DEVELOPING BENCHMARKS FOR COMPARING RELATIONAL AND OBJECT-ORIENTED DATABASE SYSTEMS

Becky L. Lakey

B.A., Whitman College, 1975

A thesis submitted to the faculty
of the Oregon Graduate Center
in partial fulfillment of the
requirements for the degree
Master of Science
in
Computer Science and Engineering

July 31, 1989

The thesis "Developing Benchmarks For Comparing Relational and Object-Oriented Database Systems" by Becky L. Lakey has been examined and approved by the following Examination Committee:

Jacob Stein
Adjunct Assistant Professor
Thesis Research Advisor

David Maier
Professor

Robert Babb
Associate Professor

Denise Ecklund
Software Engineer
Mentor Graphics

This thesis is dedicated first and foremost to my family. Without their support, encouragement, and prodding, I would still be working in a laboratory wondering if there wasn't something else in life. Now, I know there is and I like it.

I also dedicate this thesis to my advisor, Jacob Stein, who did not get too discouraged with his first graduate student, but kept me going with encouragement and help. He also understood how finishing a thesis can sometimes take a long time!

I wish to acknowledge the following people for their help and support while I was involved in preparing and writing my thesis: Dave Maier (the Edit King), Robert Babb, Denise Ecklund, and the engineers at Servio-Logic Corporation. In addition, many thanks go to Frank Binns and Kea Grilley for allowing me the time to complete this thesis.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

Various business and non-business applications, such as office information systems, hypertext, and VLSI, are incorporating databases for data management support. Many of these applications model entities considerably more complex than the simple data structures relational database systems were originally developed to handle. Proponents of object-oriented technology believe that the object-oriented paradigm is better-suited for those applications requiring complex objects.

Eventually, benchmarks will be needed to compare and evaluate existing and future relational and object-oriented database management systems. We discuss what criteria should be met when selecting a suitable benchmark and the issues and problems to be addressed during implementation. After selecting our benchmark applications, we defined the entities, operations, and data at the conceptual level and determined what database systems we would test. We present the benchmarks selected, discuss how we implemented these benchmarks, and present our results. In addition, we offer our views on the ease of implementation and use of each database system, including maintenance and reusability.

CHAPTER 1

INTRODUCTION

Currently, various business and non-business applications are incorporating databases for data management support. In some of these applications, e.g., VLSI design [CFH83], office information systems [SSL83], and hypertext [Mey86], the entities modeled are considerably more complex than the simple data items relational database systems were originally developed to handle. The values referenced by these complex entities may be other simple or complex entities. These complex entities can be difficult to represent in the two-dimensional relational model, and generally require encoding to coerce each entity's structure into a flat view, or alternately, over-simplifying the application model to fit it to a relational database scheme. Proponents of object-oriented technology believe that the object-oriented paradigm is well-suited for those applications requiring complex objects.

Eventually, the need will arise to compare and evaluate existing and future relational and object-oriented database management systems. Consequently, performance benchmarks for comparison and evaluation need to be developed.

The research presented in this thesis proposes and discusses various benchmarks that would address the need above. The major issues we consider are what and how a benchmark should measure and the necessary criteria that must be included in a reasonable database management system benchmark. Five possible benchmarks are proposed and the two implemented are discussed in detail. We primarily focused on available relational and object-oriented database management system, but we feel the benchmarks could be used with all types of database management systems.

1.1 CHARACTERISTICS OF RELATIONAL AND OBJECT-ORIENTED SYSTEMS

Relational Systems

The relational model, utilized by the Ingres[1] database management system, consists of a fixed set of basic types (integers, characters, and strings) and a fixed set of operations on those types (arithmetic, comparison) [MSO86]. The model also has a predefined set of type constructors, i.e., tuple and relation, that can be manipulated by a fixed group of operations (insert field, retrieve tuple, modify relation). All tuples in a relation must be homogeneous, with each tuple containing identical attributes and each attribute's type restricted to the same fixed type.

In the relational model, it is not always possible to model the real-world

[1]Ingres™ is a trademark of Relational Technology, Inc.

structure of an entity directly. Attempting to do so results in either an over-simplified database scheme or encoding the scheme to mimic the entity structure, thereby necessitating decoding by an application program. The relationship between entities and subentities is represented using fields that contain keys for other relations. These foreign keys are often difficult to maintain and are not always associated with the real-world structure of an entity. The consistency of these foreign keys, assuring that each foreign key is the key of an actual tuple, is known as *referential integrity*.

Sometimes, it is difficult to coerce the entity's real-world representation into the flat view required by the relational model. Generally, a designer attempts to maintain the database in "normalized" form, which requires decomposing real-world entities into subentities and complicates application code.

The data manipulation languages found in conventional database systems do not support general computations. Consequently, entities must be passed through an interface to a general-purpose programming language that will perform the computations. In most instances, one language is embedded within the other; an application is written in the programming language and includes calls to the database system for data manipulation. With this type of arrangement, *impedence mismatch* (or structural mismatch), where there is a loss of information when data is transferred between two structurally and semantically different languages, often ensues.

Object-Oriented Systems

The object-oriented model, as exemplified by Smalltalk-80[2] or GemStone[3], is comprised of three main concepts, **object, message**, and **class**. An object is analogous to a tuple. Most objects are divided into instance variables, similar to a tuple's attributes. Each instance variable contains a value, which is another object. Certain objects cannot be decomposed further. These are considered *atomic* objects and include SmallIntegers, Characters, and Booleans.

Objects communicate through messages. These messages generally are requests to change an object's state or return a value. Every object has a standard set of messages it responds to (its *protocol*). A *method* is associated with each message denoting how to perform the request. An object can only respond to messages associated with that object. The instance variables of an object cannot be directly accessed from any method outside of those methods defined for that object.

A class defines a collection of objects sharing the same internal structure and methods. Each object is an *instance* of its class. Since all instances of a class share the same methods, "any difference in the response by two instances is determined by a difference in the values of their instance variables" [StB86] and the message arguments passed.

Classes are arranged hierarchically; a class's format and methods are automatically inherited by instances of any class defined as its subclass (i.e., a subclass

---

[2]Smalltalk-80™ is trademark of ParcPlace Systems.

[3]GemStone™ is a trademark of Servio-Logic Corp.

inherits the properties of its parent and the parent's ancestors). A subclass is an extension of its superclass that may introduce new instance variables and methods and may redefine (or block) the behavior of its superclass.

In contrast to the relational model where the user is restricted to a fixed set of data types and operations, object-oriented systems enable the user to define new data types (classes) and new messages associated with those types. The object's behavior is thus encapsulated by binding its structure to the set of operations associated with it.

In object-oriented systems, an entity's real-world structure can be modeled more directly; an entity is represented as a single object rather than multiple tuples spread through multiple relations. Consequently, complex entities may be modeled with less encoding. Most object-oriented systems also support set-valued entities, where the set elements are not restricted to homogeneous elements, but can be of arbitrary type.

Object-oriented systems incorporate one language that can be used for data querying and manipulation, general computation, and system administration. This language is used for all computations in an application, thereby eliminating the problem of impedence mismatch.

1.2 RELATED WORK

Most literature concerning database system performance analyses focuses on relational database systems. It is still too early in the development of object-oriented database systems to find many articles that describe their performance.

Wisconsin Benchmarks and TP1

Bitton, et al., designed a systematic approach for benchmarking relational database systems, commonly referred to as the "Wisconsin Benchmarks" [BDT83]. Their goal was to "develop a scientific methodology for performance evaluation of database management systems" [BDT83].

Five relational systems were benchmarked: DIRECT, Britton-Lee IDM/500, 'university' Ingres on a VAX 11/750 under UNIX 4.1[4], 'commercial' Ingres on a VAX 11/750 under VMS[5], and Oracle. The database consisted of four relations, each populated using random number generators and containing a different number of tuples. The number of attributes in each relation were the same. Various relational operations (i.e., selection, projection, join, aggregates, and updates) formed the set of queries, which measured the cost of these operations. The benchmarks were executed in the single-user mode; elapsed time was the primary measurement of performance.

After analyzing the benchmark results, the group concluded that the benchmark's primary limitation was that of testing the systems in single-user mode. Even though the only means of determining the effects of a system's hardware design, operating system features, and query execution algorithms was to execute the benchmark in a stand-alone environment, the designers felt that the testing situation did not realistically approach a multi-user environment. The group also concluded that the Wisconsin

---

[4]UNIX™ is a trademark of AT&T.

[5]VMS™ is a trademark of Digital Equipment Corp.

benchmarks did not exhaustively compare the various systems tested.

The TP1 benchmarks [Ano85] measure transaction throughput and are targeted towards numerous, short transactions. We speculate that many applications developed with object-oriented database management systems, especially for computer-aided design, will make use of fewer and longer transactions.

Benchmarks For Relational and Object-Oriented Systems

Another group of researchers developed a set of benchmarks which "would measure the response time performance from a database system for simple queries" [RKC87]. The set of benchmarks were designed to be independent of the data model presented by a particular system; the group strove to come up with data that could be presented in both relational and object-oriented systems. The benchmarks execute simple operations on single objects or records (as compared to complex operations on objects spread over several records). Response time was defined as the real time that elapsed between when a program invoked the database system with a specific query, and when the result was returned.

The benchmarks included the following operations: Key lookup (referred to as Name Lookup), range lookup, group lookup (equivalent to using the value of a foreign key for selection), single-step path traversal (referred to as Reference Lookup), class insertion (class is analogous to a relation in their benchmarks), sequential scan of all instances of a class, and database open. Since these benchmarks were intended to be independent of the database systems, the design group did not specify queries using a

specific syntax (e.g., SQL), but gave an explanation of what they intended in each operation. All of the benchmarks were implemented on Sun Microsystem's two database systems, Unify[6] and Ingres, and were executed on Sun workstations with local databases. Within Unify, the user would define the data in an object-oriented model, which was then mapped and stored relationally. The benchmarks were also run on a modified version of Unify, Rad-Unify, that uses a large main memory cache. Each database system was also tested on a database located remotely on a network of workstations.

The researchers concluded that compiling relational queries or allowing a programmer to access a local database at the single-table level (avoiding the query processor) could increase the speed by a factor of five to ten. Also, if much of the database could be cached in the workstation's main memory, execution would also be speeded up. Remote access resulted in approximately 30 milliseconds extra per record transferred (and per operation invoked in Unify) and more in Ingres (exact numbers were not reported). Finally, the authors felt that additional optimization techniques might generate a faster database system.

The analyses above compared various database systems with no hardware or software extensions made to the systems. The next two performance analyses address the effect of hardware and software configurations on database management system performance.

---

[6]Unify[R] is a registered trademark of Unify Corporation.

Dividing Ingres Between Two Computers

Hagmann and Ferrari proposed dividing a database system's software into a front-end and a back-end and implementing the back-end on a second computer [HaF86]. They specified that the back-end was to have no specialized hardware or an operating system finely-tuned for database operations. Benchmarks were run against six different configurations. They decided to use the Ingres database system running under a predecessor of UNIX 4.2 The front-end would reside on a VAX 11/750, while the back-end would be on a VAX 11/780. Research Ethernet and TCP/IP network and transport protocol were the facilities for data transferral.

The six configurations consisted of the following:

Ingres

> This was the existing, nondistributed Ingres database management system.

Smart Disk

> All of the file system functions were transferred to the back-end.

In the rest of the configurations, the file system functions remained in the back end.

Access Methods

> The access methods of the database system were converted to run on the back-end. This included the software that gets, replaces, finds, inserts, and deletes tuples.

Inner Loop

> The software that was responsible for nontrivial queries (i.e., those queries where the database system had to read and process more than one record) was transferred to the back-end.

Decomposition

> In this configuration, the query was parsed and validated in the front-end and then sent to the back-end for full execution.

Parser

> The back-end executes all the database processing, including the query processing. It returns a data stream or an error message to the front-end.

Moderately complex queries were executed against two different database schemes, in order that the results generated would be less dependent on a particular database.

The authors found that no configuration uniformly performed better than another; depending on the performance criteria selected as most important, each configuration could be considered to have performed the best. Also, each configuration utilized varying amounts of resources. **Ingres** used the fewest total CPU cycles, since all decisions were made on one machine. The **Smart Disk** configuration did not perform as well as the authors anticipated; they found that it performed suitably for a small amount of buffering, but as the buffering load increased, the performance factor did not. The **Access Methods** and **Inner Loops** configurations were concluded to be unsuitable for relational database systems. In the **Access Methods**, the network load was the heaviest; the **Inner Loop** had the best CPU usage distribution between the ends, but it was felt that the coordination effort needed between the two ends made this configuration impractical. The **Decomposition** and

Parser configurations would only be practical in relational database systems because they were dependent on the specific relational system used.

Hardware Improvements to Database Machines

Rather than altering the software portion of the database manager system, Dewitt and Hawthorn proposed using relatively inexpensive computer hardware to improve the system's performance [DeH81]. To this end, they specified five *generic* classes of database machine architectures:

CS - conventional system

Database system manager runs on a single processor.

PPT - processor-per-track system

This architecture processes selection operations 'on the fly'. It includes a mass storage device consisting of a large number of cells. The storage device is connected to a global data bus which transmits selected tuples to the host processor.

PPH - processor-per-head system

In this architecture, each head of a moving-head disk includes associative processing logic.

PPD - processor-per-disk system

A processor (or a set of processors) is situated between a standard disk and a memory device where selected tuples will be transferred. The processor operates as a filter to the disk and forwards only those tuples that match the selection criteria.

MPC - multiprocessor cache system

The architecture in this class is composed of a small set

of general purpose processors and a three-level memory hierarchy.

The last four classes are connected to a host processor which accepts and compiles the queries and assists in executing specific complex queries that the back-end architecture is unable to handle.

DeWitt and Hawthorn benchmarked these five architecture types on selection and join operations and aggregate functions. These benchmarks measured the *total* system workload necessary to process a query, not the elapsed time for the machine to process the query. The benchmark tests on the selection operation demonstrated that the **CS** and **PPD** architectures performed very well when an index existed on the attribute being qualified. Therefore, if an index was always maintained, these designs were the most cost-effective when processing selective queries. But, if indexes were not maintained on the selected attribute, they found that the **PPH** design was the better performer. In the join operations, the authors discovered that the **PPD**, **PPH**, and **PPT** architectures performed terribly. They concluded that if an architecture design executed faster on only selection operations, it was better to ignore the back-end and take the **CS** approach, which was to perform a sort-merge join on the host processor. For complex queries, it was seen the **MPC** design executed the fastest.

IRIS

The IRIS Database Manager System, Version 2.0, is an object-oriented prototype being developed and tested at Hewlett-Packard [FBC86] [LDF86]. The aim of this database system is to provide "generalized database support for a wide variety

of applications" [LDF86]. The interfaces provided to IRIS's DBMS are designed for specific object-oriented languages and will enable various applications coded in different programming languages to share information and transparently access persistent objects.

The Hewlett-Packard group developed a benchmark which compared their IRIS prototype to their relational database system, Allbase. Both IRIS and Allbase are built on top of the same relational storage manager. The benchmark database consisted of five relations that included between three and five attributes. No performance data was given by the authors; they stated that generally, IRIS performed two to four times slower than the relational system. There were some queries and updates where IRIS performed at the same speed as Allbase, though.

Lyngbaek, et al. [LDF86], admitted that their schema did not test the entire range of IRIS's capabilities; the sample set of queries did not cover derived functions or the inheritance mechanism. They also noted that part of the query set included updates and disjunctive queries which IRIS was unable to handle at the time of testing. But, the group was encouraged and felt that IRIS had demonstrated the potential to become competitive with commercial relational systems. Since the prototype is still very new, it hasn't the performance fine-tuning of more mature relational systems.

The intention of this section was to share with the reader some of the work that has been done in benchmarking relational and object-oriented database systems. Researching this area has demonstrated that, while there is little trouble obtaining literature on relational database system benchmarks, limited information exists on

object-oriented database systems. This is most likely because object-oriented database systems are still very new and the systems being developed are still in the prototype and testing stages.

1.3 MEASUREMENT CRITERIA

Three characteristics differentiate benchmarking object-oriented database systems and traditional record-based systems (e.g., relational database management systems). One, the newer applications where object-oriented database management systems are applied involve modeling more complex entities. Definition of these entities is generally in conceptual terms rather than the physical or logical terms normally used in relational database management systems. This affects those applications where an entity's fields may contain values of different types. In relational database management systems, a foreign key field can only reference a single relation, in which all tuples are of the same type. Object-oriented database management systems allow fields to reference various types of entities. The benchmark therefore should include path traversal where entity types are unknown in advance or aren't restricted to be of a specific type.

Two, in object-oriented database management systems, the process of moving from the conceptual model to the execution model is handled by one language. Operations that may entail numerous calls to complete in a relational database management system can often be completed via one call in an object-oriented database

management system. While traditional benchmarking methods tend to ignore the substantial body of programming language code needed for an application, this may not represent the object-oriented database management systems fairly, as they combine data definition, data management, and general computation into a single language. The benchmark must therefore also address this issue.

Finally, traditional benchmarks measure performance in terms of elapsed response time to query completion and transaction throughput. Ease of implementation and use, maintenance, and reusability are generally not considered. As applications become more complex, these characteristics may become more critical, e.g., the user may be willing to trade performance for ease of maintenance. Object-oriented database management systems address the need for easier implementation, maintenance, and reusability. Therefore, benchmarks for object-oriented database management systems should emphasize measuring the complete application process, from development to maintenance, not just the execution speed.

In selecting benchmark applications for object-oriented and relational systems, there are many issues and problems that must be considered. Finding an application that is easy to populate may not be a simple process. The application may be ideal for benchmarking purposes, but populating it with data may be difficult. Many applications managed by object-oriented systems are characterized by multiple connections between objects. Randomly generating data exhibiting realistic connectivity and distribution patterns may be difficult or unachievable. Therefore, the benchmark must be easy to populate with realistic data.

The data in an application may vary greatly in its degree of data-sharing,

nesting depth, and entity size. A hypertext application may exhibit a high degree of data-sharing and arbitrary depth, while a text application may demonstrate a low degree of data-sharing and a bounded depth. Database management systems should consequently be able to manage trees, DAGs, and cycles. In addition, an application may contain data that is not necessarily stable, e.g., one update could modify multiple entities, or one field could reference more than one entity. Database management systems will vary considerably in their ability to manage complex structures and the various operations upon these structures.

The complex real-world structures generally found in the newer applications may be captured as single data items in object-oriented systems, whereas relational systems may require referencing multiple relations to access the entire structure. Additionally, in relational systems, manipulating complex structures may require complex programming language code since the user is restricted to a fixed set of operations and complex operations are implemented by embedding these operations in the programming language code. Therefore, one conceptual operation may require multiple database calls for execution. In object-oriented systems, the user has the capacity to define new operations, where the operations may execute on a single object, a set of homogenous objects, or a set of heterogenous objects.

An application developed on one relational system can easily be ported to another relational database management system with little or no revision. The same cannot be said of applications developed on an object-oriented system. Currently, there is no single formal model for object-oriented systems. The object-oriented schema and methods implemented on one system are not necessarily compatible with

those on another object-oriented system. In particular, object-oriented systems can differ in their deletion techniques (implicit versus explicit), indexing techniques (collections versus classes), and inheritance (single versus multiple). Consequently, considerable revisions to the schema and methods may be necessary before an application developed on one object-oriented database management system could function on another object-oriented system.

As discussed previously, the object-oriented systems have one language that covers data definition, management, and computation, in contrast to relational systems, where there are separate languages for data definition and computations. Therefore, what may conceivably be simple and time-saving to accomplish in an object-oriented system is not necessarily so in a relational system.

Finally, it can be argued that the schema and operations implemented in the benchmark may not be optimal and better results might have been generated using different schemas and methodology. It was not our intention to present our implementations as absolute, but to design and develop benchmarks that could be used as starting points and references to evaluate and compare various object-oriented database management systems and relational database management systems.

1.4 RESEARCH GOAL

Our goal in this thesis research is to develop benchmarks that can be used to compare relational and object-oriented database management systems, where the

methodology is not biased toward either relational or object-oriented systems. The benchmarks were implemented on two different database systems, Ingres and GemStone, due to their ready availability. Because time was a factor in this study, we did not concentrate on matching the operating environments in both database management systems; therefore, our conclusions are based on the relative performance within each system.

Each benchmark emphasized an application incorporating complex entities. For each benchmark, we first determined a conceptual model for the complex entities used in the application and a set of operations that were to be executed in the benchmark. Additionally, due to the time constraint within this study, we required real-world applications with pre-existing data, thereby, overcoming the necessity of generating our own data, randomly or otherwise. The conceptual model was then mapped into relational and object-oriented schemes and the intended operations implemented in each database management system. Finally, the operations were executed within each database management system and statistics collected that characterized each system's performance.

The remainder of this thesis discusses the work involved in completing this study. Chapter 2 discusses the relational and object-oriented database systems used, while Chapter 3 covers benchmarking issues and benchmark selection. In Chapter 4, we present the methodology employed in implementing the selected benchmarks. The statistics collected on each system and an analysis of the systems' performance are given in Chapter 5. Finally, conclusions and suggestions for future work are presented in Chapter 6.

CHAPTER 2

THE DATABASE SYSTEMS

The criterion for selecting a relational and an object-oriented database system centered on availability. Potential relational candidates included two database machines, DIRECT and Britton-Lee's IDM/500, and three relational database management systems, Oracle, and 'university', and 'commercial' Ingres. From the time Ingres was introduced in 1976 [SWK86], it has been used extensively and is widely respected as a proven relational database system. While 'commercial' Ingres includes various performance enhancements (a different query optimizer, sort-merge join strategies, query-tree caching, and buffer management controlled by the database system [BDT83]) not found in the 'university' version and would have been desirable to use, the 'university' version was already installed at Oregon Graduate Center (OGC) and available for immediate use. Since the database machines and Oracle were also unavailable for evaluation, 'university' Ingres was chosen as the relational system for the analysis.

At the time of decision, most object-oriented database systems (Postgres, Vbase, IRIS) were in experimental form and undistributed. One object-oriented database

system, GemStone, a product of Servio-Logic, was being beta-tested at various sites, and available for use. Although GemStone was not available at OGC, Servio-Logic made arrangements to allow benchmark development and testing at their site. GemStone was therefore selected to be the object-oriented database system to analyze.

The following sections briefly describe the Ingres and GemStone database systems.

2.1 UNIVERSITY INGRES - VERSION 8.7

Ingres (Interactive Graphics and Retrieval System) is a relational database system, designed and written in C, and implemented on top of UNIX. Each relation is stored as a separate UNIX file divided into 512-byte blocks (pages). Tuples are maintained as records; records are packed into the blocks as tightly as possible with no record divided between blocks.

