, DC, May 1986.

[Blas77] M. Blasgen and K. Eswaran, "Storage and Access in Relational Databases," *IBM Systems Journal*, 16(4), 1977.

[Bora90] H. Boral et al., "Prototyping Bubba, A Highly Parallel Database System" *IEEE Trans. on Data and Knowledge Eng.*, 2(1), March 1990.

[Bric76] P. Brice and S. Sherman, "An Extension of the Performance of a Database Manager in a Virtual Memory System using Partially Locked Virtual Buffers," *ACM Trans. on Database Systems*, 6(1), June 1976.

[Care88] M. Carey et al., "A Data Model and Query Language for EXODUS," *Proc. of the 1988 ACM SIGMOD Conf.*, Chicago, IL, May 1988.

[Care89] M. Carey et al, "The EXODUS Extensible DBMS Project: An Overview," in *Readings in Object-Oriented Databases*, S. Zdonik and D. Maier, eds., Morgan-Kaufman Publ. Co., 1989.

[Care90] M. Carey et al, "An Incremental Join Attachment for Starburst," *Proc. of the 1990 VLDB Conf.*, Bisbane, Australia, Aug. 1990.

[Catt90] R. Cattell and J. Skeen, "Engineering Database Benchmark (DRAFT)," Sun Technical Report - Database Engineering Group, April 1990.

[Chan88] A. Chang and M. Mergen, "801 Storage: Architecture and Programming," *ACM Trans. on Computer Systems*, 6(1), Feb. 1988.

[Chan89a] E. Chang, "Effective Clustering and Buffering in an Object-Oriented DBMS," Ph.D thesis, Computer Science Tech. Report CSD 89/515, Univ. of California, Berkeley, June 1989.

[Chan89b] E. Chang and R. Katz, "Exploiting Inheritance and Structure Semantics for Effective Clustering and Buffering in an Object-Oriented DBMS", *Proc. of the 1989 ACM SIGMOD Conf.*, Portland, OR, June 1989.

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

[Cock84] W. Cockshott et al., "Persistent Object Management Systems," *Software-Practice and Experience*, vol. 14, 1984.

[Cope84] G. Copeland and D. Maier, "Making Smalltalk a Database System," *ACM Trans. on Database Systems* 4(4), Dec. 1979.

[Cope90] G. Copeland et al., "Uniform Object Management," *Proc. of the Intl. Conf. on Extending Database Technology*, Venice, Italy, March 1990.

[Date86] C. Date, "An Introduction to Database Systems," pg. 56, Addison-Wesley, Reading, MA, 1986.

[Date87] C. Date, "A Guide to The SQL Standard," pp. 113-120, Addison Wesley, Reading, MA, 1987.

[Deux90] O. Deux et al, "The Story of O2," *IEEE Trans. on Knowledge and Data Eng.*, March 1990.

[Dewi84] D. DeWitt et al., "Implementation Techniques for Main Memory Database Systems," *Proc. of the 1984 ACM SIGMOD Conf.*, Boston, MA, May 1984.

[Dewi85] D. DeWitt and R. Gerber, "Multiprocessor Hash-Based Join Algorithms," *Proc. of the 1985 VLDB Conf.*, Stockholm, Sweden, Aug. 1985.

[Dewi90] D. DeWitt et al., "A Study of Three Alternative Workstation-Server Architectures for Object-Oriented Database Systems," Computer Science Tech Report #907, Univ. of Wisconsin, Jan. 1990.

[Drav88] R. Draves and E. Cooper, "C Threads," Computer Science Tech. Report 88-154, Carnegie Mellon Univ., June 1988.

[Drav90] R. Draves, "A Revised IPC Interface," *Proc. of the USENIX Mach Workshop*, Burlington, VT, Oct. 1990.

[Duch89] D. Duchamp, "Analysis of Transaction Management Performance," *Proc. of the 11th Symposium on Operating System Principles*, Litchfield Park, AZ, Dec. 1989.

[Grae90] G. Graefe, "Parallel External Sorting in Volcano," Computer Science Tech. Report, University of Colorado, May 1990.

[Eppi89] J. Eppinger, "Virtual Memory Management for Transaction Processing Systems," Ph.D thesis, Computer Science Tech. Report 89-115, Carnegie Mellon Univ., Feb. 1989.

[Fish87] D. Fishman et al. "Iris: An Object-Oriented Database Management System," *ACM Trans. on Office Info. Sys.* 5(1), Jan. 1987.

[Ford88] S. Ford et al., "ZEITGEIST: Database Support for Object-Oriented Programming," *The 2nd Workshop on Object-Oriented Database Systems*, Oct. 1988.

