Persistent LISP:
Storing Interobject References in a Database

By

Margaret Helen Butler

A.B. (University of California) 1982

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

in the

GRADUATE DIVISION

OF THE

UNIVERSITY OF CALIFORNIA, BERKELEY

Approved: [illegible] .... [illegible]
Chairman Date

Randy H. Katz 11/19/87

[illegible] 11/19/87

....

Persistent Lisp:
Storing interobject references in a database

Margaret Helen Butler

Abstract

Considerable recent activity has been directed toward providing data management facilities for artificial intelligence and computer-aided design. A major characteristic of the data in these environments is that data objects may contain references to other data objects. These *interobject references* introduce many issues that are studied in this dissertation. Since Lisp is a common language with interobject references, it has been used to study their characteristics when stored in a database management system. First, this dissertation reports on a persistent Lisp prototype in which Lisp stores application data in the database management system Ingres. Two Lisp applications were converted to use this prototype and their performance is analyzed.

Initial testing of a data intensive application displayed a slowdown of a factor of three over the performance when all data was stored in virtual memory. The retrieval rate from the database was thirty-five objects per second. In contrast, the update rate was only 10 objects per second. Therefore, applications with many retrieves but few updates will perform fairly well.

Three techniques were studied for their impact on performance, namely, caching data in virtual memory, prefetching data, and clustering data in the database. If an object is cached in virtual memory after it is fetched from the database, the performance of the example persistent applications improved by a factor of 25. Subsequent experiments incorporated a mechanism that predicted what data would be accessed next. The performance of persistent applications improves up to 30% when data is prefetched.

Clustering was implemented by storing more than one object in a single database tuple. Various forms of clustering were analyzed and a kind that has proven effective for Lisp data was implemented and tested. This clustering technique improves performance up to 40%.

Once interobject references are stored in a database, it becomes possible to alter a reference such that an object stored in the database will no longer be accessible. In virtual memory environments, automatic storage reclaimers, or garbage collectors, are utilized to find and delete unreachable objects. Existing storage reclamation algorithms are analyzed to determine their disk I/O characteristics. The features that

appear to be necessary include being able to reclaim data a little at a time, and reclustering the data. Algorithms that do not recluster data cause applications performance to degrade over time. In addition, algorithms that scavenge the entire memory at once require an application remain idle for unrealistic amounts of time.

All the miles of a hard road are worth a moment of true happiness.
Arnold Lobel, *Fables*

For Momma and Daddy

Acknowledgements

First, I would like to thank the readers of my dissertation: Michael Stonebraker, Randy Katz, and George Lakoff. Michael, my research advisor, gave me intellectual support and encouragement. Because of his vision of this dissertation, it has greatly improved over time. He read many drafts and made many red-marks, which helped refine the style of the final product. I am indebted to him for his patience and for continuing to have faith in me. Randy allowed me to vent my frustration when my morale was low. He gave me words of encouragement and advice. I am very grateful for his support. George is a great linguist who did me the favor of being my third reader. He has always given me positive feedback in an environment that provides mostly negative. Thank you George, just for being you.

Ross Bott receives special recognition for allowing me to port Polymnia to the machines at Pyramid Technology Corporation. Toward the end of my research, I needed to rerun all my tests using a new version of Ingres, but the machine I had been using at U.C. Berkeley was sold for scrap. Ross arranged for me to consult for Pyramid and gave me single-user time on their benchmarking machines. The results of these tests significantly changed the flavor of my dissertation.

All famous acknowledgements include someone from early in the acknowledger's life. This one will be no different. I would like to thank my seventh grade Math teacher, Don Lipsih. He instilled in me a belief in myself. He was the first person to say that I should do something special with my life. Without his intellectual encouragement early in my life, I don't know where I would be now.

I would like to acknowledge the members of the CSUA who gave me their unconditional support during the most difficult period of my life. When those closest to me turned away, a group of very special people gave me the love and encouragement I wasn't getting elsewhere. I want to especially thank David, Paul, Perry, Steve, and Chris.

The members of the BAIR group deserve recognition for their support and friendship. I was a BAIR during my first two years as a graduate student. I received financial and emotional support. Peter Norvig allowed me to use his parser and generator for many tests of Polymnia. Jim Martin lent me his living room floor on many nights when I needed a place to sleep in Berkeley. Peter and Jim, thank you for your friendship.

Doug. He taught me to be independent. He gave me enormous support and encouragement. He has enriched my life beyond words. My eternal love and gratitude are his.

My deepest gratitude is to my parents, Janice and Dan Butler, who instilled in me a sense of wonder and curiosity about the world. They allowed me to climb trees and to strive for things that were seemingly out of reach. They encouraged me to explore ideas and to form my own opinions. They never discouraged my naivete as childishness. The qualities that they instilled in me kept me going when graduate school got rough. Therefore, this dissertation is dedicated to my parents for without them it would not have been completed.

M.H.B.

Table of Contents

CHAPTER 1

Introduction

So come and walk awhile with me and share
The twisting trails and wondrous worlds I've known.
But this bridge will only take you halfway there--
The last few steps you'll have to take alone.
Shel Silverstein, *A Light in the Attic*

Artificial Intelligence (AI) and Computer Aided Design (CAD) applications manage large amounts of data that has historically been stored in files. As the amount of the data has grown, better ways of accessing the data than are typically provided by a filesystem have become desirable. Since the purpose of a database management system (DBMS) is to provide better access to data, attempting to couple these applications with a DBMS seems natural.

Past efforts to couple complex applications with traditional DBMS have failed to produce practical systems because many complex applications have difficulty using traditional data management systems to store their data. This difficulty is a result of an *impedance mismatch*[Pren78] between the data used in complex applications and the data model supported by relational data management systems. Most DBMS are designed for business data applications; whereas, the complex applications have highly structured data that is often interrelated to other data. As a result, the performance of these applications when coupled with a traditional DBMS has not been acceptable.

For complex applications to be well-served by a DBMS, the problems created by the impedance mismatch must be resolved. Some researchers claim these problems will be solved by a DBMS with a new data model which has been termed an Object-Oriented Database

(OODB) [Maie86]. This new data model is fashioned after that of SmallTalk [Gold83] in which type hierarchies and object identity are supported. Other researchers have shown how this data model can be implemented using a Relational DBMS (RDBMS) [Rowe86]. Alternatively specialized data managers for each application can provide exactly the data management functions needed for a particular application [Cock84, Bato86, Care86]. Regardless of the form the data manager takes for these applications, the storage problems of complex data must be addressed.

Complex data differs from traditional data in the richness of its types. Traditional data types are numerical types, strings, and characters. The new types include rules and lists for AI, and boxes and wires for CAD. Applications may need to define their own types. Much of this complex data contains interobject references.[1]

Storing interobject references introduces some problems, but can also provide some advantages. Two problems must be addressed:

1: A data manager must decide how to store a reference to an object.
2: Some objects may only be accessible to applications if something references them. Hence, removing objects that are no longer referenced becomes an issue.

Interobject references can provide some advantages, though, because of the semantics of a reference. For example, if an application contains two objects, *objectA* and *objectB*, and *objectA* contains a reference to *objectB*, when the application accesses *objectA* it is possible that the application may access *objectB* soon. Data managers can use this information to help improve an application's performance in two ways. For example, *objectB* could be placed on the disk near *objectA*. When *objectA* is retrieved, *objectB* could be retrieved at little additional cost. This is an example of *clustering*. Another way to help an application's performance would be to use interobject references to help *prefetch* data. Even if they aren't stored

[1] For the purposes of this dissertation, an *object* refers to any piece of data. An *interobject reference* is colloquially known as a pointer.

near each other, the data manager could fetch *objectB* after *objectA* is fetched so that if *objectB* is accessed the application needn't wait for it to be fetched.

The thesis of this dissertation is that:

> *A traditional DBMS coupled with a language interface provides the functionality necessary to support interobject references without unreasonably degrading an application's performance.*

This dissertation explores how an RDBMS can be extended to provide storage management for complex data which contains interobject references and how applications can maintain good performance. Three techniques are studied that will potentially improve performance, namely, caching, prefetching, and clustering. In addition, automatic storage reclamation algorithms are analyzed to discover how their implementation would affect the performance of an application.

A prototype system that couples Lisp with a DBMS has been implemented to explore performance and interface issues. Lisp was chosen because it contains simple, uniform data structures that heavily rely on interobject references. Because of the simplicity of the structures, the research focuses on the issues regarding interobject references rather than superfluous problems that might arise if more complex structures are used. Two applications developed by the Berkeley Artificial Intelligence Research Group at the University of California have been used for testing the performance of this persistent Lisp system. These applications will be described in Chapter 3. The knowledge base of each application is stored in a DBMS and the applications are executed. Any data necessary to understand the program input has to be fetched into the Lisp environment from the DBMS. The performance of each *persistent* application is compared with the performance when all the data is in virtual memory. This gives a basic performance cost of using persistent data.

Data is often accessed many times by an application; therefore, after an item is fetched from the DBMS it may be advantageous for the datum to remain in virtual memory until the application completes. If the datum is not *cached* in virtual memory, every access to that datum would require a database retrieval. The applications are monitored to discover how helpful caching actually is.

Initially, all data is fetched from the DBMS when the application accesses it and it is cached in virtual memory. Since the datum is not retrieved until it is accessed by an application, the application must pause while the datum is being retrieved. Hence, no parallelism between the application and the DBMS can be realized. The amount of time an application spends waiting for data from the DBMS may be lessened if the DBMS can fetch data in parallel with the execution of the application. To fetch data before it is requested, a mechanism must exist with which to predict what data may be needed by the application. Interobject references are helpful in predicting what objects may next be needed by an application. Various prefetch methods are tested that use the interobject references of a fetched object as hints for what may next be needed.

When retrieving large amounts of data from a DBMS, the amount of work performed by the DBMS can be characterized by the number of queries executed and the number of tuples retrieved. In the initial system, each object is stored in a single tuple and only one tuple is retrieved per query. One way to lessen the work performed by the DBMS is to have more than one object retrieved per query. The simple way to achieve this is to store more than one object in a single tuple. If two objects are stored in a single tuple, one query will retrieve two objects. Hence, theoretically, the number of queries executed can be cut in half. This technique will only prove useful if both objects are needed at the same time. This is another case where interobject references can be used as hints to determine which objects may be accessed together. This kind of clustering is implemented and tested to verify its performance benefits.

Another issue is that of scavenging the database to remove data that is no longer accessible by applications yet has not been explicitly deleted. Inaccessible data is a side-effect of having interobject references; when an interobject reference is altered, it may leave an unreferenced object in the DBMS. In many situations, this unreferenced object cannot be accessed by any application. Hence, for all practical purposes, this object is garbage and it should be removed. Algorithms such as Mark and Sweep, Copy-Compact, and Reference Count [Cohe81] have been designed to perform garbage collection but they are all optimized for a virtual memory environment. The performance of the algorithms can be characterized by the amount of I/O that they require. The I/O requirements become even more important when they are operating on data stored in a DBMS on a disk. The properties of these algorithms are analyzed to determine their performance impact. Since these algorithms may be greatly affected by the type of data they operate upon, how their performance correlates with differing data structures is also studied.

In the rest of this dissertation, the topics outlined above will be detailed and analyzed. To begin, Chapter 2 surveys many of the current development efforts aimed at supporting complex applications with a focus on how these systems address issues related to storing interobject references. Chapter 3 contains a thorough description of the persistent Lisp system that was implemented along with performance results. The performance impact of various cache management strategies, including prefetching, is explored in Chapter 4. In Chapter 5 the effectiveness of clustering data in the database is evaluated. The last topic investigated in this dissertation, storage reclamation algorithms, is described in Chapter 6. This dissertation concludes in Chapter 7.

CHAPTER 2

A survey of persistent systems

The geographer is much too important to go loafing about. He does not leave his desk. But he receives the explorers in his study. He asks them questions, and he notes down what they recall of their travels. And if the recollections of any one seem interesting to him, the geographer orders an inquiry into that explorer's character.
Antoine de Saint Exupery, *The Little Prince*

2.1. Introduction

A goal of many data management projects is to ease the writing of applications that deal with complex data. Three quite divergent approaches have been taken to achieve this goal. One approach is to build a general data manager that allows applications to make custom extensions to it. These are termed *extensible data managers*[Ston86a, Lind87]. The second approach is to provide a language that incorporates data management facilities with which to write complex applications. These languages are often called *persistent languages*[Atki82, That85, Obri86]. Distinct from these two approaches are projects that involve automating the design of specialized data managers. The third approach is to provide a set of building blocks out of which a custom data manager can be easily built. These database toolkits are referred to as *database generators*[Bato86, Care86].

The designers of extensible data managers believe that complex applications vary in their needs; hence, the only way to provide for application developers is to give them a fully functional database and tools they can use to customize the DBMS to their needs. The researchers who take the persistent language approach agree that applications will vary in their needs, but not all applications will need a general DBMS; therefore, applications that do not need the functionality will not want to pay a performance price for it. The people who are building database generators concur with persistent languages researchers that application developers will only want to pay a performance price for features that they need. In addition, they feel that many applications will have common needs. The designers of database generators are convinced that a DBMS toolkit will reduce the amount of development duplication across applications. Database generators will not be discussed any further in this survey because the focus of the generator projects is how to provide a toolkit, rather than how to store complex data. Persistent language and extensible database projects concentrate more on the storage of complex objects; hence are more relevant to the research described in this dissertation.

Many common issues need to be addressed by projects taking either the persistent language approach or the extensible database approach. Of interest here are the following:

How are interobject references stored in the data manager? in VM?
Is data cached? prefetched?
Is clustering of data supported?
Is garbage collection provided to remove objects no longer logically reachable?

All of these issues are related to interobject references which complex data often contains. Systems that provide for complex applications should address these questions.

In this chapter, seven systems are described. The systems were chosen based upon the amount of information available about them and how thoroughly they addressed issues relating to complex objects and interobject references. Some are ongoing research projects, others are commercial systems. The next seven sections contain descriptions of these systems and the answers to the above questions are given whenever possible. The last section of this chapter compares the systems.

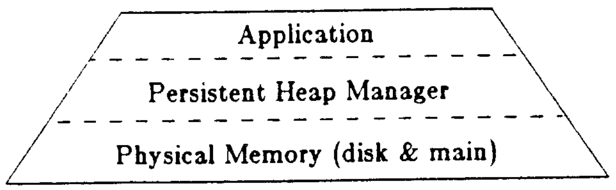


Figure 2.1: Architecture of POMS

2.2. Persistent Object Management System

The Persistent Object Management System (POMS) [Cock84] is a programming language and data manager in one system. POMS is the most recent version of PS-Algol [Atki82], a version of Algol that incorporates persistence. In Algol, all dynamically allocated data resides in a heap. In PS-Algol, at an application's conclusion, all reachable heap data is written onto disk-based heaps. Therefore, all heap objects are potentially persistent objects.

The architecture of POMS applications is shown in Figure 2.1. An application runs on top of the persistent heap manager. This heap manager handles all interactions with the disk. Hence, the persistence of heap structures is transparent to applications written in the language.

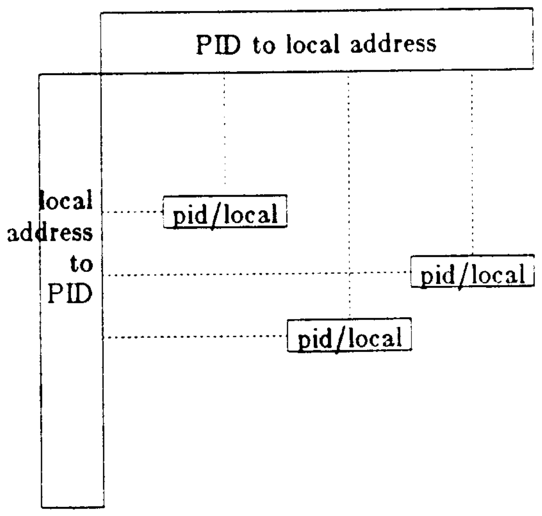


Figure 2.2: The PIDLAM: Mapping local addresses to PIDS in POMS

References to persistent heap objects are also handled transparently. Each heap object is assigned a persistent identifier (PID) with which other objects reference it. When a program dereferences a PID, an address fault occurs because all PIDs are negative and a routine is called to decipher the PID. A PID-to-Local-Address Map (PIDLAM), shown in Figure 2.2, is kept in local memory. To decipher the PID, the routine first looks it up in the PIDLAM to see if the object has previously been read from disk and allocated a local memory address. If it is not in local memory already, the PID is translated to obtain the disk address of the object; then it is mapped into local memory, allocated a local address, and entered into the PIDLAM. Once a piece of data is brought into virtual memory, it can be accessed by its local address.

The translation from PID to disk location has three steps, as shown in Figure 2.3. A PID is composed of a logical block address and an offset. The logical address must be translated into a relative block address, which in turn is mapped onto a physical block address. The offset from the PID is added to the physical block address to obtain the actual disk address. Hence, POMS incorporates the duties usually delegated to a filesystem and a virtual memory system.

As is typical in filesystems, the unit of transfer between disk and local memory is a block. When a structure is referenced, the block on which the structure resides is read into local memory, not the entire heap. By not reading the entire heap, the traffic from disk to local memory is reduced and local memory does not become cluttered with unnecessary data.

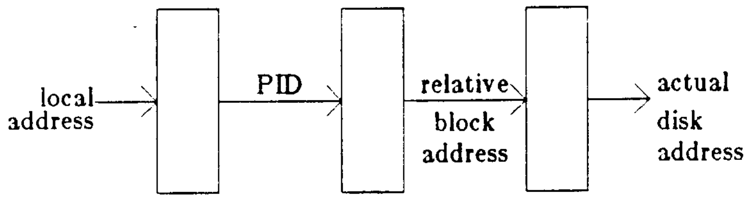


Figure 2.3: Translating local addresses into disk addresses in POMS

Just as with Algol, POMS objects are strongly typed. That is, at compile-time, all variables and objects are checked for type compatibility. Hence, POMS must supply a mechanism for type checking at run-time since the objects stored on disk are not known at compile-time. In POMS, the type of an object is termed it's *class*. When an object is read from disk, its class must be easily discernible so that type clashes can be caught. Within heaps, objects are clustered in blocks based upon their class. That is, each block contains only one class of object. Hence, each block can contain a description of the class of the objects contained in that block. When an object is referenced and its block is read from the disk, the class description contained in that block is compared to the expected class. If the two classes are not compatible, a run-time type clash has occurred and the program receives an error.

As was just mentioned, POMS objects are clustered based upon their type. This is in contrast to clustering together objects that reference each other. The designers of POMS believe that objects will most likely reference objects of their same class. No evidence to support or deny this belief has been revealed.

POMS supports sharing of data via heap level locks: only one application at a time is allowed to access a heap. Since applications can create multiple heaps, the granularity of sharing can be made smaller by splitting a single heap in two. This would allow two applications to each access half the data. When an object is dynamically allocated, the heap on which it should be allocated must be specified by the application. In this way, data can be spread out amongst multiple heaps.

POMS does not incorporate a garbage collector because its designers claim that no data is permanently written to the disk that is not reachable. At an application's conclusion, each heap is traversed and each block that contains any reachable data will be written to the disk. The following scenario is possible, though, which will leave unaccessible data permanently on the heap: an application creates a data structure that consists of one more object than will fit in a single block. Therefore, the data is on two blocks, one of which contains only one object. The next time the application runs it only accesses the block with more than one object in it and alters the data structure such that the lone object is no longer accessible. The block with the unreachable object will never again be accessed. In this way, garbage will collect on the disk and will never be detected. This simple scenario illustrates how unreachable objects can collect on disk. This same situation can occur in more complex situations with more than one object left unreachable.

In summary, POMS supports persistent structures of arbitrary complexity by storing them on a disk-based heap. Object references on the disk are in the form of PIDs; whereas, in local memory the references become local addresses. Persistent data is accessed in exactly the same way as nonpersistent data; hence, the persistence is transparent. Sharing of persistent data is supported by locking an entire heap. The garbage collection scheme is an optimistic one that may occasionally allow unreachable data to become permanent.

2.3. Opal/GemStone

GemStone [Cope84] is a programming system for database applications. As is shown in Figure 2.4, the system consists of three major components: Gem, Stone, and Agent. Stone is the object manager and contains the concurrency control mechanisms. Gem is a per process execution unit that can interpret, compile, and execute code. Originally GemStone was designed for applications written in Opal, a SmallTalk-based [Gold83] language; but, the designers realized that users may desire to program in other languages. Therefore, the Agent component of the system was built to communicate to other languages. Currently C and Pascal are supported.

Persistence in Opal is associated with a class, or type. Any class is allowed to be persistent and all elements and subclasses of that class will inherit the persistence property. When a class is declared persistent, the class may specify that one of its fields should be

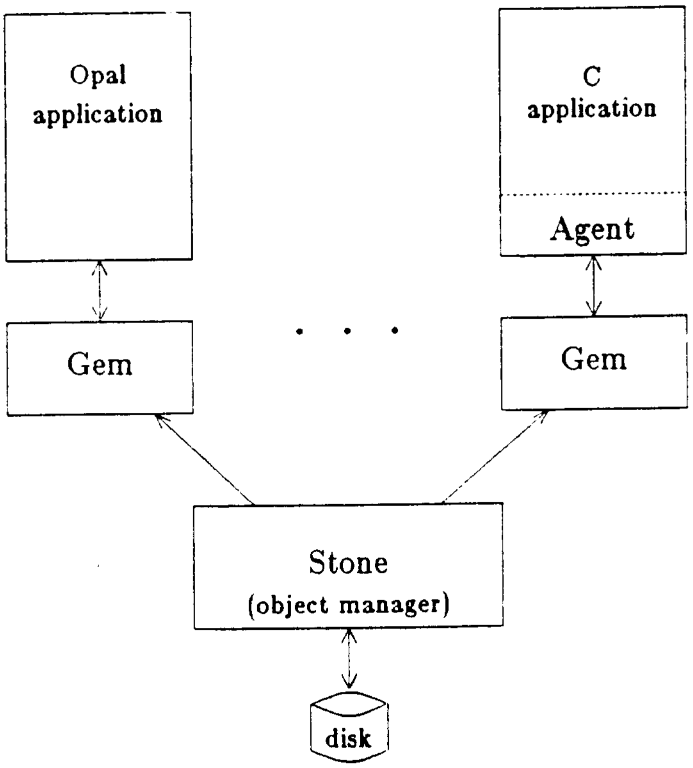


Figure 2.4: The architecture of GemStone

indexed. A B-tree index is built for faster access to objects of the class when referenced by the indexed field. Although multiple programs may have objects of an indexed class, only objects reachable by the program that requested the index will be accessible via the index. This ensures that applications that do not utilize indexing do not pay a penalty for accessing objects of a class that another application has indexed. Another feature that has been added to help deal with persistent objects is type specification. In SmallTalk, an object's instance variables may take on any value. In Opal, an application can specify that an instance variable is confined to be of a particular type.

Objects in GemStone are identified by Object-Oriented Pointers (OOPs). An object table in Stone maps OOPs onto physical locations. An application may contain a large number of persistent objects that are local to the application (i.e., not shared with other applications). To speed access to these local persistent objects, they may be referenced by their physical location. Hence this avoids the cost of looking the object up in the object table. This mode of reference is used only for pointers between objects.

GemStone also supports versioning of objects. Theoretically, no garbage collection is needed since all objects are reachable by some version of the database. Nonetheless, garbage collection is needed because past versions of the database may become inaccessible. GemStone uses a shadow page mechanism to allow recovery after a system crash. A hybrid garbage collection scheme has been implemented that runs on the shadow pages of the database while an application runs on the real pages. If an object on a shadow page is determined to be unreachable, the corresponding object on an active page is deleted.

In the future, GemStone, intends to support clustering of objects together that will be accessed at the same time. This may be accomplished by taking hints from an application or by adding mechanisms to Gem that would attempt to cluster objects based upon their interrelationships.

In summary, GemStone is a programming system and data manager that supports object identity, versioning, and B-tree indices on objects. Object references are either physical pointers or OOPs which map onto physical pointers through a table. In addition, GemStone supports access from multiple languages. A hybrid garbage collection scheme is utilized to remove objects which no application may access. Because the system is still under development, many features are being added and improved at the current time.

2.4. Trellis/Owl

Trellis/Owl [Obri86] is another SmallTalk-based language that has been extended to encompass persistence. In Trellis/Owl persistence is associated with a collection of objects rather rather than with a class as in GemStone. For example, an application may define a class *Student*; instead of specifying that all Students are persistent, an application can declare a variable *Pers-Students* to be a *db-collection* of Student. Any object of class Student can be added to Pers-Students, thereby becoming persistent. All objects of class student that are not added to Pers-Students remain temporary.

The disk storage and management of db-collections is handled by the Object Repository Manager as shown in Figure 2.5. Each application has its own object manager that transfers data to and from the disk for the application. Db-collections are different from files in that set-like operations are defined for them. The actual format of db-collections on the disk is hidden from the application.

Trellis/Owl allows multiple programs to share data using share-by-copy semantics; that is, each application that accesses a particular piece of data gets a separate copy of that datum in the application's virtual memory. If the object is updated and written back to disk, a new version is created. Hence, if two applications update the same item at the same time, two new versions will be created and the current version will be the one written last. If applications require stricter sharing, Trellis/Owl provides semaphore-like protection mechanisms

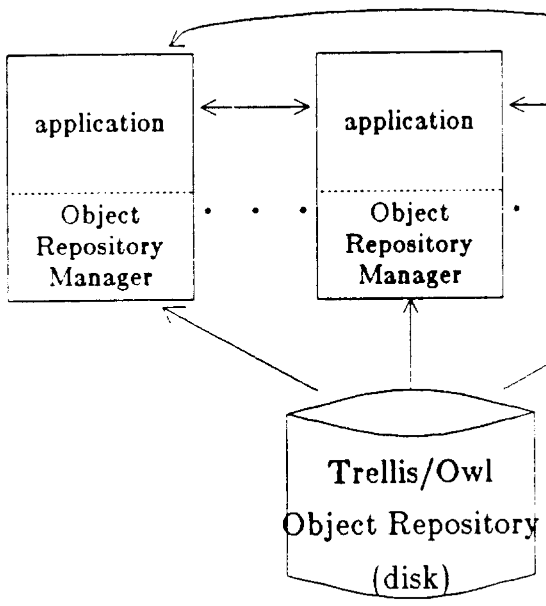

application
Object Repository Manager
application
Object Repository Manager
application
Object Repository Manager
• • •
• • •
Trellis/Owl Object Repository (disk)

Figure 2.5: The architecture of Trellis/Owl

which applications may use to synchronize with each other. Therefore, concurrency control mechanisms are outside the object repository manager and must be developed on an application-by-application basis.

The object manager for Trellis/Owl is built specifically for its own data. Hence, the data manager understands the semantics of the data. That is, a pointer is recognized as distinct from an integer and other structures. Therefore, the object manager can cluster objects together on disk that it thinks will be accessed together based upon the object's structure. In Trellis/Owl this clustering is termed *chunking*. When one object in a chunk is referenced, all objects in the chunk are fetched.

The designers of Trellis/Owl feel all data stored in the object repository is permanent. Hence, Trellis/Owl provides no automatic garbage collection. Old versions cannot be explicitly deleted; therefore, once an object is stored in the repository, it remains there forever.

In summary, Trellis/Owl is a language that supports persistence by providing access to an object repository. Sharing is supported by the repository and the language provides some transaction facilities to help coordinate shared access. Intelligent clustering and fetching of data is also provided by the repository.

2.5. OM

OM [Mish84] is a persistent Lisp system designed for the Apollo series of machines. OM extends T (a dialect of Lisp) to include a persistent heap manager. The architecture of OM (shown in Figure 2.6) is very similar to that of POMS. The main difference is that in POMS all heaps are persistent; whereas, OM has persistent and nonpersistent heaps. The application uses different operations on persistent heap data than on nonpersistent heap data. Although, according to the designer of OM, complete transparency would be better, distinguishing between persistent and nonpersistent was simpler to implement Instead of rewriting low-level Lisp functions, which may have unpredictable results, the implementer of OM simply added some new functions. An advantage to this approach is that an application programmer has more control over the data; that is, the programmer is aware during an application's development when persistent data is being operated upon. This may lead to more efficient code.

An application may access more than one heap. To accomplish this, each heap when created is given a unique identifier with which all applications may reference it. Each heap is stored as a file in the Apollo filesystem and has a maximum size of 500 Megabytes.

When an application first references a persistent heap, the heap is mapped into its virtual address space. Thereafter, the data on the persistent heap is accessed similarly to non-persistent heap data. When an application completes, the heap is written back to its files.

Applications are allowed to share persistent heaps, but only one application at a time may access a heap. If a finer granularity of sharing is desired, a heap can be split into two smaller ones (as in POMS), or the heap objects can each be allocated semaphores. Applications can program their own concurrency control protocols and coordinate sharing using semaphores.

Each heap is individually scavenged to remove garbage. When a heap is being garbage collected, no applications may access it. Since a heap object may reference an object in a different heap, a list of objects referenced from other heaps must be kept for each heap. A hybrid garbage collection scheme was developed that uses this list to ensure that an object referenced from outside the heap in which it resides, but not by any object in its own heap,

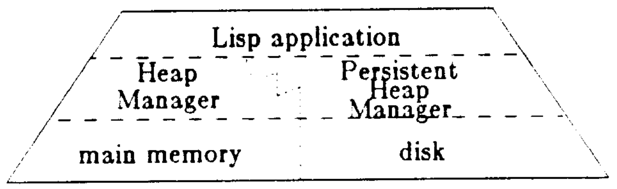


Figure 2.6: The architecture of OM

will not be deleted. Since heaps are scavenged individually, circular garbage that spans more than one heap will not be detected.

OM incorporates garbage collected, persistent heaps into Lisp. These heaps are of limited size but are not limited in number. The granularity of sharing is an entire heap, although applications may achieve a finer degree of sharing using semaphores.

2.6. Lisp with Persistent Virtual Memory

Texas Instruments is developing a new Lisp environment which provides persistent data for their Explorer workstation [That85]. Figure 2.7 shows the architecture of TI's persistent virtual memory (PVM). Recoverable virtual memory is provided by shadowing each page on the paging disk. All writable virtual memory pages are automatically allocated two pages on disk. When a page from virtual memory is swapped out, it is written to the older of the two pages. A system checkpoint can occur whenever all dirty pages from virtual memory have been written to the disk. In case of a system crash, VM can be rolled back to the last checkpoint.

Layered on top of the recoverable VM is the garbage collector. An aging garbage collection scheme [McEn86] is used which seems to be the best for large virtual memory systems. Chapter 6 will describe this type of garbage collection.

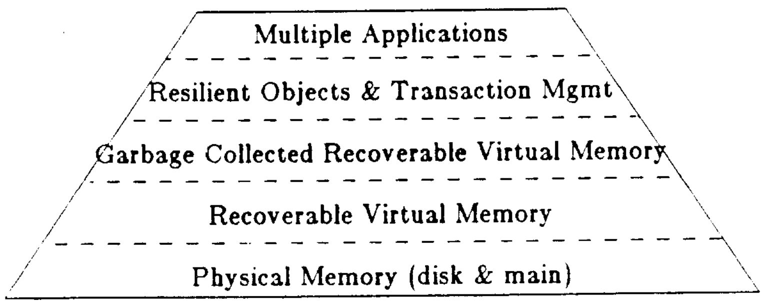

Multiple Applications
Resilient Objects & Transaction Mgmt
Garbage Collected Recoverable Virtual Memory
Recoverable Virtual Memory
Physical Memory (disk & main)

Figure 2.7: The architecture of TI's persistent virtual memory

Resilient objects and transaction management are implemented on top of the garbage collector. The entire virtual memory is recoverable to the last checkpoint, but individual objects can be made recoverable to their last update if they are designated as resilient. A resilient object will survive a system crash even if its page in VM is not part of the current checkpoint. Resilience is implemented using undo and redo logs. Each time an object is updated, it's old value is written to the undo log and it's new value is written to the redo log before the change is actually made to the object. If a resilient object is not in a checkpoint when the system crashes, the system can reconstruct the value using the undo and redo logs.