Ingres's memory management is handled by UNIX. An Ingres-generated read request prompts UNIX to move the required page(s) from secondary memory into memory buffers and return the required byte string to the user. (Ingres is configured with 40 buffer pages.) Should a page that is already in the buffers be referenced again, no disk I/O takes place.

Three types of storage organizations are supported by Ingres: **heaped, hashed**, and **isam**. When a relation is created and filled, the relation file is automatically organized as a *heap*; tuples are stored in the file in the order they were added. In the

heap organization, data is retrieved by sequentially searching the entire relation file. Therefore, sequential scans generally consume most of the database system's time in query and update operations. The heap storage structure has low system overhead and is best suited for small relations, temporary relations, or transitional storage structures due to a COPY operation.

Users may stipulate that a relation be maintained in a *hashed* or *isam* organization, which are *keyed* (indexed) storage structures: "The storage location of a tuple within the file is a function of the value of the tuple's key domains" [SWK86]. These structures allow rapid access of specific pieces of a relation when a user supplies the key values. **Hashing** creates and utilizes a table of buckets, determined by the user-specified key values, and is most suitable for conditional accesses on a specific key value. Relations designated to have an **isam** (indexed sequential access method) organization maintain a sparse index on their key values and are nearly sorted. This type of organization is desirable in situations where selection is based on a range of values that the key value must fall in.

Ingres's data manipulation language, QUEL (QUEry Language), allows a programmer to query and update a database without worrying about data structure implementation or memory management algorithms. Customized user programs may be written in EQUEL (C plus QUEL, i.e., a C program with QUEL embedded in it). EQUEL is a preprocessor and allows a programmer to code programs in C, make UNIX system calls, and include QUEL commands to Ingres. Terminal and disk I/O are supported in EQUEL via C operations. QUEL also provides an operation facilitating bulk data loading. The user formats a UNIX file of data to Ingres

specifications and then invokes the QUEL COPY command to download the data to a pre-existing relation in the database.

For this study, Ingres was resident on a MicroVaxII running 4.3 BSD UNIX. The MicroVaxII has one disk drive. On this drive resides the operating system and all system and user programs and data belonging to the MicroVaxII. As a result, potential bottlenecks can develop when the operating system and the database system come in conflict with each other.

2.2 GEMSTONE - VERSION 1.3

GemStone is an object-oriented database management system that encompasses the features of a traditional database management system while providing the object-oriented features discussed in Chapter 1. It is written in C and implemented on VMS.

GemStone is composed of a **Stone** process and one or more **Gem** processes. The Stone process is a resource monitor that allocates object-oriented pointers (OOPs) and coordinates transaction commits. Both Stone and Gem use OOPs to refer to objects and employ an *object table*, mapping the OOP to a physical location.

A Gem process provides secondary storage management, authorization, transactions, recovery, and associative access support. Objects are stored on pages; those objects too large to fit on one page are divided into pieces, and organized in a tree structure spanning several pages.

GemStone employs five basic object storage formats:

self-identifying

This storage format includes integers, characters, and booleans.

byte

Strings and floats are included within this format.

named

Access to an object's subobjects is through unique identifiers, the object's instance variables.

indexed

An object's subobjects are accessed by number (e.g., an Array).

non-sequenceable collections (NSC)

In this storage format, the instance variables are anonymous, i.e., elements are not identified by name or index, but are accessed by their values.

Both self-identifying and byte storage structures are considered atomic, i.e., there is no internal structure. The NSC structures may be constrained to contain only specific kinds of objects[7] and are sometimes referred to as *constrained Collections*.

Within GemStone, a user may create indexes on a **path**. A path is composed of the instance variable names of some subpart of an object. Two types of indexes are implemented. An *identity* index facilitates a search on the identity of an element's subobject and is unconcerned with the state of the subobject (its value). An *equality* index aids in searching a collection based on the value of its objects; it will also assist in range searches on values. Indexes are implemented as B-trees; one B-tree exists for

[7]An object O is a kind of its class and its class's superclasses.

each component in the path name.

GemStone's programming language is OPAL, which is used for data definition, data manipulation, and general computation. OPAL is similar to Smalltalk; it encompasses the programming features of the Smalltalk language and also supports associative access, data typing, and multi-user environments.

The GemStone database system provides workstation tools for editing and database browsing. These programs, called the *OPAL Programming Environment* (OPE), include the **OPE Workspace**, where OPAL code may be entered, tested, debugged, and committed to the database, the **OPE Browser**, which provides templates for class and method definitions and facilitates review of previously-defined classes and methods, and the **OPE Bulk Load/Dumper**, a tool allowing automatic loading and unloading of data. One utility recently added to the GemStone system is **Topaz**, an interactive debugging environment. This utility allows the user to download OPAL class definitions and program code to a GemStone process and interactively test this code.

Direct terminal and disk I/O facilities are not available through OPAL. Therefore, various libraries of C functions (the GemStone C Interface) or Smalltalk classes and methods (the GemStone Smalltalk Interface) are provided to allow data transferral between workstations and the host system, and enable the user to create interactive user interfaces. In this way, the user is able to send OPAL code to GemStone for execution, receive GemStone data objects for further manipulation or display, and perform various system functions (i.e., transaction commitments, beginning and ending a GemStone session).

The GemStone database system used in this study runs on a VAX 11/750 under VMS 4.5. The operating system resides on one disk drive (the swap disk), while all other data, programs, etc., are placed on several other disk drives (the data disks). Since the operating system is removed from the data disks, operating system and database system conflicts can be lessened.

CHAPTER 3

THE BENCHMARKS

Benchmarking attempts to quantitatively evaluate a system's external and internal performance [BoD84], [DHK85]. *External performance measurements* include the elapsed response time to process various requests, while *internal performance measurements* evaluate the work distribution within a database system.

External performance is measured by inserting checkpoints into a database application's flow of control. At each checkpoint, the system's clock is accessed and the current time is stored. Generally, checkpoints will be inserted immediately before and after a query, with the difference between the two times representing the elapsed response time. While a query is being processed, no further operations, such as printing, sorting, or calculations should be performed on the data; this restriction guarantees that the result is a strict measurement of the database system's time to process the request (i.e., no additional overhead is incurred during the processing operation). The checkpoints are encoded into a program using the database system's embedded language facility and executed by the host system. Demurjian, et al.,

recommend inserting checkpoints directly into the database system's source code rather than utilizing the system's embedded language facility [DHK85]. In this way, any overhead incurred due to the embedded language is avoided. This approach was not feasible for this study; therefore, the checkpoints are included in the program code of each test. Pre-query statistics were collected to determine what overhead was incurred.

Internal performance measurements evaluate a system's work distribution by measuring CPU time and I/O activity. CPU cycles are generally consumed by the software executing the query, the overhead procedures, i.e., path selection access and buffer pool management, and the operating system functions which initiate disk operations. I/O activity includes retrieval of the query data into the buffer pool (reads) and storing the query results into secondary storage (writes) [BoD84].

3.1 APPLICATION CRITERIA

In selecting benchmark applications, we first designed a conceptual model for each potential application. In this model, we defined the data entities, determined a set of operations on the entities, and specified a data set. Once the conceptual model was defined, we needed to map its components to their logical counterparts in Ingres and GemStone. Figure 3.1 illustrates the mapping from the conceptual level to the logical level.

Each potential benchmark described an application modeling complex structures, where the structures differed in their degree of depth and object-sharing. The data in

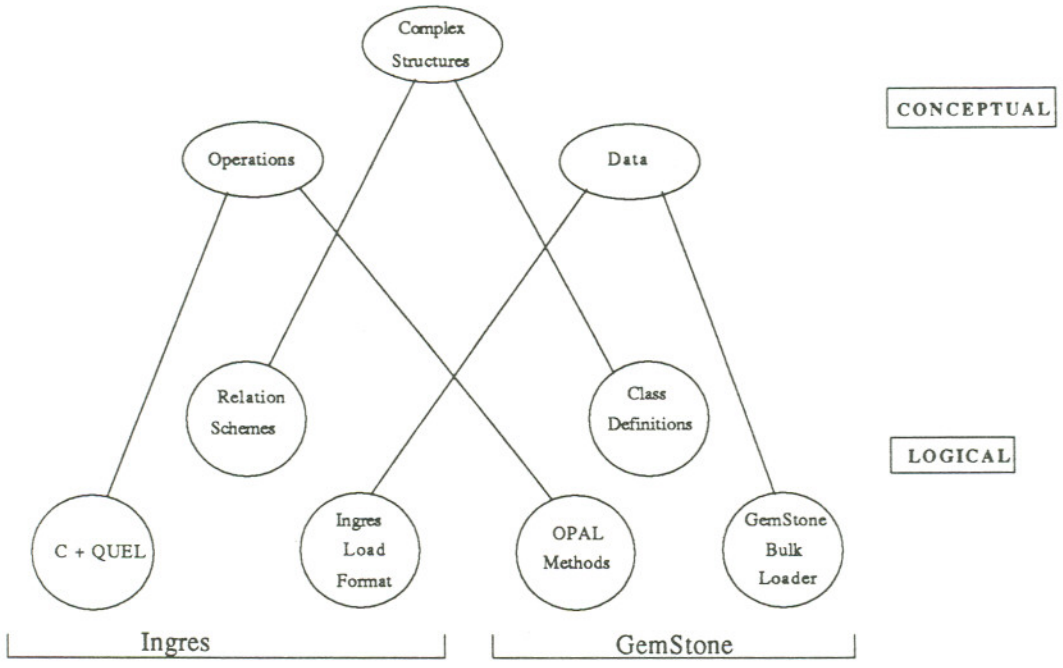


Figure 3.1

an application that exhibits a known fixed number of levels and no cycles demonstrates **bounded acyclic** depth. A Text application is an example of bounded acyclic depth. In an application where no fixed maximum level can be ascertained in advance and no cycles exist on the data, the application is said to exhibit **unbounded acyclic** depth. Applications involving parse trees may be considered examples of this. **Cyclic** data is found in those applications whose data exhibit cycles, i.e., the depth of the data is not well-defined. An application that cross-references programs where recursive procedures are found is an example of cyclic data.

The degree of object-sharing in an application is characterized by the average number of references to an object. We defined three levels of object-sharing. **No**

sharing is characterized by no object-sharing in the application, that is, one reference per object (and hence, the data is tree-structured). If the maximum number of references to an object is known in advance, the application is said to exhibit **limited sharing**. Again, a Text application may be an example of limited sharing. Finally, an application demonstrating **arbitrary sharing** permits an unbounded number of references to objects. A Hypertext application can be considered to display arbitrary sharing.

The conceptual operations for each application consisted of two queries and two updates. In choosing the operations, we tried to include at least one complex operation, i.e., one that resulted in multiple object updates or referenced numerous types of entities.

We strove for real-world applications for our benchmarks that included between 5,000 and 10,000 data items. This large amount of data thus precluded hand-generating the data. Randomly-generating the data was considered, as specified by Bitton, et al. [BDT83], but that approach was rejected due to the simplistic data generated and the difficulty of producing realistic object-sharing and connectivity. Consequently, we decided to concentrate on real-world applications that had pre-existing data. Additionally, to avoid any bias towards object-oriented database schemes, we wanted complex entity applications previously encoded into relational schemes by someone else. We then worked backwards to the conceptual scheme.

Once the conceptual model was determined for each potential application, we concentrated our efforts on mapping it to its logical model in Ingres and GemStone. This entailed mapping the conceptual scheme to Ingres relations and GemStone class

definitions. Since we were focusing on applications that already had relational schemes in place, the mapping to Ingres relations was relatively simple. Translating the conceptual scheme to GemStone class definitions was more time-consuming, since there was no precedent to work with and correlating relational scheme definitions to object-oriented schemas is not straightforward.

In Ingres, the conceptual operations were mapped to EQUEL programs and involved working with two languages (C + QUEL). If the intended operation was not part of Ingres's predefined set of operations, we had to devise some means of implementing that operation within the confines of that predefined set; this generally entailed multiple Ingres calls to implement the operation. Mapping the conceptual operation to GemStone involved developing an OPAL method, and possibly several supporting methods, to implement that operation. Once the method was defined, executing the operating entailed one OPAL call.

Finally, we had to map the application's data set to Ingres and GemStone databases, i.e., we needed to load the database. In Ingres, this step was fairly simple, since the applications included pre-existing relational schemes and data that could be loaded with little or no pre-processing. For GemStone, it was not quite as simple. The data had to be pre-processed first into a format that the GemStone Bulk Loader could use. Once the data was loaded, it needed further manipulation to 'wire' the object instances together.

Upon selecting the two applications that would be used and performing the conceptual to logical mapping, commands were then entered into the code to generate statistics on the following parameters:

1) elapsed response time per request,

2) CPU time involved per request,

3) I/O activity generated per request.

Statistics were also gathered on the amount of time required to generate and load the data for each benchmark.

Each benchmark was executed in single-user mode without indexes to gather baseline statistics. Those applications where indexing was appropriate were further tested with indexes in place. Boral, et al., recommended that benchmarks be initially run in single-user mode to measure a system's performance under optimal conditions, expose update anomalies, and provide a "picture of the resources required by different queries" [BoD84]. Multi-user and distributed modes were not possible to benchmark for two reasons: one, the time-frame of this study prohibited testing the systems in the multi-user mode, and two, neither system is a distributed system.

3.2 REJECTED APPLICATIONS

A literature search identified five relational schemes for complex object applications. This section discusses the applications that were considered and rejected, and the reasons why they were rejected.

Vdd

Vdd (VLSI Design Database System) can be used to store VLSI chip descriptions in a relational database and also serve as the focus for the VLSI tools necessary for chip design [CFH83]. Once the chip description is entered into the database, the designer can employ a conventional database management system for query processing and data manipulation or access the data using the set of VLSI tools.

This application represented potential cyclic paths and arbitrary object-sharing, but was rejected due to the time and effort it would take to coerce an existing VLSI data set into the structure required by Vdd.

Storage of Block-Structured Programs

Another application involved storing block-structured computer programs in a relational database management system. The database management system would enable users to reference different parts of a program without a sequential search. It would also aid the user in identifying various relationships between procedures and functions, and variable declarations and usages. The relational schema was based on Linton's schema for the Model programming language ([Lin83], [Lin84]) with the objects exhibiting cyclic data and arbitrary object-sharing. Further research into the Model language revealed that it was very similar to Pascal, with a few extensions. Since we had a better chance of obtaining Pascal programs with the OGC community than we had of gathering Model programs, we investigated the possibility of designing

a relational schema based on a subset of Pascal, using the Linton Model schema as a guide. A database of pre-existing programs would be filtered to remove those expressions which weren't part of the subset.

We determined that this application would be extremely time-consuming in the data-generation phase. Linton reported [Lin83] that 10,000 lines of code and four months of full-time programming were required to design and implement the parser to populate the database. Consideration was given to requesting Linton's database and parser, but this was rejected due to time limitations and a lack of familiarity with the Model language.

We also investigated using a utility that was designed and developed by Eugene Rollins at OGC [Rol82b], [Rol82a], the Syntax Analyzer Constructor. This utility could generate abstract syntax trees for a Pascal program which would then be used to produce the necessary relational tuples and objects for the revised Linton schema on Pascal programs. Since this approach required developing an LL1 grammar and debugging the utility, it was considered too time-consuming for our needs. Consequently, we abandoned the Linton Model application.

Persistent LISP Objects

This application was developed by Margaret Butler at Berkeley and addresses those situations where LISP programs "need to manipulate data structures that persist between program invocations" [But86]. Butler designed an interface, Polymnia, between Ingres and LISP that allows the user to declare a variable to be persistent and

store the value of that persistent variable in Ingres. Since LISP objects are loosely-typed, representing them in Ingres, which is strongly-typed, required introducing additional attributes to indicate an object's type and to store tuple identifiers (the identifier may be the object's value or a reference to another tuple in another relation).

This application represented cyclic data and limited object-sharing. Initially, we intended to include this application in our study and began implementing the relational and object-oriented schemas. As time and the project progressed, we determined that the time-frame for this project precluded completing this particular application; both the data preparation and the implementation of the schema and operations would be extremely time-consuming. Therefore, although this application suited our requirements very well, we opted to drop it from the study.

3.3 SELECTED APPLICATIONS

In this section, we present a detailed description of the applications that were selected and discuss how we mapped the conceptual model to its relational and object-oriented counterpart.

Documents

Currently, document processing is handled by text editors that include their own facilities for storing and manipulating data. Stonebraker, et al., introduced the idea of

storing documents in a database management system whose services could then be available to the tools or users that require them [SSL83]. These services include concurrency control, crash recovery, and access control. The system's database query language could also be used to aid document manipulation. The group proposed storing documents as *ordered* relations and designing a text editor that uses the relational system's facilities. In order for this idea to be successful, the relational system's facilities must support variable-length strings and ordered relations.

In this application, a text document is divided into chapters, sections per chapter, paragraphs per section, lines per paragraph, words per line, and the relative position of words within lines. Initially, we considered the relationship of sentences per paragraph and words per sentence, but after further consideration, concluded that the initial document preparation for data loads would be substantially more difficult if sentences were a part of the relational schema. We determined that the line data included enough information to generate acceptable queries and updates, and therefore, chose to use only line-oriented relationships.

Figure 3.2 shows an example of a document divided into the components above. This document consists of five chapters. The first chapter contains three sections, with the initial section composed of ten paragraphs. Paragraph 1 contains seven lines, the first two lines being *This is a test.* and *This is only a test.* As mentioned previously, each line has a text and word relationship. In **Line** 1, the **Text** value is the entire first line, while **Words** separates the first line into its component words, with their position in the line defined (i.e., word 1 -> *This*, word 2 -> *is*, etc.). This model is used in designing and implementing the relational and object-oriented schemas for

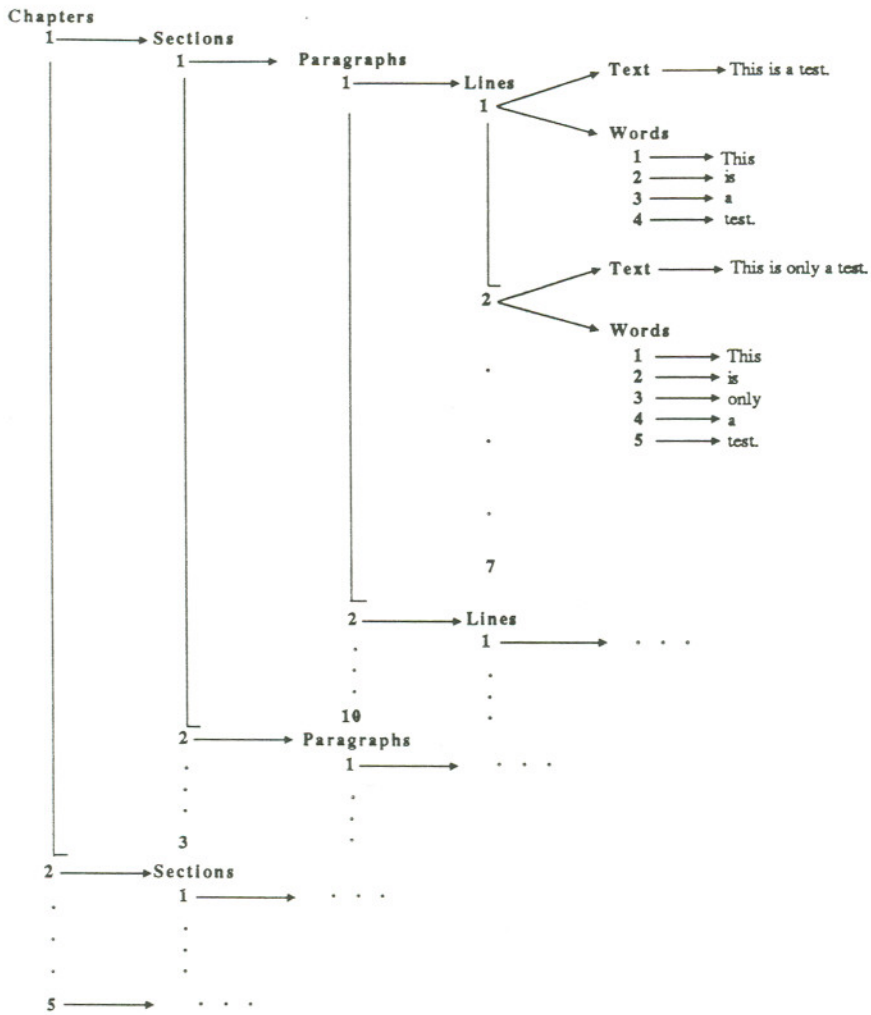


Figure 3.2

this application.

The data used for this application is the text of a master's thesis which, in its raw form, consists of the thesis text and the commands required to format the document. Initial data preparation included identifying and labelling the required components, dividing the document into those components, and arranging the

components into a format suitable for loading into Ingres and GemStone.

This application addresses bounded acyclic depth (the depth level is 6) and no object-sharing. In Figure 3.2, the diagram should illustrate that any path from the top will always lead to some object at the bottom, with no cycles or lost paths to contend with. Also, no object is referenced by more than one object.

The operations in this application consist of two updates and two queries. The first update entails substituting all occurrences of one word with another word. The second update involves moving specific chapters, sections, paragraphs, or lines to another location in the document. In the first query, the database is searched for all the occurrences of a specific word and the chapter, section, paragraph, line, and word position values corresponding to each occurrence of that word are returned. The second query finds and returns a specific text line designated by the position of that text line in the document (e.g., find the text line designated by "Chapter 1, Section 2, Paragraph 3, Line 4").

Hypertext

In a hypertext system, related materials are connected non-sequentially. One such system is the *Intermedia* hypermedia/hypertext system, designed and implemented by the IRIS (Institute for Research in Information and Scholarship) group at Brown University [Mey86], [SmZ87], [GSM86]. Intermedia "provides a framework for a collection of editors written in an object-oriented language, each capable of allowing sophisticated connections between pieces of information in its documents" [SmZ87].

In the current version, the editors include a word processor, graphics editor, and a scanned-image viewer.

Documents are connected by user-defined **links**. To create a link, a user selects a block in one document (the source block), another block in a second document (the destination block), and issues a command to link the two. A **block** may encompass anything that can be selected in a document, i.e., a word, line, paragraph, figure, picture, etc. There is no restriction to the number of blocks defined within a document, nor is there a limitation on the number of links attached to a block. All links and their associated documents are stored in a **web** and may be shared by numerous users having access to that web.

A conceptual model illustrating a tiny web and its components is presented in Figure 3.3. Four documents, **D1, D2, D3,** and **D4** are shown. Each document contains one or more blocks, labelled **B1** through **B6**. The links, **L1** through **L6**, connect the documents by beginning at a source block and ending at a destination block, contained in some other document. One block can be designated both a source and a destination block and may be associated with several links. For example, block **B1** represents the source block for links **L1** and **L2**, and blocks **B2** and **B3**, respectively, denote the destination blocks for those links. Block **B3** is also the source block for link **L3**.