[Haas90] L. Haas et al, "Starburst Mid-Flight: As the Dust Clears," *IEEE Trans. on Knowledge and Data Engineering*, March 1990.

[Hans87] E. Hanson, "A Performance Analysis of View Materialization Strategies," *Proc. of the 1987 ACM SIGMOD Conf.*, San Francisco, CA, May 1987.

[Hans88] E. Hanson, "Processing Queries Against Database Procedures, A Performance Analysis," *Proc. of the 1988 ACM SIGMOD Conf.*, Chicago, IL, May 1988.

[Horn87] M. Hornick and S. Zdonik, "A Shared, Segmented Memory System for an Object-Oriented Database," *ACM Trans. on Office Info. Sys.* 5(1), Jan. 1987.

[Jhin88] A. Jhingran, "A Performance Study of Query Optimization Algorithms on a Database System Supporting Procedures," *Proc. of the 1988 VLDB Conf.*, Los Angeles, CA, Aug. 1988.

[Jhin90] A. Jhingran and M. Stonebraker, "A Comparison of Two Representations for Complex Objects," Computer Science Tech. Report ERL M90/32, Univ. of California, Berkeley, April 1990.

[Katz87] R. Katz and E. Chang, "Managing Change in a Computer-Aided Design Database," *Proc. of the 1987 VLDB Conf.*, Brighton, England, Aug. 1987.

[Kemp90] A. Kemper and G. Moerkotte, "Access Support in Object Bases," *Proc. of the 1990 ACM SIGMOD Conf.*, Atlantic City, NJ, May 1990.

[Kern78] B. Kernighan and D. Ritchie, "The C Programming Language," Prentice-Hall, Englewood Cliffs, NJ, 1978.

[Kim90] W. Kim et al., "Architecture of the ORION Next-Generation Database System," *IEEE Trans. on Knowledge and Data Engineering*, March 1990.

[Knut68] D. Knuth, *The Art of Computer Programming: Fundamental Algorithms, Vol. 1*, Addison-Wesley, Reading, MA, 1968.

[Knut73] D. Knuth, *The Art of Computer Programming: Sorting and Searching, Vol. 3*, Addison-Wesley, Reading, MA, 1973.

[Kuma87] A. Kumar and M. Stonebraker, "Performance Evaluation of an Operating System Transaction Manager," *Proc. of the 1987 VLDB Conf.*, Brighton, England, Aug. 1987.

[Lehm89] T. Lehman and B. Lindsay, "The Starburst Long Field Manager," *Proc. of the 1989 VLDB Conf.*, Amsterdam, The Netherlands, Aug. 1989.

[Li86] K. Li and P. Hudak, "Memory Coherence in Shared Virtual Memory Systems," *Proc. of the 5th Annual ACM Symposium on Principles of Distributed Computing*, Aug. 1986.

[Lind87] B. Lindsay et al., "A Data Management Extension Architecture," *Proc. of the 1987 ACM SIGMOD Conf.*, San Francisco, CA, May 1987.

[Lind88] B. Lindsay, IBM Almaden Research Center, personal communication, Fall 1988.

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

[Mack86] L. Mackert and G. Lohman, "R* Optimizer Validation and Performance Evaluation for Local Queries," *Proc. of the 1986 ACM SIGMOD Conf.*, Washington, DC, May 1986.

[Mack89] L. Mackert and G. Lohman, "Index Scans Using a Finite LRU Buffer: A Validated I/O Model," *ACM Trans. on Database Systems* 14(3), Sept. 1989.

[Maho86] M. Mahon et al., "Hewlett-Packard Precision Architecture: The Processor," *Hewlett-Packard Journal*, 37(8), Aug. 1986.

[Maie86a] D. Maier et al., "Development of an Object-Oriented DBMS," *Proc. of the 1st ACM OOPSLA Conf.*, 1986.

[Maie86b] D. Maier and J. Stein, "Indexing in an Object-Oriented DBMS," *Proc. of the Int'l Workshop on Object-Oriented Database Systems*, Pacific Grove, CA, Sept. 1986.

[Maie89] D. Maier, "Making Database Systems Fast Enough for CAD Applications," in *Object-Oriented Concepts, Database and Applications*, W. Kim and F. Lochovsky, eds., pp. 573-581, Addison-Wesley, Reading, MA, 1987.

[Mash90] J. Mashey, MIPS Corporation, personal communication, based on panel discussion at the *1990 Microprocessor Forum*, San Jose, CA, Oct. 1990.

[Moha89a] C. Mohan et al., "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging," IBM Research Report RJ6649, Jan. 1989.