The virtual memory is shared by all applications running on a single workstation. Since the workstations are intended to be single-user, sharing among multiple users is not supported. PVM does not extend memory beyond the size of regular VM. Hence, it does not get rid of the need for a filesystem. Extending persistence beyond the current VM size and supporting multiworkstation sharing are projects currently being worked on at TI.

Clustering of data is provided by the VM system in the form of CDR-coding [Bobr79], a technique that is successful for Lisp data. Clustered pages are the unit of fetching from the disk.

In summary, PVM extends the Explorer workstation environment to have recoverable, garbage-collected virtual memory. In addition, transactions and resilient objects are supported. The VM pages contain data clustered using typical Lisp environment techniques. Currently, the persistence is not extendible beyond the size of virtual memory. Hence, many applications which call for large amounts of data must still use files for the permanent storage of their data. The designers hope to alleviate this need for files in the future and to address the problems of sharing amongst users.

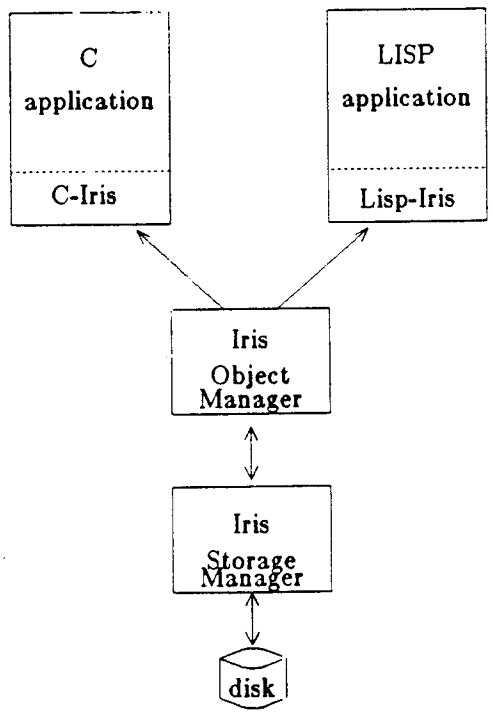


Figure 2.8: The architecture of Iris

2.7. Iris

The Iris project [Derr85] provides multilanguage access to data stored in a general purpose DBMS. The Iris architecture is shown in Figure 2.8. At the heart of the Iris system is a conventional relational storage subsystem similar to the RSS in System R [Astr76] which is called the Iris Storage Manager. On top of the IRIS Storage Manager is the Iris Object Manager which implements the Iris data model. The data model is based upon the functional model of DAPLEX [Ship81]. The Object Manager supports type hierarchies, inheritance of properties, versioning, and user-defined types and operations. Access to the Object Manager can be made via an Interactive Query Interface, a Lisp application or a C application. Lisp and C have interfaces written for them that incorporate the Iris database object model. The Lisp application interface will be used to describe what is provided to an application.

When an application accesses a persistent datum, that datum is requested by the interface from the Iris object manager. An application may view a piece of data as a single item, but it may actually be represented by multiple tuples in the database and may need multiple queries to retrieve. The object manager will turn the request for a particular datum into the appropriate queries to retrieve the datum.

Iris supports caching of objects in an application's virtual memory. This caching facilitates multiple accesses to the data without incurring repeated cost to retrieve it. For an application to cache data values in its virtual image, it must explicitly request the interface to lock the data. Thus, data that is cached cannot be accessed by another application.

The Iris provides clustering of data items by type. A single block on the disk contains tuples from a single relation; hence, tuples of the same type are stored on one page. As was previously mentioned, a single datum to an application may actually consist of multiple tuples of different types in the database. These separate pieces of a single datum are not stored together on the disk. Hence, if a piece of data occupies multiple tuples of different relations, it will take multiple queries and disk reads to retrieve it.

The Iris DBMS does not provide garbage collection, neither do the current language interfaces. Data will only be deleted if an application specifically requests it. It is feasible that garbage collection could be built into the language interfaces or the object manager, but it is currently not implemented.

Iris provides permanent storage in a DBMS for data. This data may be accessed from applications written in multiple languages. Locking of objects is performed upon request, as is object deletion. Tuples from a single relation are clustered together on a page. Data items are fetched individually upon demand.

2.8. Postgres

Postgres [Ston86a] is a fully relational DBMS that allows applications to easily attach new functionality to the DBMS. Each running application sends requests to a server process that manages interactions with the disk (Figure 2.9). Postgres supports triggers, which they call *alerters*, with an additional process called the *Postmaster*. The Postmaster can communicate with every server process. The purpose of the Postmaster is to notify servers when alerters are set off that require a server process be notified. An object interface is also being built for Postgres [Rowe86] to accommodate the needs of many complex applications. This discussion of Postgres will primarily deal with the object interface.

When an application accesses an object, it will be retrieved from the database and stored in an object cache that is accessible to the application. An object index, similar to the object table of GemStone, determines whether or not a given object is in the cache. This index is stored in the cache, not the database. All references to database objects are done using object identifiers that must be mapped onto the object through the object index. This alleviates the problem of mapping interobject database references into VM addresses when an object is read into virtual memory. The consequence is that all interobject references must through the object index, thus are more costly.

Applications may specify which data should be prefetched when an object is retrieved from the database. These prefetch hints given by an application are stored in a Prefetch Relation. When an object is retrieved from the database, a retrieval is done on the Prefetch Relation to see if the retrieved object had any associates that should be prefetched. If prefetching is successful, it will cut down on the time the application wait for data.

Some objects may actually be composed of multiple tuples. To lessen the number of retrievals required when demand fetching such objects, an object can be precomputed and stored in the Precompute Relation. The precomputation of objects is handled by the data-

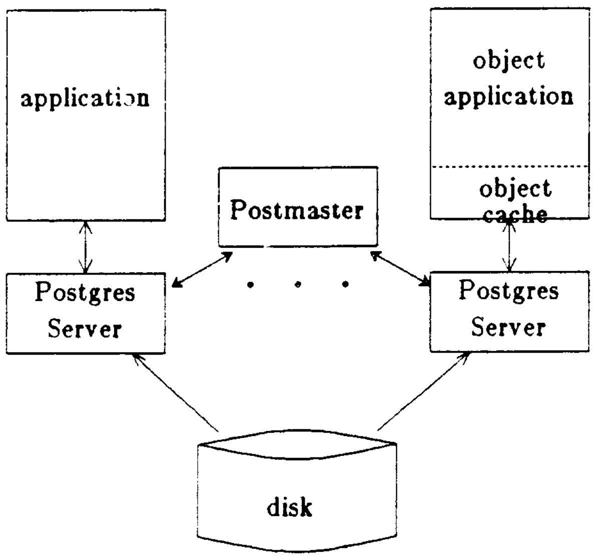


Figure 2.9: The architecture of Postgres

base system, not by the application. Triggers are used to invalidate the precomputed image whenever a portion of the object is updated.

Triggers are also used to help propagate updates to object caches. When an object is fetched by an application, the application may specify that if the object is updated, it should be notified. A trigger is then set on the object that will be executed is the object is updated. This trigger will send a message to the application that set the trigger notifying it of the update.

Objects must be explicitly deleted. The designers feel that permanent data should only be removed upon request. Hence, no form of garbage collection is provided.

Postgres, like Iris, places tuples from the same relation on the same disk block. Hence, clustering of tuples by type is provided. Since an object in an application may consist of more than one tuple, this does not necessarily accomplish clustering of objects of the same type. Precomputation provides clustering of multiple tuples to form a single object. Instead of fetching multiple tuples from the database, one database retrieval results in the whole object.

Postgres is a new data manager that incorporates many features to facilitate complex applications. The object interface being built demonstrates how these features can be used successfully. Precomputed values allow quick retrieval; and triggers allow multiple cache consistency protocols to be implemented.

2.9. Comparing the systems

Seven divergent systems have been described: one extensible DBMS (Postgres), one that uses a general DBMS as a data manager for programming languages (Iris), and five programming languages that incorporate data managers to varying degrees. These systems were chosen in part because of their diversity. Figure 2.10 depicts how transparent the storage dichotomy is to an application programmer. If applications do not distinguish between persistent and nonpersistent data, the system will be on the far left. Whereas, if each operation on a persistent object is explicitly distinct from those on nonperistent objects, the system will be on the far right. POMS and PVM are the most transparent with GemStone just a little less so because data types must be declared persistent. Because sharing is explicit in Trellis/Owl and locking in Iris, these two systems are much less transparent. PVM does not provide sharing among users so part of this comparison isn't fair. In OM, each operation on a persistent object is distinct from that of a nonpersistent object, but the interfaces are similar. The designers of Postgres' object interface believe that applications may take longer to write, but they will perform better, if all dealings with persistent objects are nontransparent.

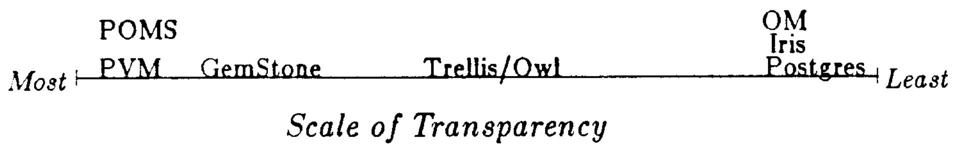


Figure 2.10: Comparison of overall transparency

All of the systems offer some form of caching, but in some systems data is only cached if the application requests it. Hence, the amount of transparency correlates with the type of caching. In Postgres and Iris, the least transparent systems, data is only cached if requested explicitly. In POMS and PVM, the most transparent systems, data is automatically cached.

Just as the degree of caching can be predicted from the amount of transparency, whether or not clustering is offered can similarly be predicted. Figure 2.11 depicts the comparison scale for clustering. *Most* designates systems that offer clustering based upon the structure of the data. This includes PVM, Trellis/Owl, and POMS. The clustering POMS offers has not been shown to be as effective as semantic clustering. If each object was represented by one tuple in a relation, then Postgres and Iris would supply the same level of clustering as POMS. Nevertheless, the designers of both Postgres and Iris admit that a single object from an application may be implemented composed of tuples from many relations. Hence, Postgres and Iris do not support clustering based on object type as does POMS. Postgres' precomputation of objects is equivalent to a type of clustering. This form of clustering allows a single object from an application to be retrieved in one query. Therefore, implementing this form of clustering could prove to be a performance benefit for a system such as Postgres. GemStone does not offer clustering currently, but plans on providing clustering based upon the semantics of the data. Systems to the left of middle on transparency are to the left of middle on clustering.

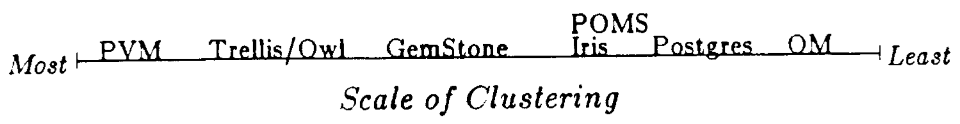


Figure 2.11: Comparison of clustering supported

Whether or not a system provides automatic garbage collection of all objects no longer logically reachable correlates with the transparency scale as Figure 2.12 shows. Some systems, such as Trellis/Owl, believe that an object has an infinite lifetime; hence, garbage collection is superfluous. The designers of GemStone originally felt this way, but eventually came to recognize that there will exist data that becomes unnecessary at some point in time and it may not be apparent to applications. Only PVM and GemStone offer full garbage collection. Both OM and POMS provide a form that is not 100% accurate, but it should delete most unreachable objects. Iris, Postgres, and Trellis/Owl will only delete an item if an application specifically requests it.

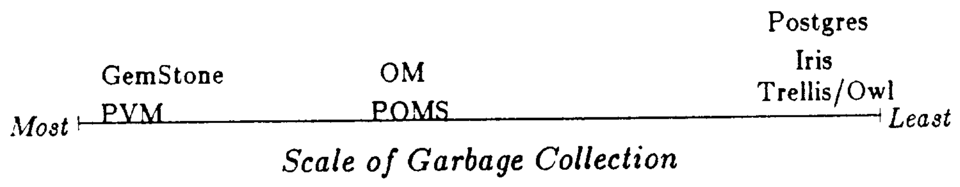


Figure 2.12: Comparison of garbage collection algorithms

Lastly, the correlation between transparency and granularity of sharing becomes clear when looking at Figure 2.13. PVM doesn't support sharing at all and it is one of the most transparent systems. POMS supports it only at the granularity of a heap. Only the systems that allow applications to explicitly set locks support sharing at the granularity of an individual object. It is clear that transparency is sacrificed to support sharing.

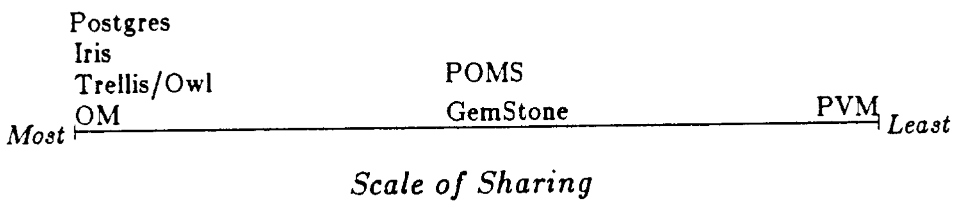


Figure 2.13: Comparison of sharing

The systems built using a general DBMS (e.g. Iris and Postgres) offer little transparency, clustering, and garbage collection, but they offer the most in terms of concurrency control; whereas, systems that are specifically designed for handling complex data structures tend to offer more storage and fetching features, but do poorly in providing fine granularity sharing (e.g. POMS and PVM). The system described in the next section, Polymnia, is an experiment using a commercial DBMS to try to provide features, such as clustering, that may lessen the cost of using persistent data. The features discussed in this section (clustering, caching, interobject references, storage dichotomy, and garbage collection) are analyzed for potential benefits; some are implemented to check actual improvements.

CHAPTER 3

Polymnia: a persistent Lisp prototype

If we wished to change the structure we could do so, without any fear of some primal urge welling to the surface and sucking us back into some atavistic pattern. Vested interests might be loud in their protest, but our deepest biological nature would not. We are, after all, the ultimate expression of a cultural animal; we have not totally broken free of our biological roots, but neither are we rules by them.

Richard E. Leakey, *Origins*

3.1. Introduction

Polymnia[1] [Butl86a] is a persistent Lisp system built from Franz Lisp [Fode83] and Ingres [Ston81]. Lisp is an easily adaptable programming language and Franz is the version that was readily available in 1985 when this research began. Ingres is the locally available database system that offers the most features and the best support.

The architecture of the Polymnia system is depicted in Figure 3.1. Portions of the Lisp system have been rewritten so that when persistent data is encountered Lisp routines can be executed that will handle the persistent data. This segment of code communicates with an Equel/C [Allm76] process via a Unix pipe. The Equel/C process in turn communicates with the database management system Ingres. Data requests start from the Lisp process and are translated via the Equel/C portion into database queries. The data is returned from the database to the Equel/C program and converted into an intermediary format and sent to the Lisp process.

[1] Polymnia is the Greek muse of mimicry. This prototype mimics Lisp structures in a DBMS.

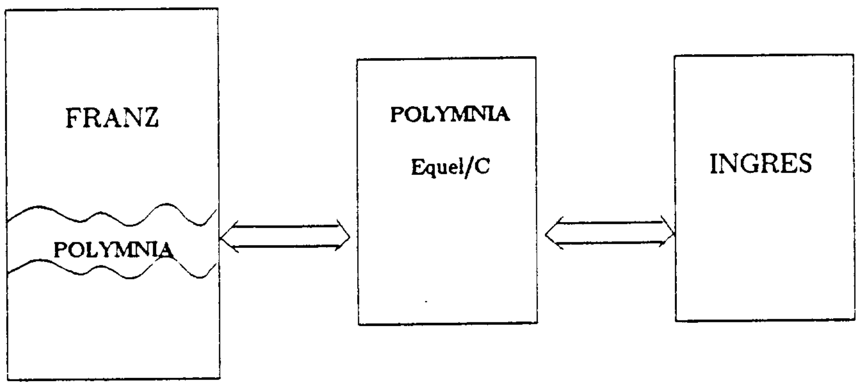


Figure 3.1: Architecture of Polymnia

The chosen architecture is similar to that of Iris. A general data manager is at the core of the system. There is a layer between the data manager and the programming language that contains the semantic understanding of Lisp data and knowledge about the storage mechanism. The DBMS itself cannot distinguish between Lisp data and other data; and, the Lisp system does not know how or where the data is stored. It could be spread out among many DBMSs or stored in any other format (e.g. a file).

An alternative to this architecture would be to have the Lisp system communicate directly with the DBMS. This would reduce the number of layers in the system which should result in better performance. To implement the alternate architecture, a reliable general mechanism must exist with which to communicate with the DBMS from a programming language. Such a mechanism does exist: it is termed the "IIWRITE" library. These are the routines that Equel/C uses to communicate with the DBMS. Problems occur when trying to use the IIWRITE library with Lisp [Laru83]. The functions in the IIWRITE library attempt to speed communication to Ingres by encoding and careful buffering. The resulting connection between Lisp and Ingres is tenuous because it can easily lose synchronization when an interrupt is fielded by the Lisp interpreter. The loss of synchronization is most likely caused by Lisp flushing buffers that are used by the IIWRITE functions.

Although having an extra layer in the system (i.e., the Equel/C layer) may cause some performance degradation, there are advantages to this approach. Using the three-layered approach, a small amount of changes were made to Franz Lisp, and no changes were made to Ingres. The bulk of Polymnia is encoded in the Equel/C layer. Franz Lisp is a very large, complex system, as is Ingres; without the Equel/C layer, building Polymnia would have involved more complex additions and changes to Franz Lisp and/or changes to Ingres. In addition, any extension to the system would involve changing either Franz Lisp or Ingres. Changes to the Polymnia are easier to implement in the Equel/C layer due to its simplicity. Therefore, due to the difficulties communicating directly from Lisp to the DBMS and the advantages of having a simple layer in the middle, the alternate architecture was deemed less desirable.

Each part of this system will be specified separately. Section 2 contains a description of the Lisp portion. The database and Equel/C portion are explained in Section 3. An example is detailed in Section 4 to show how the pieces fit together. A test has been conducted to measure the cost of updates to and retrievals from the database. The application used for the test and the results are analyzed in Section 5. In Section 6, the various factors that effect the performance will be summarized as will the benefits of the approach described here.

3.2. Polymnia from a Lisp perspective

3.2.1. Design goals and issues

When designing a persistent system, a primary goal should be ease-of-use for the intended users. If a person writing a program in Lisp need not make a distinction between operating on persistent data and nonpersistent data, the program will be easier to write. This will make persistent data easier to use by making it easier to design a system using temporary data, and to gradually move into a persistent environment as the system is debugged. In other words, the persistence of data should be transparent to the application program. This also allows for better performance evaluation of the system since an existing application can be run to measure performance, rather than inventing one for evaluation.

Another goal was to build something that was general: not specific to one version of Lisp; therefore, all changes were limited to ones that could be written in Lisp. The interface package that was built runs under Franz Lisp but could easily be ported to other Lisp systems. When a persistent application executes, the interface package is loaded into the Lisp environment. This package contains recoded Lisp functions that check for persistent data and invoke special routines whenever persistent data is encountered.

Handling different data types in a uniform way is also a goal. If a general mechanism can be built that handles many types of data, it should be more easily extended to handle additional types. There are many kinds of Lisp values: for example, arrays, vectors, lists, symbols, and numbers. The Lisp types that will be implemented in Polymnia are the following: symbols, integers, structured types (similar to Pascal records), and lists of values. In addition to symbols being a kind of value, symbols are also Lisp's equivalent of variables: each symbol can take on a value of any type; that is, a symbol is an untyped variable. The fields of a structure are also typeless. A list is an even more complex type because a list may contain an indeterminate number of values with varying types. If these data types can be handled in a uniform way, the mechanisms for handling persistence may be more easily extended to encompass other types.

The prototype must be built in a modular fashion so different fetching and storing alternatives can be explored. This will enable strategies to be tested to determine which ones give the best performance.

The goals for the Lisp portion may be summarized as follows:

1: An application should not need to distinguish between persistent and nonpersistent data.

2: Alterations to test different fetching and storage strategies should be easily accomplished.
3: All data types should be treated as uniformly as possible.

All of these goals involve making persistent data easy to use, easy to extend, and easy to tune. The implementation consequences of these goals are discussed in the following few paragraphs.

One way to lessen the performance cost of a persistent system is to buffer or cache data. Upon fetching an object from the database, the system is designed to store the object in virtual memory so when next referenced, it need not be fetched from the database. This has the advantage that the cost of a database fetch may be amortized over a number of accesses; but, this has the disadvantage that virtual memory may become cluttered over time with persistent objects that are no longer needed. Another difficulty caused by caching data occurs when the data is shared. Concurrency control mechanisms need to be developed that handle multiple copies of shared data. Although a very interesting topic, the issues surrounding concurrency control are beyond the scope of this dissertation. These and other caching issues are covered in Chapter 4.

Data must be retrieved automatically from the database to achieve the transparency goal. Therefore, when a datum is referenced, the Lisp system must be able to determine whether or not it has been fetched, and if not, a retrieval must occur before the reference can be processed. To determine this, associated with each persistent object in virtual memory is information that specifies the database object to which it corresponds and whether the object has been fetched from the database yet.

Upon an object reference, the system must also determine how much data to fetch. This can be determined by the operation being executed. Fetching only the data needed to execute the current operation will save time and will lessen the cluttering of memory. Fetching an entire list when one element of the list is referenced may save time in the long run if the whole list will be needed. In Polymnia, only the portion of a list necessary to complete an operation is fetched. This is termed *partial fetching*. The example detailed in Section 4 will clarify what partial fetching of a list means and how it is accomplished.

In the following sections, the implementation will be described separately for each Lisp type as it pertains to each of these issues.

3.2.2. Determining persistence

3.2.2.1. symbols

Every persistent Lisp value needs to be tagged so that functions can easily determine that it is a persistent value. Inside a Lisp application, every symbol whose value is persistent is specially declared as such by calling the function **declare-persistent**. Upon such declaration, different actions are taken depending upon whether or not the symbol currently has a value. To illustrate the different actions, the statement *(declare-persistent foo)* will be evaluated. If *foo* is bound to a particular value, that value will be declared persistent; whereas, if *foo* is unbound, declaring it persistent will cause it to be bound to *foo_UNBOUND* which Polymnia understands as the persistence tag for the symbol *foo*.

The low-level Lisp functions have been altered to examine their arguments to check for persistency. When one of these low-level functions is called, it checks to see if any of its arguments have the special value that indicates a persistent symbol (e.g. foo_UNBOUND). If any argument is persistent, the function initiates the retrieval of the symbol's value from the database. The function then executes using the retrieved value. If the function is one that alters the value of a symbol, an update of the symbol in the database is requested before the function executes.

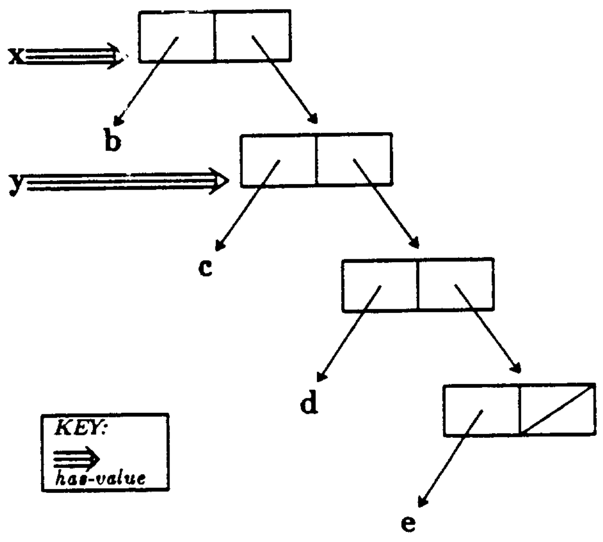


Figure 3.2: Sample lists
The values are set as if the following operations had been performed:
(setq x '(b c d e))
(setq y (cdr x))

3.2.2.2. lists of values

Although a list may contain an indeterminate number of values with varying types, it can be viewed as a set of related, very simple structures. A list (inside a Lisp system) is composed of **cons cells** which are structures that each contain two values. Lists are built by stringing cons cells together via one of the values. One value, which will be called *first*, refers to the first element of the list and the other value, which will be called *next*, refers to the rest of the list. Figure 3.2 depicts the structure of the list (b c d e).

When a symbol's value is read-in from the data manager, if its value is a list, each **cons** cell in the list is *registered* so that functions can determine that the cell is part of a persistent value: that is, the virtual memory address of the cons cell is hashed into an array which serves as a *catalog* of persistent cons cells. This address catalog stores in virtual memory whether or not a cons cell's value has been retrieved from the database. In addition to the catalog of cons cell addresses, each object is also registered in a separate catalog by it's database identifier.

Persistent cons cells are treated by functions in an identical manner to persistent symbols. That is, when a low-level Lisp function is executed, if its value is a persistent cons cell that has not yet been fetched from the database, the function initiates the retrieval of the cell from the database. If the cons cell will be altered, a request to update the tuple representing that cell in the database is sent to the DBMS.

3.2.2.3. structures

A structure type in Lisp is similar to a record type in Pascal. To define a structure in Lisp, one uses the function **defstruct**. For example, *(defstruct (edge) label start finish)* defines a structure type that represents an *edge* in a graph and has three fields, *label*, *start*, and *finish*. The process of defining a structure includes defining a constructor function (*make-edge*) and accessor functions (*label*, *start*, and *finish*). The constructor function is called to allocate the new structures. The accessor functions return the corresponding field of a data structure. For example, *(start edge1)* will return the *start* field of *edge1*. Accessors are also used with the function *setf* to set the field values. For example, *(setf (label edge1) 'name)* sets the *label* field of *edge1* to have the value *name*.

Defstruct also takes some optional arguments that can specify, for example, how the structure should be built or referenced. A new option was added to allow persistence to be specified. Each structure type in Polymnia when defined may specify the option **:persistent**. If it is defined as such, all created structures of that type will be stored in the database. Two additional fields are defined for these structures that contain the database identifier of the structure and a tag that indicates whether or not the structure has yet been fetched from the database.

To lessen the amount of data sent from the DBMS to Lisp when a structure is retrieved from the database, only the values of the fields are sent (not fieldname-value pairs). Hence, an ordered list of the fields (or accessor functions) must be kept that corresponds to the order the fields will be received from the database. Upon definition of a structure, a list of field-names as they appear in the definition is stored and the database sends the field values in that order.

When a persistent structure type is declared, the accessor functions that are defined are different than those defined for nonpersistent structures. For persistent structure types, the accessor functions are defined such that they check for persistence of the structure being accessed. If a structure being accessed is persistent and not in memory, then it is retrieved into memory which entails fetching all its fields' values. If any field of a structure is altered, an update is sent to the DBMS.

3.2.3. Partial Fetching and Caching

Partial fetching of lists is implemented in *Polymnia* by fetching the fewest **cons** cells necessary to complete an operation. In addition to being recoded to check for persistence, functions also know how much of a list is needed to complete its execution. If a list is partially fetched, it needs to be tagged so that functions can determine that the complete value is not yet in virtual memory. The catalog that registers persistent cells' addresses also keeps track of whether the value of the cell is resident in virtual memory or not. When a Lisp value is fetched that references a cell that is not in memory, a cons cell is allocated but is registered in the address catalog as *not yet fetched.* This indicates that the value of that cons cell has not yet been fetched from the database. The database identifier for the unfetched cons cell is also recorded, so that when the cell is accessed it can be retrieved from the DBMS. Since each cons cell in a persistent list is registered, functions can check for this when trying to operate on a persistent cell. The catalog that contains the information about whether or not a cons cell has been fetched also houses how much of the list starting from the cell has been fetched. For example, it may state that all cells reachable via this cell have been fetched or that all cells reachable by following *next* pointers have been fetched. This information is useful to determine whether or not a sufficient amount of a persistent value has already been fetched from the database before a function begins to execute.

Once a cons cell's value is fetched into virtual memory, it need never be fetched again because the cell in virtual memory will be altered to contain all the data that has been fetched from the database.[2] When a symbol's value is retrieved from the database, the symbol is changed in virtual memory so that it's value can be accessed in the same way as a nonpersistent value. The same is true for structures. Section 4 contains a detailed example that indicates precisely how the fetching works.

3.3. Database portion of Polymnia

3.3.1. Storing Lisp Values in a Database

The storage of Lisp in a DBMS is not a trivial matter due to the varied values Lisp *variables* can take on. As was mentioned in the previous section, a Lisp value can be one of the following: a symbol, a numeric value, a structured type, or a list of values. Symbols and numbers are easily handled by current database management systems: a symbol can be represented by a string and a number can be represented by the appropriate numeric type (e.g. integer or real). Storing other Lisp values presents problems to a traditional DBMS. Besides having odd types (such as list), the symbols and structure fields are untyped. This lack of typing conflicts with the traditional DBMS policy of strong typing. One approach to the problems presented by Lisp values is to treat each type as a different abstract data type (ADT) or complex object [Ston83, Lori83]. According to these approaches, each ADT (or complex object) is treated as a single value. That is, the DBMS does not know the internal structure of the data. Hence, the DBMS can only store and retrieve the data as a whole. The only

[2] This implies nothing is ever thrown out of the *cache* which is the case for the tests reported in this chapter. All issues related to caching are discussed more thoroughly in Chapter 4.

operations that the DBMS can perform on the data are provided from a source outside the DBMS (i.e., the implementer of a particular ADT). Whether storing nontraditional data as a black box or as a structure that the database manager can decipher is a controversy that has no widely accepted answer. Opponents of this approach say that for many applications (e.g. CAD) an ADT mechanism will not perform well enough; the only way to achieve the requisite performance is to have the special data types built into the DBMS [Maie86]. ADT proponents say that an ADT mechanism is a general way to extend an existing general DBMS: the designers of a DBMS cannot foresee all of its uses and an ADT mechanism will allow application builders to add their own data types [Ston86b]. Some of the new DBMSs, such as Postgres [Ston86a] and Iris [Fish87], provide ADT facilities that are a more integral part of the DBMS. These ADT facilities should provide better performance and may overcome other objections to this approach.

The issue of whether or not ADTs are an appropriate mechanism for storing Lisp values can be addressed by reviewing two requirements of the Lisp interface: namely, partial fetching and shared sublists. If a list was stored as an ADT, it would have the restriction that it must be accessed as a whole, not in parts. This would not easily facilitate partial fetching. In addition, separate lists may share subparts; this would be difficult to capture using ADTs. If each list is stored as a single value, when two lists share part of their values, the shared portion would be duplicated. If the shared portion is duplicated, the database cannot easily duplicate the property that altering one list's value changes the other list too. As was mentioned in section 3.2.2.2, lists are composed of two valued structures; this composition would be lost if lists were an ADT.

The approach used in Polymnia is to store all values in a general way using data types provided by the DBMS. In addition, a way for fields of a relation to vary in type has been devised: that is, a *union-type* field has been implemented in the Equel/C portion of Polymnia.

By specifying how to represent union-types in a traditional DBMS, all Lisp values can be stored in a uniform way: each structure type can be represented by a relation with a union-type field for each field of the structure; a symbol can be a string with an associated union-type value; and a list can be treated in its cons cell form, in which it is composed of two-valued structures.