The IRIS group chose to store the web information in Ingres; in this way, data persistence, concurrency and access control are supported by Ingres. Since the data is comprised of complex, hierarchical structures, flattening the data into relations makes data retrieval and manipulation potentially awkward. Consequently, the IRIS group felt that an object-oriented database might be better suited for storing the web data.

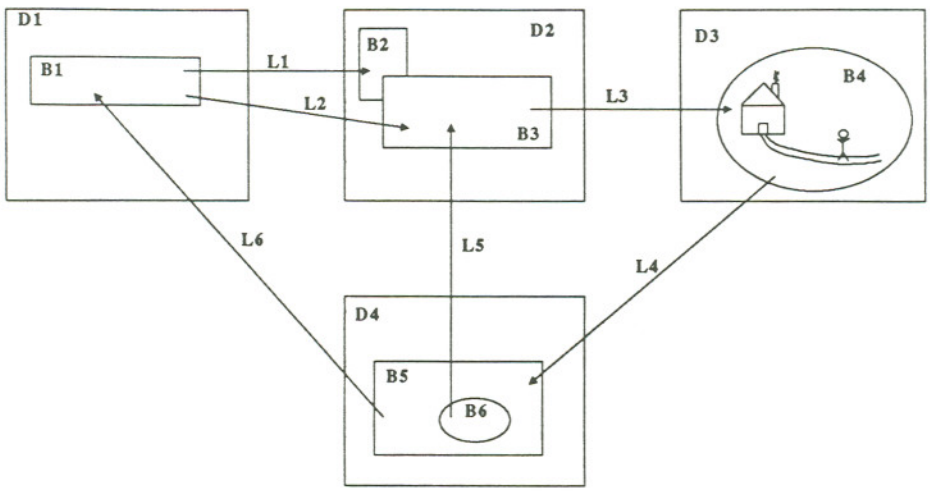


Figure 3.3

Because Intermedia is currently in use at Brown University, we contacted the IRIS group and inquired into the possibility of using some of their data. The group was very willing to help, and sent data consisting of one web corresponding to an English course. We informed the group it was not necessary to include the original document texts in the data sent; we were not intending to implement hypertext, only to query the tuples in the database. The IRIS group dumped the data from their Ingres relations used for that particular web and sent it in a format that could be used to load our Ingres relations. The data sent included extra relations that were unnecessary for this study as well as some empty relations we had intended to use in our database scheme. Also, Intermedia's relations that corresponded with ours had extra attributes we did not intend to incorporate in our schema. As a result, the conceptual models had to be modified slightly. The data files sent needed very little preparation for

loading into the GemStone database.

This application represents cyclic paths and arbitrary object-sharing. Referring to Figure 3.3, it is apparent that a search operation has the potential of executing forever because a cycle was encountered. For example, a query could begin at **D1**, follow links **L2, L3, L4**, and **L6**, return to **D1**, and begin again. This schema also exhibits a high level of object-sharing; many links could reference the same block and many blocks can reference the same document (i.e., **B3** is the destination block for links **L2** and **L5**, while blocks **B2** and **B3** are both in document, **D2**).

The hypertext application contains two queries and two updates. The first update involves relocating a specific link, i.e., a starting link will point to a different ending block. For example, in Figure 3.3, we might update link **L2** to point to block **B5** rather than block **B3**. In the second update, a new link is added to the web. Referring to Figure 3.3, we might choose to add a link **L7**, connecting block **B3** to block **B5**.

Our first query determines if a path exists between two documents by traversing all the starting links found in each document visited. The query executes a breadth-first search and halts when a path is found, no path exists, or a cycle is encountered prior to finding the desired end document. For example, Figure 3.4 contains a small web, consisting of seven documents and their blocks and links. The user might want to determine if a path exists between **D1** and **D3**, beginning at **D1**. Following links **L1** (or **L2**) and **L3**, we see the traversal has reached **D3** and thus, the user is informed that a path does exist. But, if the user inquired if a path existed between **D1** and **D4**, a negative response would be generated; the traversal would follow links **L1, L2, L3**,

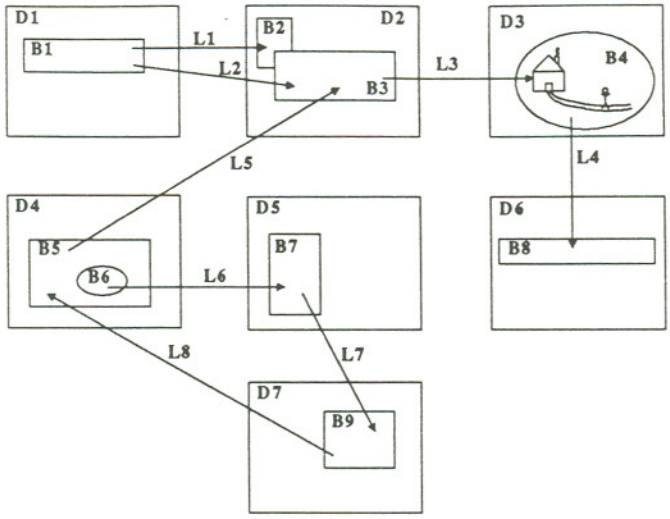


Figure 3.4

and **L4**, and stop in **D6** because **D6** has no starting links to follow. Lastly, should the user attempt to find a path between **D4** and **D1**, beginning at **D4**, a cycle response would be seen since following links **L6, L7**, and **L8** would return the search to **D4** which is where it began (of course, links **L5, L3**, and **L4** will take the search to **D6** and halt, but all the starting links in each document must be traversed).

In our second query, all the pertinent information necessary to illustrate one document, its starting blocks, the end blocks associated with those starting blocks, and the documents the end blocks are contained in is gathered. This information consists of the initial document, each starting block, each end block associated with the starting blocks, and the document for each end block. Figure 3.5 illustrates what the resulting information might contain. **D1** is the initial document with blocks **B1, B2**, and **B3** identified. The end blocks and documents associated with blocks **B1, B2**, and **B3** are

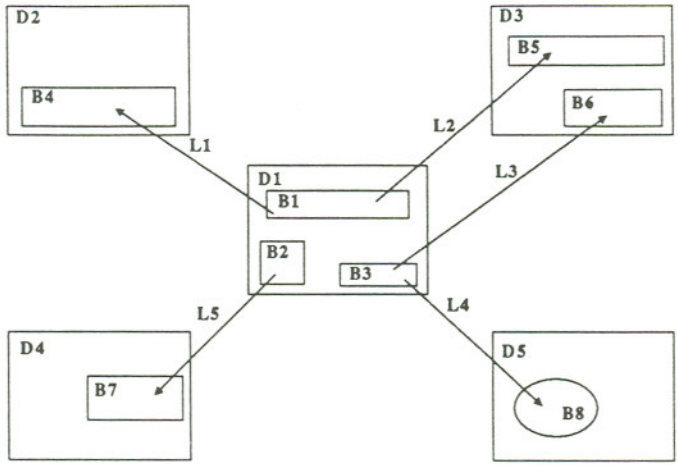


Figure 3.5

stored (**B4-B8, D2-D5**). The links denoting the block associations are then added to the information (**L1-L5**). It should be emphasized that this query only retrieves the required data; nothing further is done with that data in this study (i.e., no pictures are drawn).

CHAPTER 4

BENCHMARK IMPLEMENTATION

The implementation of each benchmark followed the protocol diagrammed in Figure 4.1. Initially, the conceptual model was mapped into relational and object-oriented schemas. The data used for each benchmark was then processed for database loading and subsequently loaded into Ingres and GemStone databases. Next, the operations were implemented in each system's language and tested on sample data. Finally, each operation was executed on real data and statistics collected.

4.1 THE DOCUMENT BENCHMARK

4.1.1 Conceptual to Logical

The relation schema used for the Document benchmark was patterned after the one presented by Stonebraker, et al. [SSL83], and is presented in Figure 4.2. The

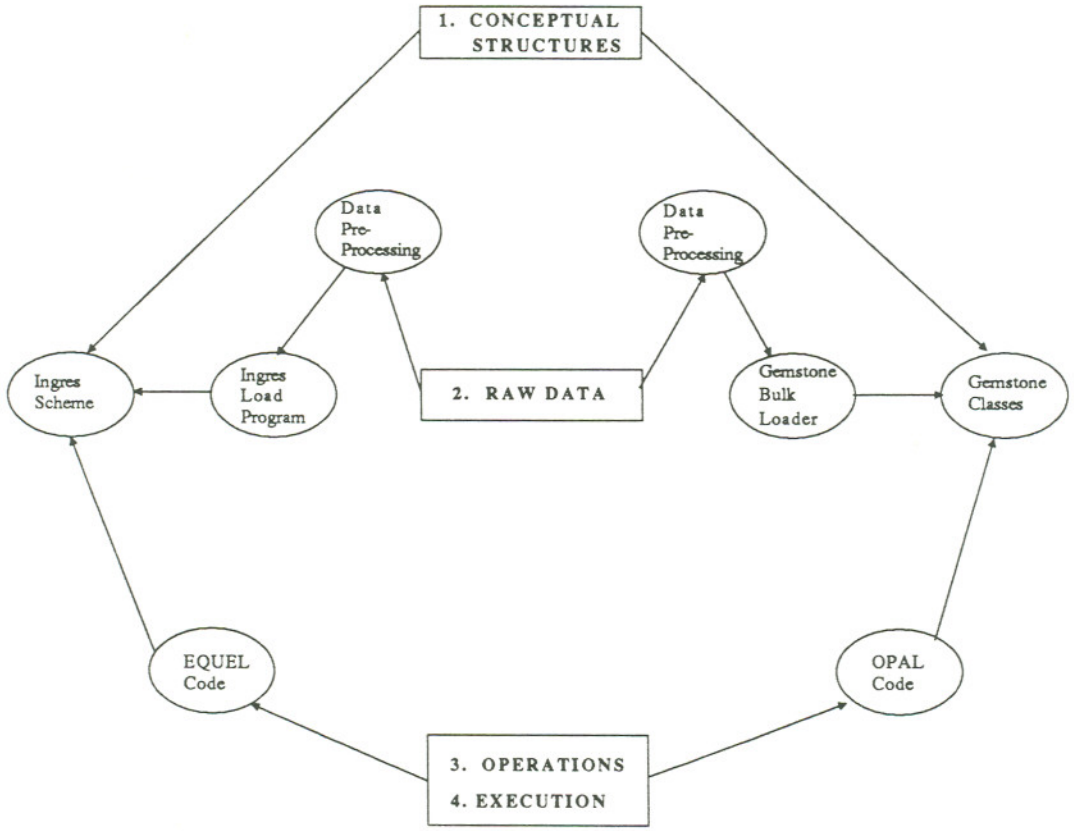


Figure 4.1

attributes defined for each relation denote the relationship between that relation and its components. For example, in the **Chapter** relation, the relationship between chapters and sections is defined by the attributes *chapid* and *sectid*. The **Section** relation defines the relationship found between the sections and paragraphs, while in the **Paragraph** relation, the relationship between paragraphs, lines and text is defined. Each tuple in the **Line** relation represents a specific word and its position within a line. The **Text** relation stores the entire line of text associated with a particular line, including punctuation. We considered omitting the **Line** relation and extracting the

Doc Database

Relation Chapter

chapId *sectId*

Relation Section

sectId *paraId*

Relation Paragraph

paraId *lineId* *tLineId*

Relation Line

lineId *wordPos* *word*

Relation Text

tLineId *text*

Figure 4.2

line-word relationship from the **Text** relation. Since this would have required a scanner to do the extraction, we rejected this approach. Also, we did not include blank lines in the **Line** relation as we felt they were unnecessary for our purposes. Since the actual contents of the document are stored redundantly in the **Line** and **Text** relations, update anomalies are possible.

Each chapter, section, paragraph, and line in the document is assigned a globally unique integer identifier, used for referencing that component in the database only; it is not intended to reflect the ordering of the components in the document. For example, Chapter 1 may contain two sections and Chapter 2 contain three sections.

The relationship between the chapters and their sections is denoted by five tuples stored in the **Chapters** relation, one tuple for each section. These tuples would contain the values **<1 1>, <1 2>, <2 3>, <2 4>** and **<2 5>**, respectively, the first value associated with *chapid*, and the second value, *sectid*. The *sectid* value increments with each section encountered to ensure that each tuple in the database is unique and that no conflict will occur when referencing a tuple. This type of tuple format is found throughout the entire schema, with the actual word and text values stored in the **Line** and **Text** relations, respectively.

Once we determined the relational schema, we created an Ingres database labeled **Doc** to contain the relations above. The type of each attribute was defined to be a 2-byte integer, with the exception of *word* (**Line**) and *text* (**Text**), that were defined to be an 80-byte character type.

The GemStone schema used in this application is diagrammed in Figure 4.3 and

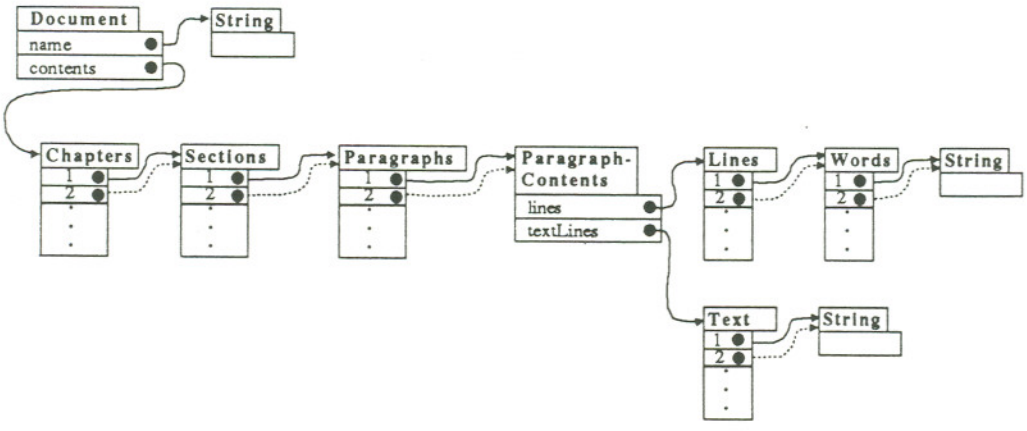


Figure 4.3

closely resembles the conceptual model shown in Figure 3.2. The **Document** class is analogous to the Ingres **Doc** database and was defined as a subclass of class **Object**. Two instance variables were defined in **Document**, *name*, referencing the name of a document (e.g., **Doc**), and *contents*, referencing the entire document. The **Chapters, Sections, Paragraphs, Lines, Text** and **Words** classes were defined as subclasses of **SequenceableCollection** class and are analogous to an Array type; the number corresponds to an object's position in the document. For example, in the **Chapters** class, position 1 refers to Chapter 1 and will reference an object of class **Sections** that contains the sections found in Chapter 1. This type of referencing pattern was utilized within all **SequenceableCollection** subclasses above. **ParagraphContents** was also defined as a subclass of **Object** class, with instance variables *lines* and *textlines* defined within the subclass. The instance variables in **Document** and **ParagraphContents** were not constrained to be a specific type, although in Figure 4.3, each is shown referencing a specific class type. The dotted arrows shown in each **SequenceableCollection** subclass in Figure 4.3 signify another instance is referenced and that each instance referenced will be the same class type throughout the collection.

4.1.2 Database Loading

Data preparation consisted of formatting the document to yield the appropriate UNIX and VMS files needed to load each database system. The document used for this benchmark was a UNIX file containing **troff** source of a Master's thesis. Initial document preparation consisted of removing the formatting commands and sections

considered unnecessary for the final document data (i.e., the Title Page, the Table of Contents, the Bibliography, etc.). The thesis contained six chapters and two appendices. We decided the first appendix was unnecessary for our needs and deleted it, but the second appendix contained program code we felt could prove interesting to query. Therefore, we left it in the document and labeled it Chapter 7.

We processed the document through the **nroff** utility provided by UNIX, tagging each section and paragraph to allow easy identification of these components in later processing steps and removing all blank lines.

The reformatted thesis still required a substantial amount of processing before the necessary files could be created. This processing was accomplished through several UNIX tools and utilities, specifically, **vi, awk**, and **sed**. Using these tools, 1) chapter and line tags were inserted into the thesis, 2) each chapter, section, paragraph, and line tag was labeled with the appropriate integer value (i.e., for the relational scheme, each tag was assigned a globally unique integer value, while in the object-oriented scheme, each tag value was locally unique), and 3) the pertinent UNIX and VMS files were generated and formatted to each database's loading specifications.

Initial database loading attempts in both systems used small subsets of the data to ensure the data was correctly formatted and the loading operations understood. In Ingres, we used the COPY operation to load the **Doc** database. Once the files were copied into the database, it was ready to be used. The number of tuples contained in each relation is listed below.

Chapter	37
Section	199

Paragraph	2751
Line	19,711
Text	2751

In GemStone, large data files are copied into a database using the **OPE Bulk Loader** tool. Like Ingres's COPY operation, the OPE Bulk Loader requires the data be in a particular format within a file. The Bulk Loader reads the file, creates and instantiates one object for each line of the file, and stores the objects in a user-specified collection. After we had loaded the data into GemStone, we had to take the data and construct our **Document** object. Methods were designed and tested to facilitate this step. Once we were assured our methods worked, we constructed the **Document** object and stored it in the GemStone database.

4.1.3 Operation Implementation

In this section, we discuss the implementation of each operation specified in our conceptual model. Each operation was assigned a test name (Test 1, Test 2, etc.) for easy referral. For each operation (test), three sets of parameters were tested.

Test 1

> **Replace all word and text occurrences of a word** (the target word) **with another word** (the replacement word).

We used the Ingres REPLACE operation which employs pattern-matching characters to denote the target and replacement words. The operation did not require additional C code in the EQUEL program to perform the word substitutions. We attempted to select words that did and did not occur frequently in the document. The words replaced in the programs and the ratio to their total word count were:

1. a -> A (5.25%)
2. the -> THE (9.39%)
3. process -> PROCESS (1.35%)

In the GemStone schema used in this application, the **Word** and **Text** objects are positioned at the bottom of the **Document** object. To replace all the target words with the replacement words, we had to implement a depth-first search on the **Document** object. We designed a method that executed this search, assigned it the message selector **replace:with:**, and included it in the **Document** class. An example expression for this update would be

Document replace: 'the' with: 'THE'.

The OPAL program for this update invoked the **replace:with**; method on the set of parameters listed above.

Test 2

Relocate specific chapters, sections, paragraphs, or lines within the document.

In order to implement this update in Ingres, we had to impose an order on the relations. This was accomplished using the ORDEREDN operation provided by Ingres. ORDEREDN adds an extra attribute, **lid1**, to a pre-loaded relation and assigns that attribute an integer value for each tuple, signifying the tuple's place in the relation. This field is different from the user-defined attributes because it is dynamically adjusted by Ingres during appends, deletes, and replacements to maintain a consistent ordering of the tuples in the relation. In append operations, if the user specifies an **lid1** value, the new tuple is inserted in front of the tuple associated with the specified **lid1** value and all **lid1** values following the new tuple are incremented by 1. If a user declares an **lid1** value in deletion operations, all **lid1** values following that tuple are decremented.

After determining the mechanics of ORDEREDN, we designed the EQUEL programs that required additional C code to supplement the Ingres operations. Ingres by itself could not express the necessary manipulations.

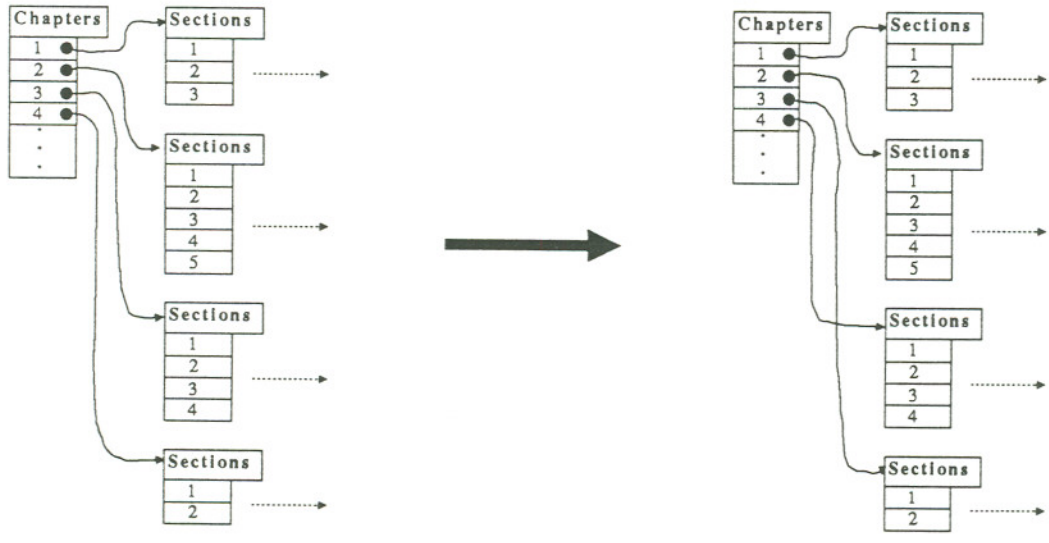
The three parameters tested in Test 2 are listed below and define the target components in terms of their globally unique tuple ids. The first integer denotes the component to be moved; the second defines the new location it should be inserted at.

1. chapter 5 2
2. paragraph 32 22
3. line 795 2206

For example, 'chapter 5' denotes that we want to insert Chapter 5 before Chapter 2. One issue we discovered that could not be addressed in this scheme was what would happen when a line was inserted between two paragraphs; there was no way to

determine if the line was the last line of the previous paragraph or the first line of the next paragraph.

The GemStone method implemented for this update had to move a specified object to a new location in the **Document** object. This entailed locating the appropriate objects and performing the necessary insertions and deletions. For example, suppose we wanted to move Chapter 4 to just before Chapter 3. Figure 4.4(a) diagrams the original **Chapters** array and a few of the sections it references. The intent is to move the **Sections** array referenced by position 4 in the **Chapters** array to immediately before position 3 in the **Chapters** array. Thus, position 3 would now reference the **Sections** array previously referenced by position 4, and all the



(a) (b)

Figure 4.4

components referenced by those positions following position 3 are adjusted down one position in the **Chapters** array. Figure 4.4(b) reflects this relocation operation.

Since we implemented the relocation method to move the intended object immediately before the designated index position, this presented a problem when we wished to move an object to the end of the array (e.g., how would one denote that position?). We solved this problem by specifying that the object to be moved was to be inserted in the target object at index 0 (zero), thereby signalling the method to add the moved object to the end of the new array object. The message selector for this method was labeled **relocate:to:**. An example demonstrating its usage is shown below.

Document relocate: oldObject to: newObject.

The **oldObject** and **newObject** arguments are defined by the user.

