IMPLEMENTING INFORMATION RETRIEVAL USING A COMBINED OBJECT-ORIENTED DATABASE/ FILE SYSTEM PARADIGM

By

Bradley Scott Rubin

A dissertation submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN – MADISON

1996

Abstract

The SHORE (Scalable Heterogeneous Object REpository) persistent object store under development at the University of Wisconsin-Madison is a hybrid of Object-Oriented Database Management System (OODBMS) and file system technologies. A primary goal of the SHORE project is to provide a system that facilitates the construction of high functionality data servers for a wide variety of applications. In this thesis, I evaluate the suitability of the SHORE system for supporting the implementation of an information retrieval server and evaluate four different distributions of function between client and server. To evaluate the tradeoffs in word index design, two architectures explore an object index versus a server index service. To evaluate client versus server placement of information retrieval functions, two other architectures explore a generic SHORE server versus a custom information retrieval server. My experience with an implementation of a prototype multimedia information server, Chrysalis, demonstrates that overall the SHORE persistent object store functionality is well suited for supporting the construction of an information retrieval server compared with the

traditional alternatives of file systems, relational database systems, or even current OODBMS. Several performance studies evaluate major architectural and implementation tradeoffs. Areas for improvement for SHORE are highlighted. Chrysalis is the first major SHORE application, so this dissertation also serves as an experience guide for other potential SHORE applications.

Acknowledgements

This work was funded by IBM in Rochester, Minnesota under the IBM Resident Study Program. I would like to thank the management team that approved my plan to go back to school for my doctoral studies. That team was Alan Larson, Keith Fisher, Steven Ladwig, and David Schleicher. I would also like to thank the management team that provided continued support for my educational effort– Ralph Christ and Michael Tomashek. In the coming years, I hope to show that companies receive a substantial return on their investments in employee educational development and that this program is an effective way to transfer technology from academia to industry.

I would like to thank my thesis advisor, Jeffrey Naughton, for his guidance and for the initial idea of combining my interest in information retrieval with the SHORE project and suggesting that the performance implications of client/server functional partitioning would make for interesting exploration. The other preliminary examination committee members, Michael Carey and David DeWitt, also provided advice that influenced this work.

I would like to thank the SHORE development team (Nancy Hall, Mark

McAuliffe, Daniel Schuh, C.K. Tan, and Michael Zwilling) for supporting my development and for implementing several unplanned additions to SHORE.

Most of all, I would like to thank my wife Debra and my sons Justin and Nathan for temporarily relocating with me to Madison so I could pursue my educational goals. While the experience was enjoyable, leaving friends and family in Rochester for almost two years was a big sacrifice. My wife and I hope this experience sets an example for our children on the importance of education, just as my parents, Jill and Gerald Rubin, had instilled in me.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Information retrieval (IR) is an important computer system application. The recent Internet technology wave is placing heavy demands on information retrieval applications in terms of both functionality and response time performance. Traditionally, information retrieval applications have not been built upon database technology. Recently, a new paradigm of extensible database systems that combine object-oriented database and file system technology have been developed. A goal of these systems is to facilitate the construction of many types of applications that require access to persistent storage.

Is this new paradigm appropriate for the implementation of information retrieval applications? Does this new paradigm satisfy information retrieval functional requirements? There are several alternatives for mapping the functionality of an information retrieval application to this paradigm. What are the performance implications of these alternatives?

In this thesis, I explore how well the paradigm of a combined object-oriented database/file system, as exemplified by an experimental Object-Oriented Database Management System (OODBMS) under development at the University of Wisconsin-Madison called SHORE (Scalable Heterogeneous Object REpository) [CDF+94], fits the requirements of an information retrieval application. SHORE combines the features of an object-oriented database (persistent objects, abstract data types, inheritance, associative data access, transaction management, concurrency control, recovery, indexing, and object caching) with the features of a distributed file system (hierarchical name space, files, directories, links, location transparency, and access control). SHORE uses a peer-peer server architecture, and can be run on a single workstation, a network of workstations, or on a parallel processor. Parallel operations (both inter- and intra-transaction) will eventually be supported.

Specifically, this thesis addresses two primary aspects of this issue:

1. How can information retrieval applications best exploit the function provided by SHORE?

2. What are the tradeoffs in the distribution of function between client and server for an information retrieval application built using SHORE?

1.1 Functional Issues

Existing systems using OODBMS for information retrieval, such as Rufus from IBM Almaden [Mes91, Sho93] and INQUERY/Mneme from the University of Massachusetts [Bro94a, Bro94b] store documents in the file system and meta data in the OODB. These systems do not store documents in the OODB because they would have to be imported/exported to a UNIX file for use with legacy UNIX applications (or these UNIX applications would have to be rewritten). Documents stored in a file system, however, do not benefit from database facilities like transaction and persistent object support.

The approach explored in this dissertation is to use objects to store both the documents and their meta data, as well as full-text index data structures. Legacy UNIX tools can access SHORE objects using a Network File System (NFS) [LS90] protocol. The objects look like files in a UNIX name space via NFS, with the document contents placed in a SHORE object `text` attribute field appearing like regular UNIX files to UNIX programs. A full text index of document contents points to document objects. Document object attribute fields can be used to provide additional query capability.

I implemented an information retrieval application, called Chrysalis [RN95], that runs on SHORE. It performs full text indexing of documents and supports multiple media formats (mail, image, audio, etc.). Chrysalis is not meant to improve the state-of-the-art in information retrieval, but to provide a reasonably good, realistic platform for exploring functional, performance, and distribution

of function issues. Chapter 2 contains an elaboration on the techniques used in information retrieval and the rationale for choices made for Chrysalis, as well as a discussion of related work in combining object-oriented databases and file systems for information retrieval.

Chrysalis performs stop list checking (eliminating common words) and term conflation (reducing like words to a common stem). It supports new/changed document detection, incremental indexing, and lazy document deletion with index garbage collection. It uses the vector-space/natural-language query model [FBY92, Sal89]. In response to these queries, Chrysalis returns a list of relevancy-ranked documents. It then presents a UNIX command prompt, allowing the user to use any UNIX command on the selected document, which is known by the filename `text`. If the document has a binary component, that data is accessed by the filename `binary`. Figure 1 shows a sample session to give a flavor of Chrysalis operation. A World-Wide Web (WWW) [LPJ+94] version of Chrysalis has also been developed, allowing a WWW browser to be used as the user interface to Chrysalis. Chrysalis is the first major application for SHORE, and resulted in several enhancements, both functional and performance, to SHORE. Chapter 3 describes Chrysalis and SHORE in greater depth.