Union-types can be implemented in the DBMS by having each type (e.g. symbol, cons, etc.) stored in a distinct relation. A tuple-value that is of a union-type can then be composed of two fields, one specifying the relation (type) and the other indicating an identifier that will allow the value to be retrieved from that relation. This two-field value will hereinafter be referred to as a *union-field.* Figure 3.3 shows the Polymnia implementation: union-types are implemented by storing a tag field (*type*) that indicates what type the union-field contains thereby determining how to interpret the accompanying value field (*value*). The *value* field is an integer that can be interpreted as either a number or an identifier for a tuple in some relation. Examples of union-fields are shown in Figures 3.4, and 3.5 which depict the storage of the list *(b c d e)* (displayed in cons cell form in Figure 3.2). The *type* and *value* subfields are interpreted as follows: if *type* has the value *"cons"*, *value* holds a referent to a tuple in the Cons Relation (i.e., a list); if *"symbol"*, a referent to a tuple in the the Symbol Relation; and, if *"int"*, the particular integer value. "Int" is interpreted similar to an immediate addressing mode and "cons" and "symbol" are similar to indirect addressing modes. In the following sections, how to store each type is described separately and examples of using union-fields will also be described.

union-field	
type	**value**

Figure 3.3: Implementation of Union-fields

Symbol Relation			
id	**name**	*value fields*	
		stype	**svalue**
0	nil	nil	
1	x	cons	0
2	b		
3	c		
4	d		
5	e		
6	y	cons	1

Figure 3.4: Schema for the Symbol Relation

3.3.2. Symbol Storage

The symbol relation contains information that identifies the symbol and that specifies the value. The exact schema used to store Lisp symbols in Polymnia is shown in Figure 3.4. Every symbol is represented by a tuple in the *Symbol Relation* with the fields *name, id,* and a union-field (*stype, svalue*). In Figure 3.4, both *x* and *y* have values that are in the Cons Relation (depicted in Figure 3.5 and described below). The symbols *b, c, d,* and *e* have no value assigned to them.

3.3.3. List Storage

Lists are broken down into the structures from which they are composed to be stored in the database. Cons cells are simple structured objects that when chained together form lists. A relation to store cons cells must have a union-field for the **first** and **next** values plus an identifier field so that other tuples can refer to this one. Figure 3.5 displays the form of the Cons Relation.

In the Cons Relation, the **first** and **next** union-fields each have three pieces: a type (*firsttype, nexttype*), a value (*firstvalue, nextvalue*), and a symbol field (*firstsymbol, nextsymbol*). The type and value fields are interpreted in the same way as the *stype* and *svalue* fields of the Symbol Relation. When the type field has the value "symbol", the extra piece of

Cons Relation						
id	*first value fields*			*next value fields*		
	firsttype	**firstvalue**	**firstsymbol**	**nexttype**	**nextvalue**	**nextsymbol**
0	symbol	2	b	cons	1	
1	symbol	3	c	cons	2	
2	symbol	4	d	cons	3	
3	symbol	5	e	symbol	0	nil

Figure 3.5: Database schema for list storage

information (the *_symbol*[3] field) holds the symbol name. A noncircular list has the same properties as a binary tree: if there are *i* internal nodes, then there are *i + 1* leaves. The leaves of Lisp lists are either integers or symbols, and are predominated by symbols. Hence, about half of the first and next values are symbols so providing an immediate mode (like "int") will be beneficial to performance because it will allow a symbol value to be known immediately, saving the otherwise requisite fetch from the Symbol Relation.

Without the *_symbol* field, every time a symbol is referenced in a list, an extra retrieval would be required to get the name of the symbol from the Symbol Relation. To illustrate this point, consider the following example which uses the relations displayed in Figures 3.3 and 3.4. The value of the symbol *y* is *(c d e)*. Retrieving that value requires four database tuples to be fetched: one from the Symbol Relation with *name* = *y*; three from the Cons Relation with *ids* = 1, 2, and 3. These tuples are sufficient to determine the structure of the value of *y* (a list) and the values that are in that structure (the symbols in the list). Assuming that the Cons Relation does not contain the *_symbol* fields, seven tuples must be retrieved because the tuples in the Cons Relation no longer contain the name of the symbol, they only contain a referent to the Symbol Relation. The referenced tuples must be retrieved from the symbol relation (tuples with ids of 3, 4, and 5) to determine the names of the symbols. This extra field of information approximately halves the number of database retrievals required to deter-

[3] The notation *_symbol* designates both the firstsymbol and nextsymbol fields

mine the value of a list.

By storing the symbol name in the *_symbol* field of the Cons Relation whenever a cell references a symbol, the database schema has become denormalized. Since symbol's are identified by their name and names do not change, this duplication will not cause update difficulties. Although much space is taken up in the database to provide the immediate symbol mode, less time will be needed to retrieve list values. Time was deemed a more critical cost factor than space.

Edge Relation						
id	*field0*		*field1*		*field2*	
	type	**value**	**type**	**value**	**type**	**value**
1	symbol	name	cons	2	edge	1

Figure 3.6: Example of a Structure Relation

3.3.4. Structures as database objects

Each structure type has its own relation composed of an identifier field and an ordered group of union-fields. Figure 3.6 depicts the schema for the structure type Edge that was defined in section 2.2.3 which has three fields. To reference one of the tuples in Edge, the type of the union-field would be "edge" and the value would be the particular tuple's id, as shown by *field2* in Figure 3.6.

Structures in Lisp can be represented in virtual memory as lists. Therefore, instead of providing a separate relation for each structure type, all structures could be treated as lists in the database too. Figure 3.7 depicts the storage of the Edge tuple from Figure 3.6 as a list. Instead of taking up one tuple (as it does in the Edge Relation), this structure takes up three tuples in the Cons Relation. The fact that these three tuples compose an Edge is determined

Cons Relation						
id	*first value fields*			*next value fields*		
	firsttype	**firstvalue**	**firstsymbol**	**nexttype**	**nextvalue**	**nextsymbol**
.						
.						
.						
4	symbol	8	name	cons	5	
5	cons	2		cons	6	
6	edge	1		symbol	0	nil

Figure 3.7: Database schema for a structure stored as a list

by Lisp and the Equel/C interface. The storage form in the database (whether Cons Relation tuples or Edge Relation tuples) may be independent of that used in virtual memory (whether stored as a list, a vector, etc.). Both schemas will be examined. Section 5 reports on a performance comparison of using the Cons Relation versus using specialized structure relations for structure storage.

3.4. Example of Polymnia in Action

The actions taken by the Lisp package and the database package when executing a few simple statements are detailed in this section. The database is assumed to be in the state shown by Figures 3.3, 3.4, and 3.5; that is, the symbol x has the value *(b c d e)*, the symbol y has the value of *(CDR x)* which equals *(c d e)*, and the one edge structure has been declared. In addition, in the Symbol Relation, the symbol e is set to have the value of the Edge tuple.

The following represents a transcript of a Lisp session. All Lisp commands are in boldface; the actions taken by Lisp are in regular-face; and, the actions taken by the database package are in italics.

(defstruct (edge :persistent) v1 v2 v3)
This defines all the persistent accessors (v1, v2, and v3)
and associates them with the type edge.
(declare-persistent x y e)
This declares that x, y, and e have values that are stored in the database.
x, y, and e are each set to a special value that indicates this.
(CAR x)

The operation CAR (which returns the value of the first field)
notices that its parameter is persistent and has not yet been
fetched so it requests the value of x from the database package.
The tuple from the Symbol Relation is retrieved that has the name field
set to x. Its value is a tuple in the Cons Relation. This tuple is
fetched and a message with the following meaning is sent to the Lisp process:
x has value cons tuple with id=0,
cons tuple with id=0 has values first=b & next=cons tuple with id=1.
Lisp allocates a cons cell, registers it as fetched with the id equal to 0,
and allocates another cons cell, registers it as unfetched with id equal to 1,
sets the first field of the cell with id=0 to the symbol b and the next field
to the cell with id=1.
(CDR x)
The value of x is a fetched cell, but the operation CDR returns the value
of the next field which is an unfetched cell with id 1; therefore,
the cons cell with id of 1 is requested.
The tuple from the Cons Relation with id of 1 is read and the following is sent to Lisp:
cons tuple with id=1 has values first=c & next=cons tuple with id=2.
The cons cell that has id of 1 is located in the catalog of cells. Its first field is
set to c; a new cons cell is allocated and registered as unfetched with id of 2;
the next field of the cell with id of 1 is set to the unfetched cell; the catalog entry
for the cell with id of 1 is changed to say it has been fetched.
(CAR y)
The operation CAR notices that its parameter is persistent and has not yet
fetched so it requests the value of y from the database package.
The tuple from the Symbol Relation is retrieved that has the name field
set to x. It's value is a tuple in the Cons Relation. The following is
sent to Lisp:
y has value cons tuple with id=1.
Lisp checks the catalog of cons cells and finds a cell already allocated and fetched
that has the id of 1, therefore y is set to have that as its value.

This example has demonstrated how the fetching mechanism works and what information is needed by the Lisp package and the database package. Whenever a database fetch is executed, a unique identifier for the object requested is known. This unique identifier is used inside the Lisp package to identify the object also.

3.5. Performance of the Basic Prototype

This section reports on testing the performance of updating and retrieving persistent values and compares the two approaches to storing and retrieving structures from Section 3.4. In the first approach, each structure type is represented by a separate relation. This will be contrasted to storing the structures in their list form, composed purely of cons cells. The task used for the test will be described followed by the results of the test. The focus of the tests is the performance a Lisp application would realize. The time the DBMS takes to operate is visible from Lisp only as a delay in processing. Also, the time used by the Lisp garbage collector is separated out. Garbage collection time is dependent upon the particular Lisp system one is using and therefore can be considered a separate cost. The cpu time used for garbage collection is subtracted out of the regular cpu time.

3.5.1. The Task: Parsing & Generating

An Artificial Intelligence program that parses English sentences into their grammatical make-up and generates sentences from the grammatical make-up was used to perform the set of tests reported here. PP[4] takes as input an English sentence and creates a data structure that contains all the syntactic information in the sentence. That is, it creates parse trees for all possible parses of the sentence. This parse tree structure is composed of persistent objects which are dynamically stored in the database as they are created during the parsing. The generator of PP retrieves all of the syntactic information stored in the database and regenerates the original sentence and the various syntactic parses the sentence has.

The test was to parse the following simple story:

> I saw the man. I saw the man on the hill. I saw the girl on the hill with the rabbit. that stupid rabbit hopped under a rock. the girl thought the rabbit ran off. this made the little girl cry. the man found the rabbit for the little girl. the happy little girl hugged the rabbit.

Each sentence's syntactic parse was stored as the value of a symbol (sentence1 through sentence8). The original sentences and all of their parses were then printed out by the generator which traverses the entire data structure. Consequently, when using persistent data, the generation phase caused all the tuples to be retrieved from the database.

[4] PP stand's for Peter's Parser. The parser and generator were written by Peter Norvig, a Research Associate in the Berkeley Artificial Intelligence Research group at the University of California at Berkeley.

The tests were performed single-user on a Pyramid 9820 dual processor machine. Franz Lisp Opus 38 was the Lisp system and RTI Ingres version 5.0 [RTI83] was the database system utilized.[5] Most of the tests were performed more than once and the reported time is the average of the tests.

3.5.2. Results of the Test

Three topics will be covered in this section: the slowdown associated with persistent data, the cost of retrieving versus updating persistent data, and the difference between using a special relation for each structure type and using the Cons Relation for structure storage. Of first consideration is the slowdown associated with the use of persistent structures.

3.5.2.1. Performance of an application with a persistent database

To use an application with a persistent database, the data first needs to be stored in the database. After the data is stored once, the application can run by retrieving only the data needed for the current use. This is to be contrasted with an application which uses a database stored only in virtual memory. Every time the *virtual* application runs, it must build the entire database in virtual memory. For the application PP, the building and storing of the database occurs during the parsing stage; the generating stage accesses the structures created during parsing. The advantage to storing data in a database is that it needn't be parsed again to generate it. In this subsection, the performance of the generating stage of PP is studied.

The time used by PP for *parsing* and *generating* without the use of persistent data is shown in the first row of Figure 3.8. *Parsing* creates a data structure in virtual memory which is traversed in the *generating* stage. The second row shows the performance of the *generating* stage when persistent structures are used. To accomplish this, the story was parsed

[5] The original system was developed and tested using RTI Ingres version 2.1 running on a Vax 11/780 [Butl86a].

	Parsing			*Generating*			*Total*		
Test:	**real**	**cpu**	**gc**	**real**	**cpu**	**gc**	**real**	**cpu**	**gc**
Original	14	6.4	3.2	5	3.5	.9	19	9.9	4.1
Polymnia	-	-	-	83	22	11	83	22	11

In seconds

Figure 3.8: Performance of Original versus Persistent

and the resulting data about the structure and meaning of the story was stored in the DBMS. First the cpu time used by the Lisp process will be examined, then the elapsed time (*real-time*), which reflects the time it takes to fetch the data from the DBMS, will be considered.

The times in Figure 3.8 show that the persistent system uses 6 times more cpu than the original system to perform the *generating* test. Much of the time used in persistent test comes from building the new structures in virtual memory. After the data is fetched from the database, it must be stored in virtual memory and allocating memory is costly. In the original system, the *parsing* stage allocates all the structures; the *generating* phase only traverses them. Since the *generating* stage in the original system cannot occur without the structures being parsed, to have a fair comparison, the original system's *generating* time used for comparison should include the parsing time. The rightmost set of columns in Figure 3.8 compares the performance of the *generating* stage using Polymnia to the performance of *parsing* plus *generating* when using the nonpersistent system. Comparing this *total time,* the original system takes 9.9 seconds of cpu and the persistent system takes 22 seconds. The extra time is used for the bookkeeping done in Polymnia: cataloging cells, setting database ids, etc. Henceforth, all comparisons with the *generating* times for the nonpersistent system will refering to this *total time.*

Subtracting the cpu-time and gc-time from the real-time for Polymnia results in the amount of time used to fetch the data from the database (i.e., 50 seconds). Since the database for PP contains 1916 tuples and 50 seconds are used to retrieve the tuples, about 38 tuples are retrieved per second. Since the tuples are not organized on the disk, these tuples are randomly placed. A random retrieval rate of 38 tuples per second is very good.

A persistent system, such as Polymnia, will perform best with applications that access a piece of persistent data multiple times after it is brought in from the database. This would allow the cost of retrieving the data to be amortized over many accesses. This application brings the data in and traverses it only once. Figures 3.8 and 3.9 graph how the difference between the performances of the persistent and nonpersistent system becomes less significant over many accesses. Each *access* in the graph represents running the *generating* test once. For the persistent system, the data is presumed to be fetched into virtual memory during the first access. Hence, the rest of the accesses cause no data to be fetched from the database. For the original system, the data is *parsed* during the first access. Figure 3.9 depicts the CPU cost and Figure 3.10 graphs the real-time. The difference between the CPU cost of Polymnia and the Original system is 12 seconds (see Figure 3.8). Over many accesses this difference becomes insignificant. The real-time difference between Polymnia and the Original system remains constant at 64 seconds over all accesses. Just as with the CPU cost, as the number of accesses increases, the significance of that 64 seconds becomes smaller.

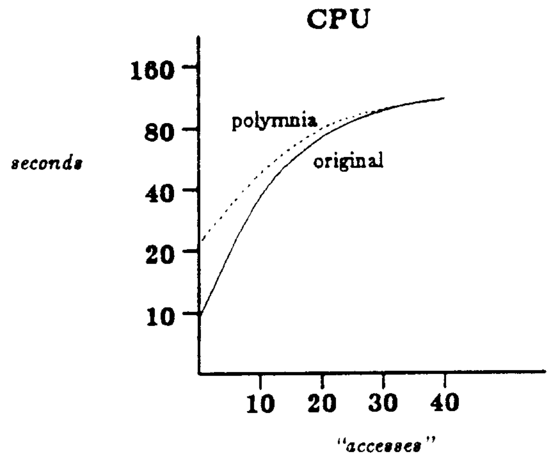


Figure 3.9: CPU usage of Original versus Polymnia over many accesses

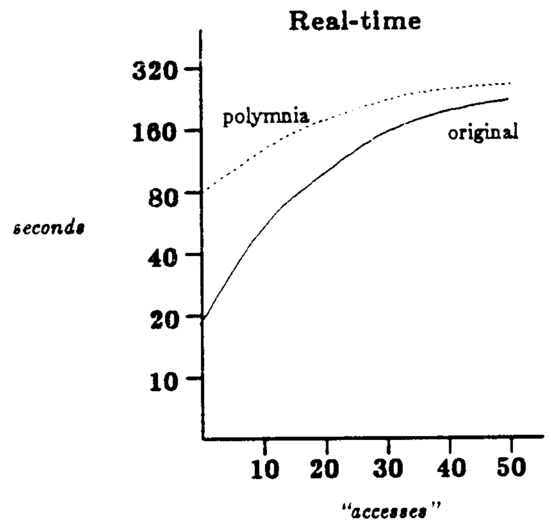


Figure 3.10: Real-time of Original versus Polymnia over many accesses

3.5.2.2. Performance of Updates versus Retrievals

The previous subsection examined the cost of retrieving persistent data. In this subsection, the cost of updating data is discussed and compared to that of retrieving data.

The *generating* phase of PP traverses the data structures but does not update them; whereas, the *parsing* stage of PP creates and updates many data structures but does not access any previously existing data. Hence, *parsing* is update-only and *generating* is retrieve-only. By comparing the performance of these two parts of PP, the cost difference between updating and retrieving persistent data can be examined.

Figure 3.11 depicts the times for the *parsing* and *generating* tests for both Polymnia

	Parsing			*Generating*		
Test:	**real**	**cpu**	**gc**	**real**	**cpu**	**gc**
Original	14	6.4	3.2	19	9.9	4.1
Polymnia	411	43	19	83	22	11

In seconds

Figure 3.11: Performance of Updating versus Retrieving

and the Original system[6]. For the *generating* stage, Polymnia uses about 120% more cpu time than the Original; whereas, for the *parsing* stage, Polymnia uses almost 600% more. This makes it clear that Polymnia must do more work when creating and updating data than when retrieving it. Not including tasks performed by both Polymnia and the Original system, retrieving data involves setting the values of the structure, whereas, updating and creating data involves sending the values of the structure to the DBMS, reading the database id from the DBMS, storing the id, and recording that the structure's value has been retrieved. Consequently, persistent updates and creates are relatively slower than persistent retrieves.

In the previous subsection, the retrieval rate was determined by subtracting the cpu-time and gc-time of Polymnia from the real-time and dividing this into the number of tuples retrieved which resulted in 38 retrievals per second. Performing a similar calculation on the *parsing* times, the update rate can be determined. The amount of time used by the database during parsing is 349 seconds. The number of tuples created plus the number updated during the *parsing* test is 3405. Therefore, the effective update rate[7] is about 9.8 per second. Hence, tuples can be retrieved almost 4 times faster than they can be updated or created.

The large difference between update and retrieval rates is caused by two factors: (1) database updates take more time than database retrievals on average; and, (2) the Equel/C portion of Polymnia executes more than one query when a tuple is created. When a tuple is created, it must be allocated a unique identifier. The identifiers are sequentially generated for each relation with the last one stored in the database. Before a new tuple is allocated, the next identifier is determined by retrieving the last generated one. This number is then incremented by one and stored back in the database. Lastly, a new tuple is inserted into the specified relation with the new identifier in its *id* field. Therefore, a request to create a tuple

[6] The quoted generating time for the Original system includes the parsing time. Recall from the previous subsection that this results in comparable executions for Polymnia and the Original system.

[7] the tuple update rate that the application experiences

results in 3 queries: a retrieval, an update, and an insert. Database updates and inserts, on average, take the same amount of time; and, typically, updates take more time than retrievals since indices may be updated. Since the retrieval rate is known (38), the DBMS update rate can be calculated. 1916 tuples are created; in addition, there are 1489 updates. To accomplish these updates and creates, 1916 retrieves occur and 5321 (1916*2 + 1489) updates occur. The retrieves take 50 seconds which leaves 299 seconds the updates. Therefore, the DBMS update rate must be 18 queries per second (less than half the retrieval rate).

The effective update rate is 9.8 tuples per second, as compared with the calculated DBMS update rate of 18 tuples per second. Hence, the effective update rate is about half what it could be. This difference can be attributed to the extra queries executed by the Equel/C interface when tuples are created. Two things can be done to speed up creates: (1) Unique identifiers could be kept outside the database; hence, generating them would not result in two queries; or, (2) A retrieve/update query could be implemented that updates a tuple and returns it which would allow the identifier generation process to take one query. Either of these options would allow the effective update rate to become closer to the calculated DBMS update rate.

3.5.2.3. Performance of using specialized versus general-purpose relations

Most data types can be constructed using cons cells. Therefore, the Cons Relation can be considered a general purpose relation for most Lisp data. Section 3.4 depicted storing Lisp structures in the Cons Relation and also showed how each type of structure could be represented by a relation in the database that was specifically designed for that structure. The performance reported for Polymnia in the previous subsections was that of using specialized relations for each structure type. In this subsection, the performance of specialized relations will be compared to that when the Cons Relation is used to store structures. Figure 3.12 depicts the results of running the *parsing* test and the *generating* test using Polymnia when special relations are used for each structure type versus when all structures are stored as composed of cons cells. First, the retrieval rates will be compared, then the update rates.

3.5.2.3.1. retrieval rates

Looking at the cpu time for generation phase (Figure 3.12), using the Cons Relation takes twice as long as when using structure relations (44 seconds versus 22 seconds). Considering the time used by the DBMS, it takes 7 times as long for this retrieval test when using the Cons Relation (357 (Cons) versus 50 (structures)). The retrieval rate when using the structure relations is about 38 tuples per second; whereas, the rate when using the Cons Relation is 14 tuples per second.

The difference in the retrieval rates is caused by the Equel/C interface. The handling of structures is simple: one is requested and one is retrieved. The handling of cons cells is more complex: a request could be for a chain of cells or for an individual one. Therefore, many extra steps are taken when retrieving one tuple from the Cons Relation than one tuple from a structure relation. The reason for the complexity is that if a Lisp function needs an entire chain of cons cell to execute, it saves communication cost by sending one request to Equel/C and having the Equel/C interface retrieve as many as are needed. PP causes each cons cell to be retrieved individually. To test whether or not the added complexity helps when retrieving a string of cons cells, an application was used that uses functions that fetch entire sublists in one request. This application is an AI program named PHRAN [Wile84].

	Parsing			*Generating*		
Test:	**real**	**cpu**	**gc**	**real**	**cpu**	**gc**
Structural Relations	411	43	19	83	22	11
Cons Relation	874	37	17	415	44	14

In seconds

Figure 3.12: Performance when using Structural versus Cons Relations

PHRAN is basically a shift-reduce parser for natural language sentences, such as English, with the bulk of the program dedicated to handling language constructs that do not fit the shift-reduce paradigm. A user of PHRAN types in a sentence and gets back a conceptual representation for the sentence. PHRAN has two major components: a processing part, and a knowledge base (KB).

The knowledge base which consists of 37,280 cons cells and 72 symbols was entirely loaded into INGRES. The symbols whose values represent the KB were declared to be persistent inside PHRAN's code. The following five sentences were parsed by PHRAN:

Yigal went to the big apple.
Yigal ran to the big apple.
Yigal gave Mary a big apple.
Mary sold an apple to Yigal.
Mary has a big apple.

Since all English word and grammar knowledge is stored in the DBMS, all relevant tuples had to be retrieved to parse them into their conceptual representations. Parsing these sentences results in 3913 Cons Relation tuples and 7 Symbol Relation tuples being fetched from the database.

Figure 3.13 reflects the timings for PHRAN when operating with a nonpersistent KB and a persistent KB. Just as with the PP *generating* test, the persistent system uses about twice as much CPU as the nonpersistent system. 3920 tuples are retrieved in 218 seconds which is 18 tuples per second. This is about 30% higher than the retrieval rate found for retrieving cons tuples during PP,but is still less than half the structure retrieval rate.

	Real	**CPU**	**GC**	**DB-Time**	**Tuples Fetched**
NonPersistent	189	118	54	-	-
Persistent	585	239	128	218	3920

Figure 3.13: Persistent PHRAN timings

Clearly, the complexity of the Equel/C interface when retrieving cons tuples is not worthwhile. This could be a factor influencing the poor retrieval rate for structures stored in the Cons Relation.

3.5.2.3.2. update rates

When using cons cells as the database storage medium, the parsing phase takes almost 900 seconds; whereas, when special structure relations are used, the *parsing* test takes close to 400 seconds. Slightly less cpu is used by the system when the Cons Relation is used (15% less than when special structure relations are used), but the elapsed time is more than twice as much. The real-time is dominated by the time the DBMS uses. Figure 3.14 depicts the reason for the cost difference. When data is stored in structure relations, a total of 1916 tuples are created; whereas, when using cons cells, 5043 tuples are created. In addition to the tuples that were created, 1489 updates to tuples are made. If the tuples involved in the updates are added to the tuples created (this gives us the total number of writes to the database that are requested by the application), 3405 tuples are written when structures are used, as opposed to 6959 when cons are used. The effective update rate when using structure relations is 9.8 per second and when using the Cons Relation is 8.5. The effective update rate makes it appear to be more costly to update the Cons Relation than a specialized relation. By examining the queries executed while doing the creates and updates, the actual DBMS rate can be determined. This may substantially differ from the effective rates.

Section 5.2.2 outlined the actions taken when a tuple is created: a create request causes 3 queries. The number of update and retrieval queries executed when using the Cons Relation and the structure relations is shown in Figure 3.14. Assuming the retrieval rate for the Cons Relation to be 14 tuples per second, 5043 retrieve queries will take 360 seconds to execute. Since 820 seconds are used for the 5043 retrieves and the 11,575 updates, using 360 for retrieving leaves 460 for updating. This indicates an actual update rate of 25 tuples per second. Since the retrieval rate was calculated to be 14 tuples per second, if updates are faster this must be a result of the complexity in the Equel/C interface for retrieves from the Cons Relation.

3.5.2.4. Summary of results

A very good retrieval rate (38 tuples per second) has been achieved for randomly placed tuples from the structure relation. This speed can improve with better DBMS or disk technology. In addition, when data is accessed many times, the initial retrieval cost becomes insignificant.

Another conclusion that can be drawn is that updating and creating tuples takes much more time than retrieving. Clearly an ideal application for persistent data is one that involves many more retrievals than updates. One technique to lessen both update and create costs would be to use a delayed update technique that would enable many updates and creates to be batched together.

The actions taken by the Equel/C portion of Polymnia when a tuple is created need to be simplified to reduce the overhead. This would bring the cost of a create closer to the cost of an update. In addition, the complex methods of retrieving strings of tuples from the Cons Relation do not benefit performance. The results indicate that the simple interface provided for structures (retrieve one at a time) allows a faster retrieval rate than the complex one provided for cons cells (retrieve one or a string).

Test:	tuples		#queries		db-time	
	created	updated	retrieves	updates	parsing	generating
Structures	1916	1489	1916	5321	349	50
Cons Cells	5043	1489	5043	11,575	820	357

Figure 3.14: Tuple and Byte count for PP test

Using a general purpose storage technique (i.e., Cons Relation) is slower than using a more specialized approach (i.e., structure relations). This is a direct consequence of the Cons Relation requiring more tuples to store the same amount of information and the slower retrieval rate. Since the time spent retrieving and updating data is correlated with the number of queries executed, techniques to compact data into fewer tuples should result in lower cost.

3.6. Conclusion

Since PP is a data intensive application, the performance achieved by it is impressive. The generating stage traverses a structure; when that structure must be retrieved from a DBMS during traversal, the application slows down by as little as a factor of four. The parsing stage creates a large data structure; this structure must be duplicated in real time in the DBMS. Due to idiosyncrasies in the way tuples are created, the application slows down by a factor of thirty. If a request to create a tuple did not generate extra queries, the slowdown would be cut in half.

PHRAN is a read-only application that does a lot of processing per datum. Therefore, PHRAN experiences a slowdown of a factor of three. This application is an indicator of the improved performance of persistent applications if data is accessed many times after it is brought into memory. This will be discussed further in the next chapter.

Polymnia is a layered system that extends Lisp to include persistent data. One factor in the performance cost of the system is the layered approach used to build it. When Lisp determines the piece of data it needs, it sends its request to the Equel/C process which parses the request and determines what query to run to retrieve the data. This query is sent to the DBMS via a pipe and is executed. The resulting data is sent to the Equel/C process and stored in a variable. This variable is examined and the data is sent in an appropriate form to the Lisp system. Lastly, the Lisp system parses the returned data, creating all the necessary structures. A data request passes through many layers before data is returned to Lisp. Reducing the number of layers should result in improved performance. But as was discussed in the introduction, this architecture is more reliable and enables a system to be more easily tuned for a particular application.

The major advantage of the approach discussed here is its generality. The approach used by Polymnia for storage of Lisp values (using union-fields with a new relation for each type) can easily be extended to encompass other types. For example, to include three element arrays, one can create a relation called Array3. If a persistent symbol is an array of three elements, it's *stype* field will have the value "array3" and it's *svalue* field will hold the index into the Array3 relation. Extending the system to encompass arrays could be implemented in many different ways; this is just an illustration of the ease with which this approach can be extended to handle other kinds of Lisp objects.

The database cost can be lessened by fetching more from the database at a time as demonstrated by the results of the structure versus cons cell study. An average of three Cons tuples had to be retrieved for each tuple in a structure relation. This had a devastating effect on the performance. In the next chapter, another strategy for receiving more data per DBMS retrieval is reported.

CHAPTER 4

Cache Management

And when they came to the place that was very far away Martin said, "Oh how happy we'll be." "Forever," said the horse. "--And ever," sighed the sparrow. And they climbed through the window one by one, except the horse. He stuck in the middle with his head on the inside and the rest of him on the outside.

Maurice Sendak, *Very Far Away*

4.1. Introduction

Just as cache memories speed up systems by buffering main memory information, persistent applications can be sped up if main memory acts as a buffer for disk-based data. By *caching* data fetched from a database in main memory, data access (and therefore an application's performance) can be drastically improved. Cache memories are successful due to the *property of locality*[Denn72] which has two aspects, *temporal* and *spatial*. *Temporal locality* means that information which is currently in use will be used again in the near future. This is the basis upon which Polymnia chose to cache persistent data in virtual memory: data that an application currently needs, it will probably need again in the future. *Spatial locality* means that portions of the address space which are in use consist of small, contiguous segments of that address space. This behavior can be expected when related data items are physically located together (e.g. array elements or clustered data). Chapter 5 studies the effects of clustering data in the database; hence, spatial locality is covered in that chapter.

A major summary of work on cache memories [Smit82] identifies many design alternatives that affect cache performance. Due to the differences between hardware cache memory and main memory used to cache disk-based data only a subset of the alternatives are relevant to study in a system that caches database data in main memory. Specifically they are: line size, fetch algorithm, cache size, update algorithm, and multicache consistency. (For the remainder of the discussion, *cache* will refer to the use of main memory as a buffer for data that is regularly housed on disk.)

Line size: The number of pieces of data transferred between secondary memory and the cache is called the line. Since each query represents one request for data, the line size for Polymnia corresponds to the contents of a single tuple. The line size can increase by placing more than one piece of data in a tuple. Clustering related pieces of data together in a tuple to retrieve them in one query is studied in the next chapter. This should be advantageous to performance if spatial locality is displayed by applications when the data is clustered.

Cache size: The amount of main memory used for the cache will clearly affect the amount of performance benefit accrued by caching data. Cache size will be discussed in Section 2.

Cache fetch algorithm: The cache fetch algorithm determines what information to cache and when to bring it into the cache. Two possibilities exist: data can be fetched on demand (when it is needed) or prefetched (before it is needed). In the tests reported in Chapter 3, Polymnia fetched data upon demand; Section 3 of this chapter relates a study in which Polymnia used various forms of prefetch.

Update algorithm and *multicache consistency*: If data is shared between processes, two separate processes may each contain a cache of the same data value. This inherently poses a consistency problem. The handling of updates is also a sensitive issue when data is shared. The many complex problems when cached data is shared are beyond the scope of this thesis.

Section 2 describes some of the effects of cache size. Differing fetch algorithms are explored in Section 3. Lastly, Section 4 summarizes the results of this chapter.

4.2. Cache size

The previous results reported for PHRAN and PP were attained by assuming the cache to have infinite size; that is, all data fetched from the database was buffered in main memory. Since the space available to a process may not be infinite, these applications running under Polymnia were analyzed to predict the performance if the cache size is limited.[1] Figure 4.2 shows the results of the analysis.