The OPAL programs were written to test the same parameters as those tested in Ingres. But, whereas the Ingres tests defined the parameters using the component's globally unique ids, the OPAL programs define each component by its position in the document. The OPAL expressions used are given below.

1. relocate: chapter 5
to: chapter 2

2. relocate: (chapter 3 (section 2 (paragraph 4)))
to: (chapter 1 (section 3 (paragraph 0)))

3. relocate: (chapter 4 (section 4 (paragraph 4 (line 4))))
to: (chapter 6 (section 3 (paragraph 3 (line 3))))

Conceptually, this would appear to give GemStone an advantage over Ingres, because it is much easier to be able to define a component by its relative position in

the document. Yet, if we had chosen this means of identifying our parameters in Ingres, it would have necessitated introducing additional program code to determine the unique component id corresponding to our parameter. Since the **Doc** database was defined and created with globally unique component ids, we decided it was fairer to Ingres to maintain this specification in our test parameters.

Test 3

> **Find every occurrence of a specific word** (the target word) **and return the chapter, section, paragraph, line id and word position associated with each occurrence of that word.**

To execute this query in Ingres, we had to join the **Chapter, Section, Paragraph,** and **Line** relations. We also needed to specify the target word using Ingres's pattern-matching characters. The three target words and the number of times they occurred in the document were:

1. and (431)
2. parallel (89)
3. fork (94)

We decided it would be interesting to determine how the statistics would be affected if the retrieval information was printed (i.e., for each word occurrence, the chapter, section, paragraph, and line ids and the word position value were printed). Therefore, we designed two sets of EQUEL programs, one where the results weren't printed and

one where they were.

We determined the best way to execute this query in GemStone was to create and instantiate an object for each target word occurrence and store the word's position values in the object. These objects would be stored in a collection that would be returned by the method upon completion. We had to perform a depth-first search to compare every word in the document to the target word. The method implementing this search and collecting the required data was assigned the message selector **retrieveSpecificWord:**. An example expression invoking this method is given below.

Document retrieveSpecificWord: 'fork'

The OPAL programs implemented for Test 3 tested the same set of parameters as Ingres. As in Ingres, we wrote programs that did and did not print the returned information.

Test 4

Retrieve a specific textline designated by its position in the document.

The Ingres implementation of this query required that the target textline be defined by that textline's unique id. Therefore, we designed a QUEL command that searched the **Text** relation for the tuple whose *tlineid* value matched the value defined in the parameter and returned that tuple's *text* value.

The parameters tested in this test were:

1. line 651

2. line 1192

3. line 2526

We used the same approach as in Test 2, i.e., we defined the parameters in terms of the textline's unique id, for the same reasons. We decided it would be appropriate to collect statistics on printed and non-printed results in this test also, and developed the necessary EQUEL programs.

We determined that this query did not require us to design a new GemStone method for implementation. Since we knew exactly where the text line was located in the document, we could retrieve it directly using OPAL's **at:** method. For example, suppose a user wished the text associated with chapter 1, section 3, paragraph 2, line 25. The following OPAL expression would perform the retrieval.

((((((Document contents) at:1) at:3) at:2) textlines) at: 25).

Separating this OPAL expression into its individual expressions yields the following (refer to Figure 4.3 for the scheme):

(Document contents) returns **Chapters**

(Chapters at:1) returns **Sections**

(Sections at:3) returns **Paragraphs**

(Paragraphs at:2) returns **ParagraphContents**

(ParagraphContents textlines) returns **Text**

(Text at:25) returns **String** corresponding to the text at that line.

The OPAL programs for this test retrieved the same textlines as Ingres, but the

parameters are phrased as component positions. The OPAL expressions are listed below:

1. ((((((Document contents) at:3) at:1) at:3) textlines) at:6).
2. ((((((Document contents) at:4) at:8) at:11) textlines) at:3).
3. ((((((Document contents) at:7) at:1) at:1) textlines) at:250).

Included are programs that didn't print the returned textline and programs that did.

In this case, it would appear that Ingres might have had an advantage over GemStone, since it only needed to search one relation (**Line**) for the unique ids and return the textlines; GemStone had to traverse each specified object to finally locate the specified textline. As in Test 2, because we wished to be consistent with respect to the scheme's specifications and our test parameters, we chose to leave each one as is.

4.1.4 Execution

At this point, we were ready to execute our test programs and collect statistics. In each EQUEL program, we used UNIX system calls to produce the statistics. To measure the elapsed time of a test operation, the **gettimeofday** system call was used. This operation stores the system's time upon invocation. In order to determine the elapsed time, two invocations were required, one immediately before executing the query, and one immediately after the query was completed; the difference between the two is the elapsed time. Since the units returned by **gettimeofday** were not in seconds, minutes, or hours, we converted them to seconds within the EQUEL program.

The cpu activity of a test operation was determined by a **getrusage** system call. This call returns various timing and disk I/O statistics for the calling process and its children process(es). (For the remainder of this thesis, I/O means disk I/O.) Of primary interest to our study were the values denoting the amount of cpu time the EQUEL program consumed executing the test operation and the I/O numbers generated during execution. Utilizing **getrusage** required stipulating the process for which we wanted the statistics (the parent or the child process). If a child process was to be measured, **getrusage** could not be invoked until that child process had terminated.

When Ingres is invoked in an EQUEL program, an Ingres process is forked. All QUEL commands included in the EQUEL program are piped to this process and executed there, with the results then piped back to the EQUEL program. Therefore, the **getrusage** call was defined to collect the statistics on the child process (i.e., the Ingres process) and was invoked immediately after the Ingres process had terminated (e.g., a QUEL EXIT call was made). The statistics collected by the **getrusage** call reflected the cpu time and I/O activity generated by the QUEL operations. In tests that included printing the results to a file, this activity was also included in the **getrusage** statistics.

Since we could not invoke the **getrusage** operation until the child process had terminated, the statistics included several Ingres operations outside the test operations. These operations included initiating the Ingres process, pre- and post-query operations (i.e., range, destroy, and modify statements), and exiting Ingres. Because these operations were not considered part of our test operations, we did not want them included in the performance statistics. We designed a separate program to collect the

getrusage statistics for only the Ingres initiation and exit operations. These value were then subtracted from the over-all query statistics to obtain a more representative performance measurement.

We were unable to incorporate both system calls in one EQUEL program because the **getrusage** values would include the activity generated by **gettimeofday**, and would not reflect the absolute cpu and I/O statistics of the test operations. Therefore, each test operation required two separate EQUEL programs; one invoked **gettimeofday** for the elapsed time measurements and the other called **getrusage** and measured the cpu and I/O activity.

For each test operation, a UNIX batch file was written that invoked the corresponding EQUEL program ten times. All output generated by the tests was captured to a file for future review. Since update operations revised the database, we restored the relations with the original data prior to executing each run, using separate programs that copied and truncated the relations.

Because our objective was to run each test program in a system without concurrent user activity, we needed to ascertain that each test was indeed executed under this condition. To accomplish this an **update** system call was made prior to each test's invocation that reported the number of users currently logged on to the system and the system load factor at that time.

The OPAL programs for each test operation were executed from the Topaz environment. Topaz is a process forked by the VMS operating system that enables the user to execute OPAL programs interactively and in batch mode. When a Topaz process is initiated, VMS begins collecting data on that process and all executions

occurring within that process. That data is captured by including two calls to the OPAL method **time** within Topaz; one call prior to executing an OPAL program and a second call immediately after execution is completed. The **time** operation returns the following values:

1) the system clock time at the point of the **time** call,
2) the cpu time consumed by the Topaz process from the time of initiation to the **time** invocation, and
3) the I/O activity generated in Topaz, also from the initiation point to the invocation call.

The results from each **time** invocation were written to an output file. By subtracting the second set of values from the first set, we were able to determine the timing and I/O statistics for that OPAL program.

Batch programs were written for each test program. These batch programs invoked Topaz, ran the OPAL program ten times, and generated an output file containing the timing and I/O statistics.

As in Ingres, when executing updates we wanted to be certain we were updating a fresh database after each set of OPAL expressions was executed. Therefore, when the set of expressions was finished executing, we aborted the transaction. This guaranteed that the database remained unchanged and that subsequent updates would be executed on the original data.

In GemStone also, our objective was to run the test suites on a system with no other users concurrently executing. To determine if this was the case, prior to running each test suite and upon completion, we invoked a VMS call **sho sys** that reported the

system activity. Since Ingres basically began with a fresh database prior to each batch execution, we attempted to approximate those conditions in GemStone by instantiating a fresh database before running each batch program.

4.2 THE HYPERTEXT BENCHMARK

4.2.1 Conceptual to Logical

The relational schema used for the Hypertext Benchmark, given in Figure 4.5, is a subset of the schema implemented by the IRIS group [Mey86], [SmZ87], [GSM86]. We used a subset because their original data included empty relations and relations they subsequently informed us were unnecessary and would be deleted in the future. All document information contained in the web sent by the IRIS group is stored in the **Docs** relation. Each document tuple contains a unique integer value used for referencing the document (*docId*), a string representing the name of the document (*docName*), and a string associated with the pathname denoting where the document is stored in the file system (*docPath*). (We reiterate here that the document contents were not included in the data; consequently, although the path value denotes where the document may be found, the document itself cannot be retrieved.) The **Block** relation stores the tuples for every block contained in the web. Each tuple contains a unique integer *blockId*, a *blkOwner* string, denoting who created that block, and a *docId*, that references a **Docs** tuple. The block definition parameters are stored in the **AppBlock**

Inter Database

Relation Docs

docId *docName* *docPath*

Relation Block

blockId *docId* *blkOwner*

Relation AppBlock

blockId *appExtent*

Relation Link

linkId *linkType* *startDocId* *startBlkId*

endDocId *endBlkId* *lkOwner*

Figure 4.5

relation. The *blockId* value references a tuple in the **Block** relation that the parameters are associated with. The *appExtent* (application extent) attribute stores the block definition parameters. Since one block may require several *appExtent* values for definition (for example, the beginning and end points of a text selection), the **AppBlock** relation will store a tuple for each *appExtent* value, each tuple referencing the same *blockId*.

Finally, the **Link** relation contains the data associated with each link. Each link tuple includes a type (*linkType*) and an owner (*lkOwner*) value. Also in the link information are the start and end ids for the document (*startDocId*, *endDocId*) and block (*startBlkId*, *endBlkId*) associated with that link. The document ids are included

in this relation to avoid a join operation when a link is retrieved, although this results in more relations requiring updates when a link is shifted.

An Ingres database labeled **Inter** was created and the **Docs, Block, AppBlock**, and **Link** relations defined. Most of the attributes were defined to be 2-byte integers; *docName* (**Docs**), *docPath* (**Docs**), *blkOwner* (**Block**), and *lkOwner* (**Link**) were defined as character strings.

Figure 4.6 illustrates the GemStone scheme for this application. Since there can be more than one web stored in a database, each web is assigned a unique identifier; we received **Web7** from the IRIS group. In Figure 4.6, we show an instantiation of Web7. We will use this diagram to explain the classes defined for the Hypertext application. Each object in the diagram is an instance of the class designated in bold print (e.g., **WebObj** represents an instance of an object of class **WebObj**).

A **WebObj** object, referenced by position 7 in **WebColl**, contains the data associated with Web7. The *links*, *blocks*, and *docs* instance variables in **WebObj** correspond to the **Link, Block,** and **Docs** relations, respectively, shown in Figure 4.5. Each instance variable is constrained to contain a collection of their associated objects (i.e., *links* -> **LinkColl** objects, *blocks* -> **BlkColl** objects, and *docs* -> **DocColl** objects). Each Web would contain these collections which are stored as relations by the IRIS group. Included in **WebObj** is an instance variable, *owners*, also constrained to point only to a collection (**OwnerColl**) of owner strings.

Each collection's class definition also stipulates that its component objects must be a specific type. In our schema, **LinkColl** contains only **LinkObjs, BlkColl** contains only **BlkObjs, DocColl** contains only **DocObjs, OwnerColl** contains only

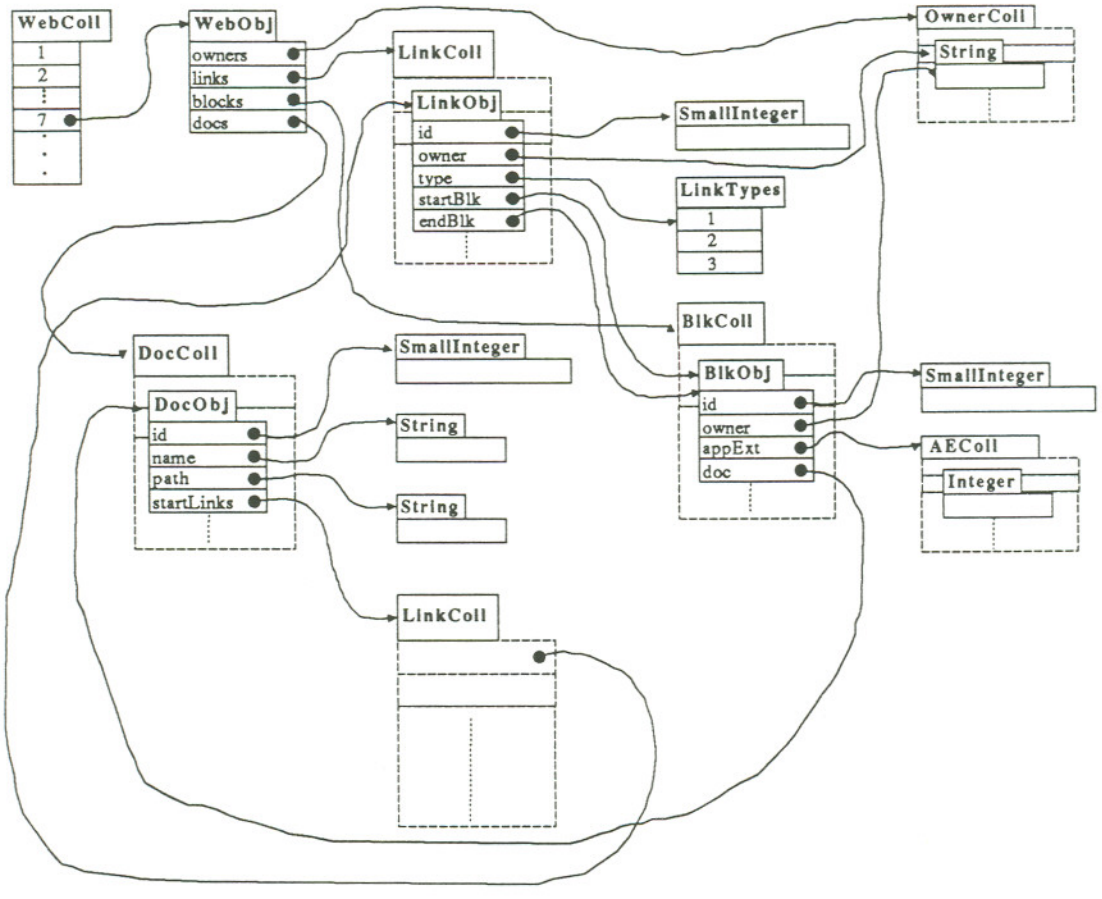


Figure 4.6

Strings, and **AEColl** (AppExtent Collection) contains only **Integers**. These constraints are necessary for adding indices.

Each **LinkObj** corresponds to one tuple in the Ingres **Link** relation; except for *startDocId* and *endDocId*, a **LinkObj** contains the same data as a **Link** tuple. The *owner* instance variable references the owner string associated with that link, while the *type* variable references a single value found in **LinkTypes**. Both *startBlk* and *endBlk* are constrained to reference specific **BlkObjs**. These **BlkObjs** are stored in **BlkColl**.

The instance variables found in a **BlkObj** correspond to those attributes found in the Ingres **Block** and **AppBlock** relations. The *owner* variable references an owner string, while *doc* is constrained to reference a **DocObj** identifying the document containing that particular block. The *appExt* instance variable must contain an **AEColl** set that stores all the *appExt* values connected with the **BlkObj**.

The *id*, *name*, and *path* instance variables found in a **DocObj** match the Ingres **Docs** relation attributes. Each points to a **String** or a **SmallInteger** value. Included in each **DocObj** is an instance variable, *startLinks*, that can only reference a **LinkColl**. This collection contains references to all the **LinkObjs** whose starting blocks are found in that particular document.

4.2.2 Database Loading

We received the data for this benchmark from the IRIS group at Brown University. The data was sent in UNIX files, formatted for direct input into Ingres relations. Since some of the UNIX files contained extra fields, some pre-processing was necessary to delete those fields. We used Ingres's COPY operation to fill the relations. The following lists the number of tuples contained in each relation.

Docs	1723 tuples
Block	1331 tuples
AppBlock	1376 tuples
Link	909 tuples

In order to load the GemStone database, we first had to define classes and

methods to accept the data during loading. Once this was accomplished, we used the GemStone Bulk Loader to load the VMS data files into the database. At this point, the data was still in pieces; we needed to "wire together" those pieces into a Web7 object. We implemented additional methods to perform this process. Finally, we invoked these methods and constructed and instantiated our Web7 object and stored it in the GemStone database.

4.2.3 Operation Implementation

In this benchmark, we wanted to ascertain what effect indexing would have on execution time and the other statistics. Consequently, our scheme had to be designed to accept indexes. Since GemStone cannot index array structures at this time, any collections we intended to index in the Hypertext schema could not be arrays.

We also wanted to determine how printing the results would affect our statistics. Therefore, for each operation, we designed three test cases. The first test did not print the results or utilize an index. The second test did print the results, but still did not execute on indexed collections. Our third test case utilized indexing, but did not print the results.

Test 1

Determine if a path exists between two documents by traversing the starting links found in each document encountered.

Implementation of this query followed the algorithm given in Figure 4.7 which computes transitive closure. We should note that when a cycle was encountered, the execution immediately halted, thus potentially overlooking valid paths. The source and destination documents are specified by their names. The set of parameters used

```
startDocName = user-defined document name.
endDocName - user-defined document name.

startDocId = DOC.id where (DOC.name = startDocName).
endDocId = DOC.id where (DOC.name = endDocName).

START <- {startDocId}.
VISITED <- {startDocId}.

while TRUE do
      END <- {LINK.endDocId} where {START.startDocId = LINK.startDocId}.

      if (END == 0)
            EMPTY SET.          /* no path exists between the two documents
                                */

      if (END.endDocId == endDocId)
            MATCH.              /* path exists between the two documents */
            quit.

      delete from END where (END.endDocId == VISITED.docId).

      if (END == 0)
            CYCLE               /* every document in END is also a member
                                of VISITED.  Therefore a cycle has been
                                encountered in the traversal and the search
                                is halted. */

      /* none of the above conditions matched - continue search */

      VISITED <- {VISITED U END}.
      START <- {}.
      START <- END.
      END <- {}.

end WHILE.
```

Figure 4.7

in this test were

1. Crusoe Defoe
2. Crusoe Footprint
3. Defoe_Sources Defoe

where the first parameter indicates the source document and the second parameter denotes the destination document.

Prior to executing the EQUEL program that queried the indexed database, we created the appropriate indexes. We indexed permanent relations only. We excluded the temporary relations because they were too small to justify the overhead of generating indexes. The **Docs** and **Link** relations were indexed, since these were the only permanent relations searched in the query. The **Docs** relation was modified to an **isam** storage structure, using *docName* as the key value. The **Link** relation had a **hashed** storage structure, declaring *startDocId* as its key value. In hindsight, we should have used the hashed storage structure for both relations, since both were conditionally searched on their key values.

The GemStone method retrieved the source and destination **DocObj** objects using the **detect:** message, which searches a collection sequentially, looking for matches. Within the **detect:** expression, we used a selection block rather than an ordinary block, signalling OPAL to execute the search operation using indexes if possible. By using selection blocks, we hoped to decrease the time spent retrieving the **DocObj** objects. We assigned the method the message selector **doesPathExistBetween:and:**. An OPAL expression demonstrating its usage is shown below.

Web7 doesPathExistBetween: 'Crusoe' and 'Defoe'

Three OPAL expressions were implemented, each specifying one of the sets of parameters defined above and incorporated into OPAL programs. Before we executed the OPAL program that would query the indexed database, we created an equality index on the *name* instance variable in the **DocObj** objects contained in **DocColl**, since this was the only collection that had to be searched. The rest of the data in this test was retrieved by following the object references.

Test 2

> **For a specific document, collect all the information necessary to display that document, its starting blocks, and the documents and end blocks linked to each starting block.**

We implemented this query following the algorithm given in Figure 4.8. As in Test 1, we defined the initial document by its name. The primary detail addressed during the implementation process in both database systems was the nature of the data structure in which we collected the data. Once we had completed that step, we then wrote the test programs.

The parameters we used were:

1. Defoe
2. Defoe_Essay
3. Pope OV

Preceding the execution of the program that queried the database containing

```
docName = user-defined document name.

docId = DOC.id where (DOC.name = docName).
docPath = DOC.path where (DOC.name = docName).

Data Structure <- docId, docPath.

TEMP <- {LINK.startBlkId, LINK.endBlkId, LINK.endDocId}
      where (LINK.startDocId = docId).

Data Structure <- {APPBLOCK.appExtent}
      where (TEMP.startBlkId = APPBLOCK.blockId).

Data Structure <- {APPBLOCK.appExtent}
      where (TEMP.endBlkId = APPBLOCK.blockId).

Data Structure <- {DOC.name, DOC.path}
      where (TEMP.endDocId = DOC.id).
```

Figure 4.8

indexes, we created **hashed** storage structures on the permanent relations that were searched (*docname* in **Docs**, *startDocId* in **Link**, and *blockId* in **AppBlock**.)

The GemStone method implemented for this test was assigned the message selector **retrieveAppExtent:** and was used in an OPAL expression resembling the following:

Web7 retrieveAppExtent: 'Pope OV'

We wrote three OPAL programs, each invoking this method with one of the parameters listed above as an argument. The same index as in Test 1 was created prior to executing the OPAL program designated to query the indexed database for the same reasons given in Test 1.

Test 3

Relocate a specific link by modifying it to point to a different ending block.

Test 4

Add a new link to the web.

These two updates were not implemented in this study, due to time limitations.

4.2.4 Execution

The execution of the tests for this benchmark and the statistic collection followed the same protocol discussed in the Document Benchmark (Section 4.1.4).

CHAPTER 5

ANALYSIS OF BENCHMARK RESULTS

This chapter presents the statistics collected for each benchmark test. Since Ingres and GemStone were run under different operating systems and on dissimilar machine configurations, direct comparison between the two is not possible. Therefore, for each benchmark, we will analyze the performance of each system in terms of the statistics generated and discuss possible reasons for those statistics.