1.2 Function Partitioning

In this section, I describe the four basic configurations built to evaluate tradeoffs in function partitioning. One tradeoff involves how to implement the full-text

```
QUERY: Show me some documents on object-oriented databases

RESULT:
    Rank Type  Title
(0) 32   Mail: RE: Pls send OODB info
(1) 26   Text: CS736 Class Notes
(2) 14   TRpt: SHORE OODBMS
(3) 4    News: OODBs useful?

SELECT DOCUMENT: 0

Subject: RE: Pls send OODB info
From:    Jeff Naughton
To:      Brad Rubin
Date:    8/1/94

$ cat text      // Display mail document
...

$ mail -fo text // Mail mail document
...
```

Figure 1: Sample Chrysalis Session

document index, as a persistent index object or using SHORE server B+ tree index support. The object index is an application object that resides in the object cache at run time while the server index is a SHORE server built-in index function that resides in the SHORE page cache at run time. Another tradeoff involves where the information retrieval function should reside, in the client or in a custom-function server.

The object index implementation structure is shown in Figure 2. The user interacts with the Chrysalis information retrieval application and also has a file system view of the document objects via the NFS interface. Chrysalis is bound with the SHORE client code. An object cache caches objects referenced during a transaction. Updated objects are written back to the server at the end of a transaction. The client code communicates with the SHORE server via Remote Procedure Calls (RPC) ([Blo92] describes RPCs in general). In the basic configuration, shared memory is used to move data referenced in these RPCs when the client and server share the same workstation. Multiple clients are supported, either on the same workstation or on different workstations. In the experiments discussed in this dissertation, all processes are local to a single workstation. There is a generic server shipped with SHORE. The SHORE literature refers to this as the "SHORE Value Added Server". I refer to this server as the "Generic Server" (GS). This server performs data reads and writes and logging on a local disk. It also has a page cache that reduces the amount of data I/O to disk. The SHORE client service and the object cache are replicated for each client.

The common index is implemented as an object and is replicated in all object caches, and each client's referenced document set (DOC) also appears in their respective object caches when referenced. There are also data structures called posting list objects (PL), that also occupy the object cache. This structure, described in Chapter 3, is a list of documents where a given word appears as well as a count of how many times it appears in each document.

Figure 3 shows an alternative functional partitioning which implements the full-text index using the native SHORE index facility, a B+ tree located in the server, instead of using an object index. The index is not replicated, but additional RPCs are now needed for the index operations. The document sets and posting lists still appear in their respective object caches when referenced.

Figure 4 shows the collapse of the Chrysalis server and the SHORE server, eliminating the extra process and communication overhead found in Figure 2. The multiple client object caches now reside in the server, although they remain unshared. This configuration represents a SHORE concept called a custom Value-Added Server, which is a SHORE server customized for a particular application. In this case, it is an Information Retrieval Server (IRS). Figure 5 corresponds to the Figure 3 implementation that uses SHORE server index, but using the IRS partitioning. Chapter 5 discusses the implications of a single shared server object cache, although one was not implemented for this effort.

Chapter 4 discusses the performance characteristics of all four configurations using three workloads and two document collections, in addition to several other