When all data accessed by PHRAN is cached, 138 data requests are sent to the database. By monitoring the operations that request data, it was determined that if data were not cached, 3851 requests for data would be made. If 138 requests result in 3920 tuples being fetched, assuming that the ratio of requests to tuples fetched remains the same, one can predict that 3851 requests will result in 109,391 tuples being fetched. To fetch 3920 tuples takes 218 seconds (DB-time); therefore, fetching 109,391 tuples is expected to take 6084 seconds. If there is no caching, no extra cpu cost is needed to process persistent data and store it in memory; hence, the CPU/GC cost is expected to be close to that of persistent PHRAN when all data is in the cache. This cost has been measured to be 252 seconds. Therefore, the total time that PHRAN is expected to take when the cache is of size zero is 6336 seconds (approximately 1.75 hours). The cost is dominated by the data retrieval time.

The average number of accesses per tuple can be calculated by dividing the number of tuples retrieved if there is no cache by the number when there is a cache (3851/138 = 28). Each fetched tuple is accessed an average of 28 times by PHRAN. Since only a subset of Lisp operations could be monitored to check if the data is accessed, this is not the actual average

[1] The caching of data is so integral a part of Polymnia that the system could not accurately be altered to remove it.

Cache size:	**Lisp Requests**	**Tuples Fetched**	**CPU/GC**	**DB-Time**	**Real**
		PHRAN			
infinite	138	3920	367	218	585
no cache	3851	109,391	252	6084	6336
		PP -- generating			
infinite	4986	4994	33	50	83
no cache	21,930	21,974	33	220	253

Figure 4.2: Expected PHRAN & PP timings with and without caching

number of accesses; it is the minimum average number of accesses. This also only accounts for new accesses not accesses when a value is returned by a function. The actual average accesses per value is much higher.

The generating phase of PP does not do a lot of processing per datum. The minimum average number of accesses is 4.4. Hence, if there was no caching, the cost of using the database would only triple. That is, instead of taking 83 seconds, generating would take 253 second.

To summarize the results of this section, caching is of great advantage to persistent systems. The amount of performance improvement depends upon the access characteristics of the application. Since the size of main memories has been growing rapidly, allowing a persistent system to cache as much data as the system desires to cache seems feasible. The benefits of caching all objects break down when data can be shared. If data sharing is supported, limiting the amount of time an object can be cached and the number of objects cached may be desirable.

4.3. Prefetching: varying the fetch algorithm

From the previous tests, the use of persistent data (in terms of the elapsed time) was shown to slow an application by a factor of three or four. One way to possibly lessen this delay would be to have the data for the Lisp application fetched from the database before it was requested. This section describes a test performed to measure the performance when prefetching was used.

4.3.1. Prefetching Test Results

The system must be able to predict what data will be accessed next based upon past references to make prefetching worthwhile. When accessing data that contains interobject references, the system can predict that an object referenced by the accessed object may be needed. This technique proves to be fairly successful for Lisp data.

When an unfetched cons cell is referenced, the following extra cells could be requested:

next:the cell pointed to by the next field of the cell that was referenced;
first&next:the cells pointed to by the first and the next pointer
of the cell that was referenced;
all-nexts:all cells reachable transitively via following only next pointers;
2-all-nexts:do an *all-nexts* fetch from the referenced cell and from it's first pointer, too.

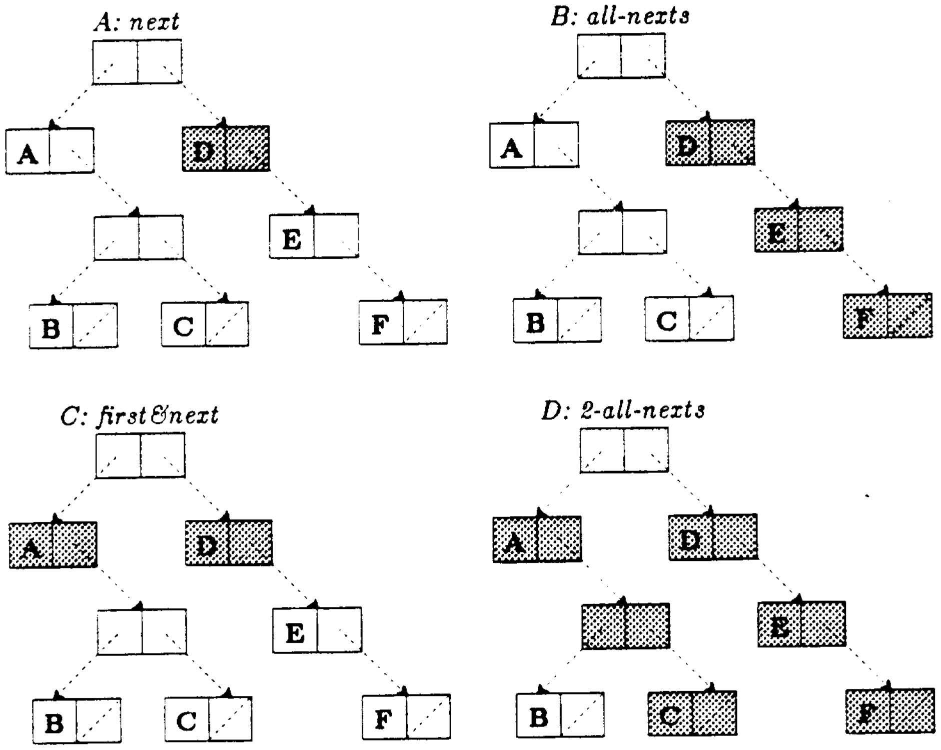


Figure 4.3: Prefetch methods tested

Figure 4.3 depicts the list *((A (B) C) D E F)* in cons cell form. When the dotted cell is fetched, the ones filled with slashes represent the extra cells that would be prefetched if the labeled prefetch method is employed.

Both PHRAN and PP were tested using the various prefetching strategies. Due to the circular nature of PP's data, neither the *all-nexts* nor *2-all-nexts* strategies could be tested on PP. The prefetching is performed by the Equel/C layer of Polymnia. This layer is limited in function because it does not record which tuples have already been retrieved. Hence, doing a transitive fetch, such as an *all-next* request, would result in an infinite loop.

Due to the different access characteristics of the applications, the prefetching tests have different purposes. Since PP needs all of the data in the database, the PP prefetching tests show the maximum parallelism that can be achieved using these techniques. On the other hand, the PHRAN test uses only 1/10 of its database, so its prefetching test will determine the accuracy of the predictors and whether the accuracy (or lack thereof) effects performance.

For all the different prefetch strategies, the Lisp cpu time was approximately equal; therefore, Figure 4.4 shows only the difference between the real-time and the cpu-time which is called DB-time. The tests were performed on a dual processor machine; therefore, the

Kind of Prefetch:	**DB-Time**	**Tuples Fetched**	**Lisp Requests**
	PHRAN		
no prefetch	218	3913	138
next	203	3913	139
first&next	212	4118	385
all-nexts	230	3913	699
2 all-nexts	181	3913	114
	PP		
no prefetch	357	4994	4986
next	282	4994	4986
first&next	255	4994	4986

In seconds

Figure 4.4: Performance of PHRAN & PP with Prefetching

DBMS can operate in parallel. Hence, the actual time used by the DBMS may be greater than DB-time, because some of the operation may be done parallel. The number of *Lisp requests* is the actual number of commands sent to the Equel/C interface from Lisp. Some prefetch methods require more than one command (e.g. *first&next* requires two commands); hence, the number of Lisp requests will increase even though fewer requests are forced by the application.

When prefetching is used, if the predictor is not accurate the resulting performance could be very poor because much unnecessary data would be fetched from the DBMS. According to the results for PHRAN in Figure 4.4, the number of cells fetched when employing *next*, *all-nexts*, or *2-all-nexts* is the same as that when no prefetch is employed. Therefore, all data that was prefetched was subsequently referenced. This indicates the method of prediction was 100% accurate. For PHRAN the greatest performance improvement was realized for the *2-all-nexts* prefetching strategy (it reduced DB-time by 17%). The communication cost to Equel/C is low since this method produces the fewest number of Lisp requests. Every cell fetched is used; hence, no unnecessary data is fetched. The pattern of access for PHRAN causes *all-nexts* to have the poorest performance which in part could be the result of the high number of requests sent to Equel/C. One of the functions PHRAN executes requires that a portion of its knowledge base equivalent to that fetched by a 2-all-nexts request be retrieved from the database. When the all-nexts prefetch method is employed, before this function executes, another operation executes on the knowledge base and an all-nexts prefetch is requested. The function that requires the second layer of all-next then issues one data request for each sublist it needs that was not retrieved by the all-nexts request.

For the PP test, since all tuples are needed, the best performance should result when the DBMS is working continuously while the application is working (i.e., achieving maximum parallelism). For all the methods tested with PP, each cell required one Lisp request; hence, all prefetching methods result in the same number of requests. The more prefetching that occurs, the greater the performance benefit. This is a direct result of increased parallelism between the application and the DBMS. By prefetching *first&next*, the DB-time gets reduced by 30%.

4.4. Conclusion

Section 2 stressed the impact that caching has on an applications performance. Without caching, each PHRAN object would have been fetched 28 times and each PP object, 4.4 times. Obviously, an infinite size cache will allow the best performance since all objects can be cached. Due to the increasing size of current main memories, assuming a persistent system may cache any data it desires in main memory is reasonable. Only when objects are shared do these caching assumptions breakdown. Some of the systems surveyed in Chapter 2 addressed the problems of sharing cached objects. The best way of handling this situation is an area for future research.

Section 4 explores a technique for reducing the effective database fetch time. If the DBMS can operate in parallel with an application without harming the application's performance, prefetching is a simple way of allowing parallel execution: when an application requests a piece of data, that data is returned; then the DBMS fetches something that it predicts the application will need next. (In the case of Polymnia, the Lisp system actually determines what may be needed next, not the DBMS itself.) The effectiveness of prefetching is determined by how accurate the predictor is and how much can be done in parallel. For PP, the oracle was 100% accurate since all data were needed; an interesting result is that for PHRAN a perfect oracle is easy to devise. Although the PHRAN test only required 10% of its database to be fetched, the least accurate prefetching method caused only 5% more than necessary number of cells to be fetched. Predicting that a cell referenced by a next pointer will be accessed is 100% accurate.

Whether or not data is cached may determine whether persistent data is too costly to use. Much research has been done on hardware caches; some of this work can be applied to caching persistent data in main memory. Namely, prefetching can be of extreme benefit to a system that caches persistent data. If an accurate oracle can be devised for an application, prefetching can be very beneficial. Using the semantics of the data (i.e., following interobject references) appears to produce very accurate prefetching methods. In addition to the access pattern determining how accurate the oracle is, it also determines how much parallelism can be realized. Without sharing, caching cannot hurt performance and it can be a great benefit depending upon the application.

CHAPTER 5

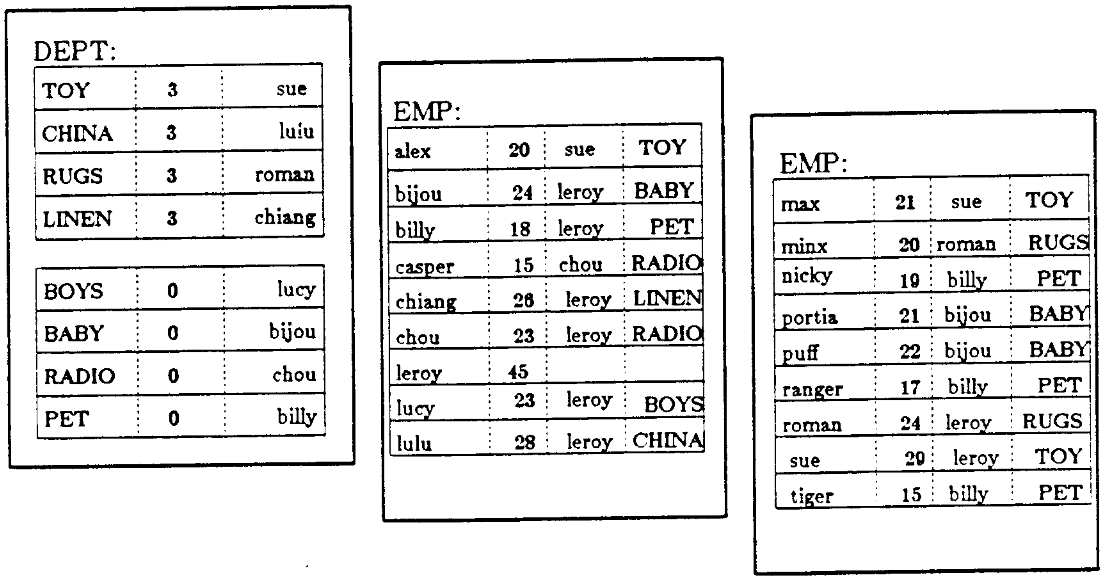
Physical Placement of Tuples

Without a doubt there is such a thing as too much order.
Arnold Lobel, *Fables*

5.1. Introduction

When choosing storage structures for a database, the placement of tuples on pages is a factor to be considered. Often the placement is determined by the choice of a primary index. For example, if the primary index is a B-tree keyed on a particular field, then the tuples will be placed in sorted order according to the value of that field. Most work on physical database design focuses on choosing efficient structures for storing a set of homogeneous tuples. This chapter explores the properties that may be needed for efficient implementation of persistent data such as that of Polymnia in which the set of tuples to be stored are not necessarily homogeneous.

Figures 5.1 and 5.2 illustrate some traditional database placement strategies. There are two relations, EMP [ename, esal, edept, emgr] and DEPT [dname, dfloor, dmgr]. In Figure 5.1, the EMP Relation is stored sorted by *ename* (this might be the result of the primary index being a B-tree keyed on ename) and the DEPT Relation is hashed on *dfloor* which results in the tuples being clustered by dfloor. In Figure 5.2, each employee tuple is clustered near the department in which the employee works. This might be caused by issuing a statement such as: *cluster EMP by DEPT where EMP.edept=DEPT.dname.* This serves two purposes: it allows queries that are joined on the fields edept and dname to execute more quickly;



DEPT:

TOY	3	sue
CHINA	3	lulu
RUGS	3	roman
LINEN	3	chiang

BOYS	0	lucy
BABY	0	bijou
RADIO	0	chou
PET	0	billy

EMP:

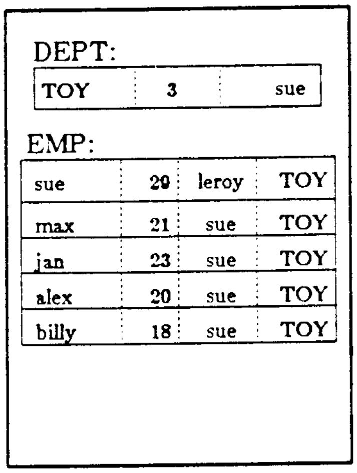
alex	20	sue	TOY
bijou	24	leroy	BABY
billy	18	leroy	PET
casper	15	chou	RADIO
chiang	26	leroy	LINEN
chou	23	leroy	RADIO
leroy	45		
lucy	23	leroy	BOYS
lulu	28	leroy	CHINA

EMP:

max	21	sue	TOY
minx	20	roman	RUGS
nicky	19	billy	PET
portia	21	bijou	BABY
puff	22	bijou	BABY
ranger	17	billy	PET
roman	24	leroy	RUGS
sue	29	leroy	TOY
tiger	15	billy	PET

Figure 5.1: Page layout when EMP is sorted on ename and DEPT is hashed on dfloor

and, it clusters all employees together who work in the same department making it faster to access all employees in one department. If employees were scattered across pages, as in Figure 5.1, more disk reads would be necessary for those two kinds of accesses than if employees are clustered.



DEPT:

TOY	3	sue

EMP:

sue	29	leroy	TOY
max	21	sue	TOY
jan	23	sue	TOY
alex	20	sue	TOY
billy	18	sue	TOY

Figure 5.2: Traditional db clustering

To illustrate this point, assume there are E employees and D departments with NP tuples per page (for simplicity the size of EMP tuples and DEPT tuples are equal). There are P pages of tuples, where P $=\lceil (E+D)/NP \rceil$. Assuming the employees are evenly distributed across departments, there are E/D employees per department. To execute the query *Retrieve emp.all where emp.edept=TOY* will take a varying number of page reads depending upon the tuple placement strategy. If the tuples are clustered as shown in Figure 5.2, only one page will be read. If the tuples are placed as in Figure 5.1, two pages must be read. In a worst case scenario, if the tuples are scattered across pages, five pages may be read. The average cost per tuple, which is obtained by dividing the number of pages read by the number of tuples retrieved, if objects are clustered is approximately 1/NP; whereas, if objects are not clustered, up to 1 page per object will be read. Clearly, placement strategy can greatly effect disk I/Os.

The access characteristics of the persistent data of Polymnia are not similar to that of traditional data such as in the above EMP example. Considering the Cons Relation from

CONS:

1	symbol A	cons 5
5	cons 2	cons 10
10	symbol G	symbol NIL
15	cons 13	cons 8
8	int 1	cons 9
9	int 2	cons 16
16	int 3	cons 12
12	int 4	symbol NIL
7	symbol D	symbol NIL

CONS:

2	symbol B	cons 3
3	cons 6	cons 4
4	symbol F	symbol NIL
6	symbol C	cons 11
11	cons 7	cons 14
14	symbol E	symbol NIL
13	int 5	cons 17
17	int 6	cons 18
18	int 7	symbol NIL

Figure 5.3: CDR-coded Cons Relation

Chapter 3 (simplified in Figure 5.3), the value of the *next* field often indicates the tuple that will be accessed next. Virtual memory tests were performed to determine how to place cells so as to minimize paging in Lisp environments. These studies resulted in a technique of organizing cons cells in memory called CDR-coding [Bobr79]. CDR-coding is similar to placing cons cells according to the depth-first traversal of a list. To illustrate a CDR-coded ConsRelation, the two lists *(a (b (c (d) e) f) g)*[1] and *((5 6 7) 1 2 3 4)*[2] have been stored in a Cons Relation using placement fashioned after CDR-coding (Figure 5.3) and also have been stored sorted on the id field (Figure 5.4). CDR-coding is distinct from the traditional clustering depicted in Figure 5.2 in that CDR-coded placement is based not upon having a similar value in one field, as is the placement of the EMP tuples in Figure 5.2, but upon one tuple refering to another in a particular field.

CONS:

1	symbol A	cons 5
2	symbol B	cons 3
3	cons 6	cons 4
4	symbol F	symbol NIL
5	cons 2	cons 10
6	symbol C	cons 11
7	symbol D	symbol NIL
8	int 1	cons 9
9	int 2	cons 16

CONS:

10	symbol G	symbol NIL
11	cons 7	cons 14
12	int 4	symbol NIL
13	int 5	cons 17
14	symbol E	symbol NIL
15	cons 13	cons 8
16	int 3	cons 12
17	int 6	cons 18
18	int 7	symbol NIL

Figure 5.4: Cons Relation sorted on id

[1] This list begins at the cons cell with id=1.

[2] This list begins at the cons cell with id=15.

In this Chapter, the issues surrounding CDR-coding the database and other placement strategies are explored. The next section contains a discussion of the theoretical benefits and drawbacks of particular placement strategies using equations to generate expected access costs. Section 3 will report the results of an experiment performed on PHRAN and PP. How the data itself effects clustering is explored in Section 4. Problems that must be addressed order for a traditional DBMS to provide these placement strategies are covered in Section 5. Section 6 contains a summary of the results.

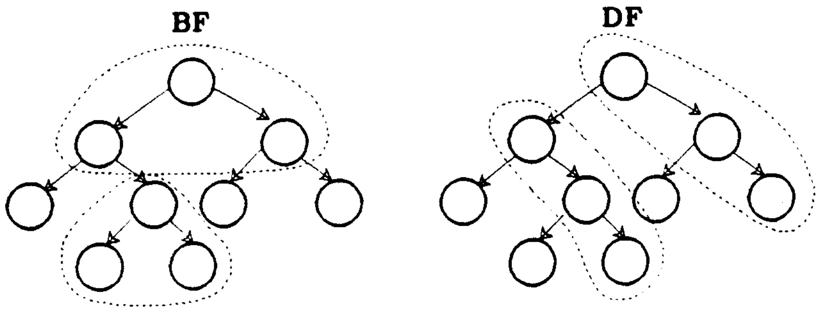


Figure 5.5: Breadth-First versus Depth-First Placement

5.2. Analytical Clustering Benefits

5.2.1. Placement versus Access Strategies

The introduction mentioned a depth-first placement strategy called CDR-coding. Placement strategies other than CDR-coding may be desirable in a persistent environment. To simplify the analysis, two placement strategies will be compared. Figure 5.5 depicts the two opposing strategies. If a breadth-first (BF) strategy is employed, the circled objects in the left figure will be collocated; whereas, the circled objects on the right would be collocated if depth-first (DF) placement is employed. Using a BF technique, each node is placed near it's parent. A DF technique places nodes together similar to CDR-coding.

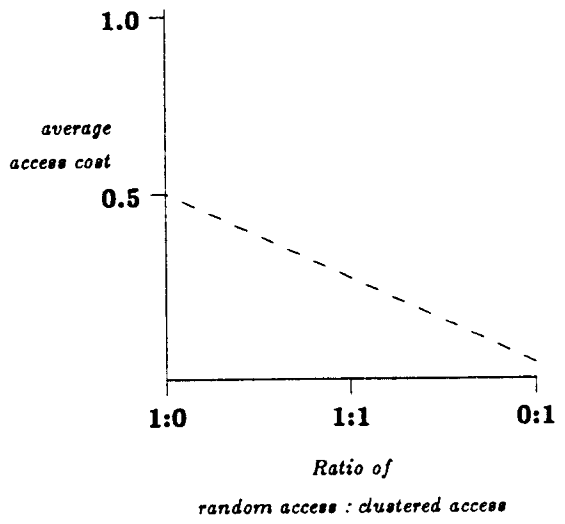


Figure 5.6: Random versus perfect match access

When a placement strategy and an access pattern match perfectly, performance will be optimal. When choosing a placement strategy, the access cost for various access patterns should be considered since no placement strategy will be perfect. Figure 5.6 shows that when the accesses are random (i.e., following either of two pointers is equally likely) the average access cost is 0.5. No matter what the placement strategy, whether BF or DF, if the access pattern is random, half of the references will be to objects not on the page;[3] hence, the expected average cost is 0.5. As the accesses begin to match the placement strategy, the cost decreases uniformly until the placement strategy and the access pattern are exactly match. When they are exactly the same, the cost is 1/NP (0.05 for NP=20), since as soon as an object is referenced all other objects on the same page will also be referenced. If the ratio of random accesses to clustered accesses is equal, the average access cost is 0.275.

Not all objects that may be stored in a database have only two fields with pointer values. Figure 5.7 shows how the cost increases as the breadth increases when each pointer is equally likely to be followed. When the breadth is 1, the cost is 1/NP (for NP=20, it is 0.05).

[3] There are NP objects on a page, each with 2 pointers; hence, there are NP*2 pointers. Since there are only NP objects on the page, only a half of the references can be to objects on the page.

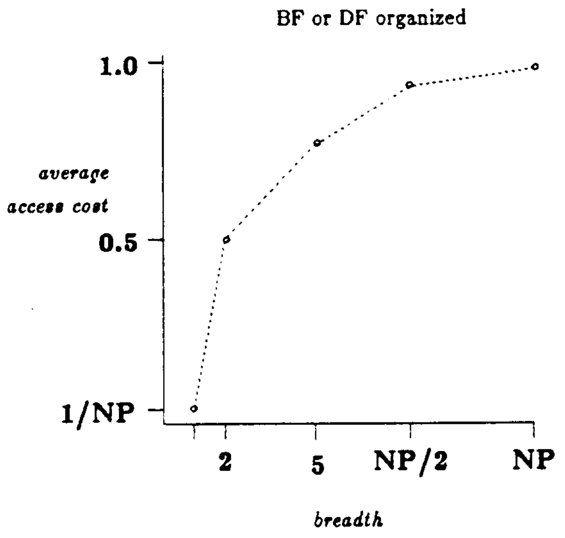


Figure 5.7: Random access cost as breadth increases

These costs assume that 20 objects fit on a page regardless of the breadth. The cost increases rapidly and approaches an average cost of 1: at a breadth of 5, the cost is 0.8; at 10, the cost is 0.9; at NP (20), the cost is 0.95. Random access cost of both BF and DF degrade equally as the breadth increases.

The probability that a particular field will be followed can often be determined in advance. For example in Lisp, the ratio of CAR operations to CDR operations executed in a particular application can be easily determined by measurement. This allows one to determine the probability of following a right pointer versus a left pointer in a Cons cell. Figure 5.8 shows how the BF and DF placement strategies vary in average cost as the ratio of CAR to CDR operations changes. The BF placement strategy is unaffected by the ratio since an equal number of left and right pointers refer to objects off the page. In contrast, the DF strategy causes an average access cost of 1 if only CARs are performed and an access strategy of 1/NP if only CDRs are performed.

When tuples are placed on pages, the strategy may allow for variance on the ideal to accommodate otherwise unqualifying tuples. That is, if a DF-right strategy is employed and a

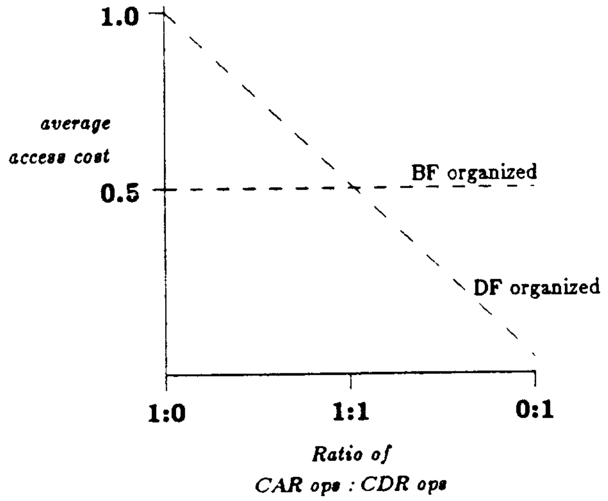


Figure 5.8: Average cost as ratio of pointer followed varies

tuple is added to the database that has only a left pointer value, the tuple should be placed on the page with the object it references to the left if that page can accommodate the tuple. Not only will access pattern not be perfect, but the actual placement may be imperfect. Figure 5.9 shows that BF and DF access costs vary complementarily as the placement strategy varies between the two.

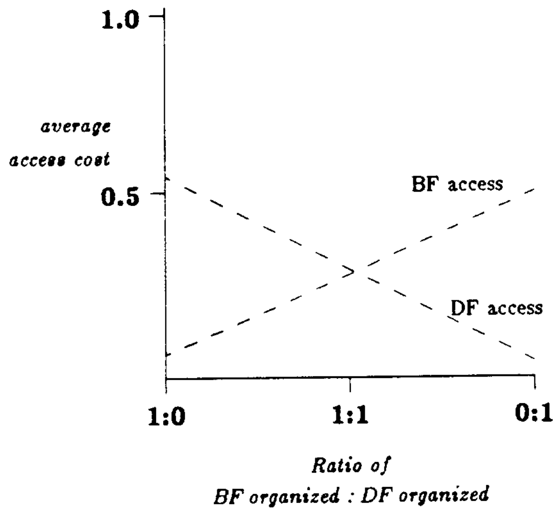


Figure 5.9: Access cost as placement varies

5.2.2. Object deletion and clustering

Although the database allocation mechanism may place tuples in accordance with a clustering scheme, the clustering may gradually diminish as tuples are deleted. As clustered objects are deleted from pages, internal fragmentation of the cluster will occur. This cluster fragmentation causes the average cost of accessing objects to increase. For example, if NP/2 objects are deleted from each page on which NP objects were clustered, two pages must be read to access NP objects.

Figure 5.10 depicts how cluster fragmentation degrades the cost of accessing objects over time. As objects are accessed, some of these accesses will be deletions. The three different lines in Figure 5.10 represent the expected average access cost for delete ratios of 1/10, 1/100, and 1/1000 (from left to right); that is, on average 1 object is deleted every 1, 10, or 100 accesses. The expected average cost of accessing an object when NP objects are clustered on each page is 1/NP (0.05), which indicates that only 1 page is read to access NP objects. As objects are deleted, fewer clustered objects can be accessed from a single page read, therefore the expected cost of accessing objects approaches 1. A cost of 1 indicates so many objects have been deleted that no objects are clustered on pages, therefore each object access causes 1 page read. The greater the number of objects deleted (i.e., the higher the delete ratio), the quicker the expected cost of accessing an object reaches 1.

5.2.3. Summary of Analysis

In most of the comparisons between breadth-first and depth-first placement, the two strategies perform similarly. Varying the ratio of CAR to CDR operations is the only analysis in which the two strategies demonstrate their potential performance differences: breadth-first maintains an average cost of 0.5 regardless of the ratio and depth-first varies from 1 down to 0.05 as the number of CDRs increase. Hence, breadth-first allows a more stable, predictable average cost; whereas, depth-first placement may result in a very low or a very high average

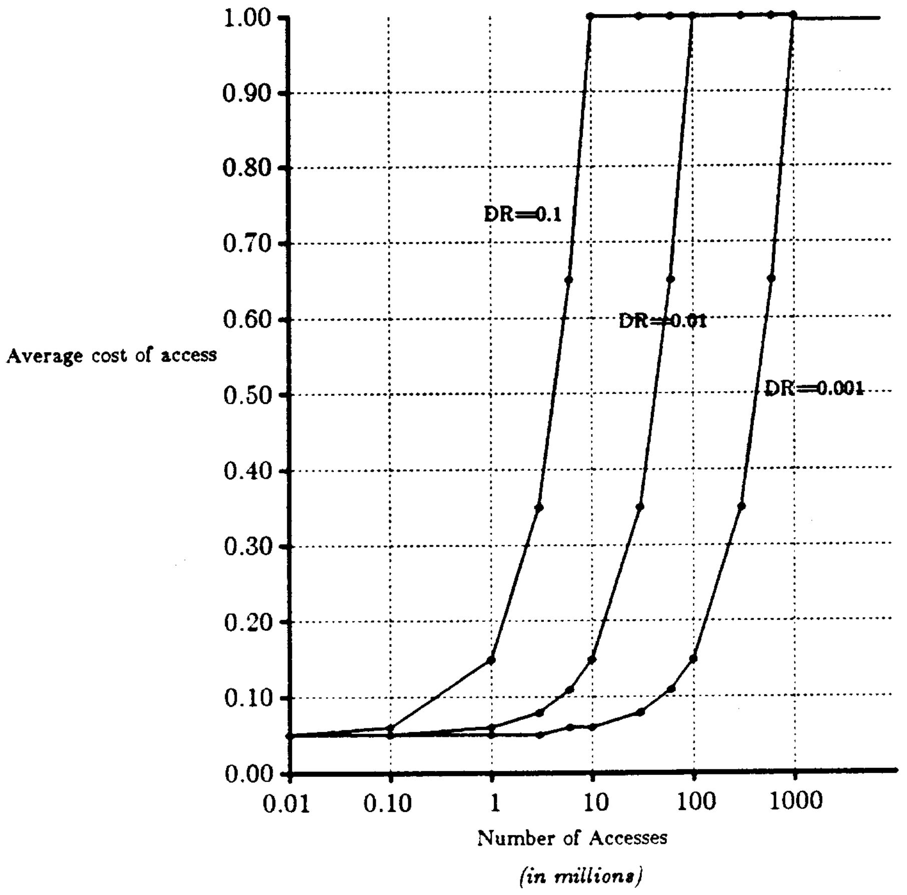


Figure 5.10: Access cost as objects become unclustered

cost depending upon an application's access characteristics.

To retain clustering, the effects of deletes must be overcome. One way of alleviating this problem is to recluster as part of a storage reclamation process. This will be discussed in Chapter 6.

5.3. Measured Performance of Clustering

The database for PHRAN and that for PP are two very different data structures. PHRAN contains over 37,000 tuples that are connected in a tree structure; whereas PP contains less than 5000 tuples that form a tightly connected network. Testing the effectiveness of clustering on these two distinct databases should produce results that can be extended to other data structures.

The DBMS does not support the kind of clustering needed. Therefore, a new relation was created in which 2 tuples from the Cons Relation would be stored together. Polymnia software enforced the kind of clustering that was desired. Figure 5.11 depicts the DoubleCons Relation. The technique for storing tuples together worked as follows: starting at the root of the structure stored in Cons Relation, copy the current tuple into the DoubleCons Relation; copy the tuple referenced by the *next* field into the second half of the same tuple. When two objects are stored in the same tuple, the first one's *next* value refers to the second object. Hence, this can be made implicit; that is, the first object's *next* field need not be stored since it will always refer to the second object. Of course not all objects contain next values that are pointers; therefore, sometimes the first object will be the only object in the DoubleCons Relation. The tag field indicates how many objects are stored in the tuple.