The statistics collected in each test represent **elapsed time, cpu time,** and **I/O activity. Elapsed time** is wall-clock time to execute an operation. This value includes the cpu time consumed by the user process and the system during the operation, and overhead. Part of the overhead consists of page swapping for page faults, managing various system file structures, and maintaining the catalogue files associated with a database. The **cpu time** value represents the total cpu time consumed by both the user process and the system during operation execution. The value shown for **I/O activity** reflects the number of times the operating system had to perform reads and writes from and to secondary storage while the operation was being processed. In order to better

understand what the I/O statistic represents, below is a discussion of Ingres and GemStone's data storage and retrieval mechanisms.

Ingres is built on the UNIX file system where each relation is stored in a UNIX file on 512-byte pages. (We confirmed this number with the UNIX system call **statfs**.) UNIX employs a buffer pool (cache) whose size is 40 pages, established by Ingres. All file I/O is handled through this buffer pool. During a file **read**, the buffer pool is searched first to determine if the desired page is in the pool. If the page is already resident, no I/O occurs and the data is returned directly to the caller. If the page is not found in the buffer pool, it must be swapped in from secondary memory. In some cases, this action could require two I/O calls, one to swap in a piece of the page table, and another to transfer in the required data itself. At the time of a file **write**, the updated data is written to the buffer pool. When the buffer pool is full (or the entire relation is updated), the pages from the pool are written to secondary storage.

Within each Ingres database, there are six system relations, automatically created and filled by Ingres, that describe the database. During program execution, these relations are referenced and updated. Consequently, they are also being swapped in and out of the buffer pool, contributing to I/O.

In GemStone, each object is referenced by its *object-oriented pointer* (OOP). OOPs are allocated by the Stone process, which is built on the VMS file system. The OOP values for all the objects in a database are stored in an *object table*. The object table maps each OOP to its physical location. Although an object table can conceivably contain 2^{31} possible OOPs [MSO86], the designers expected that the portion of the object table for objects needed during a session would be small enough to fit

into main memory. The OOPs for each object's instance variable values are clustered together (except for large objects), although the objects they reference may be spread around.

All object manipulation is handled via the object's OOP. The only time secondary storage is accessed is when specific object values are requested, i.e., printing, updates, and various operations (arithmetic, comparisons, string manipulation, etc.). Therefore, the I/O generated by a GemStone program might represent retrieving data from secondary storage and writing data to secondary storage. The values rarely include object table operations, since theoretically, the entire object table resides in main memory and no extra I/O is incurred from page-swapping. Also, because GemStone does not require extraneous system files to support a database, there is no additional I/O due to system file maintenance. As in Ingres, GemStone employs large buffer pools for file I/O.

We first discuss the Document benchmark statistics, then the statistics on the Hypertext benchmark, and afterwards comment on specific benchmark issues.

5.1 THE DOCUMENT BENCHMARK

Test 1

> **Replace all word and text occurrences of a word** (the target word) **with another word** (the replacement word).

The results for Test 1 are given in Figure 5.1, which includes the number of tuples updated in both the **Line** and **Text** relations.

In the Ingres implementation, the entire **Line** and **Text** relations were searched in each test run. Therefore, a portion of the I/O statistics reflects the fact that the entire relation had to be brought into memory. Included in this number is I/O resulting from reading the page table. If the page table was too large to fit into main memory, the I/O may be increased from swapping in the appropriate pages of the page

---

Test 1.1 - 'a' -> 'A'

Tuples updated: 1035	**Ingres**	**Gemstone**
Elapsed Time	85.22 sec	382.57 sec
CPU Time	51.92 sec	356.79 sec
Direct IO	1497	596

Test 1.2 - 'the' -> 'the'

Tuples updated: 1851	**Ingres**	**Gemstone**
Elapsed Time	95.70 sec	380.07 sec
CPU Time	58.27 sec	355.61 sec
Direct IO	1632	605

Test 1.3 - 'process' -> 'PROCESS'

Tuples updated: 267	**Ingres**	**Gemstone**
Elapsed Time	63.03 sec	323.84 sec
CPU Time	45.84 sec	312.38 sec
Direct IO	927	233

Figure 5.1

---

table. The I/O statistic also includes the I/O generated from reading and updating the system catalogue files associated with the **Doc** database.

Since the **Line** and **Text** relations had to be completely searched in all three tests, we hypothesized that the difference in the statistics between the tests was due to the difference in the number of tuples updated. Recall that when a tuple is updated, the revisions are stored in the buffer pool. When the pool is full or the entire relation is updated, the pool is then written to secondary storage. Therefore, as the number of tuples that needed updates increased, the number of revisions stored in the buffer pool and the number of writes to secondary storage increased. To test this hypothesis, we designed a program that tested a parameter not found in the document ('becky' -> 'BECKY'). By doing this, we hoped to approximate the I/O when no updates occurred. We found that, without updates, the tests would generate approximately 560 I/O hits. Therefore, the difference in the elapsed time and I/O between the three tests appears due to the variation in the number of update operations that occurred.

In the GemStone implementation, execution entailed a depth-first search of the entire **Document** object. Updating the **Word** and **Text** values required retrieving those values from disk. We hypothesized that the majority of I/O activity generated by the GemStone tests was due to accessing the disk to retrieve the **Word** and **Text** values. Although it would seem reasonable to assume that Test 1.2 would generate the largest values, due to the greatest number of tuples to update, this was not reflected in the GemStone statistics. The results of Test 1.1 and Test 1.2 are nearly identical. At this time, we have no reasons for why this happened.

Test 2

Relocate specific chapters, sections, paragraphs, or lines within the document.

Figure 5.2 shows the statistics generated in Test 2. In the Ingres implementation, we used Ingres's ORDEREDN operation to maintain an order on the

---

Test 2.1 - 'chapter 5 2'

	Ingres	Gemstone
Elapsed Time	9.56 sec	1.02 sec
CPU Time	5.89 sec	0.88 sec
Direct IO	121	5

Test 2.2 - 'paragraph 32 22'

	Ingres	Gemstone
Elapsed Time	178.54 sec	1.24 sec
CPU Time	163.94 sec	1.01 sec
Direct IO	261	7

Test 2.3 - 'line 795 2208'

	Ingres	Gemstone
Elapsed Time	1456.43 sec	1.66 sec
CPU Time	1414.28 sec	1.34 sec
Direct IO	2002	10

Figure 5.2

---

relations. As the size of a relation increases, the time it takes to maintain order on the relation increases. Notice the difference in the elapsed time values between Test 2.1 and Test 2.3. In Test 2.1, the **Chapter** relation, containing 37 tuples, was modified and re-ordered. In Test 2.3, we modified and re-ordered the **Line** relation, which contained 19,711 tuples.

The elapsed time value includes the time spent searching a relation for the desired tuples, inserting and deleting the appropriate tuple, and re-ordering the relation. We hypothesize that the majority of the elapsed time and I/O is spent re-ordering the relation.

In GemStone, it is unnecessary to retrieve the entire collection of objects to reorder elements. The relocation operation involves rearranging the objects' OOPs rather than the objects, and thus, requires no disk access for specific data values in the objects. Since the OOPs are stored in an object table and it is assumed that the object table can completely fit into main memory, the only I/O required is to bring the object table into memory.

Reviewing the statistics on the three tests, we noticed a slight increase in the elapsed time and I/O from Test 2.1 to Test 2.3. We don't feel the differences are very significant, and could be attributed to the size of the arrays that had to be searched. In most instances, a **Line** array was larger than a **Chapter** or a **Paragraph** array. The results are so close between the three tests that the differences might be due to run-time variation and not program implementation or scheme design.

Test 3

Find every occurrence of a specific word (the target word) **and return the chapter, section, paragraph, line id and word position associated with each occurrence of that word.**

The statistics generated for Test 3 are shown in Figures 5.3-5.5. Each query included tests that printed the results and tests that didn't print the results.

In the Ingres implementation, the chapter, section, paragraph, and line ids and the word position value for each occurrence of a particular word in the document had to be retrieved. In order to facilitate this retrieval, the **Chapter, Section, Paragraph,**

Test 3.1 - 'and'

Number of occurrences: 431

Part 1 - No Results Printed	**Ingres**	**Gemstone**
Elapsed Time	640.93 sec	298.03 sec
CPU Time	609.82 sec	289.20 sec
Direct IO	1426	16*
Part 2 - Results Printed	**Ingres**	**Gemstone**
Elapsed Time	647.85 sec	442.70 sec
CPU Time	611.54 sec	388.38 sec
Direct IO	1459	848

* These results are suspect. See explanation in text.

Figure 5.3

Test 3.2 - 'parallel'

Number of occurrences: 89

Part 1 - No Results Printed	**Ingres**	**Gemstone**
Elapsed Time	453.39 sec	217.80 sec
CPU Time	422.37 sec	206.83 sec
Direct IO	1354	106
Part 2 - Results Printed	**Ingres**	**Gemstone**
Elapsed Time	454.37 sec	251.76 sec
CPU Time	423.70 sec	234.41 sec
Direct IO	1392	360

Figure 5.4

and **Line** relations had to be joined. Each relation had to be read into main memory, searched, and the appropriate tuple returned to the program. All of this activity contributed to the elapsed time and I/O. When we initially began reviewing the statistics, we were surprised that the elapsed time values were so much longer than those generated in Test 1, since we were only retrieving the words, not updating them. Further consideration, though, led us to realize that, in this operation, four relations were joined, one containing over 19,000 tuples (**Line**). Since a join can be an expensive operation, and four joins proportionately more expensive, we concluded that these large numbers weren't so surprising after all.

We noticed that there was very little difference in the Ingres statistics between the two parts of each test. This was expected because, for any operation, Ingres must retrieve the data in each relation from secondary storage. Since the data is already in

Test 3.3 - 'fork'

Number of occurrences: 94

Part 1 - No Results Printed	**Ingres**	**Gemstone**
Elapsed Time	470.74 sec	224.49 sec
CPU Time	437.02 sec	213.40 sec
Direct IO	1390	128
Part 2 - Results Printed	**Ingres**	**Gemstone**
Elapsed Time	466.30 sec	255.68 sec
CPU Time	436.53 sec	239.45 sec
Direct IO	1348	316

Figure 5.5

main memory, we felt printing the data shouldn't significantly affect the I/O. In Test 3.3, the I/O for Part 2 is less than Part 1 and suggests that there is very little overhead incurred when data is printed. We speculate that the difference between the two parts is negligible and might be attributed to run-time variation.

When implementing this query in GemStone, the entire **Document** object was searched, depth-first. Each word was accessed and compared to the target word. If the word matched, the **Chapter, Section, Paragraph, Line**, and **Word** array positions leading to that word were stored in a separate object. We expected Test 3.1 to yield much higher values than Test 3.2 and Test 3.3, since it retrieved the largest number of values. But, as can be seen, at least in Part 1, this was not the case. This may be explained as follows.

In the initial program execution for Part 1, the tests were executed in the

following order: Test 3.1, Test 3.2, Test 3.3. Later, we speculated that Test 3.1 might be exhibiting high statistics based on the fact that it was the first to execute and had incurred unnecessary overhead due to system warm-up and page retrieval. Test 3.2 and Test 3.3 would not be affected since the data was already resident in memory. Therefore, we devised a second run that executed Test 3.1 by itself 11 times, and ignored the first results. The database was opened at the beginning of the run and closed upon completion of the run. These are the results reported. Since the database was not closed between each test execution, we suspect these results are too low. We should have repeated the initial batch program and executed Test 3.1 twice, once at the beginning and once at the end, to obtain more representative results. Due to time constraints and machine unavailability, though, we were unable to repeat this test. Consequently, the results of Test 3.1, Part 1, cannot be considered valid.

The difference in the GemStone elapsed time and I/O between Parts 1 and 2 were expected. In Part 1, the data structure that was created and instantiated with the retrieved values was not printed, but stored on disk. In Part 2, the data structure was retrieved and each object printed out. The I/O value reflects the I/O activity to print to a file.

Test 4

Retrieve a specific textline designated by its position in the document.

The results for Test 4 are shown in Figures 5.6-5.9. As in Test 3, each query included tests that printed the results and tests that did not.

Test 4.1 - 'line 651' (chapter 3 section 1 paragraph 3 textLine 6)

Part 1 - No Results Printed	**Ingres**	**Gemstone**
Elapsed Time	4.56 sec	0.75 sec
CPU Time	4.25 sec	0.60 sec
Direct IO	62	6
Part 2 - Results Printed	**Ingres**	**Gemstone**
Elapsed Time	4.70 sec	0.57 sec
CPU Time	4.27 sec	0.48 sec
Direct IO	62	4

Figure 5.6

When the Ingres database was queried, the entire **Text** relation was searched for the tuple corresponding to the specified unique tuple id. Therefore, part of the I/O reflects the number of I/O hits taken to transfer the entire relation into main memory

Test 4.2 - 'line 1192' (chapter 4 section 8 paragraph 11 textLine 3)

Part 1 - No Results Printed	**Ingres**	**Gemstone**
Elapsed Time	4.50 sec	1.43 sec
CPU Time	3.51 sec	0.96 sec
Direct IO	54	6
Part 2 - Results Printed	**Ingres**	**Gemstone**
Elapsed Time	4.80 sec	1.45 sec
CPU Time	4.26 sec	0.87 sec
Direct IO	60	12

Figure 5.7

Test 4.3 - 'line 2526' (chapter 7 section 1 paragraph 1 textLine 250)

Part 1 - No Results Printed	**Ingres**	**Gemstone**
Elapsed Time	4.54 sec	1.18 sec
CPU Time	4.22 sec	0.81 sec
Direct IO	55	9
Part 2 - Results Printed	**Ingres**	**Gemstone**
Elapsed Time	4.42 sec	0.83 sec
CPU Time	4.25 sec	0.66 sec
Direct IO	60	7

Figure 5.8

and read it, because Ingres does not stop once it has located the correct tuple. Since this query performed no updates to the database, we expected the I/O to reflect only the activity generated by file reads. We postulated that the only file write included in the I/O statistic would be when the text line was printed. Because the data was already in main memory, this write action should have contributed very little to the I/O. Therefore, we expected the results between Parts 1 and 2 in each test to be fairly similar. Reviewing all three tests, our expectations were proven correct. Also, since each test was theoretically performing exactly the same operation (i.e., retrieving exactly one tuple from the **Text** relation), we anticipated that the elapsed time values between each test would be very similar. This also was demonstrated in the statistics.

For this query, we did not have to code a specific GemStone method, but could retrieve the desired text line utilizing the primitive method **at:**. Consequently, weexpected fairly low response times and I/O in our results. The statistics verified our

expectations. The negligible differences between Part 1 and Part 2 in each test might be attributed to the data having already been retrieved into memory, and thus, writing the result to the file had very little impact on the statistics.

We hypothesized that since each test executed approximately the same operation, the statistics between the three tests would be fairly close. The actual results support this hypothesis. The elapsed times are so small that the differences between the runs can be regarded as insignificant, and again, possibly attributed to run-time variation.

5.2 THE HYPERTEXT BENCHMARK

Test 1

Determine if a path exists between two documents by traversing the starting links found in each document encountered.

The statistics generated in Test 1 are shown in Figures 5.9-5.11. Recall that this query incorporated an infinite loop that was exited when an answer was determined (**EMPTY SET, MATCH,** or **CYCLE**). The number of loops each test cycled through before an answer was generated is shown in the Figures.

Reviewing the Ingres statistics, we found that, as the number of loops cycled through increased, the elapsed time values lengthened. For example, Test 1.2 required the least number of cycles to return an answer and exhibited the shortest elapsed time.

Test 1.1 - Crusoe Defoe -> EMPTY SET		
Loops Cycled: 6		
Part 1 - No Results Printed	**Ingres**	**Gemstone**
Elapsed Time	58.59 sec	14.60 sec
CPU Time	34.74 sec	14.10 sec
Direct IO	811	5
Part 2 - Results Printed	**Ingres**	**Gemstone**
Elapsed Time	58.70 sec	14.79 sec
CPU Time	34.66 sec	14.44 sec
Direct IO	802	8
Part 3 - With Indexes, No Results Printed	**Ingres**	**Gemstone**
Elapsed Time	52.23 sec	14.90 sec
CPU Time	28.20 sec	14.44 sec
Direct IO	786	8

Figure 5.9

Test 1.3 displayed the longest elapsed time and cycled through the loop the greatest number of times. When we examined the Ingres program implementing this query, we determined that, for each cycle through the loop, up to four separate relations had to be searched (**Link, Start, End**, and **Visited**). In Tests 1.1 and 1.2, the size of the **Start** and **End** relations were fairly small, each containing one tuple per cycle and **Visited** incrementing by one in each loop. But, in Test 1.3, the size of these three relations was substantially larger. For example, in loop 5, **Start** and **End** in Tests 1.1 and 1.2 each contained 1 tuple and **Visited** contained 5 tuples. In contrast, in Test

Test 1.2 - Crusoe Footprint -> MATCH		
Loops Cycled: 5		
Part 1 - No Results Printed	**Ingres**	**Gemstone**
Elapsed Time	49.36 sec	8.07 sec
CPU Time	30.14 sec	7.74 sec
Direct IO	681	5
Part 2 - Results Printed	**Ingres**	**Gemstone**
Elapsed Time	50.36 sec	8.07 sec
CPU Time	29.98 sec	7.70 sec
Direct IO	690	5
Part 3 - With Indexes, No Results Printed	**Ingres**	**Gemstone**
Elapsed Time	45.29 sec	6.57 sec
CPU Time	24.56 sec	6.32 sec
Direct IO	677	6

Figure 5.10

1.3, **Start** and **End** contained 72 tuples and **Visited**, 279 tuples. We therefore were not surprised with these statistics because searching and comparing relations requires time, and the larger the relations to search and the more times they must be searched and compared, the longer the execution time. Examination of the I/O revealed the same trend; as the number of loops to cycle through increased, the I/O increased.

We anticipated that the Ingres results between Parts 1 and 2 in each test would not be significantly different and this was proven by the statistics. As covered in the Document discussion, the elapsed time and I/O were not increased significantly by

Test 1.3 - Defoe_Sources Defoe -> CYCLE

Loops Cycled: 9

Part 1 - No Results Printed	**Ingres**	**Gemstone**
Elapsed Time	487.89 sec	122.74 sec
CPU Time	440.76 sec	116.39 sec
Direct IO	1975	64
Part 2 - Results Printed	**Ingres**	**Gemstone**
Elapsed Time	491.46 sec	122.12 sec
CPU Time	440.85 sec	115.55 sec
Direct IO	2015	69
Part 3 - With Indexes, No Results Printed	**Ingres**	**Gemstone**
Elapsed Time	177.77 sec	135.35 sec
CPU Time	135.04 sec	128.36 sec
Direct IO	1478	115

Figure 5.11

printing results, since the data already resided in main memory. We did expect, and the results demonstrated, that the values in Part 3 would be lower than Part 1, due to the indexes created. In Part 3, both the **Docs** and **Link** relations were indexed. During query execution, the **Docs** relation was searched once, while the **Link** relation was searched as many times as the loop was entered. Tests 1.1 and 1.2 did not exhibit large differences between Parts 1 and 3 (Part 3 in Test 1.1 ran 1.12 times faster than Part 1, while, in Test 1.2, Part 3 executed 1.09 times faster). But, Test 1.3 did display a considerable difference between the two parts, Part 3 executing faster that Part 1 by

a factor of 2.94. We surmise this can also be attributed to the size of the relations examined in each cycle.

In the GemStone statistics, we noticed the same pattern as seen in Ingres. Test 1.2 executed in the least amount of time; Test 1.3 ran the longest. Again, we suggest this was due to the number of loops each test had to cycle through. The larger I/O in Test 1.3 may be correlated to the size of the temporary collections that needed to be searched and compared.

The GemStone statistics between Parts 1 and 2 show little or no difference. The answers generated by this query were pre-defined strings, signifying the outcome of the query. There was no data collected or stored in an object to be used at another time. Consequently, we suspect the GemStone overhead when writing the answer strings was negligible and did not affect the elapsed time or I/O substantially.

Prior to executing Part 3, an equality index was created on the *name* instance variable for all the **DocObj** elements in the **DocColl** collection. Since we were performing an equality search on the *name* instance variable (which required accessing the disk for the values), we hypothesized that the search would take less time than sequentially searching the entire **DocColl** collection. Within the GemStone method for this query, there were two searches on the **DocColl** collection, one to retrieve the starting **DocObj** element, and the other to retrieve the ending **DocObj** object. We did not create indexes on the temporary collections due to the overhead in creating indexes. We also did not create an index on the **LinkColl** set, since there was no need to search it for matches. Once we had the required **DocObj** objects, we were able to retrieve their associated **LinkObj** objects by following the *startLinks* reference. Recall

that *startLinks* in **DocObj** references a **LinkColl** set that contains references to all the **LinkObj** objects whose starting point is in that document. (Refer to Figure 4.6.) Therefore, a search of the global **LinkColl** set was unnecessary.

We expected the results in Part 3 to be lower than Part 1. The index should have decreased the elapsed time, due to the decrease in search time. The I/O should also have been decreased because there would not be as many file reads to retrieve the *name* values.

As can be seen in our results, this did not always happen. The elapsed time value in Test 1.2, Part 3, is less than in Part 1, but the I/O is slightly larger. Tests 1.1 and 1.3 display greater elapsed time and I/O in Part 3 than in Part 1. This behavior demonstrates that indexing does not always improve performance. When indexes are specified, index pages are created. Consequently, when a query is performed on an indexed object, there is a small overhead incurred when retrieving the index pages. Also, different OPAL methods may be used on indexed objects, possibly necessitating more pages to be retrieved. Therefore, if the amount of data looked up is small, the overhead is not recovered.

Test 2

> **For a specific document, collect all the information necessary to display that document, its starting blocks, and the documents and end blocks linked to each starting block.**

The results for Test 2 can be seen in Figures 5.12-5.14, including their number of starting links.

The Ingres results generated in Parts 1 and 2 of each tests, as expected, were not significantly different. In Part 3, where indexes were created on the **Docs, Link,** and **AppBlock** relations, the elapsed time was decreased by approximately 40% in each test; the I/O decreased slightly. Utilizing indexes did decrease the execution time of the query significantly between Parts 1 and 3 (overall, the elapsed time increased with the number of starting links within each document). This behavior makes sense, since

Test 2.1 - Defoe

Starting Links: 7

Part 1 - No Results Printed	**Ingres**	**Gemstone**
Elapsed Time	62.08 sec	4.12 sec
CPU Time	57.70 sec	3.94 sec
Direct IO	262	4
Part 2 - Results Printed	**Ingres**	**Gemstone**
Elapsed Time	61.93 sec	9.17 sec
CPU Time	57.74 sec	7.44 sec
Direct IO	260	31
Part 3 - With Indexes, No Results Printed	**Ingres**	**Gemstone**
Elapsed Time	37.07 sec	3.31 sec
CPU Time	32.37 sec	3.22 sec
Direct IO	248	4

Figure 5.12

Test 2.2 - Defoe_Essay

Starting Links: 8

Part 1 - No Results Printed	**Ingres**	**Gemstone**
Elapsed Time	67.67 sec	8.67 sec
CPU Time	63.28 sec	7.24 sec
Direct IO	265	40
Part 2 - Results Printed	**Ingres**	**Gemstone**
Elapsed Time	67.80 sec	13.82 sec
CPU Time	63.29 sec	11.16 sec
Direct IO	258	63
Part 3 - With Indexes, No Results Printed	**Ingres**	**Gemstone**
Elapsed Time	39.80 sec	4.29 sec
CPU Time	34.72 sec	3.76 sec
Direct IO	252	14

Figure 5.13

the amount of information to be retrieved is directly proportional to the number of starting links.