[Moha89b] C. Mohan and F. Levine, "ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging," IBM Research Report RJ6846, Aug. 1989.

[Moha90] C. Mohan and D. Haderle, "Single Table Access Using Multiple Indexes: Optimization, Execution and Concurrency Control Techniques," IBM Research Report RJ7341, March 1990.

[Moss88] J. Moss and S. Sinofsky, "Managing Persistent Data with Mneme: Designing a Reliable, Shared Object Interface," in *Advances in Object-Oriented Database Systems*, vol. 334 of *Lecture Notes in Computer Science*, pp. 298-316. Springer-Verlag, 1988.

[Moss90] J. Moss, "Working with Persistent Objects: To Swizzle or Not to Swizzle," Computer Science Tech. Report 90-38, Univ. of Massachusetts, May 1990.

[Myer86] G. Myers et al., "Microprocessor Technology Trends," *Proc. IEEE*, 74(12), Dec. 1986.

[Obje90] Objectivity Inc., Menlo Park, CA, 1990.

[Oren90] J. Orenstein, Object Design Inc., personal communication, Spring 1990.

[Rash87] R. Rashid et al., "Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures (DRAFT)," Computer Science Tech. Report, Carnegie Mellon Univ., Feb. 1987.

[Rich89] J. Richardson, *E: A Persistent Systems Implementation Language*, Ph.D thesis, Computer Science Tech. Report #868, Univ. of Wisconsin, Aug. 1989.

[Rich90] J. Richardson, "Compiled Item Faulting: A New Technique for Managing I/O in a Persistent Language," *Proc. of the 4th Intl. Workshop on Persistent Object Systems: Design, Implementation and Use*, Marthas Vineyard, MA, Sept. 1990.

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

[Sche90] H. Schek et al., "The DASDBS Project: Objectives, Experiences, and Future Perspectives," *IEEE Trans. on Data and Knowledge Eng.*, 2(1), March 1990.

[Schu90] D. Schuh et al., "Persistence in E Revisited — Implementation Experiences," *Proc. of the 4th Intl. Workshop on Persistent Object Systems: Design, Implementation and Use*, Marthas Vineyard, MA, Sept. 1990.

[Sell87] T. Sellis, "Efficiently Supporting Procedures in Relational Database Systems," *Proc. of the 1987 ACM SIGMOD Conf.*, San Francisco, CA, May 1987.

[Seve76] D. Severance and G. Lohman, "Differential Files: Their Application to the Maintenance of Large Databases," *ACM Trans. on Database Systems* 1(3), Sept. 1976.

[Shap86] L. Shapiro, "Join Processing in Database Systems with Large Main Memories," *ACM Trans. on Database Systems* 11(3), Sept. 1986.

[Shek88] E. Shekita et al., *An Introduction to the EXODUS Storage Manager*, user documentation from the EXODUS Extensible DBMS Project, Univ. of Wisconsin, Madison, May 1988.

[Shek89] E. Shekita and M. Carey, "Performance Enhancement Through Replication in an Object-Oriented DBMS," *Proc. of the 1989 ACM SIGMOD Conf.*, Portland, OR, June 1989.

[Shek90a] E. Shekita and M. Carey, "A Performance Evaluation of Pointer-Based Joins," *Proc. of the 1990 ACM SIGMOD Conf.*, Atlantic City, NJ, May 1990.

[Shek90b] E. Shekita and M. Zwilling, "Cricket: A Mapped, Persistent Object Store," *Proc. of the 4th Intl. Workshop on Persistent Object Systems: Design, Implementation and Use*, Marthas Vineyard, MA, Sept. 1990.

[Ship81] D. Shipman, "The Functional Data Model and the Data Language DAPLEX," *ACM Trans. on Database Sys.* 6(1), Sept. 1987.

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

[Spec88] "The Guide to the Camelot Distributed Transaction Facility: Release 1," A. Spector and K. Swedlow eds., Carnegie Mellon Univ., 1988.

[Ston81] M. Stonebraker, "Operating System Support for Database Management," *CACM*, 24(7), July 1981.

[Ston84] M. Stonebraker, "Virtual Memory Transaction Management," *ACM Operating Systems Review*, 18(2), April 1984.

[Ston87] M. Stonebraker et al., "Extending a Database System with Procedures," *ACM Trans. on Database Sys.* 12(3), Sept. 1987.

[Ston90] M. Stonebraker et al., "The Implementation of POSTGRES," *IEEE Trans. on Data and Knowledge Eng.*, 2(1), March 1990.

[Trai82] I. Traiger, "Virtual Memory Management for Database Systems," *ACM Operating Systems Review*, 16(4), Oct. 1982.