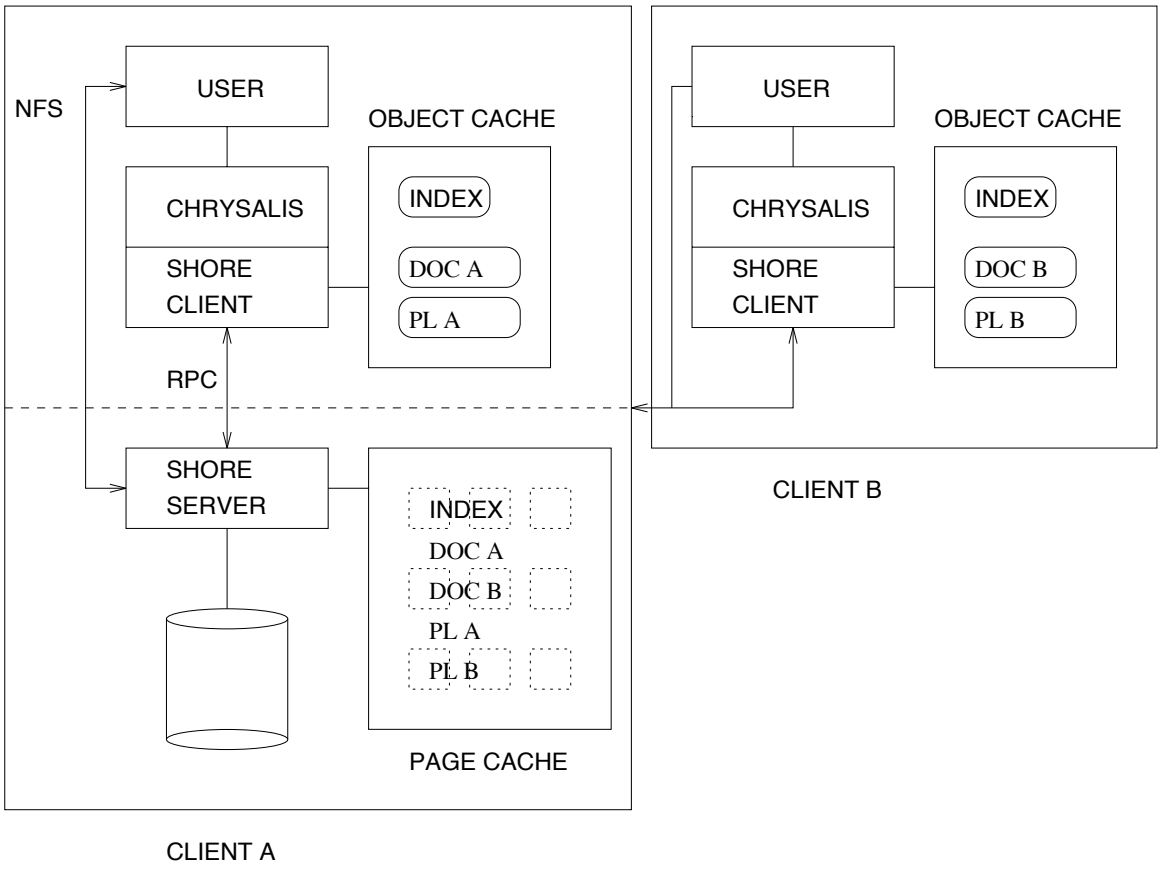

Figure 2: Object Index

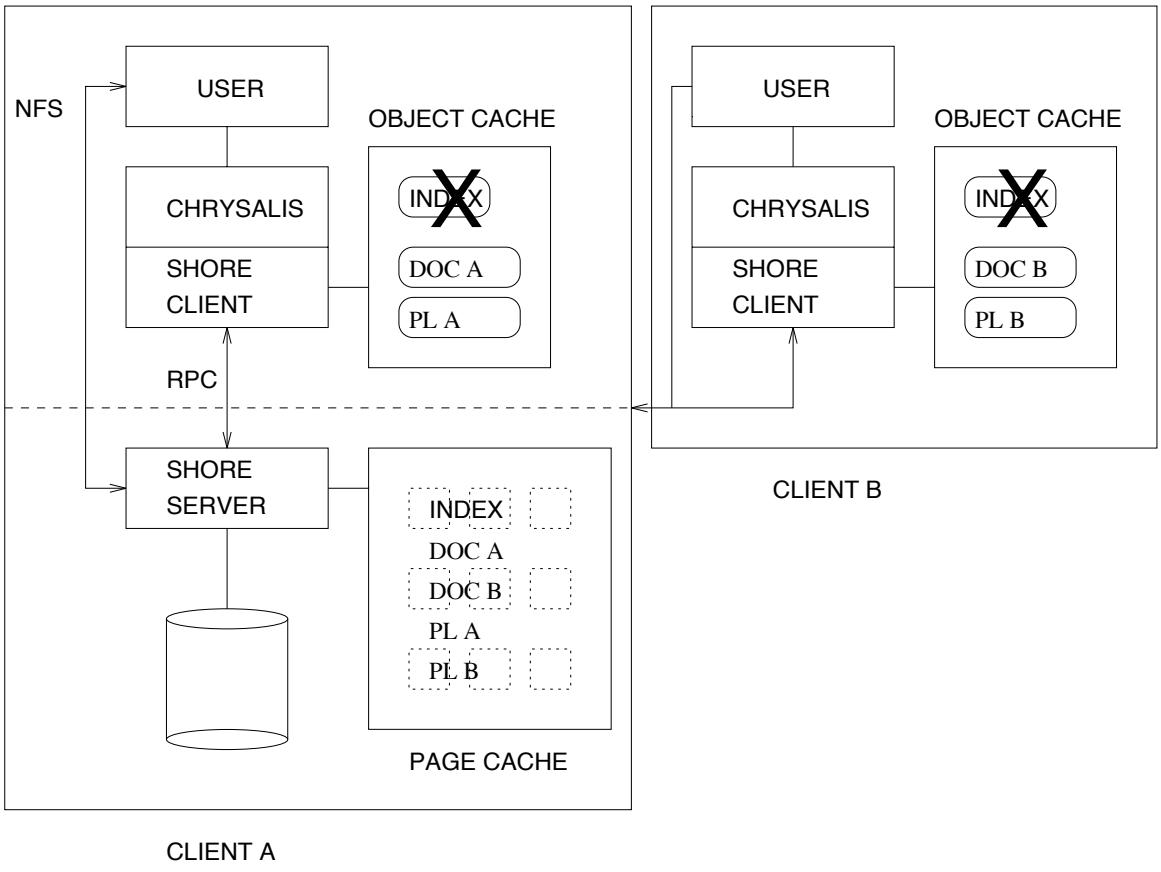

Figure 3: Server Index

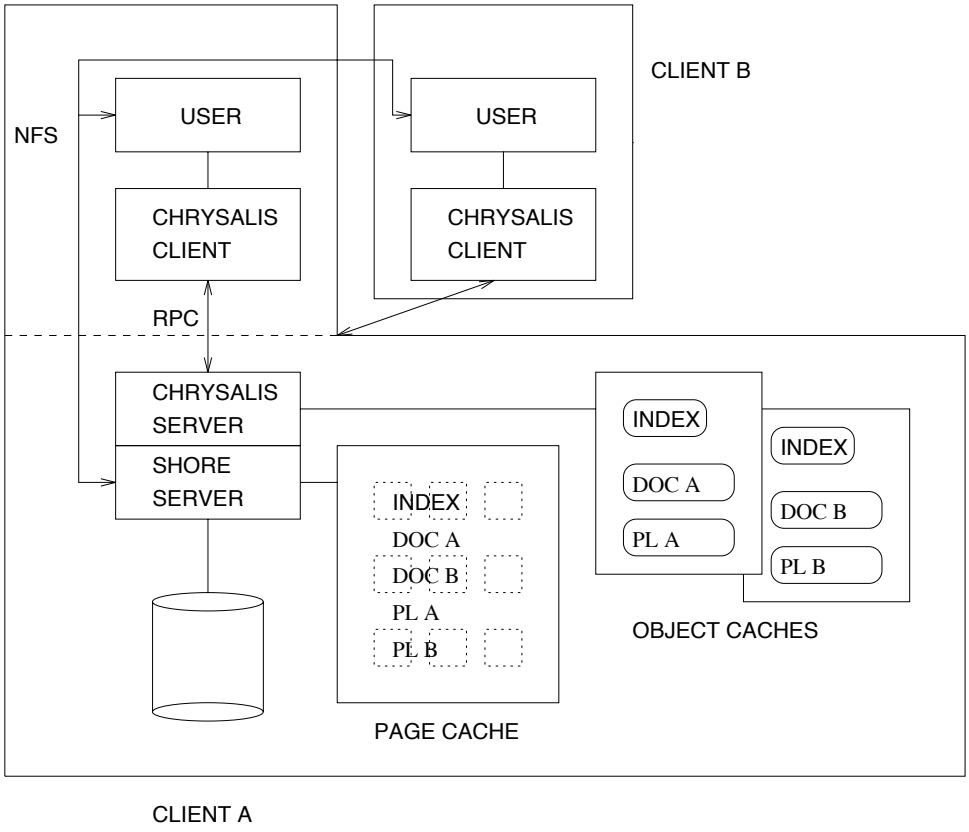

Figure 4: IRS with Object Index

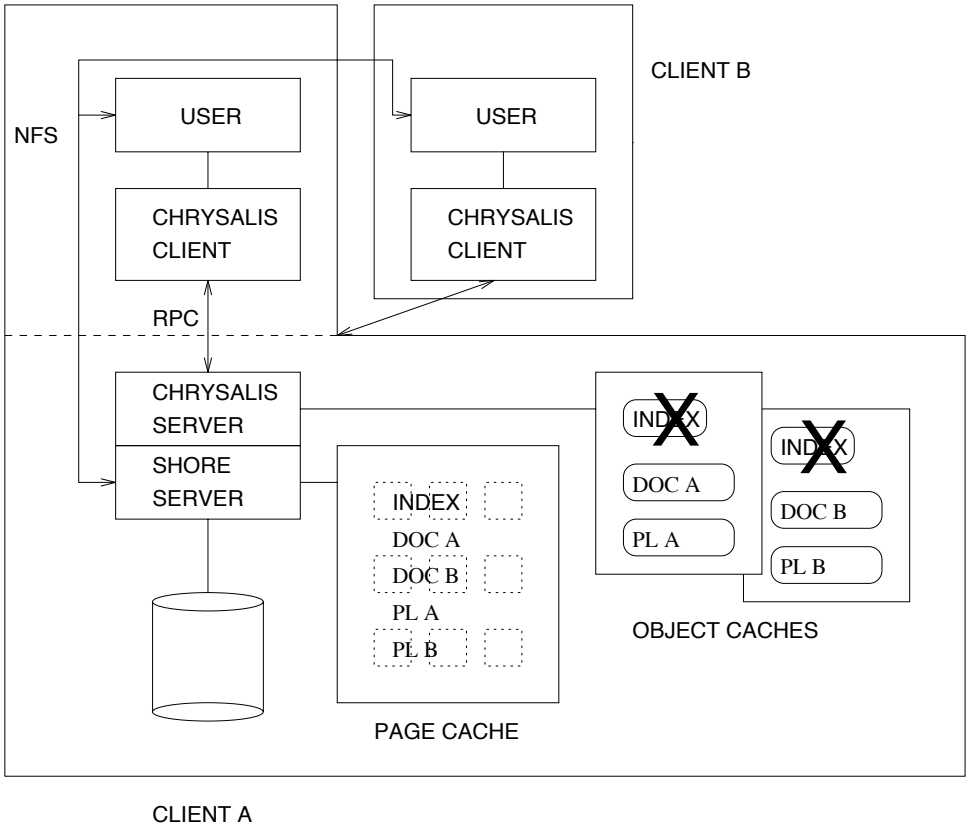

Figure 5: IRS with Server Index

performance studies. Chapter 5 summarizes the conclusions for this dissertation. To summarize, the four configurations are:

1. Generic Server, Object Index (Figure 2)
2. Generic Server, Server Index (Figure 3)
3. Information Retrieval Server, Object Index (Figure 4)
4. Information Retrieval Server, Server Index (Figure 5)

The three workloads used in Chapter 4's performance study are:

1. `Load-` reads UNIX files and creates SHORE document objects. This is an I/O intensive workload.
2. `Build-` builds a full-text index of the SHORE document objects. This is a CPU intensive workload.
3. `Query-` queries the index, using multiple users. This is a multi-threaded, read-only workload.

The two document collections used in the performance study are:

1. `Technical Reports-` a collection of 512 University of Wisconsin-Madison computer science technical report abstracts.
2. `Reuters-` a collection of 22,173 Reuters news articles from 1987.

1.3 Multi-server Issues

While this dissertation does not address multi-server issues, the basic issues are presented here for completeness. Figure 6 shows a multi-server configuration. Although this configuration was not supported when this thesis work was done, it may help resolve some problems discussed in later chapters. For example, this configuration shows a client that also contains a SHORE server with a page cache, which should help improve performance by allowing inter-transaction caching. Other interesting index configuration issues arise. Should there be one index over the data managed by all servers or should there be an index hierarchy? The former fits in well with the location transparency goals of SHORE's server-to-server communication facilities. The latter looks much like the resource discovery problem found in larger networks of workstations. There may be opportunities for productive parallelism using this structure for information retrieval. These questions are left for future work.