DoubleCons Relation

id	*first1*			*first2 or next1*			*next2*			**tag**
	type	**value**	**symbol**	**type**	**value**	**symbol**	**type**	**value**	**symbol**	

Figure 5.11: The DoubleCons Relation

Cons Relation

id	*first*			*next*		
	type	value	symbol	type	value	symbol
0	symbol	1	a	cons	1	
1	symbol	2	b	cons	2	
2	cons	3		symbol	0	nil
3	symbol	3	c	cons	4	
4	symbol	4	d	symbol	0	nil

DoubleCons Relation

id	*first1*			*first2 or next1*			*next2*			**tag**
	type	value	symbol	type	value	symbol	type	value	symbol	
0	symbol	1	a	symbol	2	b	cons	1		2
1	cons	2		symbol	0	nil				1
2	symbol	3	c	symbol	4	d	symbol	0	nil	2

Figure 5.12: Cons Relation and DoubleCons Relation storing (a b (c d))

Figure 5.12 demonstrates how the list (a b (c d)) is stored in both the Cons Relation and the DoubleCons Relation. This same technique for compacting and clustering objects was used to store the data in the QuadCons Relation, in which up to four objects are stored, and the OctCons Relation in which up to eight objects are stored.

A factor in determining whether clustering will benefit performance is whether compaction occurs. Not every list or sublist will divide perfectly into sets of 2, 4, or 8. In each relation, there will be tuples that are only partially filled. For example, in the DoubleCons Relation in Figure 5.12, the tuple with id=1 contains only 1 cons cell. If null-valued fields take up as many bytes as fields with non-null values, then partially filled tuples will use up many empty bytes. To demonstrate this, the tests on PHRAN and PP were conducted using two different database storage techniques: PHRAN data were stored uncompressed; whereas, PP data were stored compressed. Due to limited resources, PP and PHRAN could not be tested with both compressed and uncompressed data. Although testing the benefits of clustering on an uncompressed database may seem to have no purpose since it will clearly perform poorer than a compressed one, the test was undertaken to prove the importance of compression. Since not all commercial DBMS support compression, demonstrating how crucial it is for clustering to benefit performance is a worthy task.

Relation:	**Number of Objects Per Tuple**								**Avg Objects Per Tuple**
	1	2	3	4	5	6	7	8	
Cons	37,280	-	-	-	-	-	-	-	1
DoubleCons	4634	16,323	-	-	-	-	-	-	1.8
QuadCons	1922	9121	2712	2640	-	-	-	-	2.3
OctCons	2626	8777	1917	249	86	344	5	978	2.8

Figure 5.13: Tuple counts for PHRAN's Clustered Relations

5.3.1. Clustered PHRAN

Table 5.13 shows the breakdown of how many tuples contain each possible number of cons cells for each relation. DoubleCons is the only relation in which most of its tuples contain the maximum number of cons cells (78% hold two cons cells). As can be seen, the OctCons has only 978 tuples with 8 objects in them (7%); the QuadCons has 2640 full tuples (16%). In fact, for each relation the majority of tuples contain two cons cells: in QuadCons 56% of the tuples hold 2 cons cells; in OctCons, 67%.

Ideally, the data can be compacted onto fewer pages when clustering since the some of the *next* fields can be made implicit. Figure 5.14 depicts for each relation the number of tuples, the number of bytes per tuple, and the total number of bytes actually needed to store the PHRAN database. Although the OctCons Relation should allow the most compaction, the OctCons Relation actually uses 43% more bytes than the Cons Relation to store the PHRAN data. (The number in parentheses is the number of bytes that would have been used if maximum compression occurred; OctCons potentially would have used half the number of bytes of the Cons Relation.) The OctCons Relation does not realize any data compression because, the storage structure utilized for these tests allows a null value takes up as much space as an actual value. Therefore, an OctCons tuple containing 2 cons cells uses as many bytes as one with 8 objects. As Figure 5.13 illustrates, the average number of cons cells per tuple is only 2.8 in the OctCons Relation. 1.8 megabytes of data space is left empty in the OctCons Relation. Only the DoubleCons Relation realizes any data compression; it occupies

Relation:	tuples	bytes per tuple	total Mbytes
Cons	37,280	58	2.1 (2.1)
DoubleCons	20,957	84	1.7 (1.5)
QuadCons	16,395	136	2.1 (1.3)
OctCons	13,182	240	3 (1.1)

Figure 5.14: Byte counts for clustered PHRAN (no compression)

25% fewer pages than the Cons Relation.

The performance of PHRAN when operating with each different relation follows in a fairly predictable manner from the byte data. Figure 5.15 shows that when PHRAN operates using the OctCons Relation, it takes more time to run than when using the Cons Relation as might be predicted from OctCons using more space. The database portion of Polymnia uses 218 seconds during the PHRAN-Cons test (DB-time); during the Phran-DoubleCons test, 201 seconds are used. The 25% compaction gained when using the DoubleCons results in a 8% performance improvement in the time it takes to fetch the tuples from the DBMS. Although the database did not realize any compaction in the QuadCons Relation, the PHRAN-QuadCons test shows a 6% improvement. This is probably the result of executing fewer queries; 40% fewer queries are executed.

Since the tuples in the DoubleCons, QuadCons, and OctCons Relations contain more than one cons cell, fewer tuples may be fetched. Since only one tuple is retrieved for each query, fewer tuples means fewer queries are executed. If fewer queries are executed, total retrieval time may decrease. Figure 5.15 depicts the correlation between DB-time, number of tuples fetched (queries executed), number of bytes fetched, and the number of tuples retrieved per second. The number of queries and the number of bytes read appear to interact, both effecting the amount of time it takes to fetch the data. PHRAN-DoubleCons runs fewer queries than PHRAN-Cons (72% of PHRAN-Cons) and needs fewer bytes (4% less) which results in a lower cost. PHRAN-QuadCons runs many fewer queries (57% of PHRAN-Cons),

Relation:	**DB-time**	**tuples fetched**	**Kbytes read**	**Tuples/second**
Cons	218	3913	222	18
DoubleCons	201	2806	230	14
QuadCons	204	2243	298	11
OctCons	235	2112	495	9

Figure 5.15: Performance of Clustered Phran

but requires 30% more bytes to be read. Clearly, the savings in running fewer queries results in better performance. PHRAN-OctCons requires 54% of the tuples of PHRAN-Cons, but 116% more bytes to be read. Hence, the savings in fewer queries is overridden by the number of bytes to be read.

Figure 5.15 also shows the tuple retrieval rate. Simply by adding the complexity to the Equel/C interface to handle double-sized tuples, the retrieval rate falls more than 20%. The number of tuples retrieved per second steadily decreases with each clustering due to the increased complexity of Equel/C interface. The interface becomes slightly more cumbersome as the number of fields in a relation increases. Although the retrieval rate may decrease, clustering may still result in savings due to fewer tuples being retrieved.

5.3.2. Clustered PP

Although the data structure for PP is quite distinct from that of PHRAN, the average number of objects per tuples when the structure is clustered is fairly close to that of PHRAN's data (see Figure 5.16). The biggest difference is that the PP-OctCons results in 2.3 objects per tuple; whereas, PHRAN-OctCons has 2.8. This difference is a result of PP having a tightly interconnected structure; it has no long strings of cells connected via *next* pointers as does PHRAN.

For PP's data structure, the DoubleCons Relation is 82% full (only a slight bit fuller than for PHRAN). Just as with the PHRAN data structure, a small percentage of the tuples

Relation:	Number of Objects Per Tuple								Avg Objects Per Tuple
	1	2	3	4	5	6	7	8	
Cons	4987	-	-	-	-	-	-	-	1
DoubleCons	491	2248	-	-	-	-	-	-	1.8
QuadCons	289	1554	202	246	-	-	-	-	2.2
OctCons	269	1529	184	101	20	25	18	41	2.3

Figure 5.16: Tuple counts for Clustered Relations

Relation:	tuples	bytes per tuple	total Kbytes
Cons	4987	58	282
DoubleCons	2739	84	214
QuadCons	2291	136	195
OctCons	2187	240	244

Figure 5.17: Byte counts for clustered PP (compressed)

in the QuadCons and OctCons test are full and the vast majority contain two objects. The QuadCons contains 11% full tuples and 68% with two objects; and, the OctCons contains 2% full and 70% with two objects. Even though the objects may not be maximally compacted in tuples, the storage structure in the database for this test suppresses null fields. Hence, greater compaction should be realized by PP's data than PHRAN's. Figure 5.17 contains the figures for total number of bytes used. All test result in compaction, although the OctCons realizes the least.

Figure 5.18 depicts the performance realized when using the various clustered relations. The greater the clustering allowed, the better the performance. The tuples per second decreased from 14 to 11 with the added complexity of retrieving clustered tuples, but unlike PHRAN remains at 11 for all the cluster tests. This steady retrieval rate allows the decrease in tuples for each clustered relation to result in a decrease in retrieval time. This is in contrast to PHRAN in which, although the number of tuples retrieved decreased, the retrieval time increased due to the decreasing retrieval rate.

Relation:	DB-time	tuples fetched	Kbytes read	Tuples/second
Cons	357	4994	282	14
DoubleCons	262	2746	214	11
QuadCons	219	2298	195	11
OctCons	197	2194	244	11

Figure 5.18: Performance of Clustered PP

5.3.3. Generalized Results

Although PHRAN and PP have very distinct data structures, they displayed very similar clustering characteristics. For both data sets, the clustered relations were dominated by tuples that contained two objects. In addition, in only the DoubleCons test were the majority of tuples full; a minor percentage of tuples were contained the maximum number of objects for the QuadCons and OctCons tests. Despite the small amount of clustering, all tests except the PHRAN-OctCons test showed improvement upon the unclustered performances.

The PHRAN-OctCons test resulted in a performance loss due to the increased number of bytes that had to be processed by the system. For the PHRAN tests, uncompacted relations were used; hence a partially filled OctCons tuple occupied the same number of bytes as a full one. More than twice as many bytes had to be read for the PHRAN-OctCons test than for the PHRAN-Cons test. Although the average number of objects per tuples was greater for PHRAN-OctCons than for PP-OctCons, PP-OctCons realized relatively better performance. This is a direct result of PP test using compacted relations. Fewer tuples and fewer bytes were retrieved for PP-OctCons.

From this study, one can conclude that the number of queries needed to fetch the data has a great an impact on performance, but so does the number of bytes that must be read. The next section explores more fully how the characteristics of the data effect the compaction realized when attempting to cluster the data.

5.4. Data Characteristics and Compaction

Clustering PHRAN's data did not result in much compaction. The data characteristics and structure that will result in compaction when CDR-coded are explored in this section. To assist in understanding the problems, a few examples will be analyzed first. Figure 5.19 illustrates a few different structures which will be discussed.

The average number of objects per tuple is a good indicator of how much compaction has occurred. To determine the average objects per tuple, objects strung together via next pointers are counted. These are the objects stored together when the structure is CDR-coded; therefore these strings of objects will be called CDR-strings. Figure 5.19 depicts three different structures which will be referred to (from left to right) as 5.19A, 5.19B, and 5.19C. The data of 5.19A contains 8 CDR-strings each containing 8 objects. If the objects can be clustered eight to a tuple, this data will fill 8 tuples with 8 objects each. Hence, the average number of objects per tuple is 8. The structure from 5.19B has CDR-strings of size 4, 3, 2, and 1. If CDR-coded in the database, it will result in 1 tuple with 4 objects, 1 tuple with 3 objects, 2 tuples with 2 objects, and 4 tuples with 1 object apiece which is an average of 1.875 objects per tuple. Lastly, the data in 5.19C contains 1 CDR-string of length 8 and 7 of length 1. When stored in the database, this results in an average of 1.875 objects per tuple also.

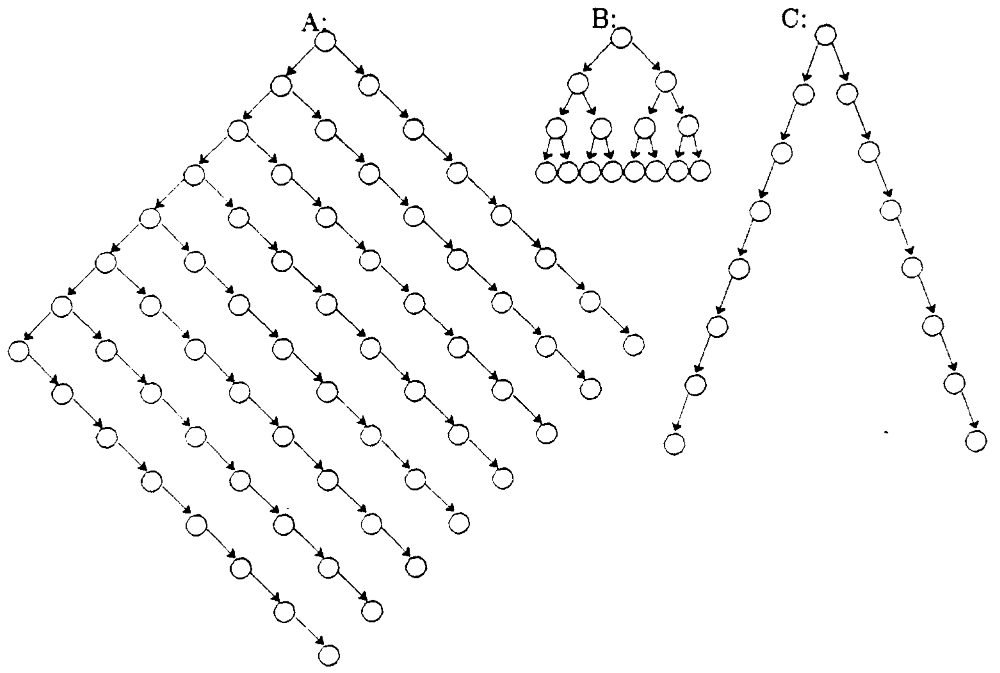


Figure 5.19: Sample Structures

The average CDR-string (ACS) correlates with the average objects per tuple (AOT). If the average CDR-string is less than or equal to the maximum number of objects (MAX) that can be stored in a tuple, then the average CDR-string equals the average objects per tuple. When the average CDR-string is greater than the maximum tuples per object, the average objects per tuple can be calculated as $\frac{ACS}{n+1}$ where $ACS = (MAX * n) + r$ and $0 \leq r < MAX$.

ACS can be approximated given the breakdown of first (F) and next (N) pointers. The expected ACS can be calculated as $\frac{F + N}{F}$. If the first and next pointers are evenly distributed, this gives a fairly accurate estimate of ACS. One indication of an even distribution is that the maximum CDR-string multiplied by the maximum CAR-string is approximately equal to the total number of objects.

The results of using these equations to analyze the examples in Figure 5.19 are shown in Figure 5.20. For all of the examples, the expected ACS is the same as the actual ACS and AOT; therefore, it is a fairly accurate indicator for the average objects per tuple. This holds true even for 5.19C which does not have an even distribution.

Using these equations, the ratio of next pointers to first pointers can be analyzed for its effect on the clustering of a structure. Figure 5.21 shows the average objects per tuple as the ratio increases (MAX = 8 for this graph). As the ratio of next pointers to first pointers increases, the average CDR-string linearly increases. As the average CDR-string increases, the average objects per tuple increases to MAX, then fluctuates geometrically each time

Figure	*N*	*F*	*estimated ACS*	*AOT*
A	56	8	8	8
B	7	8	1.875	1.875
C	7	8	1.875	1.875

Figure 5.20: Estimated objects per tuple versus actual

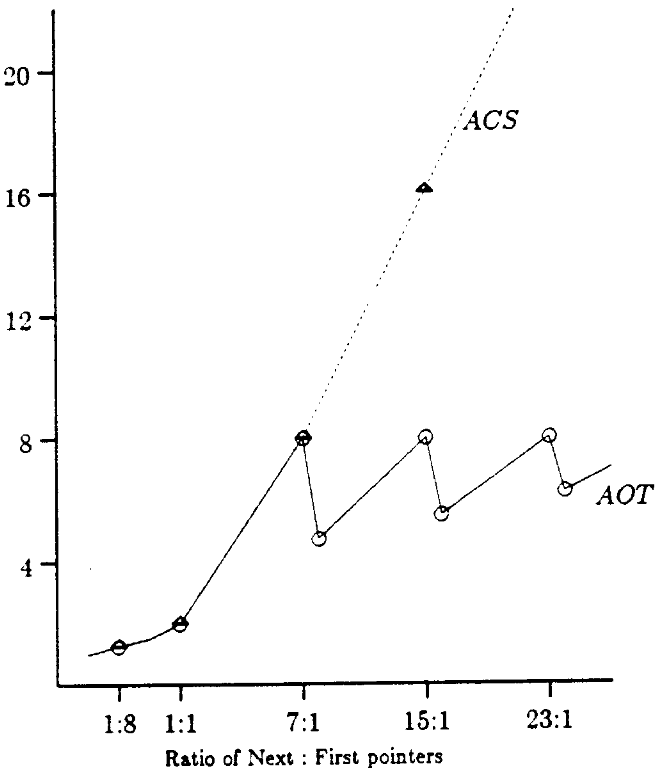


Figure 5.21: Expected correlation between next:first ratio and AOT

reaching MAX and dipping down only to rise again. Each dip in the curve is half the previous dip, therefore it gradually converges on MAX.

Looking at the bytes per tuple shown in Figure 5.14, an OctCons tuples occupies 240 bytes which is 30 bytes per object. That is about half the bytes per object for the Cons Relation (58 bytes). Therefore, for MAX=8, if an average of $\frac{MAX}{2}$ objects are stored per tuple, the actual number of bytes per object is approximately equal to that of unclustered data (i.e., 16). If the average is greater than $\frac{MAX}{2}$, the data takes less space than when unclustered. Hence, actual compaction is realized when the average CDR-string is greater than $\frac{MAX}{2}$.

In summary, the expected average CDR-string appears to be closely correlated with the average objects per tuple; and, the average objects per tuple is directly correlated with the amount of compaction realized when CDR-coding the database; hence, the expected average CDR-string will indicate whether or not data will be compacted. As the ratio of next pointers to first pointers increases, the expected average CDR-string length also increases. When the ratio is greater than $\frac{MAX}{2}$ the expected CDR-string length will also be greater than $\frac{MAX}{2}$ and will eventually converge on MAX. Therefore, if the ratio of next to first pointers is greater than $\frac{MAX}{2}$ the structure is expected to realize compaction when CDR-coded.

5.5. Problems clustering in traditional DBMS

If a DBMS provides clustering for the type of data that Polymnia handles, two problems must be addressed: union-fields and ambiguous cluster requests. Examples of each of these problems are discussed below.

5.5.1. Clustering on union-fields

The breadth-first and depth-first clustering described in the previous section is different than traditional database clustering: instead of collocating tuples based on similar values in the same field, tuples are collocated based on similar values in different fields. For example, in Figure 5.2 the EMP tuples are collocated based on the ename field having the value "TOY"; whereas, in Figure 5.3 if next-type is "cons" and next-value equals another tuple's id, the two Cons tuples are collocated. If next-type is a structure type (e.g. "edge" as described in Chapter 3), the tuple from the corresponding structure relation whose id equals the value of next-value should be collocated. Hence, clustering on a particular field's value may warrant more than one relation be collocated.

The syntax of the cluster statement could be something like:

cluster c by (r=c.next-type) where c.next-value=r.id.

This statement allows for *r* to be any type of tuple. In addition, this statement allows for the type of *r* to change dynamically as the value of *c.next* changes. For example, a cons cell may change from refering to another cons cell to refering to a structure. The placement mechanism must allow the clustered tuples to dynamically change value and type.

5.5.2. Resolving ambiguities

Another problem that may occur is illustrated by the following statement:

cluster c by r where c.next_value=r.id and c.next_type=cons.

If a Cons tuple is added to the database that has id=2, next-type="cons", and next-value=4; should the tuple be collocated with the Cons tuple with id=4 or the one whose next-type="cons" and next-value=2? Figure 5.22 illustrates the problem. Ideally, all three

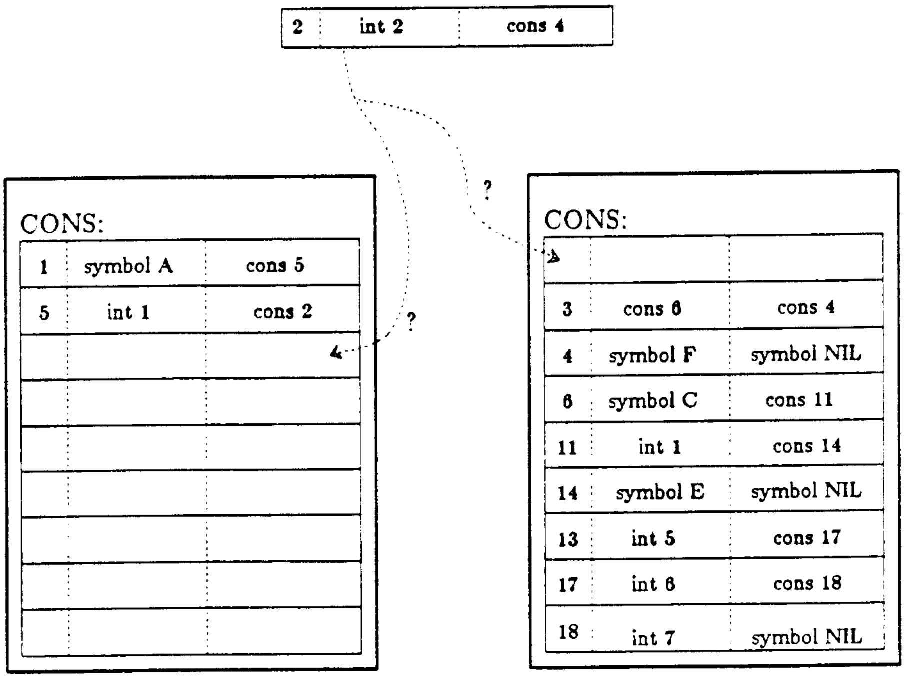


Figure 5.22: Ambiguity resulting from cluster request

tuples should be collocated, but this will not always be possible. The cluster statement must allow precedence to be specified. For example, the above cluster statement could be interpreted with the following precedence: *When adding a new tuple (c) to the database, place it on the same page as the tuple in relation r where c.next-type=r and c.next-value is the r tuple's id; if not possible, place the new tuple (r) on the same page as the tuple c where c.next-type indicates the relation to which r belongs and c.next-value equals r.id.* This interpretation indicates that the new tuple in Figure 5.22 should be collocated with the tuple with id=4.

5.6. Conclusion

Strategies for efficient placement of nontraditional data are just beginning to be explored in the database community. This chapter has reported on some initial comparisons of two possible placement strategies for data: breadth-first clustering and depth-first clustering. These two techniques are those commonly used in virtual memory allocation schemes for programming language data. Depth-first clustering gives very good performance if most of the accesses follow the clustered field; but if they don't, it gives poor performance. Breadth-first is a more stable technique when access patterns cannot be easily predicted. Either of these methods outperforms random placement, when access cost is unpredictable at all times and is likely to be poor.

Clustering needs to be an active process. Since the data may continuously change, the DBMS must accommodate the changes in ways that retain as much clustering as possible. A possible way of attaining this is discussed in the next chapter.

After analyzing the actual performance of two applications (PHRAN and PP) which access data that is clustered, one can conclude that two factors effect the cost: the number of bytes of data that must be read, and the number of queries that must be executed to retrieve the data. The initial reason for suspecting that efficient placement will lessen cost was based purely on lessening the number of queries executed to retrieve the data. In general, reducing the number of queries will result in lower cost. But, care must be taken so that by providing clustering the database does not inadvertently occupy more space. Data will not be highly clustered when the placement strategy and the data do not match well. For example, the PHRAN-OctCons used more space because cons cells that did not come in strings of eight took up as much space as those that did. Providing variable length tuples and null values that do not occupy much space alleviates this problem as is witnessed by PP-OctCons which has a lower average objects per tuple than PHRAN-OctCons, but realizes greater compaction. Even still, extra complexity is added by the Equel/C interface to handle the clustered tuples. Hence, the benefits of clustering must outweigh the added expense.

The ratio of next pointers to first pointers appears to be an excellent indicator of the amount of compaction that may be realized if the data is CDR-coded. By doing a small amount of analysis on one's data, one may determine whether clustering will lead to compaction.

Efficient placement has been shown analytically to be a possible benefit to performance, and, by actual tests, has been proven to be able to lessen the cost of a persistent system. If null values take no space, much greater compaction is realized; hence, clustering is an even greater benefit.

CHAPTER 6

Storage Reclamation

Sarah Cynthia Syliva Stout
Would not take the garbage out!...
At last the garbage piled so high
That finally it touched the sky...
Poor Sarah met an awful fate,
That I cannot now relate...
But children, remember Sarah Stout
And always take the garbage out!

Shel Silverstein, *Where the Sidewalk Ends*

6.1. Introduction

Some of the problems that occur when pointers are stored in a database were addressed in Chapters 3 and 4. The problems addressed so far have dealt with the storage of the data and the performance of applications that access the data. When applications delete or update data, an additional problem occurs: data may appear in the database that cannot be accessed by the application. This situation occurs in programming languages that support complex object structures, such as CommonLoops [Bobr86] and SmallTalk [Gold83]. To *collect* this unaccessible data, these programming environments provide automatic storage reclaimers, or garbage collectors.

The problem of storage reclamation is slightly more complex when some of the data is stored in a DBMS since actions taken by an application may affect data not accessed by that application. Figure 6.1 depicts a sample scenario. A set of interconnected objects (represented by shapes) are stored in a database on the disk. In this example environment, the *Data*Manager retrieves the objects for the *Application*. If the Application deletes the *CIRCLE*, then the *CIRCLE* is no longer accessible and can be deleted from the database. The *STAR* which is contained in the *CIRCLE* is still part of the picture because it is in the *FOOTBALL* but the *FIGURE-8* is not part of the picture any more. Ascertaining that the *FIGURE-8* should be deleted but the *STAR* should not is the process of storage reclamation or garbage collection.

Storage can be reclaimed either by an automatic method or by issuing explicit delete instructions. Current data managers delete items only when instructed to do so. As the

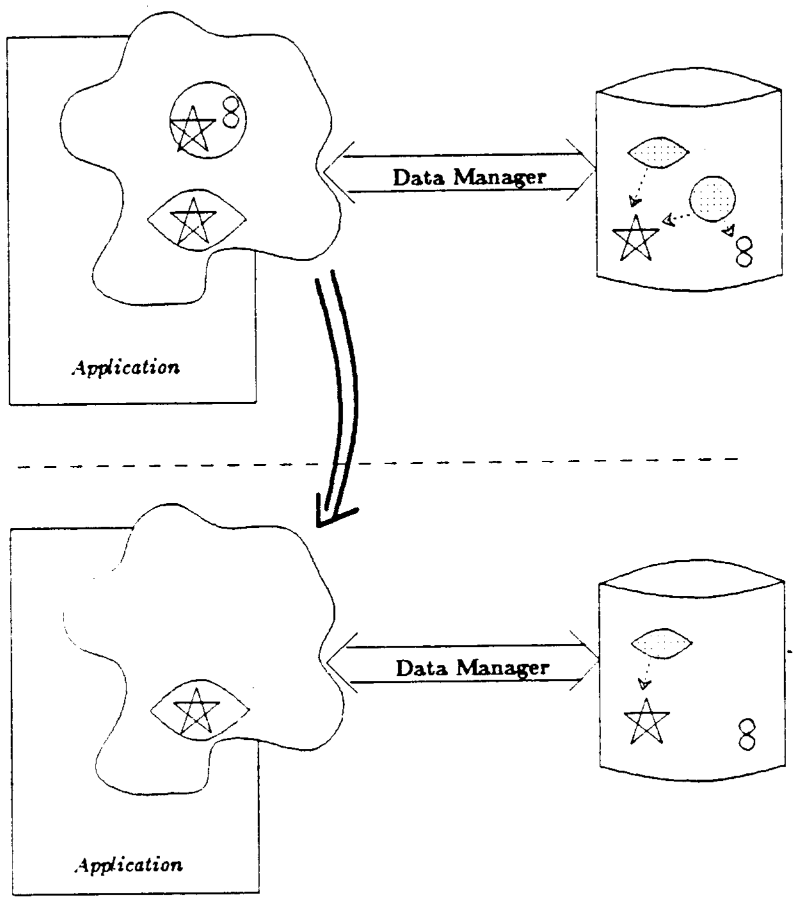


Figure 6.1: Sample Object Scenario

structures in a database grow larger and more complex, determining what objects are no longer reachable becomes exceedingly difficult to do manually. Hence, garbage collection should be an integral part of future object-oriented data managers and persistent systems[1].

All published automatic storage reclamation algorithms have been designed for main memory or virtual memory. By analyzing these algorithms as they apply to data stored in database systems, features can be identified that have prohibitive performance costs or that are beneficial to performance. The best algorithm for virtual memory may not be the best one for a database because DBMS have different performance characteristics than virtual memory systems. For example, data stored in a database tends to have a long lifetime; whereas, data in virtual memory is often temporary. Hence, garbage should be generated at a slower pace in a database system than in virtual memory. In addition, retrieving objects from a database will often cause more disk pages to be read than reading an object in virtual memory.

The focus of this chapter is an analysis of the disk input/output (I/O) behavior of existing automatic storage reclamation schemes and the amount of space they require to execute when operating on data stored in a Database Management System (DBMS). Disk I/O can often be a bottleneck, so a study of the I/O characteristics may uncover which algorithms should be avoided because of the I/O demands. Section 2 contains the model used for algorithm evaluation, the parameters varied and the values taken by them. Descriptions of the algorithms considered and their database encoding comprise Section 3. Section 4 includes the results of I/O analyses of the applicable algorithms. The space requirements of the algorithms are covered in Section 5. Section 6 concludes the study by considering how the process of storage reclamation may interact with applications. Appendix A contains all of the

[1] PO [Mish84], GemStone [Cope84], and POMS [Cock84] already incorporate automatic storage reclamation schemes into their systems.

parameterized equations used for the analysis with a brief description.

6.2. The Model for I/O Cost Evaluation

The objects used in this study are Lisp cons cells assumed to be stored in a relational DBMS as described in Chapter 2. To facilitate the algorithm descriptions, a simplified version of the database schema will be used (see Figure 6.2) and is breifly described below. The ConsRelation stores list cells individually; each representing one object. Since the right and left pointers can refer to different types, a tag field (*type*) is used to indicate the sort of object pointed to. If *type* = "leaf", the pointer refers to a leaf.[2] If *type* = "pointer", the *value* field indicates the id of the cell in the ConsRelation that is pointed to. The SymbolRelation contains the persistent Lisp symbols, or atoms, from a Lisp application. A tuple in the ConsRelation is *reachable* only if it is referenced by a symbol or by an object reachable from a symbol.

The analyzed algorithms are compared on their expected amount of disk I/O. Each algorithm has been coded in a query language that resembles Quel* [Ston83]. The paging behavior of each implementation is modeled by an equation which calculates the expected number of pages that will be read and written during the running of the corresponding reclamation scheme. To capture the I/O cost with a single number, each page read is assumed to

ConsRelation				
id	*left*		*right*	
	ltype	lvalue	rtype	rvalue

SymbolRelation			
id	name	type	value

Figure 6.2: Database Schema for Lists

[2] A leaf is any value that is stored inside a tuple (e.g. an integer or symbol), hence no additional retrievals are necessary.

have unit cost and each page write, a cost of 2.[3] Each equation includes the probability that a page has been previously read and given that it has been read, the probability that it is still buffered in main memory. Therefore, the equations express the expected amount of physical I/O.

The modeled parameters and the associated value set for each is shown in Table 6.1 and described below.