The GemStone results agreed with our expectations. The larger values in Part 2 versus Part 1 can be attributed to the system's need to access secondary storage for the data to be printed. In Part 1, references to objects are collected, which does not require retrieving the object from the disk. In Part 2, the state of the referenced object is fetched from disk. Part 3 did generate the type of numbers we expected from indexing in GemStone, i.e., the elapsed times were significantly lower than those found

Test 2.3 - Pope OV Starting Links: 18		
Part 1 - No Results Printed	**Ingres**	**Gemstone**
Elapsed Time	121.20 sec	6.59 sec
CPU Time	114.78 sec	5.88 sec
Direct IO	343	20
Part 2 - Results Printed	**Ingres**	**Gemstone**
Elapsed Time	121.89 sec	20.01 sec
CPU Time	114.85 sec	15.85 sec
Direct IO	340	83
Part 3 - With Indexes, No Results Printed	**Ingres**	**Gemstone**
Elapsed Time	71.91 sec	6.23 sec
CPU Time	65.88 sec	5.21 sec
Direct IO	271	32

Figure 5.14

in Part 1. In Test 2.1, Part 3 executed approximately 20% faster than Part 1. Part 3 in Test 2.2 was approximately 50% faster than Part 1, and in Test 2.3, the elapsed time in Part 3 was faster than Part 1 by approximately 6%. In this query, we created the same index as in Test 1 and searched the **DocColl** collection once. It is difficult to say, at this time, why indexing decreased the elapsed times in this test and made little difference in Test 1.

We expected that Part 3 would demonstrate significantly lower I/O than Part 1 due to indexing. But, this was not the case; Test 2.3 displayed greater I/O in Part 3

than in Part 1. As explained in Test 1, this can be attributed to the overhead incurred reading in index pages and executing index methods.

Unlike the Ingres results, we were unable to correlate the GemStone results to the number of starting links present in a document. Test 2.3 retrieved information on 18 starting links and generated smaller values than Test 2.2, which dealt with 8 links.

5.3 GENERAL COMMENTS

In this section, I comment on various characteristics of each database system and specific benchmark issues. In particular, I address the degree of ease or difficulty in learning each system, database loading, and general system implementation details. I also discuss several specific issues I encountered during each benchmark's design and development.

5.3.1 The Design Environment

The set of QUEL operations in Ingres was small and not very difficult to master. Since I was already familiar with the C programming language, incorporating the benchmark tests into EQUEL programs was fairly straightforward and presented few complications. The learning curve for GemStone, and especially OPAL, was much steeper than Ingres. In the beginning, I knew very little about object-oriented

languages and had to become familiar with the concepts inherent in these languages. Most of the time involved in mastering GemStone was spent learning OPAL; once I was proficient in OPAL, familiarizing myself with the rest of the GemStone system required very little time.

Because I spent the majority of my time learning the database systems while implementing the Document benchmark, that benchmark required approximately twice as much time to complete than the Hypertext benchmark. As I became familiar with each system during the Document benchmark, I realized initial designs could be revised to yield better implementations and more optimal results. In particular, I was able to decrease the number of QUEL commands used to execute an operation, thereby potentially decreasing execution times. Within OPAL, I found that incorporating as many primitive methods as possible in my expressions would significantly reduce response times. In addition, by avoiding message passing within loop conditions in OPAL, I could again decrease the execution times somewhat. As a result of all that I learned during the Document benchmark implementation, the Hypertext benchmark development required less time and was easier to accomplish. Additionally, during the implementation of the Hypertext application, I continued to discover optimization techniques that could apply to the Document application. Therefore, I would backtrack and revise the Document benchmark.

In terms of database loading, I found the Ingres COPY operation to be substantially more efficient than GemStone's Bulk Loader. Loading the Ingres database for the Document benchmark required approximately 5 minutes, in contrast to the GemStone database with took approximately 3 hours to load. (Loading the Hypertext

database in Ingres required approximately 1 minute; loading the GemStone database was about 1 1/2 hours.) Once the Ingres databases were loaded, they were immediately ready to use. Before I could query or update the GemStone databases, though, I had to construct the appropriate data objects. For the Document benchmark, I needed to build the **Document** object, which took approximately 40 minutes; the **Web7** object for the Hypertext benchmark required approximately 1 hour for construction. Therefore, the total time spent preparing each GemStone database for use in each benchmark was approximately 4 hours (the Document benchmark) and 2 1/2 hours (the Hypertext benchmark). Comparing these times to those of Ingres (5 minutes and 1 minute, respectively), it is apparent that populating the databases in Ingres requires significantly less time than that needed in GemStone.

During implementation of the Hypertext benchmark, both the Ingres and GemStone schemas were revised. In Ingres, the revision entailed redefining the **Docs** relation to include a new field, incorporating the new data into the proper UNIX file, and copying the data from that file into the new **Docs** relation. The entire process did not take long (approximately 1 hour, 30 minutes user time and 30 minutes machine time) and once it was completed, the database was ready to use.

In revising the GemStone schema, I had to redefine the pertinent class definitions and re-commit them to the database. I also needed to reformat the raw data to coincide with the revised schema and reload the database. Finally, the **Web7** object had to be constructed. The entire revision process required approximately 8 hours to accomplish (2 hours user time and 6 hours machine time).

I concluded that revising an Ingres schema is not very difficult and that the

COPY operation facilitates easy database loading and unloading. When revising a GemStone schema, redefining the classes and committing them to the database requires little time. The bulk of the revision time is spent dumping and reloading the database and constructing the required data objects. Depending on the size of the database and the constructed objects, this process could require a substantial amount of time, and in most cases, would be desirable to avoid, if possible.

5.3.2 Getrusage/Gettimeofday Issues

Within the EQUEL programs used in both benchmarks, the statistics did not take into account the C code that was included in the operations, since the C code was executed in the parent process, and the **getrusage** operation was declared to collect statistics on the child process. Consequently, I needed to determine if the included C code generated significant values that would need to be included in the performance analysis results, or if the results proved negligible, and no test re-execution was necessary. The programs that incorporated a substantial amount of C code were redesigned and rerun. The results indicated that the included C code statistics were negligible, and thus, it was not mandatory that the entire suite of tests be rerun. To be absolutely accurate, though, I feel future work should include **getrusage** calls to both the parent and child processes to generate more representative statistics.

Each test had two programs associated with it, one invoking **gettimeofday** for elapsed time measurements, and one calling **getrusage**, measuring the cpu and I/O

activity. I separated the system calls into two programs because I felt that the **getrusage** statistics would include the system work generated by **gettimeofday**. In hindsight, though, I realized that both calls could have been incorporated into one program. As long as **getrusage** was only collecting the statistics on the child process, the numbers generated by **gettimeofday** would not have been included. If future programs collected statistics on both the child and parent processes during query execution, additional work would be necessary to determine the system overhead incurred by **gettimeofday**. I suspect the overhead is minimal and would have little effect on the **getrusage** statistics.

5.3.3 Repeat Runs

In the Ingres programs, each set of tests was invoked in the same order in each cycle of the loop. I was afraid that the statistics of the first test might have exhibited larger values due to system warm-up (i.e., loading the database relations into memory), while the other two tests may have demonstrated different statistics, due to caching. It was necessary, therefore, to determine if my premise above was accurate. For each test, I designed repeat programs that invoked two of the three tests, with the original first test invoked last. The statistics collected from these repeat runs indicated that the original statistics were valid; the results between the initial runs and the repeat runs were not significantly different. Hence, I was able to use my original statistics in my analysis.

The original OPAL programs had the same format as found in the Ingres programs. As a result, I also designed new OPAL programs that reran the tests in a different order than the original tests. My results in the original runs demonstrated that the first test executed demonstrated system warm-up, increasing the statistics by approximately 1.5%. Therefore, to be consistent with the other two tests that were not affected by system warm-up, I chose to disregard the original statistics generated for the first test and use the statistics from the repeat run. The exception to this is Test 3.1, Part 1. For the reasons discussed earlier, these results must be considered invalid. It may be argued that the approach is not valid because practical applications must allow for system warm-up. But, I chose to use this approach to maintain consistency between all the test runs.

5.3.4 Specific Benchmark Issues

The reader may question why indexes were included in the Hypertext benchmark, but not in the Document benchmark. At this time, GemStone does not support indexes on array structures. I felt, however, that the array structure was the best structure to use in the GemStone Document schema because it reflected the conceptual model so closely. In many cases, a particular structure is better suited to an application and will generate better response times when used in place of indexes. In my case, I decided the array structure was more advantageous for the Document application than incorporating indexes into the schema. In taking this approach, though,

Ingres was placed at a disadvantage. Had indexes been used in the Ingres implementation, the statistics might have demonstrated faster execution times and decreased I/O activity. But, I wanted each system's implementation characteristics to be as identical as possible, which precluded using indexes in the Ingres schema.

In Test 1 in the Document benchmark (i.e., replacing words), the Ingres scheme proved suitable. The **Line** and **Text** relations represented sets that were searched during the query. Due to the nature of the scheme, I was able to execute the query without referencing relations other than the two mentioned above, and thus, avoided any join operations.

Recall that in Test 2 and Test 4 in the Document benchmark, the Ingres implementations required that the user specify the globally unique tuple id associated with the desired document component, i.e., *line 2526*. There are two alternatives I could have, and in hindsight, should have considered. One alternative would have been to include a field in each relation in the scheme that contained the relative *position* of that relation's component in the document (see Figure 5.15). Then the parameters in Test 2 and Test 4 could define a component by its relative position in the document. The EQUEL program would include code that would convert the relative position values to globally unique tuple ids and then perform the required operation. In Test 2, after the relocation operation was performed, the EQUEL program would also need to update the values in the *position* field to reflect the new order in the document. Including a *position* field in the scheme might have been difficult to instantiate, but might have decreased the update response time in Test 2, since Ingres's ORDEREDN operation would have been by-passed and my own ordering routine

Relation Chapter

chapId	*sectId*	*crelpos*

Relation Section

sectId	*paraId*	*srelpos*

Relation Paragraph

paraId	*lineId*	*tLineId*	*prelpos*

Relation Line

lineId	*wordPos*	*word*	*lrelpos*

Relation Text

tLineId	*text*

Figure 5.15

executed. It's difficult to state at this time if this revised schema would generate faster response times or decreased I/O.

The other alternative would have been to allow the user to use relative positions in the parameters and include code in the EQUEL program that converted the parameters to their globally unique tuple ids. Implementing this design would have been easier and faster than the previous alternative. The execution times would no doubt be longer than the present times, due to the parameter conversion. Also, this approach would require that a separate relation be included containing the total number of components found in the document to facilitate the conversion. The relation would need to be updated when the size of the document changed. This approach also

modifies the database scheme. Since the objective in this benchmark was to use the scheme designed by Stonebraker, et al., with no modifications, both alternatives would have modified the schema, contrary to the objective.

Ideally, I should have been consistent between Ingres and GemStone, i.e., used either relative positions or unique tuple and object ids as parameter arguments. In retrospect, the easiest and possibly best solution would have been to define the query parameters for each system in terms of a component's position within the document. This type of declaration reflects the real-world structure of the data.

Finally, the results printed by Ingres in Test 3 of the Document benchmark, the global unique tuple ids are used. For example, for the word found at position **chapter 3 section 2 paragraph 1 line 3 word position 7**, Ingres printed out "chapter 3 section 5 paragraph 50 line 196 word position 7". This type of response does not convey the relative position of that word in the document. Again, as in Test 2, I should have been consistent between the two systems and had Ingres also print the component's relative position. Printing the relative position would have required additional EQUEL to convert the Ingres response above to the desired notation, resulting in longer response times.

In the Hypertext benchmark, one objective was to determine the effect indexing would have on the statistics. Before I could create the indexes, I had to declare constraints on all the instance variables that I intended to index. I encountered problems in doing so. OPAL stipulates that an instance variable may not be constrained to an object from a class not previously defined. In defining the set of classes needed for the Hypertext schema, no matter how I arranged the classes, I

always encountered one class whose instance variable was constrained to a class not yet defined. This problem was finally resolved by utilizing OPAL's class hierarchy mechanism. Each problem class was divided into two separate classes. The first class (the superclass) contained a subset of the class's original instance variables. The second class was declared a subset of the first class, thereby inheriting the first class's instance variables. The rest of the class's intended instance variables were included in the second class. By splitting the problem classes into two distinct classes and taking advantage of OPAL's class hierarchy, I was able to circumvent this *circular constraint* problem.

The GemStone documentation recommends constraining only those instance variables that are to be indexed. During the initial design phase in the Hypertext benchmark, I thought I would need to create many more indexes than I actually did, so I declared constraints on numerous instance variables. Since my final implementation created one index on one instance variable, I could have revised my GemStone schema to remove the unnecessary constraints and possibly avoided the circular constraint problem.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

Our initial objective in this thesis was to compare Ingres and GemStone using our benchmarks. But, early in the design stage, it became apparent that we could not accomplish this objective. The version of Ingres we were testing ('university') runs under UNIX, while GemStone is implemented on top of VMS. Since each was run under different operating systems, direct comparison of the two systems was unrealistic. Consequently, the premise of the thesis was revised to concentrate on developing two benchmarks that could ultimately be used to compare relational and object-oriented database systems. These benchmarks specifically addressed applications that incorporated complex entities as data objects.

Although 'university' Ingres is a well-respected relational database system and served our needs suitably in this study, the statistics it generated should not be considered valid for 'commercial' Ingres. 'University' Ingres does not contain the performance enhancements found in 'commercial' Ingres, and therefore, does not perform as efficiently as 'commercial' Ingres. To obtain a more representative

perspective of Ingres's capabilities and proficiency, 'commercial' Ingres running under VMS should be used. In this way, a more direct comparison of Ingres and GemStone would be possible. As stated in Chapter 2, this scenario would have been desirable, but we did not have access to 'commercial' Ingres.

We stress that the relational schemas were not completely optimized, especially the Document schema. In addition, the EQUEL code written for the Document and Hypertext benchmarks was not extensively reviewed or revised for optimal performance. Consequently, we cannot take our statistics at face value and conclude that they truly reflect 'university' Ingres. Instead, we must view the statistics with caution and realize that the results may have been more favorable had the above steps been performed.

Within each database system, we found that the system's performance is dependent on the schema implemented. For example, since the GemStone Document schema was designed so that the word and text values could only be retrieved by performing a depth-first search, a significant amount of time was spent executing the search when those values were desired. A better approach would have been to modify the schema so that the word and text values were stored in collections, with the words contained in a **Set** object to eliminate duplications, and the text values kept in a **Bag** object. In this way, word and text replacement operations could be executed more quickly, since the depth-first search would be avoided. If only one occurrence of a word was to be modified, a depth-first search would still need to be performed to locate the exact line that contained the word to be modified. At that time, the word in the text could be modified, and the **Set** collection searched for the desired replacement word. If it did not exist, it would be appended to the **Set** collection and

referenced.

Relational schemas should be evaluated to determine if they can be optimized to obtain better performance. A designer needs to ascertain if it is better to maintain a "normalized" schema or combine various relations to avoid update anomalies and join operations. The schema should be checked for extraneous foreign keys or simplistic schemas that necessitate numerous joins to retrieve the data. In our Hypertext schema, we noticed that the **Block** relation was never referenced; the required block information was retrieved from the **Link** relation. In our case, the **Block** relation was unnecessary, but other application programs might use it.

One potential bottleneck in a database system is its I/O management. GemStone attempts to alleviate part of the bottleneck by only accessing the disk when object values need to be retrieved. In Ingres, the relations containing the desired data must be retrieved from the disk prior to viewing or manipulating the data. I/O activity can be decreased in Ingres with indexes; only certain pages are retrieved versus the entire relation. We feel GemStone has a better paradigm for I/O management, but it should be stressed that 'commercial' Ingres might generate better statistics than those seen in our tests, due to its performance enhancements.

After becoming familiar with each database system, we preferred the GemStone model over the Ingres model for implementing the schema. In GemStone, we could model the real-world structure and behavior of an object. In Ingres, we had to flatten the data objects to conform to the table configuration inherent in the relational model. Since this flat representation did not reflect the real-world structure of the data entity, the schema was not as easy to remember or understand as GemStone's schema.

GemStone's model facilitated representing the relationships between objects. It was not as straightforward in Ingres to model those relationships. Foreign keys had to be introduced to the relational schema to denote relationships, frequently producing a complicated schema that was difficult to decode, or the data values did not reflect the relationship between objects (e.g., in the Document application where each component was assigned a globally unique tuple id that did not always reflect that component's relative position in the document).

GemStone was easier to use because we were able to accomplish everything (model definition, method implementation, and statistic generation) in one programming language. To implement the applications in Ingres, we had to utilize three separate languages, QUEL, EQUEL, and C and be cognizant of impedence mismatch.

In GemStone, all the class definitions, application methods, and data connected with a specific database are contained in that database and are available to the user when the database is opened. An Ingres database includes the relations containing the data and the system relations describing the database, but does not include the application programs. As a result, when an Ingres database is opened, the data is available to the user, but not the specific programs written to retrieve the data; these programs reside outside of the database and the user must know where they are located.

Although our original intention to compare the performance of Ingres and GemStone was abandoned, we feel we have successfully proposed two benchmarks that could be used to compare relational and object-oriented database systems. Using the relational and object-oriented schemas and the operations provided in each benchmark,

various relational and object-oriented database systems could be analyzed and compared. Porting the relational schema between relational database management systems should be fairly easy. But, implementing the object-oriented schemas on various object-oriented systems will not be as straightforward because there is not a single object-oriented model that all systems adhere to. Yet, each benchmark presents the conceptual model of the data entity, which can be used as a guideline in the design and implementation of the object-oriented model.

Although we feel the proposed benchmarks could be used in their present form, we believe they could be improved in the following manner. First, more than one scheme should be implemented in each database system. In this way, one can determine which scheme is most suitable for the application and the database system. But, the term 'suitability' must be defined before determining which system is best, e.g., is one looking for a fast response time or an easily understood schema?

Second, the Ingres implementation of the Document benchmark should include indexes. In not utilizing indexes, Ingres was placed at a disadvantage because it was not allowed to run at its greatest potential. Third, Test 2 and Test 4 implementations in the Document benchmark should be revised to allow the parameters to reflect the component's position in the document, matching the implementation of those tests in GemStone. Fourth, the GemStone schema designed for the Document benchmark should either be modified to store the word and text values in collections, or another schema devised, including these modifications, and both compared. Finally, the amount of code required to implement the benchmark should be included in the statistics.

BIBLIOGRAPHY

[Ano85]

Anon, et.al., A Measure of Transaction Processing Power, **Datamation** 31:7, April, 1985.

[BaD82] *

Baroody, A. J. and DeWitt, D. J., An Object-Oriented Approach to Database System Implementation, **ACM Trans. Database Systems 6**, 4 (Dec. 1982), 576-601.

[BDT83]

Bitton, D., Dewitt, D. J. and Turbyfill, C., Benchmarking Database Systems - A Systematic Approach, Technical Report #526, Computer Sciences Dept., Univ. of Wisconsin, Madison, Wis., Dec. 1983.

[BoD84]

Boral, H. and DeWitt, D. J., A Methodology for Database System Performance Evaluation, **Proceedings of ACM/SIGMOD Annual Meeting**, 1984, 176-185.

[But86]

Butler, M., An Approach to Persistent LISP Objects, **31st IEEE Computer Society International Conference Proceedings,** 1986, 324-329.

---

* Not referenced in thesis.

[CFH83]

Chu, K., Fishburn, J. P., Honeyman, P. and Lien, Y. E., Vdd - A VLSI Design Database System, **IEEE**, Jan. 1983, 25-37.

[CoM84] *

Copeland, G. and Maier, D., Making Smalltalk a Database System, **Proceedings of ACM/SIGMOD Annual Meeting**, 1984, 316-325.

[DeH81]

DeWitt, D. and Hawthorn, P., A Performance Evaluation of Database Machine Architecture, **ACM Proceedings on Very Large Databases**, 1981, 199-213.

[FBC86]

Fishman, D. H., Beech, D., Cate, H. P., Chow, E. C., Connors, T., Davis, J. W., Derrett, N., Hoch, C. G., Kent, W., Lyngbaek, P., Mahbod, B., Neimat, M. A., Ryan, T. A. and Shan, M C., IRIS: An Object-Oriented DBMS, Technical Report #STL-86-15, Hewlett Packard Co., Palo Alto, Calif, Dec. 1, 1986.

[GSM86]

Garrett, L. N., Smith, K. E. and Meyrowitz, N., Intermedia: Issues, Strategies, and Tactics in the Design of a Hypermedia Document System, **Proceedings from Computer-Supported Cooperative Work Conference,** 1986, 1-17.

[GoR83] *

Goldberg, A. and Robson, D., **Smalltalk-80: The Language and its Implementation**, Addison Wesley, Menlo Park, Ca., 1983.

[HaF86]

Hagmann, R. and Ferrari, D., Performance Analysis of Several Back-End Database Architectures, **ACM Trans. Database Systems 11**, 1 (March, 1986), 1-26.

[HaS86] *

Hawthorn, P. and Stonebraker, M., The Use of Technological Advances to Enhance Database System Performance, in **The Ingres Papers: Anatomy of a Relational Database**, M. Stonebraker (ed.), Addison Wesley, Reading, MA, 1986, 106-130.

KCB87] *

Kim, W., Chou, H. and Banerjee, J., Operations and Implementation of Complex Objects, **Proceedings of the 3rd International Data Engineering Conference**, 1987, 626-633.

[Lin83]

Linton, M. A., Queries and Views of Programs Using a Relational Database System, Technical Report #UCB/Computer Science Dept. 83/164, Univ. of California - Berkeley, Berkeley, Calif., Dec. 1983.

[Lin84]

Linton, M. A., Implementing Relational Views of Programs, **ACM 1984 Software Eng. Notes/SIGPLAN Notices Proceedings**, 1984, 132-140.

[LoP83] *

Lorie, R. and Plouffe, W., Complex Objects and Their Use in Design Transactions, **ACM Proceedings on Databases for Engineering Applications**, 1983, 115-121.