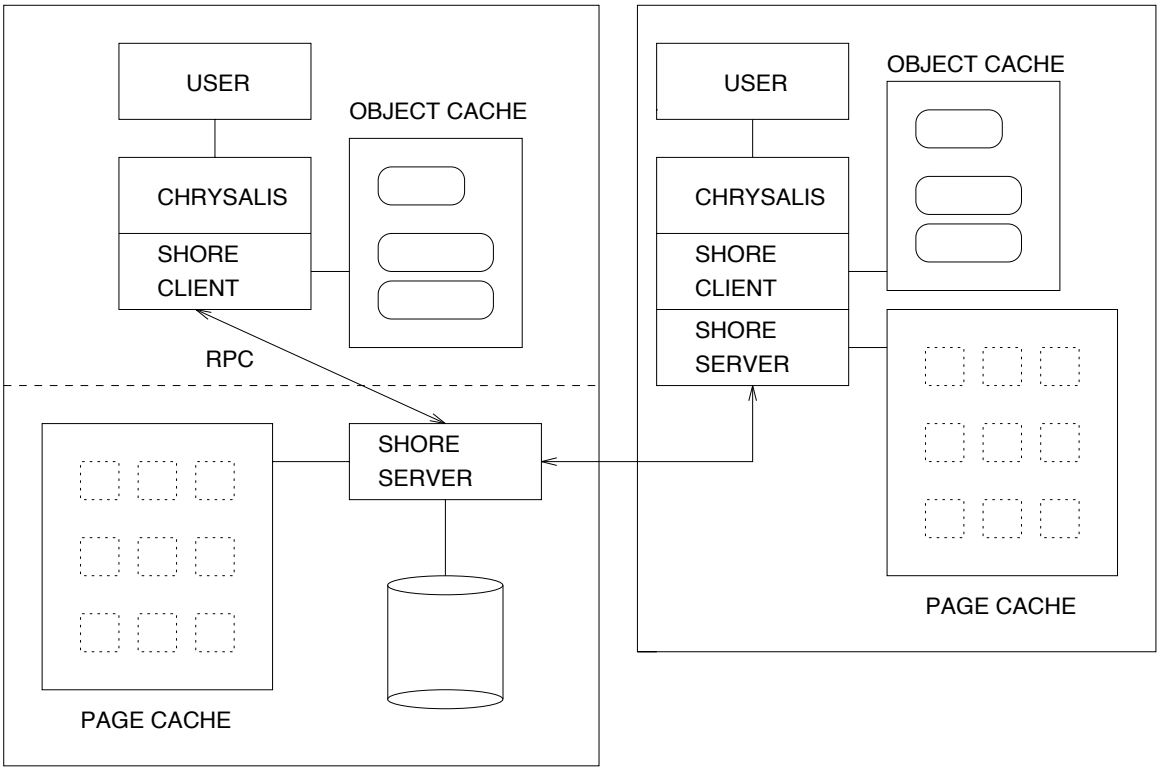

Figure 6: Multi-server Implementation

Chapter 2

Background/Related Work

The purpose of this section is to provide background information on a variety of information retrieval topics that are relevant to my thesis, as well as discuss work related to building information retrieval systems using object-oriented databases and file systems.

While the information retrieval techniques used in Chrysalis are not new, I have made several design decisions based on the information retrieval literature. The first part of this chapter explores the key issues and, in parallel, discusses the choices as well as certain implementation decisions that I have made in implementing Chrysalis. This first section may be safely skimmed by someone familiar with information retrieval technology.

2.1 Information Retrieval Background

2.1.1 Databases vs. Information Retrieval Systems

Both databases and information retrieval systems deal with the storage and retrieval of large amounts of information, but there are several characteristics that distinguish the two [SM83]. While either type of system could probably be used to support each other's intended use, the underlying model differences can make this support awkward. Databases (especially ones that follow the relational model) best deal with structured information, which consists of a number of records (tuples) with fields of defined types. Documents tend to be either completely unstructured (just a stream of ASCII characters) or semi-structured. An example of a semi-structured document is an electronic mail message. It consists of several well defined fields (subject, from, to, date, etc.) as well as an unstructured field (the mail message itself). Databases tend to operate with structured information, and information retrieval systems tend to operate with semi-structured or unstructured information.

Databases can support index creation and automatic index maintenance on record fields. In addition to indexing structured fields, information retrieval systems also maintain an index on the full-text unstructured contents of a document. In addition to indexing the existence of the words themselves, many information retrieval systems also maintain information on word position (paragraph, sentence, etc.) to support word adjacency queries. Queries in databases

usually involve Boolean combinations of record field content, while information retrieval systems usually try for a best match of the words of a query with the word profile of the documents stored in the system, returning a relevancy-ranked list of likely document matches. So, database query results tend to be precise, while information retrieval queries tend to be fuzzy.

Information retrieval is usually a read-intensive activity, with occasional writes as documents are added to the collection and indices are updated. Documents are generally not deleted once added. This read-mostly environment leads to several design decisions that are optimal for this environment, and which might not be good for a more balanced read-write ratio found in typical database applications. There are exceptions– groupware applications like Lotus Notes are document oriented yet also exhibit some traditional database characteristics. These design decisions affect areas like concurrency control and cache consistency in a distributed environment. For example, optimistic concurrency controls can be used because of the large read lock/write lock ratio. Also, multiple clients can keep cached copies of documents and indices and experience infrequent invalidation, so techniques like callback locking for cache management may be very efficient.

2.1.2 Indexing Techniques

Historically, there were two main manual indexing techniques used in the library science community, alphabetical subject headings and classification schedules [Lan79]. Alphabetical subject headings are listings of topics that describe a given piece of information. The categories are generally chosen from a fixed list of subjects. The popular Dewey Decimal System is an example of an enumerative form of this type of classification, listing a wide range of possible topics. The classification schedule is a hierarchical representation of a topic, with specificity increasing as the hierarchy is descended. One can view the alphabetical subject headings as a flattened version of the classification schedule.

The problem with both of the above techniques is that it is difficult to capture combinations of topics for synthesizing new categories of information. While several special techniques were developed to synthesize new categories, such as listing multiple topics in a fixed order, they can be difficult to use in practice. These special techniques are needed because the index complexity grows as the number of permutations of base topics for a given synthesized topic. For example, a document describing C++ and OBJECT-ORIENTED and DATABASES might have to be listed in all eight permutations. These techniques are collectively known as pre-coordinate indexing, because the synthesized categories are created during the indexing process, and the user uses these pre-created index topics during retrieval. Several extensions have been developed to improve the efficiency of the coordinate indexing technique, including specifying the role

of terms (to capture semantics) and using controlled vocabulary and thesaurus techniques.

A solution to the problem of growing complexity of synthesized topic classifications is post-coordinate indexing. This treats a description of a piece of information as a combination of base topics, like the pre-coordinate techniques, but these categories are synthesized during retrieval. This leads to a natural Boolean combination of terms for a query against an index that lists appropriate terms for each piece of information. Since there is no linear order dependency, as with the pre-coordinate techniques, the complexity grows with the combination of terms, not the permutation, and thus is more manageable and storage efficient. In the above example, the three terms can be listed in isolation, with a corresponding list of documents that contain each individual term. This technique was pioneered by Mortimer Taube in his Uniterm System.

2.1.3 Performance Metrics

Recall and precision are the two key metrics for information retrieval systems. Recall is defined as the number of appropriate documents retrieved for a user divided by the total number of appropriate documents present in the document collection, expressed as a percentage. Precision is defined as the number of appropriate documents retrieved for a user divided by the total number of documents retrieved for the user, also expressed as a percentage. Note that precision is easy to measure, because all the data is available in the query result. Recall

is very difficult to measure, because one must find all the documents in a collection that have not been found by the query. In practice, collection sampling and alternative query formulations are used to try and estimate the number of unretrieved documents.

Recall and precision are generally inversely related. To see this, consider an information retrieval system that returns ALL documents in a collection, regardless of the query. The recall will be 100%, because all appropriate documents are retrieved, but the precision will be near 0%, because many inappropriate documents were also retrieved. At the other extreme, if a single appropriate document was retrieved, and no others, then the precision would be 100%, while the recall (assuming there are many appropriate documents in the collection) would be near 0%. The recall/precision curve can thus represent the characteristics of a given information retrieval system. The user may wish to operate at various points of the curve at various times, depending on their needs. For example, if a user just wants a few initial documents on a topic, they would want high precision (they do not want to wade through many inappropriate documents) and will be willing to sacrifice high recall (they don't care, at this point, to get everything on a given topic). A researcher looking for in-depth information would want high recall (they want everything about a topic), and would be willing to sacrifice high precision (they do not care about having to wade through some inappropriate documents). In general, most systems try and achieve a default performance of about 50%/50%.