- The number of objects, **N**, represents the number of tuples in the ConsRelation. This indicates the size of the data structure. The number of objects in this study ranged from 5500 (a small database) to 100 million (a very large database).
- The number of roots, **R**, specifies the number of tuples in the SymbolRelation. To isolate how the data structure parameters effected performance, the number of roots was set to 1. More roots would simply make the overall structure denser; the effect that this

Parameter	**Values**	**Notes**
N	{5500 ... 100,000,000}	Number of objects
R	1	Number of objects in the root set
NP	20	Number of objects per page
BP	{3 ... 1000}	Number of buffer pages
A	{1 3 10 30} * N	Number of Accesses (includes reads, updates, & deletes)
UR	{.1 .01 .001 0}	Update Ratio
UDRatio	{1 4 100}	Update to Delete Ratio
DR	UR/UDRatio {.1 ... .00001 0}	Delete Ratio
IR	DR	Insert Ratio
Height	{ $\log_2 N$...N/5}	Longest acyclic root-leaf path
RLRatio	{1 ... 5}	Ratio of Right pointers to Left pointers
Density	{1 1.3 2}	

Table 6.1: Parameters modeled and their values

[3] The exact cost of a write may be more or less than twice that of a read. This assumption affects the absolute performance of each algorithm but does not effect the relative performances because each algorithm does approximately the same number of reads as writes.

would have on performance is captured by the parameter *density*.

- The number of objects per page, **NP**, represents the number of tuples in the ConsRelation that fit on a single disk page. The tuples in the ConsRelation for Polymnia utilize approximately 50 bytes so 20 tuples fit on a 1024 byte page. In addition, studies of SmallTalk showed that the average object size was 50 bytes [Gold83]. The objects per page is set to 20 for this study.
- The number of buffer pages, **BP**, signifies the number of main memory pages that the DBMS uses for buffering disk pages between queries. The number of buffer pages was varied from 3 to 1000 and only had an impact on performance for certain data characteristics as will be described in Section 4.2 of this chapter.
- The number of accesses, **A**, expresses the number of object reads, writes, creations, and deletions that occur in the database during the analyzed period. It's value was computed relative to the number of objects in the database and ranged from **N** up to 30 times **N**.
- The update ratio, **UR**, represents the probability that an access alters an object; that is, the expected number of updates divided by the number of accesses. The update ratio ranges from 1/10 down to 0. An UR of 0 indicates a read-only database.
- The **UDRatio** expresses the expected number of objects that become unreachable after an update; that is, the expected number of unreachable objects divided by the number of updates. The UDRatio ranges from 1/100 (only 1 in every 100 updates causes an object to become unreachable) to 1 (each update causes 1 object to become unreachable).
- The delete ratio, **DR**, is the expected number of objects that will become unreachable, and therefore deletable, per access. It is calculated by multiplying the UDRatio and the update ratio.

- The insert ratio, **IR**, is the expected number of new objects divided by the number of accesses. For this study, the database is assumed to be in a steady state; that is, the expected number of new objects equals the expected number of deleted objects. The IR value is calculated the same as the delete ratio, which is UDRatio*UR.

The form of the data structure is determined by the values of **Height**, **RLRatio**, and **Density**. The interpretation of these values are discussed below.

- The **Height** of the data structure represents the distance of the longest acyclic path from the root to any leaf. The height of a structure composed of N objects with two pointers each must be at least $\log_2 N$ and may be no greater than N. For the studies reported here, the heights range from $\log_2 N$ up to (approximately) N/5.[4]

- The **RLRatio** signifies the balance within the structure. It is the number of objects that are reachable by following a right pointer divided by the number of objects reachable via a left pointer. The values range from 1 to 5. These values indicate that the number of objects reachable via right pointers is always equal to or greater than the number via left pointers which cause the explored algorithms to display any idiosyncratic behavior adequately.

- The **density** signifies the number of right field or left field values that contain nonleaf pointers divided by the number of objects. The minimum density is 1 since each cell must be referred to by at least 1 other cell to be reachable. Since each object has at most 2 pointers, the maximum density for this study is 2. Figure 6.3 depicts some sample structures that have the densities used in this study.

[4] The height was actually varied by taking log N with different bases. The bases used were 2, 1.1, 1.01, 1.001, 1.0001, 1.00001, and 1.000001. but the smaller values of N did not have a legal value up to $\log_{1.000001}$. Since the Height cannot be greater than the number of objects, the greatest legal value for each N (using the logs) is approximately N/5.

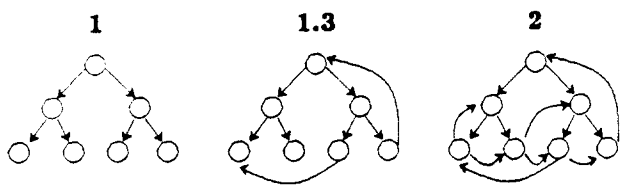


Figure 6.3: Examples of lists with varying density values

As discussed in Chapter 4, I/O performance can be greatly affected by the organization of the objects on pages. To summarize the results from Chapter 4: the cost of accessing NP objects that are perfectly clustered on a single page is 1, whereas the cost of accessing NP objects that are scattered across pages is NP. As clustered objects are deleted from pages, internal fragmentation of the cluster will occur. This cluster fragmentation causes the average cost of accessing objects to increase. For example, if NP/2 objects are deleted from each page on which NP objects were clustered, two pages must be read to access NP objects. Both breadth-first and depth-first clustering are explored in this study.

6.3. Algorithms Explored

Many algorithms have been developed to perform automatic storage reclamation [Cohe81]. The algorithms can be divided into two groups -- those that *scavenge* memory by sweeping over it, and those that perform a small portion of reclamation or garbage collection bookkeeping with each update (i.e., storage is reclaimed *incrementally*). Environments that utilize the first method of automatic storage reclamation operate in two modes: one in which garbage is being generated and one in which the memory is being scavenged to reclaim the storage space used by the garbage. On a single processor, during the generation phase a program is running; during the scavenge phase the program must pause. Incremental algorithms break up scavenging into small pieces so the pauses are shorter but more numerous. In environments that utilize an incremental method, each update takes a little extra time and, generally, there are no long pauses. All objects not reachable from the root set of objects should be reclaimed during garbage collection.

Mark and Sweep and Copy-Compact are the scavenging algorithms explored. Baker's realtime algorithm and three versions of reference counting are the incremental algorithms explored. When encoding each algorithm to run on database data, small alterations occur, but the basic flavor of the algorithm is maintained. The algorithms and their database implementation are discussed in this section.

6.3.1. Mark and Sweep

In Mark and Sweep algorithms, scavenging has two phases: a mark phase and a sweep phase. In the first phase, the data is traversed marking everything that is accessible from the root set. In the second phase, a pass is made over the memory deleting everything that is not marked. Using Mark and Sweep, a running program can experience long pauses during the scavenging of memory. In addition, clustered groups of objects gradually become fragmented as objects are deleted.

The first phase of Mark and Sweep is inherently recursive. A few methods have been explored to implement recursion in a DBMS [Banc86b], two of which are explored here: repeated execution [Ston83], and transitive iteration [Banc86a]. Figure 6.4 depicts these two different methods in pseudo-Quel. The first method executes the Quel operator **replace** repeatedly until the query results in no change to the database. The second method implements transitive iteration in Quel by keeping a temporary relation, the *MarkRelation*, that holds the values that were updated at each iteration and only uses those tuples in the next iteration.

```
/* Method 1 */
range of o,o1 is ConsRelation
range of r is SymbolRelation

/* mark all objects reachable from the root */
replace (o.mark = TRUE)
where r.type = "pointer" and r.value = o.id

/* mark all objects pointed to by a right or left pointer of */
/* an already marked object, until all are marked */
repeat
        replace* (o.mark = TRUE)
        where (o1.right_type = "pointer" and o1.right_value = o.id) or
                (o1.left_type = "pointer" and o1.left_value = o.id)
until no changes to o

/* Delete all unmarked (hence unreachable) objects */
delete o where o.mark = FALSE

/* Unmark all retained objects */
replace (o.mark = False)

----------------------------------------------------------

/* Method 2 */
range of m,m1 is MarkRelation

/* Mark all objects pointed to by the root as reachable by retrieving them into MarkRelation */
append to MarkRelation (id = r.value, mark = TRUE)
where r.type = "pointer"

/* While there are objects whose pointers have not been followed */
while count(where m.mark=TRUE) != 0 do {
        /* "Mark" all objects reachable from objects marked TRUE and mark them FALSE */
        /* Mark them only if they haven't been marked before */
        /* FALSE indicates the object has not yet had it's pointers followed */
        append to MarkRelation (id = o1.id, mark = FALSE)
        where (o.id = m.id) and (m.mark=TRUE) and
                [ (o.left_type="pointer" and o.left_value=o1.id) or
                (o.right_type="pointer" and o.right_value=o1.id) ] and
                o1.id != ANY(m1.id)

        /* All objects whose pointers were just expanded from need not be considered again */
        /* All objects not yet expanded from (FALSE) are to be expanded from next (TRUE) */
        replace (m.mark=SAVED) where m.mark=TRUE
        replace (m.mark=TRUE) where m.mark=FALSE
}
/* Delete all objects that are not represented in MarkRelation */
delete o where o.id != ANY(m.id)
```

Figure 6.4: Implementations of Mark and Sweep

6.3.2. Copy-Compact

Using a copy-compact algorithm [Mins63], the root set of objects is copied into an unused portion of memory. All objects that are reachable from the root set are also copied into this unused portion of memory. The entire memory in which the data used to reside is then reclaimed. Copy-Compact schemes require twice as much memory as is occupied by data: the memory where the data currently resides (*oldspace*) and a portion equal in size to be used for copying (*copyspace*). After each iteration of the algorithm, copyspace becomes oldspace and oldspace becomes copyspace.

As with Mark and Sweep, long pauses occur when the scavenging phase takes place because the copying takes time proportional to the number of objects plus the number of references between objects. No cluster fragmentation results from using this method because objects are reclustered in memory as they are copied. The memory will be efficiently clustered if the objects are copied in the same order as accesses occur. Copying in this manner clusters objects together on a page that will be accessed together.

Figure 6.5 depicts two Copy-Compact algorithms: one that creates a depth-first cluster and one that creates a breadth-first cluster. These are similar to the encoding used for Method 2 of Mark and Sweep. The first method (i.e., repeated execution of the replace until no tuples are altered) will not be explored due to its poor performance on Mark and Sweep which will be shown in section 4.

6.3.3. Baker's realtime

Baker [Bake76] developed a version of copy-compact that breaks up the job of object copying into pieces so that the copying can be done gradually. In each piece a bounded number of objects are copied, thereby limiting the length of the pauses in a program. Each time a new object is allocated, the pointers from k objects in copyspace are traversed.[5] This allows the scavenging of memory to be performed incrementally. If a pointer refers to an object in oldspace, the object is copied into copyspace and the pointer is updated to refer to

[5] k can be set by the algorithm implementor to limit pauses to an imperceptible amount of time.

```
/* Breadth-First */
range of o,o1 is Oldspace
range of c, c1 Copyspace
range of r is SymbolRelation
range of m, m1 is MarkRelation

/* Mark all objects pointed to by the root as reachable by retrieving them into MarkRelation */
append to MarkRelation (id = o.all, mark = TRUE)
where r.type = "pointer"

/* While there are objects whose pointers have not been followed */
while count(where m.mark==TRUE) != 0 do {
          /* "Mark" all objects reachable from objects marked TRUE and mark them FALSE */
          /* Mark them only if they haven't been marked before */
          /* FALSE indicates the object has not yet had it's pointers followed */
          append to MarkRelation (o.all, mark = FALSE)
          where (m.mark==TRUE) and
                    [ (m.left_type=="pointer" and m.left_value==o.id) or
                    (m.right_type=="pointer" and m.right_value==o.id) ] and
                    o.id != ANY(m1.id)
                    o.id != ANY(c.id)

          /* All objects whose pointers were just expanded from are moved to Copyspace */
          /* All objects not yet expanded from (FALSE) are to be expanded from next (TRUE) */
          append to Copyspace (m.all) where m.mark==TRUE
          delete m where m.mark==TRUE
          replace (m.mark==TRUE)
}
/* Make sure Copyspace is now treated as oldspace */
Swap (Oldspace, Copyspace)

---------------------------------------------------------------

/* Depth-First */

range of t, t1 RightRelation

/* Mark all objects pointed to by the root as reachable by retrieving them into MarkRelation */
append to MarkRelation (id = o.all, mark = TRUE)
where r.type = "pointer"

/* While there are objects whose pointers have not been followed */
while count(where m.mark==TRUE) != 0 do {
          /* Mark objects reachable via a right pointer */
          /* Mark them only if they haven't been marked before */
          append to RightRelation (o.all, mark = TRUE)
          where [ (m.left_type=="pointer" and m.left_value==o.id) or
                    (m.right_type=="pointer" and m.right_value==o.id) ] and
                    o.id != ANY(m1.id) and
                    o.id != ANY(t.id) and
                    o.id != ANY(c.id)
/* While some object reachable via right pointer has not been followed */
          while count(where t.mark==TRUE) != 0 do {
                    append to RightRelation (o.all, mark = FALSE)
                    where (t.right_type=="pointer" and t.right_value==o.id) and
                              o.id != ANY(m.id) and
                              o.id != ANY(t1.id) and
                              o.id != ANY(c.id) and
                              t.mark = TRUE
```

```
                append to MarkRelation (t.all, mark = TRUE)
                where t.mark = TRUE
                delete t where t.mark==TRUE
                replace t (t.mark==TRUE)
            }
        /* "Mark" all objects reachable from objects marked TRUE and mark them FALSE */
        /* Mark them only if they haven't been marked before */
        append to MarkRelation (o.all, mark = FALSE)
        where (m.mark==TRUE) and
                (m.left_type=="pointer" and m.left_value==o.id) and
                o.id != ANY(m1.id) and
                c.id != ANY(c.id) and
                m.mark==TRUE

        /* All objects whose pointers were just expanded from are moved to Copyspace */
        /* All objects not yet expanded from (FALSE) are to be expanded from next (TRUE) */
        append to Copyspace (m.all) where m.mark==TRUE
        delete m where m.mark==TRUE
        replace (m.mark==TRUE)
    }
    /* Make sure Copyspace is now treated as oldspace */
    Swap (Oldspace, Copyspace)
```

Figure 6.5: Copy-Compact Implementations

the object's copyspace location. Since pointers may still exist that refer to the object in oldspace (after it has been moved to copyspace), a marker must be left in the oldspace object that indicates the object now resides in copyspace. Each time an object in oldspace is accessed, it is also moved into copyspace. When all objects in copyspace have been traced, all the reachable objects have been copied into copyspace, and the spaces are exchanged: oldspace becomes the new copyspace and the root set of objects is copied into it; and copyspace becomes oldspace.

Figure 6.6 depicts the database implementation. In a database implementation, oldspace and copyspace can be two separate relations. Since database objects can be referenced by logical identifiers not addresses, an object can have the same logical identifier in both relations. Hence, this implementation allows no markers to be left in oldspace. In the database implementation, the effect k has on the allocate operation is represented by the variable *threshold*. The loop will execute until the amount of work completed reaches a threshold. Baker's algorithm is very similar to the breadth-first version of Copy-Compact depicted in Figure 6.5 and will be influenced by the same factors as the breadth-first methods.

```
/* Baker */
Set-up:{
range of o,o1 is Oldspace
range of c, c1 Copyspace
range of r is SymbolRelation
range of m, m1 is MarkRelation
}

Flip:{
/* Swap spaces and Mark all objects pointed to by the root as reachable */
Swap(Oldspace, Copyspace)
append to MarkRelation (id == o.all, mark == TRUE)
where r.type == "pointer"
}

On Allocate:{
/* While there are objects whose pointers have not been followed */
while count(where m.mark==TRUE) != 0 and Threshold_not_achieved do {
        /* "Mark" all objects reachable from objects marked TRUE and mark them FALSE */
        /* Mark them only if they haven't been marked before */
        /* FALSE indicates the object has not yet had it's pointers followed */
        append to MarkRelation (o.all, mark == FALSE)
        where (m.mark==TRUE) and
                [ (m.left_type=="pointer" and m.left_value==o.id) or
                (m.right_type=="pointer" and m.right_value==o.id) ] and
                o.id != ANY(m1.id)
                o.id != ANY(c.id)

        /* All objects whose pointers were just expanded from are moved to Copyspace */
        /* All objects not yet expanded from (FALSE) are to be expanded from next (TRUE) */
        append to Copyspace (m.all) where m.mark==TRUE
        delete m where m.mark==TRUE
        replace (m.mark==TRUE)
}

if count(MarkRelation) == 0 then Flip();
}

On Access:{
retrieve (object==c.all) where c.id == <given>
if object==NULL then
        retrieve object==o.all where o.id == <given>
return(object)
}
```

Figure 6.6: Implementation of Baker

6.3.4. Generation-Scavenge

To reduce the number of objects in memory that need to be considered when doing a copy, the Generation-Scavenge schemes assign ages to objects [Unga85]. The age groups are determined by the algorithm implementor. A separate portion of memory is allocated for objects of each age group. Each age group's root set consists of the objects in this age group that are pointed to by objects in older age groups. In virtual memory, the younger the object, the more likely it is to become garbage [Lieb83], so young spaces are scavenged more often than older spaces. That young objects become reclaimable sooner is caused by the use of temporary variables in programming languages. Objects that reside in the oldest space are considered to be permanent objects and their space is not examined very often to see if it contains any garbage. The pauses caused by a scavenge are greatly reduced in this scheme since only a subset of the objects are traversed at each scavenge.

Because all of the objects in the database are permanent, performance benefits based upon distinguishing between temporary and permanent data will not show much benefit in a persistent environment where the virtual environment is separate from the DBMS. Temporary variables are not likely to be stored in a database so whether age correlates with the probability of becoming garbage in a persistent environment is not clear. Studies should be performed to check whether or not such a correlation exists. Therefore, the assumption that allows Generation-Scavenge to have better performance than other algorithms (namely, that objects become garbage based upon their age) does not necessarily hold for database objects. Hence, Generation-Scavenge is not explored in this study.

6.3.5. Reference Count

If a Reference Count scheme is employed, each object in memory contains a count of the number of objects that reference it. Each time a reference is removed, the count is decremented. If the count becomes zero, the object gets reclaimed and all objects to which it points must have their counts decremented. This in turn may result in an object's count becoming zero. Hence, removal of a single reference could cause multiple deletes. This method tends not to have long pauses since the work is distributed across alterations, but due to the cascading effect of a decrement, the amount of time any alteration takes is highly variable.

In addition, this scheme does not detect circular garbage. In the simplest case, a circle of garbage will occur if *objectA* references *objectB* and *objectB* references *objectA* but no other object references either *objectA* or *objectB*. Both *objectA* and *objectB* have reference counts of one, but neither is accessible from a root object.

Figure 6.7 depicts the database implementation of reference count. These operations will only be performed when persistent objects are updated. Since DecrementChildren is

```
/* unode is the node that was just updated */
/* unode.old is the id of the object that unode used to reference */
/* unode.new is the id of the new object that unode references */
/* If a new or old value is NULL nothing happens (not even an error) */

On update:{
retrieve rc==o.refcount where o.id == unode.old
if rc===1 then
        DecrementChildren(unode.old)
        delete o where o.id == unode.old
else
        replace (o.refcount == o.refcount-1) where o.id==unode.old

replace (o.refcount == o.refcount+1) where o.id==unode.new
}

DecrementChildren(did):{
/* dnode is a variable that will hold the to-be-deleted object */
retrieve dnode==o.all where o.id==did
if (dnode.left_type === "pointer") then
        retrieve rc==o.refcount where o.id == dnode.left_value
        if rc===1 then
                DecrementChildren(dnode.left_value)
                delete o where o.id == dnode.left_value
        else
                replace (o.refcount == o.refcount-1) where o.id==dnode.left_value
if (dnode.right_type === "pointer") then
        retrieve rc==o.refcount where o.id == dnode.right_value
        if rc===1 then
                DecrementChildren(dnode.right_value)
                delete o where o.id == dnode.right_value
        else
                replace (o.refcount == o.refcount-1) where o.id==dnode.right_value
}
```

Figure 6.7: Implementation of Reference Count

recursive, it could cause a long pause during processing.

6.3.6. Logged Reference Count

To overcome the variable cost associated with decrementing the reference counts, alterations to reference counts can be logged and the log can be processed to update k reference counts at a time. This would put a bound on the number of objects that can be changed by one update. The log processing can be done during allocates (as with Baker) or could be done in the background.

Figure 6.8 depicts the database implementation of logged reference count. The major change from Reference Count (depicted in Figure 6.7) is the operation associated with an update: for Logged Reference Count the operation is much simpler. In addition, there is no DecrementChildren operation; therefore there will be no indeterminate pauses.

```
/* newleft is a new left value, newright is a new right value */
/* either of this could be null */
range of l is log

On update:{
append to log (time=$time, id=o.id, oldl=o.left_value, oldlt=o.left_type,
                    oldr=o.right_value, oldrt=o.right_type,
                    newl=newleft.left_value, newlt=newleft.left_type,
                    newr=newright.right_value, newrt=newright.right_type)
where o.id=unode.id
}

Log process time:{
for the oldest K unprocessed tuples do {
          replace (o.refcount = o.refcount-1) where o.id=l.oldl and l.oldlv="pointer"
          replace (o.refcount = o.refcount-1) where o.id=l.oldr and l.oldrv="pointer"
          replace (o.refcount = o.refcount+1) where o.id=l.newl and l.newlv="pointer"
          replace (o.refcount = o.refcount+1) where o.id=l.newr and l.newrv="pointer"
          append to log (time=$time, id=o.id, oldl=o.left_value, oldlt=o.left_type,
                              oldr=o.right_value, oldrt=o.right_type)
          where o.refcount=0
          delete o where o.refcount=0
          }
}
```

Figure 6.8: Implementation of Logged Reference Count

6.3.7. Deutsch-Bobrow reference count

Deutsch and Bobrow [Deut79] made the observation that most objects have a reference count of one. This means that most of the space typically allocated to hold the counts is wasted. They proposed keeping a table of the objects that have a zero count (newly created objects) and a table of objects that have a count greater than one. The tables hold only the addresses of the objects, no other information. When a reference to an object on the multireference table is removed, no information is kept that would indicate whether that was the last reference to the object. Therefore, once an object is on the multireference table, it cannot be removed. Their scheme operates the same as standard reference counting but uses less space because individual reference counts are not needed, although over time more space may be used because extra garbage may gather due to the permanent effect of being added to the multireference table.

```
/* Here unode designates the object whose reference count needs to be */
/* increased or decreased */
On Alloc:{
append to ZCT newnode.id
}

On Update (to unode whose reference count increases):{
retrieve (new=z.all) where z.id=unode.id
if new != NULL then
        delete z where z.id=unode.id
else
        retrieve (new=m.all) where m.id=unode.id
        if new = NULL then
                append to MRT unode.id
}

On Update (to unode whose reference count decreases) & On Delete:{
retrieve (new=m.all) where m.id=unode.id
if new = NULL then
        DecrementChildren(unode)
        delete unode
}
```

Figure 6.9: Implementation of Deutsch-Bobrow

Figure 6.9 depicts the database implementation of Deutsch-Bobrow method of reference counting. The major change from Reference Count (depicted in Figure 6.7) is the operation associated with an update when the reference count would be incremented. The operation on a decrement is very similar except no reference count is actually changed.

6.4. Input/Output Characteristics of Algorithms

Because the scavenging and incremental algorithms operate differently, different methods must be used to evaluate their cost. The critical factors that should be minimized are as follows:

the lengths of pauses, and
the expected average cost to access objects.

Incremental algorithms can most readily be evaluated on the average cost to access objects since their cost is incurred during accesses. Section 4.3 reports the increase in access cost that each incremental algorithm mandates. Scavenging algorithms are best evaluated on the length of the pause that each causes. The next two sections (4.1 and 4.2) examines the cost of all scavenging algorithms as different parameters vary. Section 4.1 examines the cost as the number of objects varies. Section 4.2 reevaluates the algorithms that 4.1 showed had the lowest costs with the data structure varying. In section 4.4 the scavenging and the incremental algorithms are compared.

6.4.1. Performance of Scavenging Algorithms as Number of Objects Varies

In performing the analysis in this section, only a few parameters are varied. For the results reported herein, the number of buffer pages (BP) had little effect so is left out of the analysis. The number of objects (N) varies from 5000 to 1 million. A structure of 1 million objects is large enough to cause some of the algorithms to exhibit unreasonably high I/O cost. The other parameters have been held constant and their settings are:

UR = 0.1
ProbDelete = 0.025
Height = N/16[6]
RLRatio = 1.2[6]
Density = 1.3
A = 1 million

These numbers represent a middle ground: some updates and deletes, a few doubly connected objects, and one access per object on average. In addition, the objects are assumed to be clustered in a depth-first fashion when the analysis begins since current Lisp systems tend to allocate cells depth-first.

6.4.1.1. Mark and Sweep

The expected I/O cost of three different methods of processing Mark and Sweep have been explored. **Mark&SweepA** represents the costs of using the repeated execution as depicted by Method 1 of Figure 6.4. **Mark&SweepB** does the transitive closure as expressed by Method 2. **Mark&SweepC** presumes the transitive closure method of Method 2, with the assumption that the temporary relation (MarkRelation) is buffered in main memory; hence, no I/O cost is incurred from accessing it.

Figure 6.10 shows the cost of using each Mark and Sweep implementation as the number of objects increases. Mark&SweepB is twice the cost of Mark&SweepC because the cost of accessing the MarkRelation is zero for Mark&SweepC. Mark&SweepA, for one million cells, has an expected I/O cost of 23 billion. A typical, random access, high speed disk can process about 50 page requests per second; hence Mark&SweepA, in this case, would take approximately 15 years to do I/O. In contrast, Mark&SweepC would take about 10 hours of I/O for one million objects.

[6] These parameters effect on performance is explored in the Section 4.2.

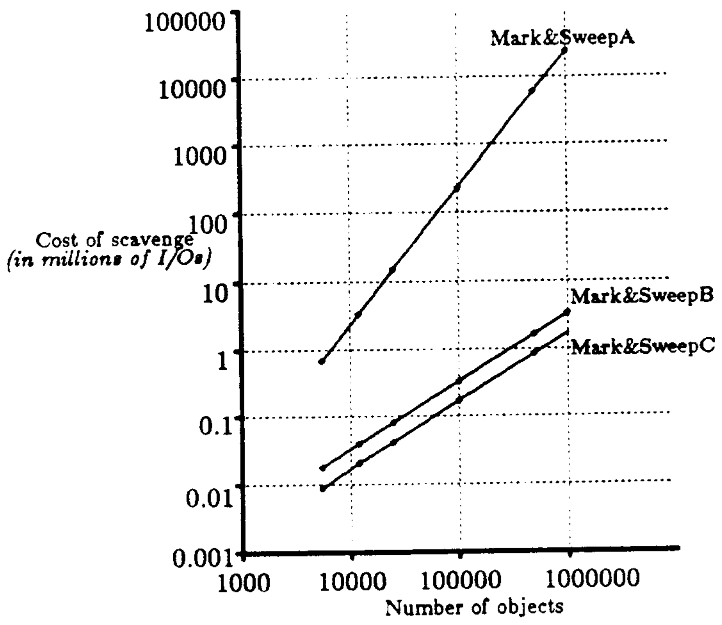


Figure 6.10: Cost of Mark and Sweep

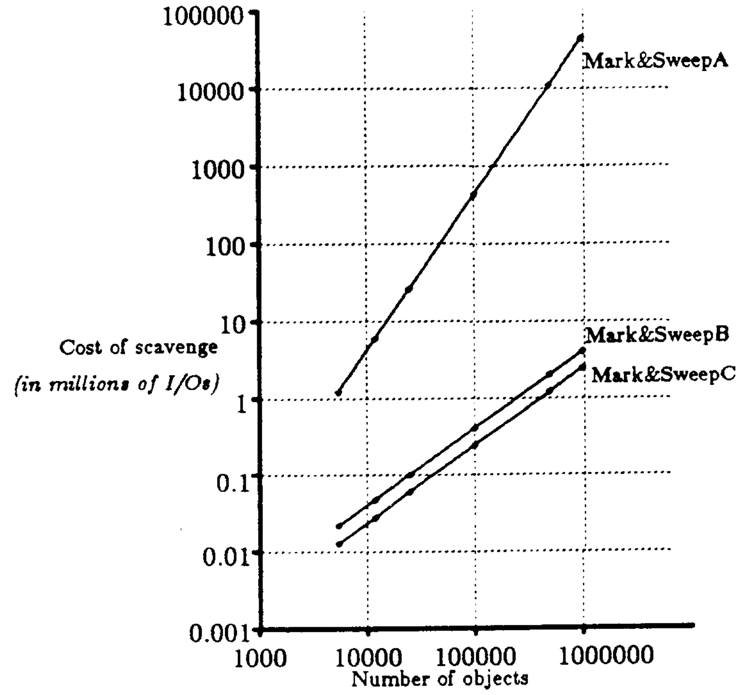


Figure 6.11: Unclustered Performance of Mark and Sweep

The costs shown in Figure 6.10 assume that the cells are depth-first clustered before the reclamation scheme executes. Since Mark and Sweep does not copy the list to retain clustering, over time the cells will become unclustered. Figure 6.11 shows the readjusted performance of Mark and Sweep when no clustering exists. The cost increases by 50% for Mark&SweepB and Mark&SweepC; whereas, Mark&SweepA doubles in cost. Therefore, Mark&SweepA requires approximately 30 years of disk I/O when one million unclustered objects are in the database. Clearly this is not a viable method to use for disk-based storage reclamation.

6.4.1.2. Copy-Compact

A Copy-Compact algorithm that creates a depth-first cluster and one that creates a breadth-first cluster are examined using the implementation techniques described for

Mark&SweepB and Mark&SweepC: a disk-based temporary relation is used to implement a transitive closure for *BreadthFirstB* and *DepthFirstB*; and, a main memory relation is used for *BreadthFirstC* and *DepthFirstC*. The method of query evaluation assumed for Mark&SweepA (i.e., repeated execution of the replace until no tuples are altered) is not explored due to its very poor performance.

Figure 6.9 graphs, for the three Shape values, the cost for the Copy-Compact implementations as the number of objects increases. DepthFirstB is more costly than BreadthFirstB because depth-first traversal requires two temporary relations: one to handle the transitive closure involved in stepping down the right pointers; and the other to keep track of the left pointers that need to be followed. Breadth-first traversal treats right and left pointers the same; therefore, one temporary relation is sufficient.

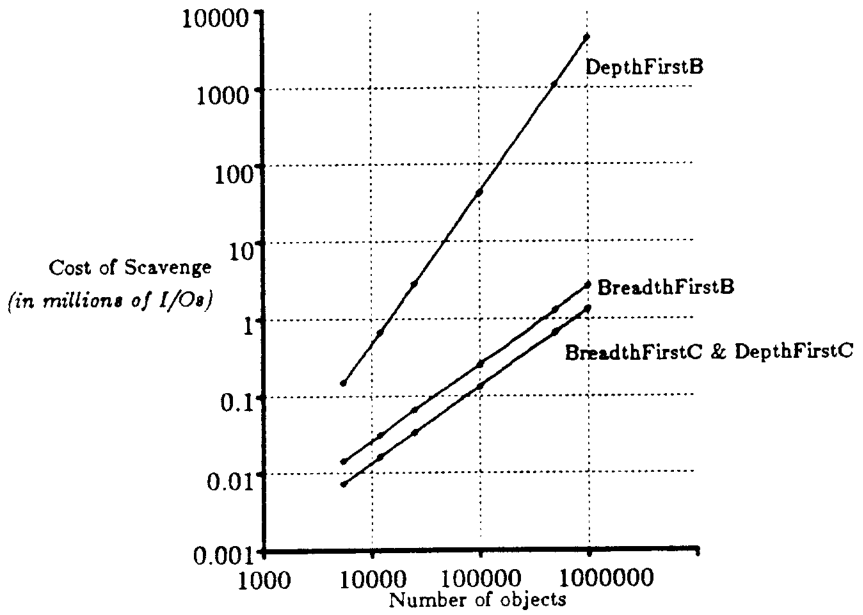


Figure 6.12: Cost for Copy-Compact

That DepthFirstB is a thousand times more costly is surprising. Upon closer examination of the algorithm, it appears that the particular data structure used for this evaluation interacted poorly with the algorithm. The effect that data structures have on each algorithm will be examined in detail in Section 4.2.