[LKM85]

Lorie, R., Kim, W., McNabb, D., Plouffe, W. and Meier, A., Supporting Complex Objects in a Relational System for Engineering Databases, in **Query Processing in Database Systems**, W. Kim, D. Reiner and D. Batory (ed.), Springer-Verlag, 1985, 145-155.

[LDF86]

Lyngbaek, P., Derrett, N., Fishman, D. H., Kent, W. and Ryan, T. A., Design and Implementation of the IRIS Object Manager, Technical Report #STL-8617, Hewlett Packard Co., Palo Alto, Calif., Dec. 10, 1986.

[Mai83]

Maier, D., Capturing More Meaning In Databases, Technical Report #CS/E-83-009, Oregon Graduate Center, Beaverton, Ore., 1983.

[Mai86] *

Maier, D., Why Object-Oriented Databases Can Succeed Where Others Have Failed, **Proceedings of IEEE International Workshop on Object-Oriented Databases**, 1986, 227.

[MaS86] *

Maier, D and Stein, J., Indexing in an Object-Oriented DBMS, Technical Report #CS/E-86-006, Oregon Graduate Center, Beaverton, Ore., May, 1986.

[MSO86]

Maier, D., Stein, J., Otis, A. and Purdy, A., Development of an Object-Oriented DBMS, **ACM OOPSLA 1986 Conference Proceedings**, 1986, 472-482.

[MaS87] *

Maier, D. and Stein, J., Development and Implementation of an Object-Oriented DBMS, in **Research Directions in Object-Oriented Programming,** B. Shriver and P. Wegner (ed.), MIT Press, Cambridge, MA., 1987, 355-392.

[Mey86]

Meyrowitz, N., Intermedia: The Architecture and Construction of an Object-Oriented Hypermedia System and Applications Framework, **ACM OOPSLA Conference Proceedings**, 1986, 186-201.

[PMS87] *

Purdy, A., Maier, D. and Schuchardt, B., Integrating an Object-Server with Other Worlds, **ACM Transaction on Office Information Systems 5**, 1 (Jan. 1987).

[Rol82a]

Rollins, E. J., Abstract Syntax in Theory and Practice, Technical Report #CS/E-82-04, Oregon Graduate Center, Beaverton, Ore., 1982.

[Rol82b]

Rollins, E. J., A Syntax-Analyzer Constructor, Doctorial Dissertation, State University of New York, Stoney Brook, 1982.

[RKC87]

Rubenstein, W. D., Kubicar, M. S. and Cattell, R. G. G., Benchmarking Simple Database Operations, **Proceedings of the ACM/SIGMOD Annual Meeting**, 1987.

[SmZ87]

Smith, K. E. and Zdonik, S. B., Intermedia: A Case Study of the Differences Between Relational and Object-Oriented Database Systems, **ACM OOPSLA Conference Proceedings**, 1987.

[StB86]

Stefik, M. and Bobrow, D. G., Object-Oriented Programming: Themes and Variations, **The AI Magazine**, Jan. 1986, 40-62.

[SSL83]

Stonebraker, M., Stettner, H., Lynn, N., Kalash, J. and Guttman, A., Document Processing in a Relational Database System, **ACM Transactions on Office Information Systems 1**, 2 (April 1983), 143-158.

[Sto86] *

Stonebraker, M., Operating System Support for Database Management, in **The Ingres Papers: Anatomy of a Relational Database**, M. Stonebraker (ed.), Addison Wesley, Reading, MA, 1986, 172-181.

[SWR86] *

Stonebraker, M., Woodfill, J., Ranstrom, J., Murphy, M., Meyer, M. and Allman, E., Performance Enhancements to a Relational Database System, in **The Ingres Papers: Anatomy of a Relational Database**, M. Stonebraker (ed.),

Addison Wesley, Reading, MA., 1986, 131-153.

[SWK86]

Stonebraker, M., Wong, E., Kreps, P. and Held, G., The Design and Implementation of Ingres, in **The Ingres Papers: Anatomy of a Relational Database**, M. Stonebraker (ed.), Addison Wesley, Reading, MA., 1986, 5-45.

[WSR81] *

Woodfill, J., Siegal, P., Ranstrom, J., Meyer, M. and Allman, E., **Ingres Reference Manual, Version 8.7**, Univ. of California - Berkeley, Berkeley, Calif., April, 1981.

[Zdo84] *

Zdonik, S. B., Object Management System Concepts, **Proceedings of the ACM/SIGOA Conference on Office Information Systems**, 1984, 13-19.

[ZdW86] *

Zdonik, S. B. and Wegner, P., Language and Methodology for Object-Oriented Database Environments, **Proceedings of the 19th International Conference on System Sciences**, 1986, 378-387.

APPENDIX A

THE DOCUMENT BENCHMARK

A.1 Creating Ingres Doc Database

UNIX command to create **Doc** database:

```
creatdb doc
```

Ingres commands to define **Doc** relations:

```
create chapter (chapid = i2, sectid = i2)
create section (sectid = i2, paraid = i2)
create paragraph (paraid = i2, lineid = i2, tlineid = i2)
create line (lineid = i2, wordpos = i2, word = c80)
create text (tlineid = i2, text = c80)
```

A.2 Ingres Commands To Load Document Database

```
copy chapter (chapid = c0, sectid = c0) from
"/ogc/students/becky/THESIS/DOC/chap.rel"

copy section (sectid = c0, paraid = c0) from
"/ogc/students/becky/THESIS/DOC/sect.rel"

copy paragraph (paraid = c0, lineid = c0, tlineid = c0) from
"/ogc/students/becky/THESIS/DOC/para.rel"

copy line (lineid = c0, wordpos = c0, word = c0nl) from
"/ogc/students/becky/THESIS/DOC/line.rel"

copy text (tlineid = c0, text = c0nl) from
"/ogc/students/becky/THESIS/DOC/text.rel"
```

A.3 Ingres Program Listing For Test1

TEST1.TEMPLATE

```
/*    In this program, all occurrances of one word are replaced with
      another word.  This program was executed three times, each run
      replacing a different word.  Included in this listing are the
      different word sets and where they were hard-coded in the program.
      This listing also contains the system calls associated with the time
      and cpu programs.  They are labeled and inserted in the code where
      they appear in their respective programs.
*/


/*    the following declarations pertain to the timing programs */

#include <sys/time.h>

            struct timeval          tp;
            struct timezone         tzp;
            long                    endtimesec,
                                    endtimeusec,
                                    starttimesec,
                                    starttimeusec;
            double                  endusec,
                                    endtime,
                                    startusec,
                                    starttime,
                                    elapsedtime;

/*    the following declarations pertain to the CPU programs */

#include <sys/time.h>
#include <sys/resource.h>

            struct rusage       resusage;
            long                systimesec,
                                systimeusec,
                                usertimesec,
                                usertimeusec,
                                inputcount,
                                outputcount;

            double              sysusec,
                                systime,
                                userusec,
                                usertime;
            int                 who;
```

```
main()
{
        /*      CPU */

        who = RUSAGE_CHILDREN;   /* = -1 */

        /*      invoke ingres on 'doc' database */

##      ingres doc

##      range of nl is newline
##      range of nt is newtext

        /*      TIME - start the timer */
        gettimeofday(&tp, &tzp);
        starttimesec = tp.tv_sec;
        starttimeusec = tp.tv_usec;

        /*      replace searchwords with targetwords in both relations */

##      replace nl (word =      "A")
                                "THE"
                                "PROCESS"

##      where nl.word =   "a"
                                "the"
                                "process"

##{
        /*      no processing */
##}

##      replace nt(text = "##0 A ##0")
                                        THE
                                        PROCESS

##      where nt.text =   "##0 a ##0"
                                        the
                                        process

##{
        /*      no processing */
##}

        /*      TIME - stop the timer */

        gettimeofday(&tp, &tzp);
        endtimesec = tp.tv_sec;
        endtimeusec = tp.tv_usec;

##      exit

        /*      CPU - collect statistics */

        getrusage(sho, &resusage);
```

```
        /*    TIME - calculate the elapsed time */

        startusec = (starttimeusec * .000001);
        starttime = (starttimesec + startusec);
        printf("start time = %lf sec", starttime);

        endusec = (endtimeusec * .000001);
        endtime = (endtimesec + endusec)
        printf("end time = %lf sec", endtime);

        elapsedtime = endtime - starttime;
        printf("ELAPSED TIME = %lf sec", elapsedtime);

        /*    CPU - pull statistics from resusage structure */

        usertimesec = resusage.ru_utime.tv_sec;
        usertimeusec = resusage.ru_utime.tv_usec;
        inputcount = resusage.ru_inblock;
        outputcount = resusage.ru_oublock;
        systimesec = resusage.ru_stime.tv_sec;
        systimeusec = resusage.ru_stime.tv_usec;

        /*    CPU - calculate and print cpu statistics */

        usersec = (usertimeusec * .000001);
        usertime = (usertimesec + userusec);
        printf("USER TIME = %lf sec", usertime);

        sysusec = (systimeusec * .000001);
        systime = (systimesec + sysusec);
        printf("SYSTEM TIME = %lf sec", systime);

        printf("INPUT COUNT = %ld", inputcount);
        printf("OUTPUT COUNT = %ld", outputcount);

        exit();

}
```

A.4 UNIX Driver for Test 1

RUNTEST 1

```
#############################################################################
#  In this shell script, time and cpu tests are run multiple times and
# the data generated collected in a file "runtest1.results".
#############################################################################


if ($#argv == 0) then
      echo "ERROR -> Specify # of runs to execute."
      exit()
else
      date >! runtest1.results

      echo " " >> runtest1.results
      echo " " >> runtest1.results

      echo "********************" >> runtest1.results
      echo "* RUNTEST1.RESULTS *" >> runtest1.results
      echo "********************" >> runtest1.results
      echo " " >> runtest1.results
      echo " " >> runtest1.results

      echo "****************** TIME *****************" >> runtest1.results
      echo " " >> runtest1.results

      @ count = 1
      while ($count <= $argv[1])
            if ($count == 1) then
                  date >> runtest1.results
                  echo "COPYING NEWTEXT/NEWLINE RELATIONS"
                                                >> runtest1.results
                  copyrel1
                  date >> runtest1.results
                  echo " " >> runtest1.results
                  echo " " >> runtest1.results
            else
                  copyrel1
            endif

            uptime >> runtest1.results
            echo "=> Time1-a" >> runtest1.results
            time1 >> runtest1.results

            uptime >> runtest1.results
            echo "=> Time11-the" >> runtest1.results
            time11 >> runtest1.results
```

```
            uptime >> runtest1.results
            echo "=> Time111-process" >> runtest1.results
            time111 >> runtest1.results

            truncre111
            echo "*****************************************"
                                                    >> runtest1.results
            @ count++
        end

        echo "****************** CPU ******************" >> runtest1.results
        echo " " >> runtest1.results

        @ count = 1
        while ($count <= $argv[1])
              copyre111

              uptime >> runtest1.results
              echo "=> CPU1-a" >> runtest1.results
              cpu1 >> runtest1.results

              uptime >> runtest1.results
              echo "=> CPU11-the" >> runtest1.results
              cpu11 >> runtest1.results

              uptime >> runtest1.results
              echo "=> CPU111-process" >> runtest1.results
              cpu111 >> runtest1.results

              truncre111
              echo "*****************************************"
                                                    >> runtest1.results
              @ count++
        end

        echo " " >> runtest1.results

        date >> runtest1.results
        exit()
endif
```

A.5 Gemstone Document Class Definitions

```
            "Chapters class"

SequenceableCollection subclass: 'Chapters'
            instVarNames: #()
            classVars: #()
            poolDictionaries: #()
            inDictionary: UserGlobals
            constraints: #()
            isInvariant: false.


            "Sections class"

SequenceableCollection subclass: 'Sections'
            instVarNames: #()
            classVars: #()
            poolDictionaries: #()
            inDictionary: UserGlobals
            constraints: #()
            isInvariant: false.


            "Paragraph class"

SequenceableCollection subclass: 'Paragraphs'
            instVarNames: #()
            classVars: #()
            poolDictionaries: #()
            inDictionary: UserGlobals
            constraints: #()
            isInvariant: false.


            "ParagraphContents class"

Object subclass: 'ParagraphContents'
            instVarNames: #('lines' 'textLines')
            classVars: #()
            poolDictionaries: #()
            inDictionary: UserGlobals
            constraints: #()
            isInvariant: false.
```

```
            "Lines class"

SequenceableCollection subclass: 'Lines'
            instVarNames: #()
            classVars: #()
            poolDictionaries: #()
            inDictionary: UserGlobals
            constraints: #()
            isInvariant: false.


            "Words class"

SequenceableCollection subclass: 'Words'
            instVarNames: #()
            classVars: #()
            poolDictionaries: #()
            inDictionary: UserGlobals
            constraints: #()
            isInvariant: false.


            "TextLines class"

SequenceableCollection subclass: 'TextLines'
            instVarNames: #()
            classVars: #()
            poolDictionaries: #()
            inDictionary: UserGlobals
            constraints: #()
            isInvariant: false.


            "Document class"

Object subclass: 'Document'
            instVarNames: #('name' 'contents')
            classVars: #('Test' 'Thesis')
            poolDictionaries: #()
            inDictionary: UserGlobals
            constraints: #()
            isInvariant: false.
```

A.6 Gemstone Bulk Loader Class Definitions

```
            "WordElement"

Object subclass: 'WordElement'
            instVarNames: #('chapid' 'word' 'sectid' 'paraid' 'lineid'
                                    'wordpos')
            classVars: #()
            poolDictionaries: #()
            inDictionary: UserGlobals
            constraints: #()
            isInvariant: false.


            "TextElement"

Object subclass: 'TextElement'
            instVarNames: #('chapid' 'sectid' 'paraid' 'tlineid' 'text')
            classVars: #()
            poolDictionaries: #()
            inDictionary: UserGlobals
            constraints: #()
            isInvariant: false.


            "FileArray class"

SequenceableCollection subclass: 'FileArray'
            instVarNames: #()
            classVars: #('TextFile' 'WordFile')
            poolDictionaries: #()
            inDictionary: UserGlobals
            constraints: #()
            isInvariant: false.
```

A.7 Gemstone Methods for Constructing the Document Object

```
classmethod: Document "retrieveThesis"

retrieveThesis

            "returns the Object found in Thesis classVariable."

            ^Thesis


classmethod: Document "setThesis:"

setThesis: anObject

            "instantiates 'Thesis' classVariable with anObject."

            Thesis := anObject


category: 'createDoc'

classmethod: Document "instantiateThesis"

instantiateThesis

"A document is created and instantiated, using the Wordfile and Textfile
Objects stored in the FileArray class.  Upon completion, the document is
stored in the 'Thesis' class variable in the Document class."

"Document instantiateThesis"

| wordfile chap index cindex length element cid sect sindex wflength sid
para pindex pid paraCont line lindex lid word textfile tflength doc tcid
sects tsid paras tpid tparacont tline textstr |

wordfile := FileArray retrieveWordFile.
wflength := (wordfile size).
chap := Chapters new.
index := 0.
cindex := 0.
[(index := index + 1) <= wflength]
      whileTrue:
      [
            element := wordfile at: index.
            cid := element chapid.
            cindex ~= cid
                  ifTrue:
                  [
                        cindex := cid.
                        sect := Sections new.
                        chap add: sect.
                        sindex := 0.
                  ].
```

```
                sid := element sectid.
                sindex ~= sid
                        ifTrue:
                        [
                                sindex := sid.
                                para := Paragraphs new.
                                sect add: para.
                                pindex := 0.
                        ].
                pid := element paraid.
                pindex ~= pid
                        ifTrue:
                        [
                                pindex := pid.
                                paraCont := ParagraphContents new.
                                para add: paraCont.
                                line := Lines new.
                                paraCont addLines:line.
                                lindex := 0.
                        ].
                lid := element lineid.
                lindex ~= lid
                        ifTrue:
                        [
                                lindex := lid.
                                word := Words new.
                                line add: word.
                        ].
                word add: element word.
        ].

"retrieve the TextFile and complete the Document contents."

textfile := FileArray retrieveTextfile.
tflength := (textfile size).
index := 0.
[(index := index + 1) <= tflength]
        whileTrue:
        [
                element := textfile at: index.
                tcid := element chapid.
                sects := chap at: tcid.
                tsid := element sectid.
                paras := sects at: tsid.
                tpid := element paraid.
                tparacont := paras at: tpid.
                tline := tparacont textLines.
                tline isNil
                        ifTrue:
                        [
                                tline := TextLines new.
                                tparacont addText: tline.
                        ].
                textstr := element text.
                tline add: textstr.
        ].
```

```
"create doc and instantiate variables."

doc := Document new.
doc addName: 'thesis'.
doc addContents: chap.

"send the document to Document class for storage and future retrieval."
Document setThesis: doc.
```

A.8 Gemstone Program Listings for Test 1

```
method: Document        "replace: with:"

replace: oldString with: newString

"replaces all oldString occurrances in a document with the newString.
This occurs in both the Word and Text objects.  The following line
demonstrates how to use this message.

"(Document retrieveThesis) replace: 'a' with: 'A'."

| doc chap sect para paracont clength slength plength llength wlength line
word aword cindex sindex pindex lindex tindex tlength oldStringsp windex
xindex yindex tline symbolarray symbol strIndex tarray textstr endPos
spOldStringsp |

oldStringsp := oldString + ' '.

chap := self contents.
clength := (chap size).
1 to: clength do: [ :cindex |
 sect := chap at: cindex.
 slength := (sect size).

 1 to: slength do: [ :sindex |
  para := sect at: sindex.
  plength := (para size).

  1 t: plength do: [ :pindex |
   paracont := para at: pindex.
   line := paracont lines.
   llength := (line size).

   1 to: llength do: [ :lindex |
    word := line at: lindex.
    wlength := (word size).

    1 to: wlength do: [ :windex |
     aword := word at: windex.
     aword = oldString
      ifTrue:
      [
       word at: windex put: newString.
      ].

    ].       "windex"

   ].        "lindex"

   tline := paracont textLines.
   tlength := (tline size).
```

```
    1 to: tlength do: [ :tindex |
     textstr := (tline at: tindex).
     "is oldString at beginning of textstr?"
     (textstr at: 1 equals: oldStringsp)
      ifTrue:
      [
       textstr deleteFrom: 1 to: (oldString size).
       textstr insert: newString at: 1.
      ]
      ifFalse:
      [
       "check if (textstr size) >= (oldString size) + 1"
       ((textstr size) >= ((oldString size) + 1))
        ifTrue:
        [
         "is oldString at the end of textstr?"
         endPos := ((textstr size) - ((oldString size) - 1)).
         (textstr at: endPos equals: oldString)
          ifTrue:
          [
           textstr deleteFrom: endPos
                   to: ((endPos) + ((oldString size) - 1)).
           textstr insert: newString at: endPos.
          ]
          ifFalse:
          [
           "is oldString within textstr?"
            spOldStringsp := ' ' + oldStringsp.
            strIndex := (textstr findString: spOldStringsp
                                 startingAt: ((oldString size) + 1)).
            strIndex ~= 0
             ifTrue:
             [
              "oldString found within textstr."
              textstr deleteFrom: (strIndex + 1)
                      to: (strIndex + (oldString size)).
              textstr insert: newString at: (strIndex + 1).
             ]
          ]
        ]
      ]

    ]. "tindex"
   ].          "pindex"
  ].           "sindex"
 ].            "cindex"
```

A.9 Topaz Driver for Test 1

TEST1.OPL

```
!-------------------------------------------------------------------------

!  This program runs three sets of word replacements.  The specific word
! which is replaced is noted in the 'remark' invocation within each
! test.
! (Note: The object 'doc' is in UserGlobals and therefore, can be
! referenced directly.)

!-------------------------------------------------------------------------

time
remark Test1.1 (a)
run
      doc replace: 'a' with: 'A'.
%
time

!-------------------------------------------------------------------------

time
remark Test1.2 (the)
run
      doc replace: 'the' with: 'THE'.
%
time

!-------------------------------------------------------------------------

time
remark Test1.3 (process)
run
      doc replace: 'process' with: 'PROCESS'.
%
time

!-------------------------------------------------------------------------

run
      System abortTransaction.
%

!-------------------------------------------------------------------------
```

A.10 VMS Driver for Test1

TEST1.COM

```
$ set noverify
$ sd user2:[lakeyb.gemstone.test1]
$ write sys$output ""
$ sho sys
$ define gemstone user2:[lakeyb.docsys]
$ startstone docstone gemstone

$ write sys$output ""
$ write sys$output "******************** TEST 1 ************************"
$ write sys$output ""
$ count = 1
$ LOOP:
$ write sys$output "Loop # ''count'"
$ write sys$output ""
$ gem
connect stone docstone
login 'becky lakey' gemstone
level 0
output push test1.out
input test1.opl
output pop
logout
exit
$ awk -f gemtime.awk test1.out > test1.time
$ count = count + 1
$ if count .le. 6 then goto LOOP

$ write sys$output ""
$ write sys$output "*****************************************************"
$ write sys$output ""

$ stopstone docstone datacurator swordfish
$ sho sys
$ write sys$output ""
$ write sys$output "*****************************************************"
$ write sys$output ""
$ exit
```

APPENDIX B

THE HYPERTEXT BENCHMARK

B.1 Creating Ingres Inter Database

UNIX command to create **Inter** database:

```
creatdb inter
```

Ingres commands to define **Inter** relations:

```
create docs (docid = i2, docname = c35, docpath = c128)
create block (blockid = i2, docid = i2, blkowner = c10)
create appblock (blockid = i2, appextent = i2)
create link (linkid = i2, linktype = i2, startdocid = i2,
            startblkid = i2, enddocid = i2, endblkid = i2,
            lkowner = c10)
```

B.2 Ingres Commands To Load Inter Database

```
copy appblock (blockid = c0, appextent = c0) from
"/ogc/students/becky/THESIS/INTER/appblock.rel"

copy block (docid = c0, blockid = c0, blkowner = c0nl) from
"/ogc/students/becky/THESIS/INTER/block.rel"

copy docs (docid = c0, docname = c0, docpath = c0nl) from
"/ogc/students/becky/THESIS/INTER/docs.rel"

copy link (linkid = c0, linktype = c0, startdocid = c0,
                  startblkid = c0, enddocid = c0, endblkid = c0,
                  lkowner = c0nl) from
"/ogc/students/becky/THESIS/INTER/link.rel"
```