There are several devices available to adjust the performance point along the recall/precision curve. A thesaurus can be used to provide synonyms for the terms in a query or in a document index. For example, the query "AIRPLANE AND CONSTRUCTION" might be modified by including the synonym PLANE for AIRPLANE. This can increase the number of relevant documents retrieved, which will increase recall. It can, however, return documents that pertain to the woodworking tool, which will decrease the precision. Thesauruses can also provide a term hierarchy, allowing more specific or more general modifications of a term. For example, the more general term AIRCRAFT might increase recall and decrease precision, while the more specific term BIPLANE might decrease recall, but increase precision.

Salton [Sal86] shows a table of typical recall-enhancing and precision-enhancing devices, most of which are described in more detail in subsections of this chapter. Recall-enhancing devices include term truncation (also known as stemming or conflation), addition of synonyms, addition of related terms, and addition of broader terms. Precision-enhancing devices include term weighting, addition of term phrases, term occurrences in sentences, and addition of narrower terms.

There are several other information retrieval metrics. Effort (related to precision) is a measure of how much work it takes a user to process a search. For example, iterative searching (having the user give feedback to a system to gain better results) takes more effort but results in a better retrieval in terms of recall and precision. Response time is the time between a user requesting information

and having that information returned to the user. The document collection subject coverage and quality are also of key importance because information query results are limited by the quality of the underlying collection.

2.1.4 Information Retrieval Experiments

The ASLIB Cranfield project [Spa81] was one of the first major information retrieval experiments. It took place at the College of Aeronautics in Cranfield, England from 1957-1966, and was funded by the National Science Foundation. The goal of the Cranfield project was to establish an experimental procedure to measure the performance of a variety of manual indexing techniques, as well as the performance sensitivities to various parameters. Although this experiment involved manual indexing techniques, and I use an automatic indexing technique in Chrysalis, this project was significant because its conclusions had a major impact on much of the information retrieval indexing work that followed.

The first phase of the project, called Cranfield 1, used 18,000 aeronautical engineering research reports and periodical articles as the document workload. Half of these were on the specialized subject of high speed aerodynamics. Four indexing systems were used in the tests: Universal Decimal Classification (a classification schedule pre-coordinate system), alphabetical subject catalog (an alphabetical subject pre-coordinate system), faceted classification (a pre-coordinate system with constrained term order), and Uniterm coordinate indexing (a post-coordinate system). The key conclusions follow:

- The experiment varied the amount of time given to an indexer to index a document. Indexing times over 4 minutes gave no real improvement in performance.
- Non-technical indexers still produced high quality indexes.
- An inverse relationship exists between recall and precision.
- Systems operated at about 70%-90% recall and about 8%-20% precision.
- A 1% improvement in precision results in about a 3% loss in recall.
- All four indexing methods gave similar performance (Uniterm came out best).

There were many criticisms of the experiment. Some dealt with experimental design issues, such as the lack of statistical significance tests on the data. Probably the biggest problem with the experiment was the technique for developing questions for use as queries against the document index. These questions were derived from the source documents, instead of being developed by an independent source.

There was a follow-up experiment called Cranfield 2. It tried to address most of the experimental design criticisms of Cranfield 1, and used a smaller document collection (1400). Most significantly, experts in the field of aeronautics generated the questions that were used to test the indexing effectiveness. This experiment also tried to focus more on the different fundamental indexing techniques, rather

than specific implementations. The main conclusion from this study is that single term index languages using an uncontrolled vocabulary perform as well or better than the traditional term phrase/controlled vocabulary index languages. This had an important influence on the development of automatic indexing.

Salton and associates [Sal68] began an extensive series of automatic indexing experiments called the SMART system. These began at Harvard University in 1964 and continued at Cornell University in 1968. These experiments covered a very broad range of techniques such as term-weighting, ranking, relevance feedback, clustering, suffixing, synonyms, and phrases. The work and conclusions are too broad to discuss here, but the results had a major impact on the automatic indexing techniques described below.

2.1.5 Manual vs. Automatic Indexing

The success of the single term results in the Cranfield 2 study naturally led to the question "Can an automated system which extracts single terms from a document perform as well as a manual indexer?" Since the cost of computers was decreasing, and the cost of the manual indexing labor was increasing, the answer to this question was of major economic importance. Unfortunately, this was not an easy question to answer. In the following, I discuss the results of two contradictory studies in this area that highlight the key issues. The results of several studies have shown that automatic indexing is just as poor, or maybe even better, than manual indexing.

A paper by Blair and Maron [BM85] looked at a large collection of documents (40,000 full-text documents pertaining to a lawsuit) that were stored in the IBM STAIRS automatic text-retrieval system. The study took over 6 months to complete and cost $500,000! Forty queries, generated by the two lawyers responsible for the lawsuit, were used in the study. STAIRS, while having some deficiencies, is generally regarded as a state-of-the-art information retrieval system. The test results showed an average precision of 75%, and an average recall of 20%. These results amazed the lawyers– they thought almost all the pertinent documents had been retrieved, instead of the actual 1/5 of pertinent documents!

There were three main conclusions in the Blair and Maron study. First, when users require high recall in a large collection of documents, they can not broaden the search request because they will be overloaded with result documents, as precision is decreased. Second, they argue that manual indexing is preferable to automatic full-text indexing when high recall is desired because the time spent typing in the documents is comparable to the manual indexing time. Third, they argue that full-text systems are not user friendly, because even trained searchers could not obtain adequate results.

Salton [Sal86] has a different interpretation of the Blair/Maron study. First, he criticizes the study for drawing conclusions about the differences between manual and automatic indexing, as well as between small and large document collections, without ever studying these variables! Second, he cites other studies of

large document collections where users did not suffer from "output overload"— when users asked for more output (higher recall), they got it. Third, he cites other studies that show that automatic full-text indexing is at least as good as, or even better than, manual indexing. Further more, automatic indexing can be performed on document abstracts (which may improve performance), which decreases the document typing entry work (and a rapidly growing percentage of documents are originated on-line, so the importance of this issue is diminishing).