The most costly of the algorithms, DepthFirstB, would take approximately 3 years of disk I/O. BreadthFirstC and DepthFirstC have equivalent performance since DepthFirstC's extra temporary relation is also kept in physical memory, hence does not incur the extra I/Os which make DepthFirstB more costly. BreadthFirstC and DepthFirstC would take about 7 1/2 hours of I/O each, in contrast with unclustered Mark&SweepC which would take 15 hours of I/O.

The objects are clustered efficiently for the storage reclamation algorithms if the access pattern of the algorithm matches the object placement pattern. This will occur if the object allocation scheme happens to allocate objects on pages that match the reference pattern of the storage reclamation scheme. Since Lisp systems tend to allocate depth-first, how well BreadthFirst Copy-Compact performs on a depth-first clustered structure is of interest. The I/O costs of the BreadthFirstB and BreadthFirstC, if the cells are actually clustered in a depth-first pattern, are indistinguishable from the costs if the cells are clustered in a breadth-first pattern. This is related to the interaction of buffering and performance described in Chapter 4. As a result, the clustering pattern of the data does not have to match the access pattern of the garbage collection algorithm to benefit the performance of a breadth-first algorithm when buffering exists.

6.4.2. Performance of Scavenging Algorithms As Infrastructure Varies

As was conjectured from Figure 6.12, the data structure and buffer space may have a great impact on performance. In this section, the form of the data structure is varied along with the amount of buffer space. The method of each algorithm that performed the best (i.e., Mark&SweepC, BreadthFirstC, and DepthFirstC) will be the only ones studied in this section. Any data structure or buffering that affects the performance of these will also affect the other algorithms.

The parameters are varied as follows:

N = 1 million, 10 million, 100 million
BP = 3, 15, 50, 1000
RLRatio = 1 ... 5
Height = $\log_2$ N ... N/5

The infrastructure of the data will have a more exaggerated effect on the performance for large N, therefore N varies from 1 million to 100 million. For simplicity, when the Height is varied the RLRatio is kept at 1; and, when the RLRatio is varied, the Height is set to $\log_{1.001}$ N.

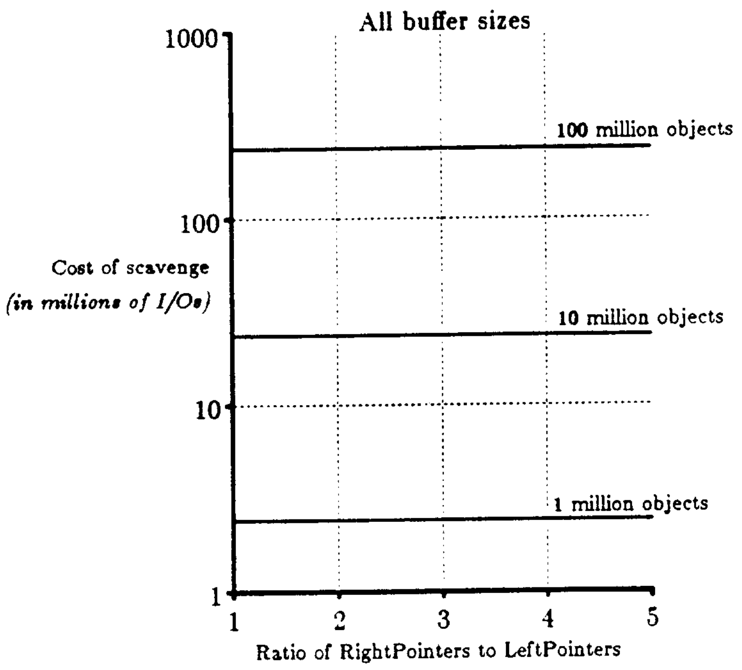


Figure 6.13: Cost for Mark&SweepC as Ratio Varies

6.4.2.1. Mark and Sweep

Since Mark and Sweep is a breadth-first algorithm, the ratio of right to left pointers does not affect its performance as is shown in Figure 6.13. This indicates that Mark and Sweep is unable to take advantage of clustering along the right pointers. As with previous tests, buffer sizes do not affect performance either. After close examination of the equation representing Mark&SweepC, for Mark and Sweep to be influenced by buffer space, BP must be increased to 5 megabytes which is enough to hold the average number of objects per level.

Height effects the performance of Mark and Sweep very little (Figure 6.14). As the height varies, the I/O remains fairly constant but has a slight increase (10%) as the structure degenerates at it greatest height. Since a greater height means more levels, and therefore more times through the *while* loop, it costs a slight bit more.

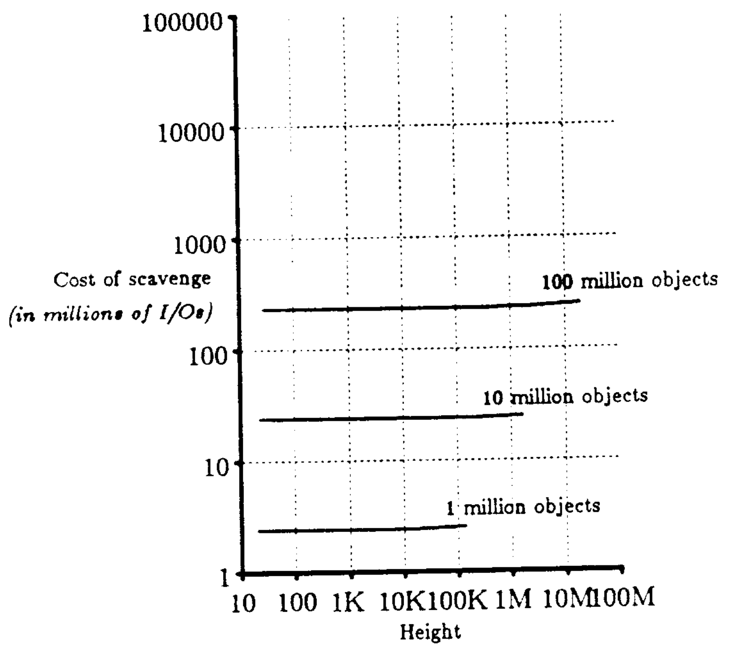


Figure 6.14: Cost for Mark&SweepC as Height Varies

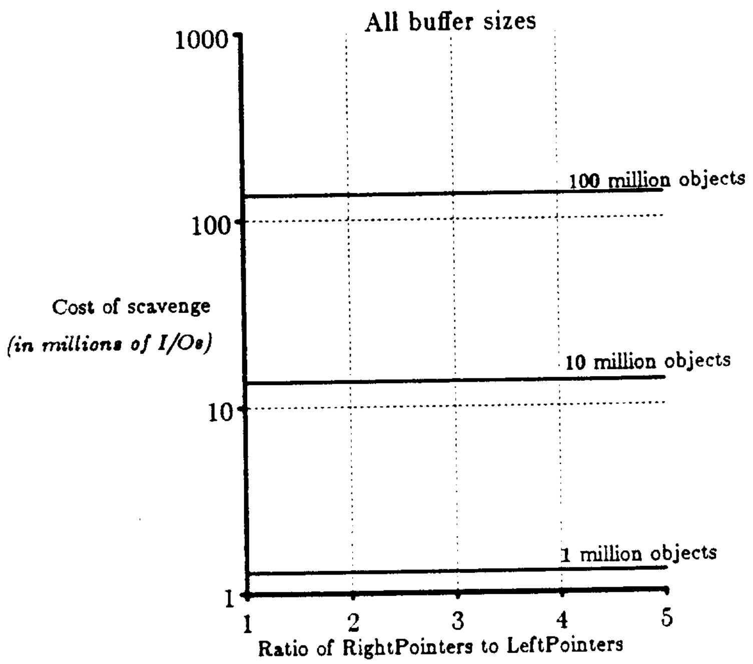


Figure 6.15: Cost for BreadthFirstC as Ratio Varies

6.4.2.2. Breadth-First Copy-Compact

As with Mark and Sweep, the ratio of left to right pointers does not affect its I/O characteristics (Figure 6.15). But, as Figure 6.16 demonstrates, as the height varies for the structure, the buffer space makes an slight impact. When only 3 buffer pages exist, the I/O increases 10% for 100 million objects as the structure degenerates to a height of 20 million. With 15 buffer pages, this increase disappears and the I/O for all numbers of objects experiences a slight decrease as the structure reaches its greatest height. As BP increases, the decrease occurs at lower and lower heights. This can be explained by the buffer space being able to accommodate a greater portion of the average objects per level.

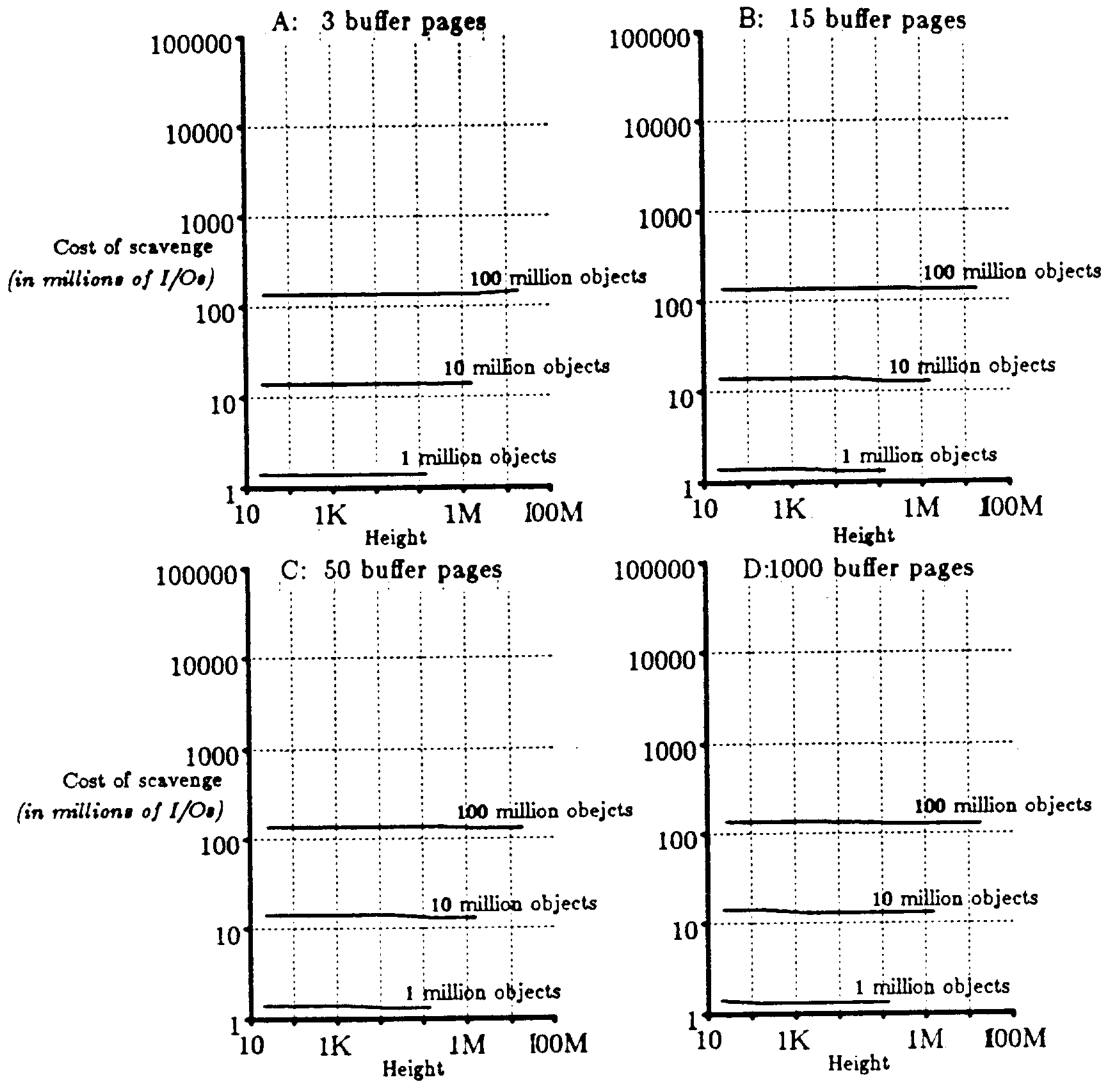


Figure 6.16: Cost for BreadthFirstC as Height and Buffer Space Vary

6.4.2.3. Depth-First Copy-Compact

The depth-first version of copy-compact is a more complex algorithm which is tuned for a particular data structure, therefore, it experiences a greater variance in performance as the data structure changes. As might be predicted, the more right pointers, the better the performance. Figure 6.17 shows the performance for 1, 10, and 100 million obejcts as the RLRatio increases and BP varies. The lower the ratio, the more I/O is incurred. The RLRatio effect on performance can be obliterated by allocating more buffers. The more objects, the more buffer space is needed to counteract the effect of the ratio. The cost correlates with $height * \lceil \frac{avg\_cells\_per\_level}{NP} \rceil$.

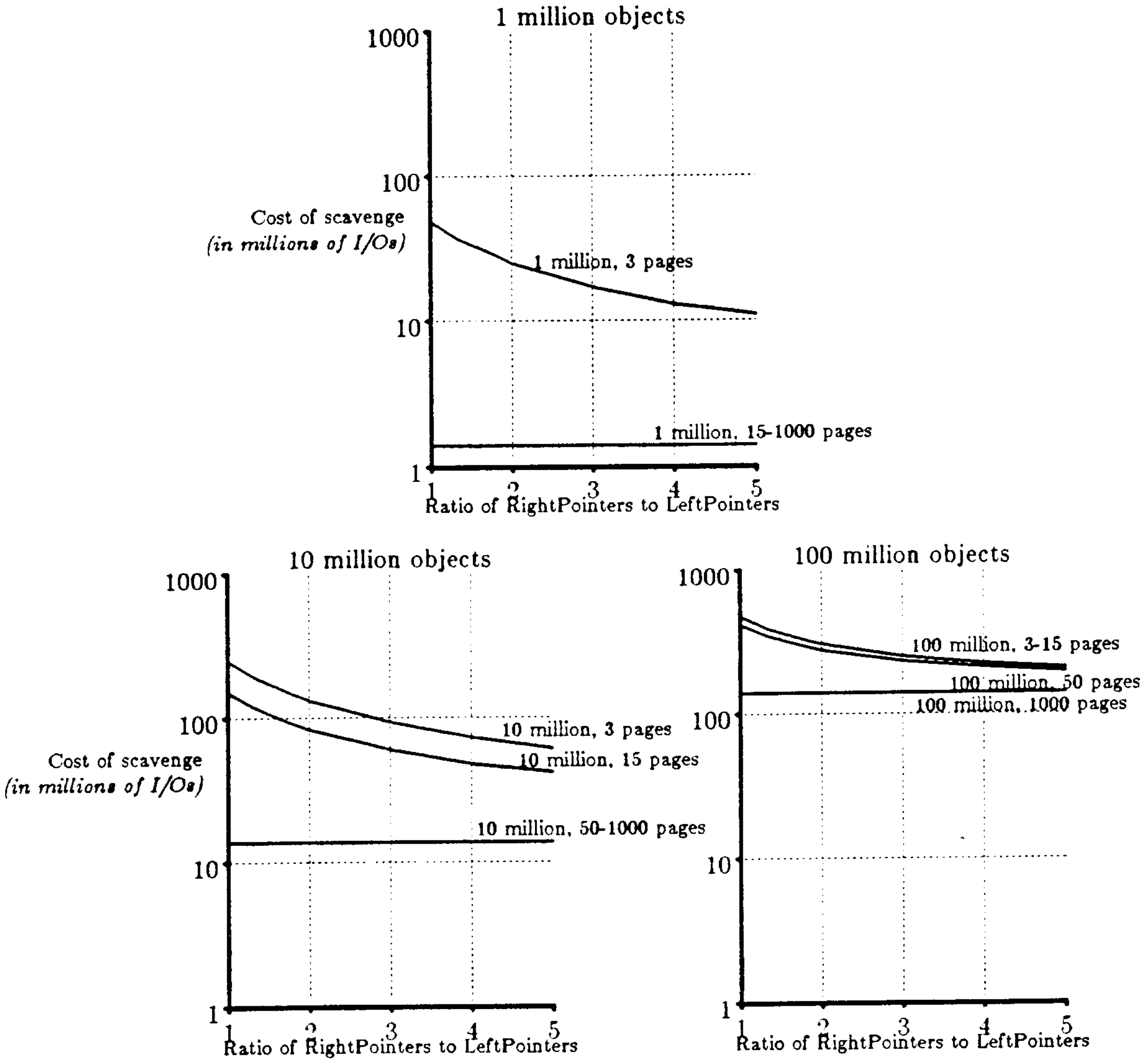


Figure 6.17: Cost for DepthFirstC as Ratio and Buffer space Vary

The Height interacts with the buffer size effecting the performance in strange ways as shown in Figure 6.18 When the structure is compact (i.e., the Height has a low value), the cost is low because every level of the structure is full and a lot of processing will occur in each pass. In addition, fewer passes through the *while* loop means less times each query is executed. When Height is a large number, the structure is very sparse, but many right pointers are strung together to attain that height. Hence, even though RLRatio is set to 1, when the Height is large, DepthFirstC performs as if the RLRatio is greater. The cost is great when the height has a middle value because the algorithm can neither benefit by a compact structure, nor by long strings of right pointers.

Mark and Sweep and the breadth-first version of Copy-Compact are not effected very much by the structure of the data or the amount of buffer space available. On the other hand, the depth-first version of Copy-Compact is greatly effected by the data's structure and the buffer space. If large amounts of buffer space are available, the depth-first algorithm will perform well regardless of the structure of the data.

6.4.3. Performance of Incremental Algorithms

In this section, the increase in access cost that is mandated by incremental algorithms will be examined. This evaluation will exclude any cost that can be deferred (e.g. log processing). All the costs not covered here will be discussed in section 4.4 which compares all the algorithms.

6.4.3.1. Baker's realtime

Since database objects can be referenced by logical identifiers, which can be the same for the oldspace and copyspace relations, this simplifies the lookup procedure: an object should be looked for first in copyspace and only if it is not found there is oldspace checked. The lookup cost increases by 50% since on average half of the objects will require a double lookup. Other

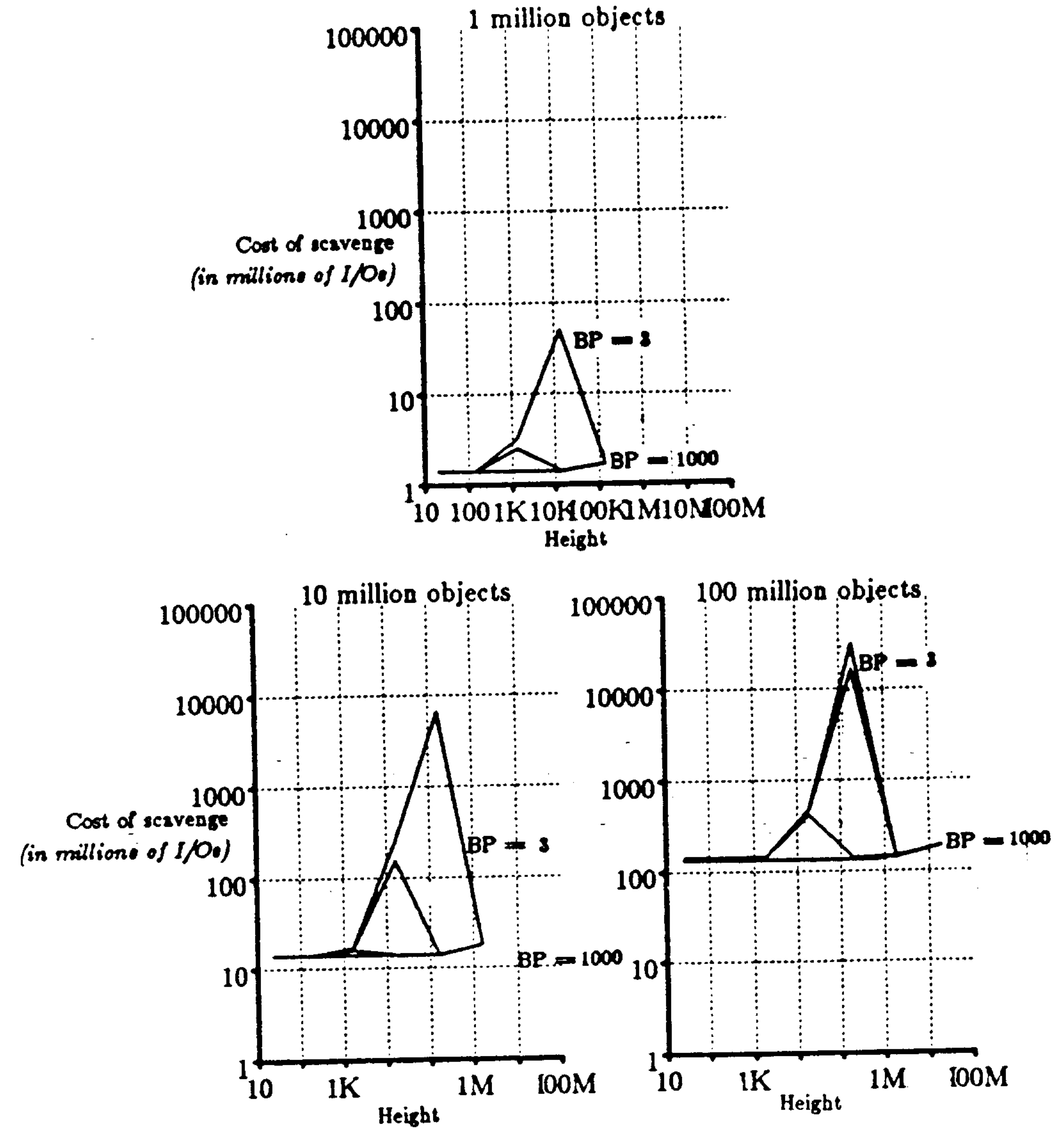


Figure 6.18: Cost for DepthFirstC as Height Varies

than the increased access cost, Baker has a total cost similar to the breadth-first version of Copy-Compact although this cost is broken up into chunks. This is because Baker is identical to breadth-first Copy-Compact done in pieces.

If the algorithm is implemented such that k objects are scavenged every m deletes where $m = p * k$ then a full scavenge of memory would be completed each time $p * N$ of the objects were deleted. The total cost of scavenging for Copy-Compact is amortized over *(p * N)/DR* accesses. Although the average increase in cost depends only upon DR, the length of the pauses depends upon m, k, and DR.

The cost to do a full scavenge of memory will be the same as Breadth-First Copy-Compact and will be influenced by the same factors.

6.4.3.2. Reference Count

Assuming no circular garbage exists, the average increase in access cost incurred because of the increased update cost in a Reference Count scheme is graphed in Figure 6.19. These amounts do not factor in increases due to loss of clustering. Each of the curves has a different value for the update ratio; the delete ratio varies along the X-axis; and, the Y-axis represents the average I/O cost. The increase in access cost ranges from 0, when no objects are updated or deleted, to 1.6, when the update ratio and the delete ratio are 0.1.

6.4.3.3. Logged Reference Count

The expected increase in the average access cost when Logged Reference Count is used is shown in Figure 6.20. Each of the curves has a different value for the update ratio; the delete ratio varies along the X-axis; and, the Y-axis represents the average I/O cost. The cost ranges from 0, when no objects are updated or deleted, to 0.3, when the update ratio and the delete ratio are 0.1. The increase in access cost is independent of DR because each update only causes a write to a log; all updating of reference counts and removal of unreferenced objects occurs at log processing time; all updating of reference counts and removal of unreferenced objects occurs at log processing time. The cost increase is very low since the reference counts are not actually updated until the log is processed. The pauses caused by processing the log are dependent upon the same factors as the pauses for Baker. The total cost of this method is shown in Section 4.4 when it is compared to other schemes.

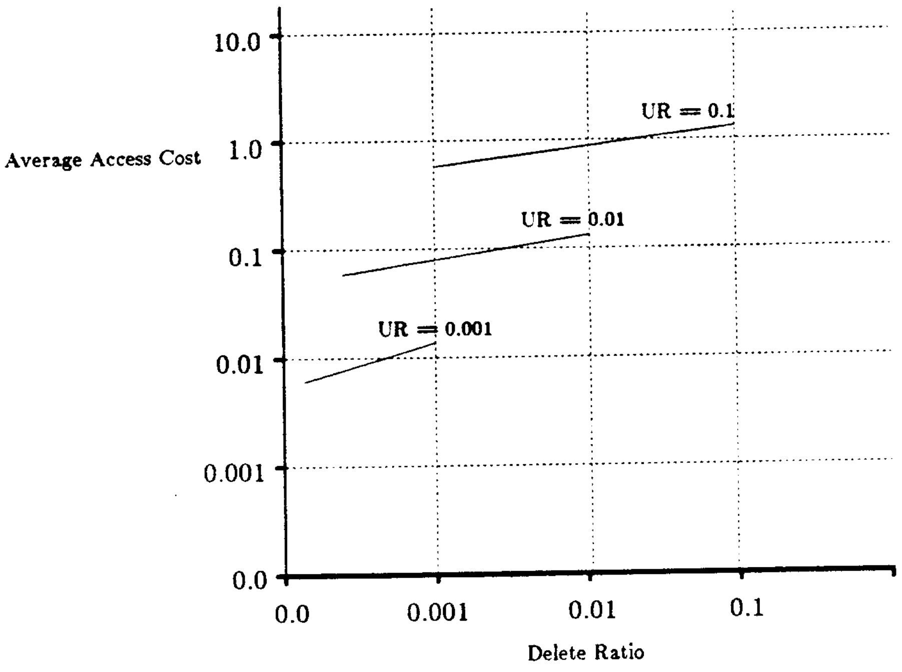


Figure 6.19: Increase in Per Access Cost for Reference Count

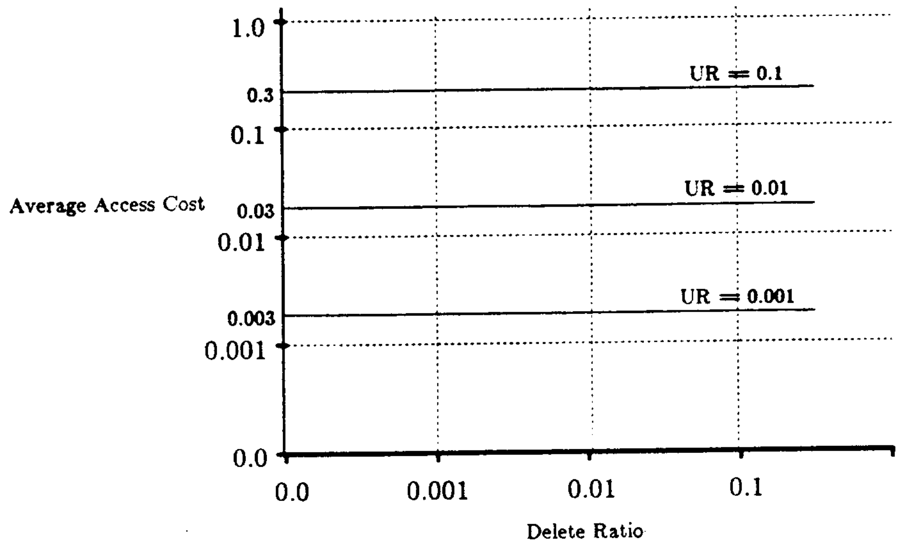


Figure 6.20: Increase in Per Access Cost for Logged Reference Count

6.4.3.4. Deutsch-Bobrow reference count

Figure 6.21 graphs the extra cost per access that the Deutsch-Bobrow method incurs. Each of the curves has a different value for the update ratio; the delete ratio varies along the X-axis; and, the Y-axis represents the average I/O cost. The cost ranges from 0, when no objects are updated or deleted, to 0.72, when the update ratio and the delete ratio are 0.1. The cost is less than Reference Count since not all updates cause changes in the reference count tables.

6.4.4. Comparing the schemes

As was stated in this section's introduction, the items to consider when comparing the I/O cost of garbage collection schemes are the length of the pauses, and the cost to access objects which is influenced by whether clustering is supported. These two factors can be captured by considering the total cost of accesses plus the garbage collection cost. Whether data is clustered or not determines the expected average access cost; adding to that the cost of

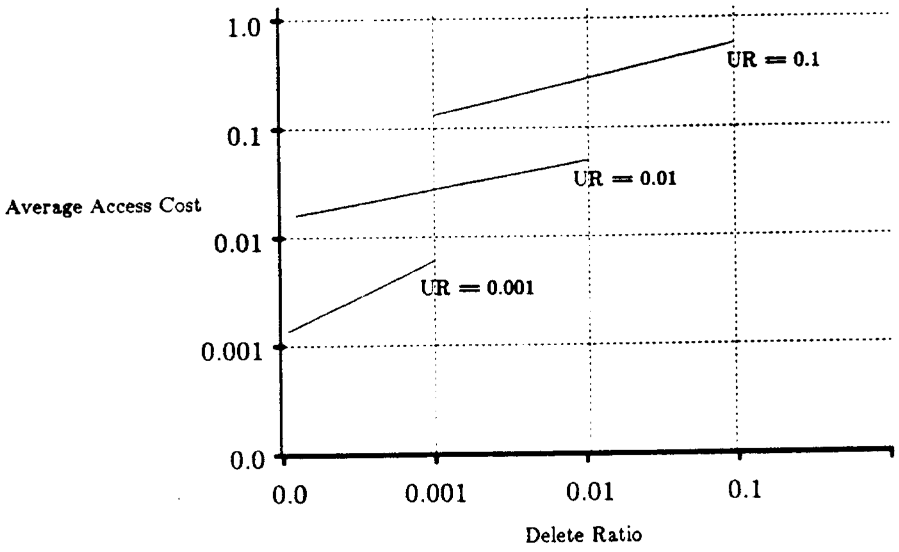


Figure 6.21: Increase in Per Access Cost for Deutsch-Bobrow

garbage collection will indicate the entire cost of a particular scheme. Dividing this total cost by the number of accesses will give the expected average cost of accessing an object when garbage collection cost is considered an access cost; henceforth, *average joint cost* refers to this conglomerated average cost.

The average joint cost of Mark&SweepC, BreadthFirstC, and Baker[7] varies depending upon how many complete scavenges occur in the analyzed interval. Figure 6.22 shows how the average joint cost varies for Mark&SweepC, BreadthFirstC, and Baker as the number of scavenges varies. BreadthFirstC and Baker have a U-shaped curve because the fewer times that compacting occurs, the higher the average access cost since objects get reclustered less often. The bottom of the U-shaped curve indicates when garbage collection should occur to minimize the average joint cost for the method. This occurs when 5 scavenges are done in

[7] As in section 4.2, only the least costly of each method is compared: specifically, Mark&SweepC, BreadthFirstC, Baker, ReferenceCount, LogRefCount, and Deutsch-Bobrow. DepthFirstC is left out since in the cases reported here, it performs equal to BreadthFirstC.