[Vald87] P. Valduriez, "Join Indices," *ACM Trans. on Database Systems* 12(2), June 1987.

[Vers90] *Versant Product Profile*, Versant Object Technology Corporation, Menlo Park, CA, 1990.

[Welc84] T. Welch, "A Technique for High-Performance Data Compression," *IEEE Computer*, 17(6), June 1984.

[Wilk90a] K. Wilkinson et al., "The Iris architecture and Implementation," *IEEE Trans. on Knowledge and Data Engineering*, March 1990.

[Wilk90b] K. Wilkinson and M. Neimat, "Maintaining Consistency of Client-Cached Data," *Proc. of the 1990 VLDB Conf.*, Bisbane, Australia, Aug. 1990.

[Yao77] S. Yao, "Approximating Block Accesses in Database Organizations," *Comm. of the ACM* 20(4), April 1977.

[Youn87] M. Young et al., "The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System," *Proc. of the 11th Symposium on Operating System Principles*, Nov. 1987.

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

APPENDIX A

ANALYSIS FOR UNCLUSTERED/CLUSTERED INDEXES

In this appendix, small to medium-sized joins with the unclustered/clustered index combination are analyzed. Much of the analysis follows directly from that presented in Section 4.4.3, and the notation used here is the same as the notation used in that section.

Standard Index-Nested-Loops

As one would expect, the analysis for the index-nested-loops algorithm with the unclustered/clustered index combination is similar to the analysis that was presented earlier for that algorithm in Section 4.4.3. Here, the analysis proceeds by noting that the cost to read the index on R, as opposed to R itself, is the same as in the earlier analysis. This follows because the cost to read the index alone is the same whether it is a clustered or unclustered index. Furthermore, the cost to probe the index on S is also the same as before due to the fact that R and S are always assumed to be relatively unclustered. As a result of these observations, the only real differences from the earlier analysis are the cost to read R′ (the result of the selection on R) and the cost to access S.

A precise estimate of the cost to read R′ can be derived by once again making use of the Yao function Y( ). However, because the index on R is an unclustered index, and because the number of objects in R′ is small relative to R in the examples that we analyze, we can safely assume that each object in R′ resides on a different page in R. As a result, the cost to read R′ is approximately $|R'| \cdot IO$.

To determine the cost to access S, recall that the index-nested-loops algorithm sorts the objects in R′ by their join attribute before the index on S is probed. Because the index on S is a clustered index, this ensures that each page in S will be accessed only once. Therefore, the cost to access S can be calculated by determining how many pages in S participate in the join, which can be determined using the same analysis that was used in Section 4.4.3. Applying that analysis again, the cost to access S is:

$P_s \cdot Y(|R|, k \cdot O_s, |R'|) \cdot IO$

Based on the above analysis, and based on the analysis for the index-nested-loops algorithm with the clustered/unclustered index combination, the cost of the index-nested-loops algorithm with the unclustered/clustered index combination is:

$2 \cdot IO + 2 \cdot \log_2 b \cdot compare$	descend the index on R
$+ \lceil \lvert R' \rvert / b - 1 \rceil \cdot IO$	scan across the leaves of the index on R
$+ \lvert R' \rvert \cdot IO$	read R′
$+ \lvert R' \rvert \cdot move_r$	extract R′ from R in memory before sorting
$+ \lvert R' \rvert \cdot \log_2 \lvert R' \rvert \cdot (compare + swap_r)$	cost to sort R′
$+ IO$	read the root page of the index on S
$+ 2 \cdot \lvert R' \rvert \cdot \log_2 b \cdot compare$	probe the index on S for each object in R′
$+ \lceil \lvert S \rvert / b \rceil \cdot Y(\lvert R \rvert, b \cdot k, \lvert R' \rvert) \cdot IO$	access the leaf pages of the index on S
$+ P_s \cdot Y(\lvert R \rvert, k \cdot O_s, \lvert R' \rvert) \cdot IO$	access S

The Remaining Algorithms

The analysis for the remaining algorithms is straightforward. For each algorithm, it should be clear that the cost to read the index on R and the cost to read R′ is the same as in the preceding analysis. Furthermore, the cost to read S in each algorithm is the same as in Section 4.4.3 because R and S are always assumed to be relatively unclustered. Finally, none of the remaining algorithms use the index on S, so that cost does not need to be considered. Based on these observations, the cost of each of the remaining algorithms with the unclustered/clustered index combination can be obtained by taking the analysis for each algorithm as presented in Section 4.4.3 and changing the cost to read R′ to be $|R'| \cdot IO$.