Most importantly, Salton cites a study in the late 1960's by Lancaster on the Medlars medical database (700,000 documents) using a manually prepared index and 300 queries. The performance varied along the expected precision/recall inverse relationship line. A high precision search (80%) had a corresponding recall value of 19%. A high recall search (89%) had a corresponding precision value of 20%. Note that the high precision results (80/19) correspond closely with the Blair/Maron results (75/20), showing, in this case, that automatic indexing performs comparably to manual indexing.

In high recall searches on Medlars, 500-600 of the 700,000 documents were typically retrieved (0.07% of the collection), while on average, 130 documents were relevant. While this is not necessarily document overload (as predicted by Blair/Maron), it does highlight the problems users face when they ask for a lot of information in a large document collection– they get a lot! It is interesting to note that the lawyers in the Blair/Maron study should be interested in operating at the high recall/low precision area of performance because of a lawyer's need

for comprehensive information, but instead were operating at the opposite end. In general, the results of both techniques are poor– far from the desired goal of getting ALL, and ONLY, the desired documents in a collection.

2.1.6 Boolean vs. Natural Language Queries

The traditional, and most commercially used, user interface for information retrieval queries is the Boolean model. Here, the user lists a series of terms, combined using the Boolean constructs AND, OR, and NOT. Document sets corresponding to each term are generated from an inverted index of terms, and Boolean set operations are performed according to the user's specifications. For example, a Boolean query might look like "(OBJECT AND ORIENTED AND DATABASE) OR OODB OR OODBMS". This can be a productive query language for an experienced retrieval user, but can be complex for more naive users. The Boolean model is intolerant of misspellings, and the use of terms in a conjunctive clause that do not exist in a document would eliminate that document, which might be a stronger action than the user might have intended.

An alternative to the Boolean query is to use a natural-language list of terms. The above query presented in this form might look like "IS THERE ANYTHING HERE ON OBJECT-ORIENTED DATABASES? MIGHT ALSO BE CALLED OODB OR OODBMS". With this technique, a similarity measure between the query and each document in the collection yields a relevancy-ranked list of documents. Studies have shown that natural language (or partial match) systems

give better performance than Boolean (or exact match) systems [BC87]. The inertia of existing Boolean systems is tremendous, so relatively few information retrieval systems support the natural language technique today.

The natural-language list of terms technique is also compatible with several other processing techniques (discussed later), such as the vector-space view of documents, term weighting, relevancy ranking, and relevancy feedback. While the Boolean query model is not excluded from using these techniques, it is less natural. Many proposals also exist that extend the Boolean model to increase its flexibility. I believe the best solution may be to use an underlying vector space model (discussed later) that supports intermixed natural language and Boolean queries. I chose the natural language user interface for Chrysalis.

2.1.7 Term Identification

During automatic indexing, the indexing program must identify index terms in the document text. Determining what is a term and what is not a term seems like an easy problem, but there are several complex considerations. There is no one "right" solution to this problem. The following highlights the different tradeoffs, as well as my decisions. In general, I follow the recommendations in Fox's chapter in [FBY92].

Terms are delimited by one or more spaces and begin with a letter. Punctuation marks are treated just like spaces, so periods and commas are not included in the word. This can lead to some problems with words that are considered one

semantic unit, but have punctuation in them (i.e. COMMAND.COM, OS/2). Hyphens are considered as punctuation as well. This helps with inconsistent usage words like "state-of-the-art" vs. "state of the art"). This hurts when the hyphenated word is a single semantic unit (i.e. Jean-Claude, F-16). This can also cause major problems when hyphenated words appear because of a syllable break at the end of a line. I chose to treat all punctuation like a space, despite these disadvantages, because trying to handle these special cases could lead to other problem cases as well.

Sometimes letter case is important to distinguish words, but these occurrences are infrequent enough, and the chance of inconsistent usage (like the first word of a sentence) is frequent enough, that I decided to convert all terms to lower case.

Numbers, in general, do not make good index terms. Certain words that contain digits, however, are important index terms (i.e. B6 and B12). I chose to define a term as not beginning with a digit, but digits in a term are part of a term.

There are several techniques used to build a lexical analyzer to parse terms from documents using the above rules, including using a lexical analyzer generator tool (like the lex tool for UNIX), trying to write a custom program to do the rule processing, and writing a custom program but using a finite state machine as a model. I found it easiest to think of the problem in terms of a state transition diagram, so chose the latter approach.

Another implementation detail is the technique used to identify the character class (letter, digit, punctuation) and do the lower case conversation. I use two tables that list all characters, and just look at the corresponding table entry for the class and case. This is much faster than a series of range comparisons.

2.1.8 Stop Lists

Words that occur very frequently (i.e. the, of, and, to) not only make poor index terms for retrieval purposes because of poor discrimination, but they also take up large amounts of index space. The ten most frequently occurring words in English account for 20-30 percent of the words in a document. A stop list is a list of frequently occurring words that are filtered from indexing and queries [FBY92].

There is no question that eliminating frequently occurring words is a good idea, but there is much debate on what this set of words should be. Frequent words are sometimes good index words. On the other hand, infrequent words are sometimes poor index words for special collections (i.e. computer in a collection of computer science technical reports), but these cases are probably best dealt with by term weighting techniques (described later). So, commercial systems tend to use few stop words to satisfy the needs of a diverse spectrum of documents. Some systems use only eight stop words (a, and, an, by, from, of, the, with).

I chose a stop list of 425 words derived from the Brown corpus of 1,014,000

words drawn from a broad range of literature in English. The method and rational for choosing this set of words is described in [Fox90]. This has, however, led to certain problems. For example, all of the letters of the alphabet are considered stop words. In indexing computer science technical reports, I not only missed index entries for documents containing references to the C programming language, but also missed C++ programming language references too (as discussed above, the ++ is treated like a space, so parses to just the letter C). These problems led to my decision to make the stop list an input to my index and query facility, allowing it to be easily modified.

There is an interesting implementation technique for stop lists. The obvious solution is to just search the list of stop words for each index and query term. A performance enhancement is to use a sorted list of stop words and perform a binary search. I chose an optimal technique using a minimal finite state machine (MFSM), where each new letter causes a state transition. If one ends up on a terminal state when finished with the word, it is a stop word. The finite state machine is minimal because some states are reused if possible. I generate a new finite state machine from a sorted stop list of words, and then just store this MFSM structure for later loading at startup and use in queries and indexing. For my 425 stop words, the MFSM has 318 states and 555 arcs. This structure is a candidate for implementation as a persistent object.