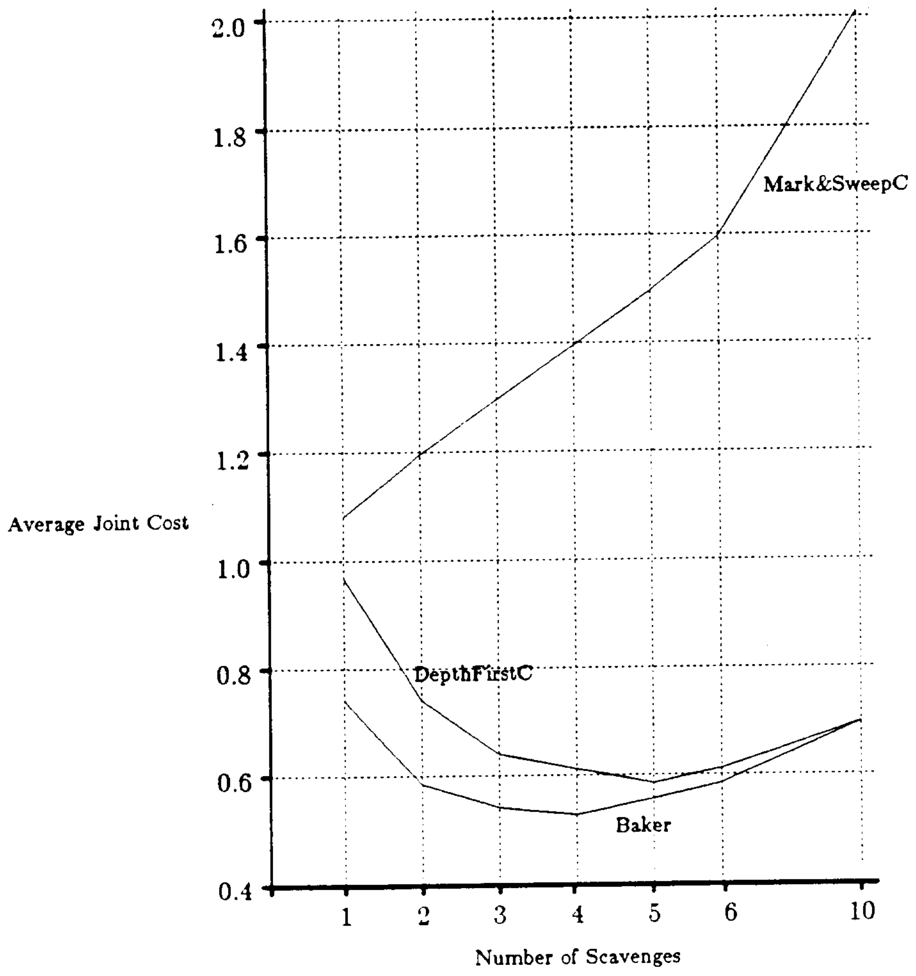


Figure 6.22: Average Joint Cost as Number of Scavenges Varies

BreadthFirstC, or, in other words, when 15% of the objects have been deleted; and, when 4 complete scavenges are done for Baker, or 20% of the objects have been deleted. To keep the average joint cost at a minimum, when BreadthFirstC or Baker is employed, they should be executed when 15% or 20% of the objects, respectively, have been deleted. Mark&SweepC has a lower joint cost the fewer times it is executed. Therefore, the optimal time to run it is dictated by how much disk-space can be allowed to be occupied by garbage. Figure 6.23 depicts the trade-off between the number of disk-pages that can be occupied by garbage and the average joint cost of Mark&SweepC. As the number of disk-pages allowed to be wasted on garbage increases, the less often Mark and Sweep needs to run, thereby causing a decrease in the average joint cost.

The bar chart in Figure 6.24 shows the average joint cost for Mark&SweepC, BreadthFirstC, Baker, ReferenceCount, LogRefCount, and Deutsch-Bobrow with varying change and delete rates over an interval of 30 million accesses. For comparison purposes, Mark&SweepC and BreadthFirstC are activated when 25% of the objects are deleted and Baker does a complete scavenge in the same interval. The first bar for each algorithm represents the cost when UR=DR=0. When no changes occur to the objects, no cost for garbage collection is incurred; therefore, this bar indicates just the cost of 30 million accesses, which is 0.05 for all

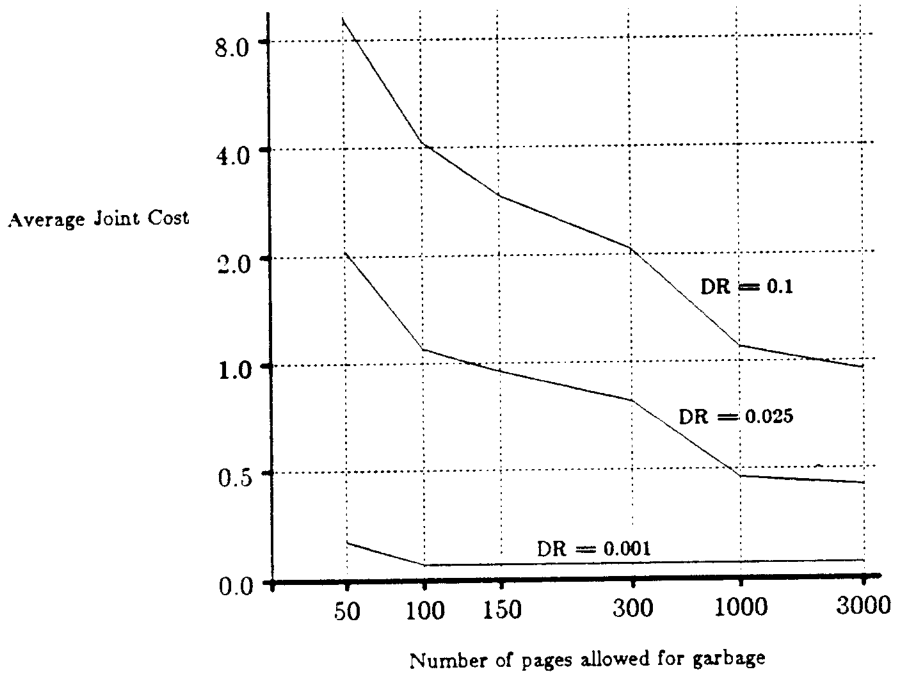


Figure 6.23: Pages of Garbage on Disk versus Average Joint Cost for Mark&SweepC

methods except Baker which, as explained in Section 4.3, may incur a 50% increase in access cost (an average joint cost of 0.075). The second bar shows the cost when the Update Ratio is 0.01 and the Delete Ratio is 0.0001. Due to the very low update and delete probabilities, the cost is not very high for any of the schemes. The Delete Ratio is not high enough to activate Mark&SweepA or BreadthFirstC during this interval. The third bar represents the cost when the Update Ratio is 0.1 and the Delete Ratio is 0.025. Mark&SweepC, BreadthFirstC, and Baker complete three scavenges of memory during this interval of 30 million accesses. The average joint cost of DepthFirstC and Baker is less than the other schemes due to the decrease in access cost that comes with reclustering the data. Baker has the least cost because it is constantly reclustering the data.

Baker seems to fit the criteria best: the pauses it causes can be very short since it is incremental; the average cost to access an object is fairly low; and, the total cost, as captured by the average joint cost, is lower than any other method. Even though Baker has the least average joint cost, it's cost is still high. Ignoring any cost increase due to loss of clustering, the base cost to access an object is 0.05; if the Baker method of garbage collection is employed, the cost increases to 0.53: a ten-fold increase. If the automatic storage reclamation could be done in the background (on a separate machine) or when the database is not being accessed, the cost of Baker would be greatly reduced. Since the reclamation of unreachable database objects is not critical, doing it during off-hours could be a big win. Baker could be easily adapted to run while the DBMS is idle.

6.5. Space requirements

Each algorithm's space requirements are dependent upon different data parameters. The bar chart in Figure 6.25 illustrates the expected number of 1Kbyte pages required for the various algorithm implementations for a structure of 1,000,000 objects with density 1.3, Height N/16, and RLRAtio 1.2. As with the I/O cost analysis, twenty objects can be stored

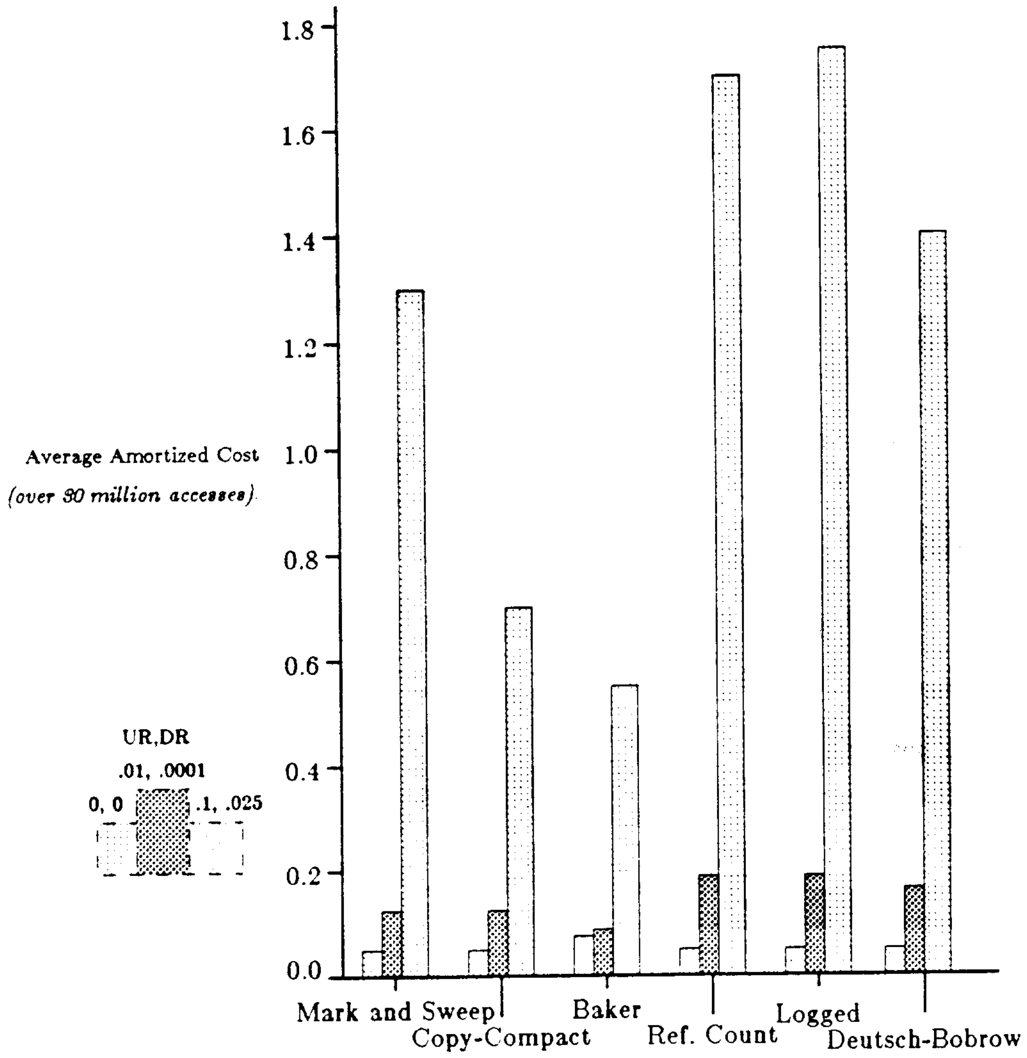


Figure 6.24: Cost Comparison with Varying Update and Delete probabilities

on a page. The left bar shows the number of disk pages needed in excess of the number used to store the objects once. The average amount of physical main memory required (if any), disregarding the DBMS buffer pages, is shown by the right bar. Mark&SweepA has the smallest disk space requirements: 1 bit per object. Deutsch-Bobrow needs to store one address (assume 4 bytes) for each multireferenced object. The Copy-Compact methods require the most space because two copies of the data exist for a short period of time. The amount of physical memory needed (the right bar) is critical for the performance of the implementations that assume temporary relations stay in physical main memory (i.e., Mark&SweepC, BreadthFirstC, and DepthFirstC). Mark&SweepC requires 5 bytes for each object, so 5000 pages are needed for 1 million objects. BreadthFirstC and DepthFirstC both require 2 pages of main memory.

In the original formulation of Copy-Compact, oldspace had to be left intact because objects in oldspace could still be referenced by other objects. When the data is stored in a database, the object identifiers are not physical addresses, as in the virtual memory version, but tuple-ids. As long as an object's tuple-id in copyspace is the same as its tuple-id in oldspace, the copy of the object in oldspace can be deleted. Using this reformulation, the

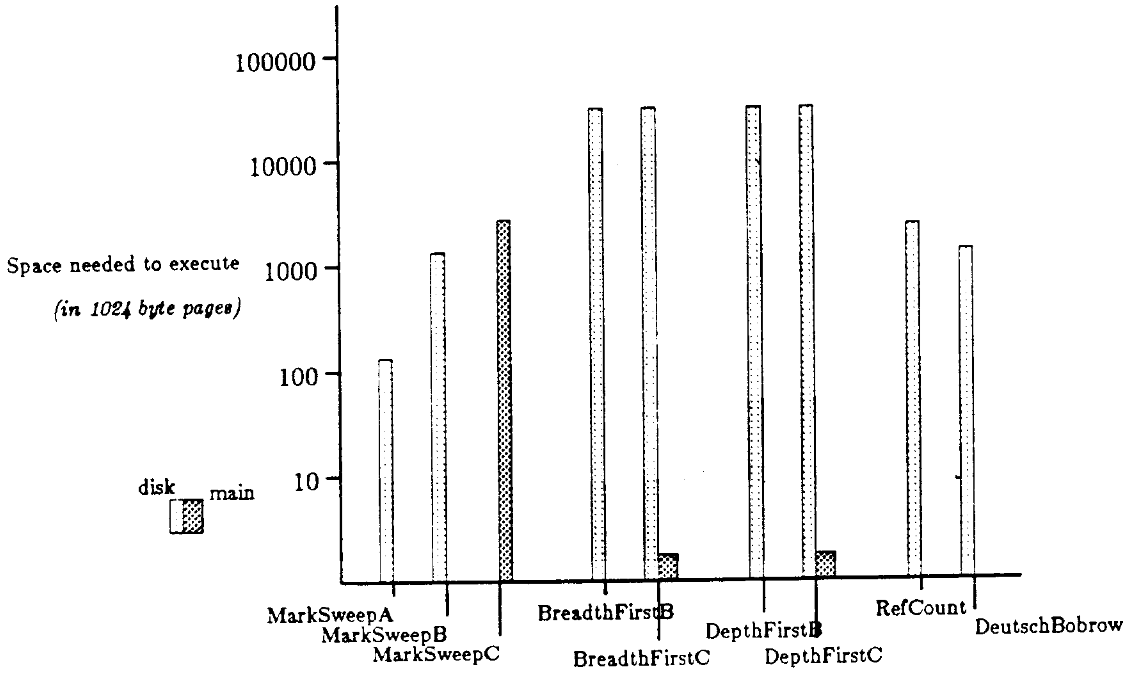


Figure 6.25: Space requirements

amount of disk space required by the BreadthFirst and DepthFirst schemes reduces to the amount needed to hold two iterations of objects: the objects whose pointers are being followed (those marked TRUE); and, the objects that are being retrieved (those marked FALSE). The average space needed for two iterations is approximately 4 pages for DepthFirst and 2 pages for BreadthFirst.

6.6. Conclusions

Many of the criteria for judging an automatic reclamation scheme are the same for virtual memory and disk-based schemes, such as:

> Causing long pauses in the program is bad;
> Scavenging incrementally is good;
> Allowing internal fragmentation is bad; and,
> Clustering improves performance.

The pauses caused by ReferenceCount and Deutsch-Bobrow are much shorter than those caused by Mark and Sweep or Copy-Compact, but the average joint cost per access of ReferenceCount is much higher than that of either of the Copy-Compact techniques. Breaking up Copy-Compact into pieces, as is done by Baker, should allow the pauses to be shorter and still maintain a low average joint cost.

The space requirements for the algorithms are dependent upon the facilities provided by the data manager. If temporary relations can be kept in physical main memory, the amount of space needed is reduced drastically. If temporaries must be treated like permanent relations, the I/O cost doubles and the disk space requirements are multiplied by ten.

A property that should be considered for choosing a storage reclamation scheme for a database is whether the cost of reclamation can be postponed to times when the database is idle. Mark and Sweep and Copy-Compact both have this characteristic, as does Baker and Logged Reference Count. If reclamation occurs at idle times, the I/O cost is one indication of how long the idle must last to complete the reclamation during this time. If the database gets a request for data during reclamation, the reclamation could pause until the database is idle again, or it could force the request to wait. The former is preferred so that the pauses realized by a user are minimized, but not all algorithms can be easily interrupted. Logged Reference Count and Baker may be easier to interrupt than Mark and Sweep. If Logged Reference Count or Baker was interrupted, at most K work would be lost; whereas, an interdeterminate amount of work could be lost if Mark and Sweep was interrupted.

An additional issue that needs to be examined is whether locking the entire database is demanded by the garbage collection algorithm. The current assumption is that the program cannot access the database during a scavenge. This restriction may not be necessary. If the storage reclaimer could run concurrently with the application, the worst possibility is that it could cause deadlock. A method like Baker which accesses only a subset of the tuples at a time is less likely to cause deadlock or to cause any lengthy delay to an application running concurrently.

The current technology for automatic storage reclamation demands long pauses when applied to disk-based data. Schemes need further exploration that reduce the pauses and reduce the I/O overhead on each access. The pauses caused when running the schemes on virtual memory are measured in fractions of seconds; the pauses that would be caused according to the I/O costs reported herein would be measured in hours. Pauses of this length are unacceptable to users and a hybrid scheme must be developed for disk-based data that is based on the characteristics of Baker and that shelters users from the cost of automatic storage reclamation.

CHAPTER 7

Concluding Remarks

I went to find a pot of gold
That's waiting where the rainbow ends
I searched and searched and searched and searched
And searched and searched and then--
There it was, deep in the grass,
Under an old and twisty bough.
It's mine, it's mine, it's mine at last....
What do I search for now?

Shel Silverstein, *Where the Sidewalk Ends*

Complex applications, such as AI and CAD applications, have data management requirements distinct from those of business applications that traditionally utilize DBMSs. The data is very rich in types and contains many interobject references. The handling of interobject references poses special problems to a data manager with regards to storage. Interobject references also invite certain features to be added to a system that can improve performance (e.g. prefetching and clustering).

In Chapter 3, the implementation of Polymnia, a system that couples Lisp and the relational DBMS INGRES, was detailed. Polymnia uses two fields in a relation to implement a *union-field* which can house any type of value, including an interobject reference. This technique is extendible to types not currently implemented in a very straightforward manner. The layered system architecture is very easy to extend with new features, but it costs in performance. The layer that exists between the DBMS and Lisp is not strictly necessary, but makes implementation and upkeep simpler. This layer has complex features for retrieving sets of cons cells, rather than just one at a time. This complexity results in cons cells being retrieved at half the rate of structures.

The most impressive part of Polymnia is the performance that was achieved. A very data intensive application slowed down by only a factor of four. The retrieval rate, as seen from Lisp, was greater than 35 structures per second (when each structure was stored in a single tuple). Using techniques described in Chapters 4 and 5 the delay caused by retrieving data from the database can be lessened.

It became clear in Chapter 4 that one reason the Lisp applications performed so well when using persistent data is that all data was cached: each structure was retrieved only once from the DBMS. Another point brought out in Chapter 4 is that prefetching data can be advantageous to performance. Since the tests were conducted on a dual processor machine, prefetching allowed the DBMS to operate in parallel with the application. The amount of time the application spent waiting for data to be returned from the DBMS was reduced by up to 30%. Of course, the performance improvement is dependent upon how accurate the oracle is that determines what to prefetch.

Chapter 5 looked at how clustering data can affect the performance of an application. Analysis showed that retrieval rates can be increased or decreased depending upon whether the placement of tuples matches the access pattern. Clustering in a manner that is very different from the access pattern can result in a retrieval rate slightly worse than if tuples were randomly placed. The performance attained when data is clustering depth-first was demonstrated to vary more depending upon the access pattern. Since a particular kind of depth-first clustering (CDR-coding) is supposed to be efficient for Lisp structures, this form was imposed upon the DBMS data to see how this affected retrieval rates. The data was CDR-coded by placing more than one cons cell into a single tuple. CDR-coding also allows data to be compacted; this turned out to be one of the critical factors. When using a non-compacting structure, the data, when clustered eight cons cells per tuple, took up twice as many bytes as when stored one cons per tuple. Hence, the application spent 10% more time waiting for data from the DBMS. When a compacting structure was used and eight cells were stored per tuple, the data took up 15% fewer bytes; hence the application spent 45% less time waiting for data. The reason so much time was saved is that when the data is clustered fewer queries need to be run to retrieve the data. When the data was clustered eight to a tuple, the number of queries executed was reduced by 60%.

Chapter 6 examined virtual memory storage reclamation techniques to see how they would perform when running in a DBMS environment. If the algorithms had to run as a regular database application using naive query processing, none of the algorithms would be usable -- they would degrade performance by too much. If global query optimization was used, or the garbage collectors were written not in a query language, the best algorithm would degrade performance by a factor of ten if it ran while an application is running. The ideal reclamation scheme is one that could execute while no application is accessing the data, or could run in parallel without affecting the application's performance. Baker's algorithm, running in the background, would be the best reclamation scheme. It would do a little reclamation at a time, allowing an application to interrupt it if the application needed data. Another result of this study was that algorithms tuned for a particular data structure perform poorly on most data; for example, the Depth-First algorithm which was designed to run well on data that had many cells connected via *next* pointers, had worse performance than the Breadth-First algorithm on all data except the extreme case of almost all cells connected via *next* pointers. In conclusion, an algorithm should be able to run in the background and do a small amount of reclamation at a time. In addition, a simple algorithm is better than a complex one.

A prototype system was built and tested using many different features, most of which proved to be beneficial under most circumstances. This work supports the thesis of this dissertation, which is:

a traditional DBMS coupled with a language interface provides the functionality necessary to support interobject references without unreasonably degrading an application's performance.

The architecture of Polymnia is similar to that of Iris and Postgres, yet Polymnia provides the transparency of GemStone and PVM. Polymnia has shown that a system built on top of a general-purpose DBMS can provide the same functionality as a system built using a special purpose data manager. In addition, the performance of applications using Polymnia did not degrade unreasonably. The thesis has been shown to be correct: a single layer between the DBMS and the application language can understand the semantics of the application language and that of the DBMS so that any desired features can be implemented.

The drawback of taking the approach used in Polymnia is that multiple applications will end up implementing very similar things. Just using programming languages as examples, a SmallTalk specific system would need a storage reclaimer and so would a Lisp system. To avoid this sort of duplication, tools that implement these services could be provided by the DBMS if the DBMS supplied a standard form of interobject references that the tools could interpret correctly no matter what application the data belonged to.

Many issues are still to be researched:

- The major issue not yet addressed by Polymnia is that of sharing data among applications. The systems that have supported sharing transparently have done so at a coarse granularity (e.g. both POMS and OM lock an entire heap). Other systems, such as Iris and the Postgres object interface, support sharing at the object level by allowing the application to explicitly set locks. Transparently providing object-granularity sharing has not been explored. The open questions include: what would a transaction be? what guarantees could an application be provided? and, how could aborts be handled transparently? If applications do not explicitly begin and end transactions, an application designer must not be affected by an abort. The software that provides the transactions must also provide transaction restart. An application designer needs to understand what data alterations are guaranteed and how other applications' actions can affect the data his application is acting upon.

- In Chapter 3, it was pointed out that updating is much more costly than reading. Facilities need to be developed that reduce the performance delay an application experiences when it updates data. One possible technique would be to batch a collection of updates together and perform them in the background. The interaction of delayed updates and sharing needs to be studied. If an update is delayed, other applications accessing the updated data will not realize an update has occurred at the same point the application making the change sees it. Therefore, different applications may see incompatible versions of the database. Also, how delayed updates and aborts affect each other should be examined. If an application updates an object, but the update to the database is delayed and before the update is made the application aborts, does the update still take place? Consistency issues such as these need to be addressed.

- Since automatic storage reclaimers have been so thoroughly analyzed, one should be built. Implementation techniques need to examined to see how to attain the optimal performance. Also, how the storage reclaimer can operate in parallel with applications should be studied.

- Chapter 4 showed that a successful prefetch algorithm can improve performance; but, it also showed a poor algorithm can hurt performance. An intelligent prefetcher should be designed and implemented that dynamically predicts what will be accessed next based upon past references. The predictors used in Chapter 4 were static: if a particular prefetch scheme was used, whether or not it was predicting correctly, it kept making the same prediction. If an oracle were designed such that when it made a prediction that was not successful, it did not predict the same way again, the oracle would customize itself for each application's access pattern. (That is, if it predicted *first* and *next* would be needed, but only *next* was needed, it could change so that next time only *next* was prefetched.)

The new application areas for which the coming generation of database management systems are trying to provide (e.g. AI and CAD/CAM) require interobject references. Polymnia demonstrates how interobject references can be implemented using a conventional DBMS. Interobject references can be used to benefit performance by clustering and prefetching. When interobject references are stored in a DBMS, reclaiming unreachable objects becomes an issue. Prefetching, clustering, and storage reclamation can be performed by the DBMS if it understands interobject references. This dissertation explored the implementation of these features outside of the DBMS because the DBMS did not provide a referential data type. This technique allows an application to be sheltered from the DBMS, but it also has a performance cost. In conclusion, although interobject references, prefetching, clustering, and storage reclamation can be implemented outside of a DBMS, if these features are provided by a data management system applications may be easier to build and better performance will be attained.

REFERENCES

[Allm76]
E. Allman, G. Held, and M. Stonebraker.
Embedding a data manipulation language in a general purpose programming language.
Proceedings of the 1976 ACM-SIGPLAN-SIGMOD Conference on Data Abstraction, Salt Lake City, Utah, March 1976, pages 25-35.

[Astr76]
M. Astrahan.
System R: a relational approach to database management.
Transactions on Database Systems 1, 2 (June 1976), 97-137.

[Atki82]
M. Atkinson, K. Chisholm, and W. Cockshot.
PS-Algol: an algol with a persistent heap.
SIGPLAN Notices 17, 7 (1982), 24-31.

[Bake76]
H. Baker.
List processing in real time on a serial computer.
Massachusetts Institue of Technology, Cambridge, Massachusetts, Ocober 1976.

[Banc86a]
F. Bancilhon and S. Khoshafian.
A Calculus for complex objects.
Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, Cambridge, Massachusetts, March 1986, pages 53-59.

[Banc86b]
F. Bancilhon and R. Ramakrishnan.
An amateur's introduction to recursive query processing strategies.
Proceedings of ACM SIGMOD , Washington, DC, May 1986, pages 16-52.

[Bato86]
D.S. Batory.
GENESIS: A project to develop an extensible database management system.
Proceedings of the International Workshop on Object-Oriented Database Systems, Pacific Grove, California, September 1986, pages 207-208.

[Bobr79]
D. Bobrow and D. Clark.
Compact Encodings of List Structure.
ACM Transactions on Programming Languages and Systems 1, 2 (October 1979), 267-286.

[Bobr86]
D. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel.
CommonLoops: merging Lisp and object-oriented programming.
Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings, October 1986, pages 17-29.

[Butl86a]
M. Butler.
An Approach to persistent LISP objects.
Proceeedings of the Thirty-First IEEE Computer Society International Conference, San Francisco, California, March 1986, pages 324-329.

[Care86]
M.J. Carey, D.J. DeWitt, D. Frank, G. Graefe, M. Muralikrishna, J.E. Richardson, and E.J. Shekita.
The Architecture of the EXODUS extensible DBMS.
Proceedings of the International Workshop on Object-Oriented Database Systems, Pacific Grove, California, September 1986, pages 52-65.

[Cock84]
W. Cockshot, M. Atkinson, K. Chisholm, P. Bailey, and R. Morrison.
Persistent object management system.
Software-- Practice and Experience **14** (1984), 49-71.

[Cohe81]
Cohen.
Garbage collection of linked data structures.
Computing Surveys **13**, 3 (September 1981).

[Cope84]
G. Copeland and D. Maier.
Making Smalltalk a database system.
Proceedings of SIGMOD 1984, Boston, Massachusetts, June 1984, pages 316-324.

[Denn72]
P. Denning.
On modeling program behavior.
Proceedings of the Spring Joint Computer Conference **40** (1972), 937-944.

[Deut79]
L. Deutsch and D. Bobrow.
An efficient incremental garbage collector.
Communications of the Association for Computing Machinery **19**, 9 (September 1979), 522-526.

[Fish87]
D.H. Fishman, D. Beech, H.P. Cate, E.C. Chow, T. Connors, J.W. Davis, N. Derrett, C.G. Hoch, W. Kent, P. Lyngbaek, B. Mahbod, M.A. Neimat, T.A. Ryan, and M.C. Shan.
Iris: An object-oriented database management system.
Transactions on Office Information Systems **5**, 1 (January 1987).

[Fode83]
J. Foderaro, K. Sklower, and K. Layer.
The FRANZ LISP Manual.
Department of Electrical Engineering and Computer Science, University of California at Berkeley, June 1983.

[Gold83]
A. Goldberg and D. Robson.
Smalltalk-80: The Language and its Implementation.
Addison-Wesley, 1983.

[Laru83]
J.R. Larus.
An Interactive Program Analysis System for Franz Lisp.
Masters Report, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, December 1982.

[Lieb83]
H. Lieberman and C. Hewitt.
A Real-time garbage collector based on the lifetimes of objects.
Communications of the ACM 26, 6, 419-429.

[Lind87]
B. Lindsay, I. McPherson, and H. Pirahesh.
A data management extension architecture.
Proceedings of the ACM SIGMOD Annual Conference, San Francisco, California, May 1987, pages 220-226.

[Lori83]
R. Lorie and W. Plouffe.
Complex objects and their use in design transactions.
Proceedings of Engineering Design Applications of ACM-IEEE Data Base Week, San Jose, California, May 1983.

[Maie86]
D. Maier.
Why object-oriented databases can succeed where others have failed.
Proceedings of the International Workshop on Object-Oriented Database Systems, Pacific Grove, California, September 1986.

[McEn86]
T.J. McEntee.
Overview of garbage collection in symbolic computing.
Texas Instruments Engineering Journal 3, 1 (January 1986), 130-139.

[Mins63]
M. Minsky.
A Lisp garbage collector using serial secondary storage.
M.I.T., Cambridge, Masachusetts, Memo 58, Project MAC, December 1963.

[Mish84]
N. Mishkin.
Managing Permanent Objects.
Department of Computer Science, Yale University, New Haven, Connecticut, PhD Thesis, 1984.

[Obri86]
P. O'Brien, B. Bullis, and C. Schaffert.
Persistent and shared objects in Trellis/Owl.
Proceedings of the International Workshop on Object-Oriented Database Systems, Pacific Grove, California, September 1986, pages 113-123.

[Pren78]
C. Prenner and L. Rowe.
Programming languages for relational database systems.
Proceedings of AFIPS 1978 National Computer Conference **47**, 849-855.

[Rowe86]
L.A. Rowe.
A shared object hierarchy.
Proceedings of the International Workshop on Object-Oriented Database Systems, Pacific Grove, California, September 1986, pages 160-170.

[RTI83]
Relational Technology Inc..
INGRES Manual, Berkeley, California, 1983.

[Ship81]
D. Shipman.
The functional data model and the language DAPLEX.
Transactions on Database Systems **6**, 1 (March 1981), 140-175.

[Smit82]
A. Smith.
Cache Memories.
Computing Surveys **14**, 3 (September 1982), 473-530.

[Ston81]
M. Stonebraker, E. Wong, P. Kreps, and G. Held.
The design and implementation of INGRES.
Transactions on Database Systems **1**, 3 (September 1976), ACM.

[Ston83]
M. Stonebraker.
Applications of abstract data types and abstract indices to CAD applications.
ACM SIGMOD International Conference on the Management of Data, San Jose, CA, May 1983.

[Ston86a]
M. Stonebraker and L. Rowe.
The design of POSTGRES.
Proceedings of ACM SIGMOD International Conference on Management of Data, Washington, DC, May 1986, pages 340-355.

[Ston86b]
M. Stonebraker.
Object management in Postgres using procedures.
Proceedings of the International Workshop on Object-Oriented Database Systems, Pacific Grove, California, September 1986.

[That85]
S. Thatte.
Persistent Memory for Symbolic Computers.
Texas Instruments Technical Report 08-85-21, Dallas, Texas, July 22, 1985.

[Unga85]
D. Ungar.
Generation scavenging: a nondisruptive high performance storage reclamation algorithm.
Proceedings of Practical Programming Environments Conference, Pittsburgh, Pennsylvania, April 1984.

[Wile84]
R. Wilensky, Y. Arens, and D. Chin.
Talking to UNIX in English: an overview of UC.
Communications of the ACM 27, 6 (June 1984), 574-593.