B.3 Ingres Program Listing For Test1

TEST1.TEMPLATE

```
/*    Given two document names, this test determines if a path exists
      between those two documents via links.  The following results are
      printed, depending on the outcome of the test:

                        path exists:                'MATCH'
                        path does not exist:        'EMPTY SET'
                        cycle encountered:          'CYCLE'

      The protocol for determining the answer is as follows:

      1.    Retrieve the docid values corresponding to the given document
            names.
                        - startid
                        - endid

      2.    Append startid to 'Visited' and 'Start' relations.

      3.    Begin infinite 'while' loop to search for the end document.

            1.    Search Link relation for all endDocId's whose
                  corresponding startDocId's = s.startDocId
                        - collect the endDocId's in a temporary relation,
                        End.

            2.    Check 'end' relation for Empty Set, Match, or Cycle.

                        - EMPTY SET
                              End relation is empty.  No path exists.

                        - MATCH
                              does one of the tuples in End match endid?
                              Yes - path exists.

                        - CYCLE
                              delete all those tuples in End which are
                              also contained in Visited.  If End then
                              contains 0 tuples, a cycle was encountered.
```

```
            3.    none of above conditions were matched - continue search.

                        - Visited = (Visited U End)
                        - remove tuples in Start
                        - append to Start the tuples remaining in End
                        - destroy End relation

      [end of while loop]

      Three sets of document names are tested, one for each condition in
      the program.  This listing also contains the system calls associated
      with the time and cpu programs.  They are labeled and inserted in
      the code where they appear in their respective programs.

      This test was run three times; the first time, the results were not
      printed, the second run printed the results, and the third run was
      invoked on an indexed database.
*/


#include <stdio.h>
#include "misc.h"

##    int                           xid;
##    int                           did;
##    int                           endid;
##    int                           startid;
##    int                           preendct;
##    int                           postendct;

##    char                          dname[36];
##    char                          *startdocname;
##    char                          *enddocname;

/*    the following declarations pertain to the timing programs */

#include <sys/time.h>

            struct timeval          tp;
            struct timezone         tzp;
            long                    endtimesec,
                                    endtimeusec,
                                    starttimesec,
                                    starttimeusec;
            double                  endusec,
                                    endtime,
                                    startusec,
                                    starttime,
                                    elapsedtime;
```

```
/*    the following declarations pertain to the CPU programs */

#include <sys/time.h>
#include <sys/resource.h>

            struct rusage           resusage;
            long                    systimesec,
                                    systimeusec,
                                    usertimesec,
                                    usertimeusec,
                                    inputcount,
                                    outputcount;

            double                  sysusec,
                                    systime,
                                    userusec,
                                    usertime;
            int                     who;

main (argc, argv)
      int argc;
      char *arv[];
{
      if (argv < 3)
      {
            printf("not enough document names");
            exit();
      }

      /*    initialize variables */

      startdocname = argv[1];
      enddocname = argv[2];

      /*    CPU */

      who = RUSAGE_CHILDREN;          /* = -1 */

      /*    invoke ingres on 'inter' database */

##    ingres inter

      /*    establish range variables */

##    range of d is docs
##    range of l is link

      /*    TIME - start the timer */

      gettimeofday(&tp, &tzp);
      starttimesec = tp.tv_sec;
      starttimeusec = tp.tv_usec;
      /*    create 'start' and 'visited' relations */

##    create start(startdocid = i2)
##    create visited(enddocid = i2)
```

```
      /*    determine startid and endid values */

##    retrieve (did = d.docid, dname = d.docname)
##          where (d.docname = startdocname) or (d.docname = enddocname)
##    {
            unpad(dname);

            if ((strcmp(dname, startdocname)) == 0)
            {
                  startid = did;
            }
            else
            {
                  endid = did;
            }
##    }

      /*    append startid to 'visited' and 'start' relations */

##    append to start (startdocid = startid)
##    append to visited(enddocid = startid)

##    range of s is start

      /*    begin infinite while loop to search for the end document */

      while (TRUE)
      {

            /*    search link relation for all enddocid's whose
                  corresponding startdocid's = s.startdocid */

##          retrieve into end (l.enddocid)
##                where (s.startdocid = l.startdocid)

##          range of e is end

            /*    check 'end' relation for Empty Set, Match, or Cycle */


            /*    EMPTY SET */

##          retrieve (preendct = countu(e.enddocid))

            if (preendct == 0)
            {
                  /*    Run 1 - No processing */

                  break;

                  /*    Run 2 - Results printed */

                  printf("EMPTY SET");
                  break;
            }

            /*    MATCH */
```

```
            xid = -1;

##          retrieve (xid = e.enddocid)
##                where (e.enddocid = endid)

            if (xid != -1)
            {
                  /*    Run 1 - No processing */

                  break;

                  /*    Run 2 - Results printed */

                  printf("MATCH");
                  break;
            }

            /*    CYCLE */

##          range of v is visited
##          delete e where (e.enddocid = v.enddocid)
##          retrieve (postendct = countu(e.enddocid))

            if (postendct == 0)
            {
                  /*    Run 1 - No processing */

                  break;

                  /*    Run 2 - Results printed */

                  printf("CYCLE");
                  break;
            }

            /*    none of above conditions were matched - continue search
*/

            /*    visited = (visited U end) */
##          append to visited (e.all)

            /*    remove tuples in start */

##          modify start to truncated

            /*    append to start the tuples remaining in end */

##          append to start (startdocid = e.enddocid)

            /*    destroy end relation */

##          destroy end

      }     /*    end of while loop */

      /*    TIME - stop the timer */
```

```
        gettimeofday(&tp, &tzp);
        endtimesec = tp.tv_sec;
        endtimeusec = tp.tv_usec;

        /*    clean up */

##      destroy end
##      destroy visited
##      destroy start

##      exit

        /*    CPU - collect statistics */

        getrusage(who, &resusage);

        /*    TIME - calculate the elapsed time */

        startusec = (starttimeusec * .000001);
        starttime = (starttimesec + startusec);
        printf("start time = %lf sec", starttime);

        endusec = (endtimeusec * .000001);
        endtime = (endtimesec + endusec)
        printf("end time = %lf sec", endtime);

        elapsedtime = endtime - starttime;
        printf("ELAPSED TIME = %lf sec", elapsedtime);

        /*    CPU - pull statistics from resusage structure */

        usertimesec = resusage.ru_utime.tv_sec;
        usertimeusec = resusage.ru_utime.tv_usec;
        inputcount = resusage.ru_inblock;
        outputcount = resusage.ru_oublock;
        systimesec = resusage.ru_stime.tv_sec;
        systimeusec = resusage.ru_stime.tv_usec;

        /*    CPU - calculate and print cpu statistics */

        usersec = (usertimeusec * .000001);
        usertime = (usertimesec + userusec);
        printf("USER TIME = %lf sec", usertime);

        sysusec = (systimeusec * .000001);
        systime = (systimesec + sysusec);
        printf("SYSTEM TIME = %lf sec", systime);

        printf("INPUT COUNT = %ld", inputcount);
        printf("OUTPUT COUNT = %ld", outputcount);

}
```

B.4 UNIX Driver for Test 1

RUNTEST 1

```
##############################################################################
#  In this shell script, time and cpu tests are run multiple times.  This
# script is invoked three times - the first time, to generate the
# statistics where the resullts are not printed (runtest1.results), the
# second run gathers the statistics for the printed results
# (runtest11.results), while the third run indexes the necessary relations
# prior to collecting the statistics (indxruntest1.results); results are
# not printed out in the indexed run.
##############################################################################


if ($#argv == 0) then
      echo "ERROR -> Specify # of runs to execute."
      exit()
else
      date >! (indx)runtest1(1).results

      echo " " >> (indx)runtest1(1).results
      echo " " >> (indx)runtest1(1).results

      echo "********************" >> (indx)runtest1(1).results
      echo "* RUNTEST1.RESULTS *" >> (indx)runtest1(1).results
      echo "********************" >> (indx)runtest1(1).results
      echo " " >> (indx)runtest1(1).results
      echo " " >> (indx)runtest1(1).results

      date >> (indx)runtest1(1).results
      echo "COPYING RELATIONS" >> (indx)runtest1(1).results
      copyrel1
      date >> (indx)runtest1(1).results
      echo " " >> (indx)runtest1(1).results
      echo " " >> (indx)runtest1(1).results

      /* Run 3 - Index appropriate relations prior to running tests */

      date >> (indx)runtest1(1).results
      echo "INDEXING RELATIONS" >> (indx)runtest1(1).results
      indxrel1
      date >> (indx)runtest1(1).results
      echo " " >> (indx)runtest1(1).results
      echo " " >> (indx)runtest1(1).results

      echo "****************** TIME *****************"
                                          >> (indx)runtest1(1).results
      echo " " >> (indx)runtest1(1).results
```

```
        @ count = 1
        while ($count <= $argv[1])

                uptime >> (indx)runtest1(1).results
                echo "=> time1 Crusoe Defoe" >> (indx)runtest1(1).results
                time1 'Crusoe' 'Defoe' >> (indx)runtest1(1).results

                uptime >> (indx)runtest1(1).results
                echo "=> time1 Crusoe Footprint " >> (indx)runtest1(1).results
                time1 'Crusoe' 'Footprint' >> (indx)runtest1(1).results

                uptime >> (indx)runtest1(1).results
                echo "=> time1 Defoe_Sources Defoe"
                                        >> (indx)runtest1(1).results
                time1 'Defoe_Sources' 'Defoe' >> (indx)runtest1(1).results

                echo "****************************************"
                                        >> (indx)runtest1(1).results

                @ count++
        end

        echo "****************** CPU ******************"
                                        >> (indx)runtest1(1).results
        echo " " >> (indx)runtest1(1).results

        @ count = 1
        while ($count <= $argv[1])

                uptime >> (indx)runtest1(1).results
                echo "=> cpu1 Crusoe Defoe" >> (indx)runtest1(1).results
                cpu1 'Crusoe' 'Defoe' >> (indx)runtest1(1).results

                uptime >> (indx)runtest1(1).results
                echo "=> cpu1 Crusoe Footprint " >> (indx)runtest1(1).results
                cpu1 'Crusoe' 'Footprint' >> (indx)runtest1(1).results

                uptime >> (indx)runtest1(1).results
                echo "=> cpu1 Defoe_Sources Defoe"
                                        >> (indx)runtest1(1).results
                cpu1 'Defoe_Sources' 'Defoe' >> (indx)runtest1(1).results

                echo "****************************************"
                                        >> (indx)runtest1(1).results

                @ count++
        end

        date >> (indx)runtest1(1).results
        truncrel1
        exit()
endif
```

B.5 Gemstone Hypertext Class Definitions

```
            "AbstractDoc class"

Object subclass: 'AbstractDoc'
            instVarNames: #('id' 'name')
            classVars: #()
            poolDictionaries: #()
            inDictionary: UserGlobals
            constraints:      #[
                                    #[#id, Integer],
                                    #[#name, String],
                              ]
            isInvariant: false.


            "AbstractBlk class"

Object subclass: 'AbstractBlk'
            instVarNames: #('id' 'doc')
            classVars: #()
            poolDictionaries: #()
            inDictionary: UserGlobals
            constraints:      #[
                                    #[#id, Integer],
                                    #[#doc, AbstractDoc],
                              ]
            isInvariant: false.


            "AbstractLink class"

Object subclass: 'AbstractLink'
            instVarNames: #('id' 'startBlk' 'endBlk')
            classVars: #()
            poolDictionaries: #()
            inDictionary: UserGlobals
            constraints:      #[
                                    #[#id, Integer],
                                    #[#startBlk, AbstractBlk],
                                    #[#endBlk, AbstractBlk],
                              ]
            isInvariant: false.


            "OwnerCollection class"

Set subclass: 'OwnerCollection'
            instVarNames: #()
            classVars: #()
            poolDictionaries: #()
            inDictionary: UserGlobals
            constraints: String
            isInvariant: false.
```

```
            "AECollection class"

Set subclass: 'AECollection'
            instVarNames: #()
            classVars: #()
            poolDictionaries: #()
            inDictionary: UserGlobals
            constraints: Integer
            isInvariant: false.


            "BlkCollection class"

Set subclass: 'BlkCollection'
            instVarNames: #()
            classVars: #()
            poolDictionaries: #()
            inDictionary: UserGlobals
            constraints: AbstractBlk
            isInvariant: false.


            "DocCollection class"

Set subclass: 'DocCollection'
            instVarNames: #()
            classVars: #()
            poolDictionaries: #()
            inDictionary: UserGlobals
            constraints: AbstractDoc
            isInvariant: false.


            "LinkCollection class"

Set subclass: 'LinkCollection'
            instVarNames: #()
            classVars: #()
            poolDictionaries: #()
            inDictionary: UserGlobals
            constraints: AbstractLink
            isInvariant: false.


            "BlkObj class"

AbstractBlk subclass: 'BlkObj'
            instVarNames: #('owner' 'appExt')
            classVars: #()
            poolDictionaries: #()
            inDictionary: UserGlobals
            constraints:      #[
                                    #[#appExt, AECollection],
                              ]
            isInvariant: false.
```

```
            "LinkObj class"

AbstractLink subclass: 'LinkObj'
            instVarNames: #('owner' 'type')
            classVars: #()
            poolDictionaries: #()
            inDictionary: UserGlobals
            constraints: #()
            isInvariant: false.


            "DocObj class"

AbstractDoc subclass: 'DocObj'
            instVarNames: #('path' 'startLinks')
            classVars: #()
            poolDictionaries: #()
            inDictionary: UserGlobals
            constraints:      #[
                                    #[#startLinks, LinkCollection],
                              ]
            isInvariant: false.


            "WebObj class"

Object subclass: 'WebObj'
            instVarNames: #('owners' 'links' 'docs' ' blocks')
            classVars: #()
            poolDictionaries: #()
            inDictionary: UserGlobals
            constraints:      #[
                                    #[#owners, OwnerCollection],
                                    #[#links, LinkCollection],
                                    #[#docs, DocCollection],
                                    #[#blocks, BlkCollection],
                              ]
            isInvariant: false.


            "WebCollection class"

Array subclass: 'WebCollection'
            instVarNames: #()
            classVars: #('LinkTypes')
            poolDictionaries: #()
            inDictionary: UserGlobals
            constraints: #()
            isInvariant: false.
```

B.6 Gemstone Bulk Loader Class Definitions

```
            "DocElement class"

Object subclass: 'DocElement'
            instVarNames: #('id' 'name' 'path')
            classVars: #()
            poolDictionaries: #()
            inDictionary: UserGlobals
            constraints: #()
            isInvariant: false.


            "LinkElement class"

Object subclass: 'LinkElement'
            instVarNames: #('id' 'owner' 'type' 'startBlkId' 'endBlkId')
            classVars: #()
            poolDictionaries: #()
            inDictionary: UserGlobals
            constraints: #()
            isInvariant: false.


            "BlockElement class"

Object subclass: 'BlockElement'
            instVarNames: #('blockId' 'blockOwner' 'docId')
            classVars: #()
            poolDictionaries: #()
            inDictionary: UserGlobals
            constraints: #()
            isInvariant: false.


            "AppElement class"

Object subclass: 'AppElement'
            instVarNames: #('blockId' 'appExt')
            classVars: #()
            poolDictionaries: #()
            inDictionary: UserGlobals
            constraints: #()
            isInvariant: false.


            "DocLinkElement class"

Object subclass: 'DocLinkElement'
            instVarNames: #('docId' 'linkId')
            classVars: #()
            poolDictionaries: #()
            inDictionary: UserGlobals
            constraints: #()
            isInvariant: false.
```

```
            "OwnerElement class"

Object subclass: 'OwnerElement'
            instVarNames: #('owner')
            classVars: #()
            poolDictionaries: #()
            inDictionary: UserGlobals
            constraints: #()
            isInvariant: false.


            "TypeElement class"

Object subclass: 'TypeElement'
            instVarNames: #('type')
            classVars: #()
            poolDictionaries: #()
            inDictionary: UserGlobals
            constraints: #()
            isInvariant: false.
```

B.7 Example Gemstone Methods for Constructing Web7 Object

```
category:   'build components'

method:          WebObj                  "buildOwnerCollection"

buildOwnerCollection

"builds the Owner Collection for a specific Web Object"

|ownerset olength oindex |

ownerset := OwnerCollection new.
olength := (OwnerFile size).

1 to: olength do: [:oindex|
      ownerset add: ((OwnerFile at: oindex) owner).
].    "oindex"

^ownerset

************************************************************

category:   'private'

classmethod:      Webcollection          "setLinkTypes:"

setLinkTypes: anObject

"instantiates 'LinkTypes' class variable with anObject"

LinkTypes := anObject.

************************************************************

category:   'initialization'

method:          WebCollection          "initializeWeb:"

initializeWeb: anInteger

"initializes an instance of a WebCollection at position anInteger with an
instance of a built WebObj."
```

```
|newWebObj |

WebCollection buildLinkTypes.
newWebObj := WebObj new.
newWebObj owners: (newWebObj buildOwnerCollection).
newWebObj docs: (newWebObj buildDocCollection).
newWebObj blocks: (newWebObj buildBlockCollection).
newWebObj links: (newWebObj buildLinkCollection:
                                    (WebCollection retrieveLinkTypes)).
newWebObj completeDocObjects.
self at: anInteger put: newWebObj.
UserGlobals at: #Intermedia put: self.
```

B.8 Gemstone Program Listings for Test 1

```
category:   'queries'

method WebObj                          "doesPathExistBetween:and:"

doesPathExistBetween: startDocName  and: endDocName

"determine if a path exists between startDocName and endDocName"

"(Intermedia at: 7) doesPathExistBetween: 'Crusoe' and: 'Footprint'"

|docColl startDocObj enddocObj startColl endColl visitedColl slcoll did
loop |

docColl := self docs.

"1.  Find the Doc obj's associated with startDocName & endDocName"

startDocObj := docColl detect: {:aDocObj | aDocObj.name = startDocName}
                              ifNone: [Nil].

(startDocObj isNil)
      ifTrue:
      [
            ^startDocName asString + 'does not exist - Goodbye'.
      ].

endDocObj := docColl detect: {:aDocObj | aDocObj.name = endDocName}
                              ifNone: [Nil].

(endDocObj isNil)
      ifTrue:
      [
            ^endDocName asString + 'does not exist - Goodbye'.
      ].

"2.  Create start & visited collections - add startDocObj to both"

startColl := (Set new) add: StartDocObj.
visitedColl := (Set new) add: (StartDocObj id).

"3.   Infinite loop searching for a path between startDocObj and
      endDocObj. -
      loop discontinued when Empty Set, Match, or Cycle found."

loop := 1.
[loop = 1]

      whileTrue:
      [
            "for each DocObj in startColl, collect the endDocObj's
            associated with each startLink object in endColl."
```

```
        endColl := (Set new).
        startColl do: [:aDocObj |
              slcoll := (aDocObj startLinks).
              (slcoll notNil)
                    ifTrue:
                    [
                          slcoll do: [:aLinkObj |
                                endColl add: ((aLinkObj endBlk) doc).
                          ]. "do"
                    ].
        ]. "do"

        "endColl Set contains all of the endDoc's associated with each
        DocObj found in startColl that had startLinks.  If endColl
        isNil,  none  of  the  starting  documents  had  startLinks
        (startblk's) associated with them - the path has ended at
        those start documents."

        (endcoll isEmpty)
              ifTrue:
              [
                    ^'EMPTY SET'.
              ].

        "endColl is not empty - check if any of its DocObj's match the
        endDocObj."

        (endDocObj in: endColl)
              ifTrue:
              [
                    ^'MATCH'.
              ].

        "none of endColl's DocObj's match endDocObj - check if any
        have already been visited."

        startColl := (Set new).
        endColl do: [:aDocObj |
              did := (aDocObj id).
              (did in: visitedColl)
                    ifFalse:
                    [
                          visitedColl add: did.
                          startColl add: aDocObj.
                    ].
        ].  "do"

        "if startColl isNil, it means that all the DocObj's in endColl
        have already been visited and checked - therefore, a CYCLE was
        found."

        (startColl isEmpty_
              ifTrue:
              [
                    ^'CYCLE'.
              ].
  ].  "whileTrue"
```

B.9 Topaz Driver for Test 1

TEST1.OPL

```
!-------------------------------------------------------------------------

!  Three sets of start and end documents are specified.  This program
! determines if a path exists abetween each set of start and end
! documents.
! The numerals in parentheses reflect what is included in Test1(1)1.opl.
! (Note: 'Intermedia' in UserGlobals references the WebColl object.)

!-------------------------------------------------------------------------

time
remark Test1(1).1 ('Crusoe' 'Defoe' - EMPTY SET)
run
      (^) (Intermedia at: 7) doesPathExistBetween: 'Crusoe'
                                 and: 'Defoe'.
%
time

!-------------------------------------------------------------------------

time
remark Test1(1).2 ('Crusoe' 'Footprint' - MATCH)
run
      (^) (Intermedia at: 7) doesPathExistBetween: 'Crusoe'
                                 and: 'Footprint'.
%
time

!-------------------------------------------------------------------------

time
remark Test1(1).3 ('Defoe_Sources' 'Defoe' - CYCLE)
run
      (^) (Intermedia at: 7) doesPathExistBetween: 'Defoe_Sources'
                                 and: 'Defoe'.
%
time

!-------------------------------------------------------------------------
```

B.10 VMS Driver for Test1

TEST1.COM

```
$ set noverify
$ sd user2:[lakeyb.inter.test1]
$ write sys$output ""
$ sho sys
$ startstone docstone gemstone

$ write sys$output ""
$ write sys$output "*************** (INDX)TEST 1(1) **********************"
$ write sys$output ""
$ count = 1
$ LOOP:
$ write sys$output "Loop # ''count'"
$ write sys$output ""
$ gem
connect stone docstone
login 'becky lakey' gemstone
omit bytevalues /* Test1.opl */
oops off  /* Test1.opl */
level 0
output push (indx)test(1).out
input (indx)test(1).opl
output pop
logout
exit
$ awk -f gemtime.awk (indx)test(1).out > (indx)test(1).time
$ count = count + 1
$ if count .le. 6 then goto LOOP

$ write sys$output ""
$ write sys$output "*****************************************************"
$ write sys$output ""

$ stopstone docstone datacurator swordfish
$ sho sys
$ write sys$output ""
$ write sys$output "*****************************************************"
$ write sys$output ""
$ exit
```

BIOGRAPHICAL NOTE

The author was born January 19, 1953 in Corvallis, Oregon. Her father was in the Air Force for 20 years and as a result, the family moved around quite a bit. In 1971, the author graduated from Charles M. Russell High School in Great Falls, Montana.

In 1971, the author entered Whitman College in Walla Walla, Washington, and graduated with a Bachelor of Arts degree in Biology in 1975. She worked in a small clinical laboratory in Richland, Washington for a year and then completed a year's internship in Medical Technology at Sacred Heart Medical Center, Spokane, Washington, graduating in 1977. From 1977 until 1985, the author worked in reference clinical laboratories and hospital laboratories in Seattle, Washington and Vancouver, Washington.

In 1985, the author began classes at Oregon Graduate Center where she completed the requirements for the degree Master of Computer Science in July, 1989. She is currently working as a Technical Marketing Engineer at Intel Corporation in Hillsboro, Oregon.