2.1.9 Stemming

Stemming, or term conflation, is a procedure that maps different variations of the same semantic root word to a common representation. For example, the words (computer, computes, computation, compute) all conflate to "comput". Note that the stem itself does not have to be a real word. Stemming improves recall and decreases index size. There are, however, many different algorithms for performing stemming, and none are perfect. Some are as simple as dropping any 's' that ends a word. I chose a rule-based algorithm called the Porter stemmer [Por80] because it is amenable to table-driven implementation and its effectiveness is comparable with other proposed algorithms [FBY92].

The Porter stemmer is a very CPU intensive algorithm. One possible performance improvement technique is to cache words and their stems. As each word is processed, a cache lookup can be performed before using the algorithm.

2.1.10 Term Weighting

The goal of a retrieval operation is to provide a list of documents, in relevancy ranked order, that best match a user's query. This can be done by viewing a representation of the document as a vector that contains a term weight for each term that appears in the document, with a weight of 0 for any term that does not appear in the document. Of course, this is a sparse vector and must be stored as such, but conceptually the document is viewed as a vector. A query can be similarly represented by a vector, with an element of 1 for each term

that appears in the query and a 0 for each term that does not appear in the query (also sparse). Although this model is most naturally used with natural language queries, it can be used with the Boolean query model as well. One method of calculating the similarity of a query with a document using the vector space model is to take the dot product of the two vectors. The resulting scalar can be used to rank the documents in order of decreasing relevancy to the query. Term weighting increases the performance of this dot product operation by giving more weight to terms that appear in a document many times, but do not appear in the general document collection too often. For example, the word COMPUTER is a poor index word in a collection of computer science technical reports, because it will likely appear in all of the documents. This can result in excellent recall, but very poor precision, so this term in a document in this document collection should have a low weight. On the other hand, multiple occurrences of the word SIMULATION in a document, coupled with a low frequency of occurrence of this word in other documents in the collection, would tend to make this a good term for retrieval; thus, this term in such a document should have a high weight. Equation 1 shows the first step towards an equation that gives the desired behavior:

$$W_{i,j} = F_{i,j} \tag{1}$$

where:

$W_{i,j}$ = weight of term i in document j

$F_{i,j}$ = frequency of occurrence of term i in document j

This results in a weighting system that is good at identifying documents that are most relevant to a query (high recall), but it puts too much emphasis on words that might appear in many documents, so precision might be much lower than it could be.

Equation 2 results in the desired behavior for words that appear in many documents. For example, if a term appears in all documents of a collection, the occurrence frequency is multiplied by 1. The multiplier will increase for words that appear in fewer and fewer documents in the collection. The bracketed term is called the inverse document frequency (IDF) factor.

$$W_{i,j} = F_{i,j} * [log_2 N - log_2 N_i + 1] \tag{2}$$

where:

$W_{i,j}$ = weight of term i in document j

$F_{i,j}$ = frequency of occurrence of term i in document j

N = number of documents

N_i = number of documents containing term i

One final step to further improve the behavior can be taken after the dot product between the query vector and the document vector is performed. This step divides the dot product result by the length of the document vector (the square root of the sum of the squares of the elements). This normalizes the result by the document size, so undo weight is not given to lengthy documents. Note that there have been hundreds of proposals for term weighting equations; the ones just described simply illustrate the basic intent. Chrysalis implements Equation 1, although other weighting techniques could be implemented as fairly easy extensions to the base implementation. Additional statistics that are incrementally updated could be kept to minimize the performance penalty for some of the more sophisticated weighting schemes.

2.1.11 Term Discrimination

Another way to classify terms as good index terms uses the term discrimination model [Sal+75]. At one extreme, terms that appear very infrequently in a document collection are poor index terms, because a user would have to specify this term exactly for the document to be retrieved. At the other extreme, terms that appear very frequently also make poor index terms, because too many documents would be retrieved when this term is used in a query. The best terms, therefore, are the ones that appear in the middle of a usage frequency list.

Terms can be moved from the very small end of the frequency spectrum by

using a thesaurus to group these words into a common, related group. For example, the terms MINICOMPUTER and MICROCOMPUTER can be grouped into the same class of COMPUTER. Thesaurus techniques, while well known, can be very collection topic specific and be difficult to generate. Stemming helps solve this problem in a more limited sense. Terms can be moved from the very frequent end of the frequency spectrum by creating compound term phrases. For example, the terms COMPUTER and PROGRAM, which might appear too frequently by themselves, can be combined to the term phrase COMPUTER PROGRAM, which not only appears less frequently, but also improves the semantic recognition. Term phrase construction techniques are known, but they are complex to implement. Recall that the Cranfield 2 results show that the single term systems performed as well as or better than the term phrase system in a manual indexing environment.

2.1.12 Relevance Feedback

Relevance feedback [Sal89] is a query technique that allows the user to mark documents in the result set as relevant (good) or not (bad). The user's initial query is then modified to reflect these differences, and another retrieval is performed. The process can be repeated. Relevance feedback is a two phase process. The first phase uses the vector-space, weighted term technique described in section 2.1.10. The feedback phase occurs when the user rates the returned documents as "good" or "bad". A new query vector is formed by adding the

good document vectors to the initial query vector and subtracting the bad document vectors. The good and bad vectors are normalized by the number of good and bad documents. The following formula illustrates the basic concept (again, there have been many modifications proposed and used):

$$Q_1 = Q_0 + \frac{1}{n_1}\sum_{i=1}^{n_1} R_i - \frac{1}{n_2}\sum_{j=1}^{n_2} S_j \tag{3}$$

where:

Q_1 = new query vector

Q_0 = initial query vector

R_i = vector for relevant document i

S_j = vector for irrelevant document i

n_1 = number of relevant documents

n_2 = number of irrelevant documents

Studies have shown very good retrieval improvement with this technique– Salton reports 30% to 60% precision improvement for a fixed recall level using one pass of relevancy feedback. Relevancy feedback is used in the Internet resource discovery tool WAIS. A performance issue is that the number of terms in the query can grow very large (hundreds), which is why this technique did not become popular until the recent availability of high performance workstations.

2.1.13 Automatic Indexing Process

Putting all of the previous results and techniques together results in an overall "blueprint" for automatic indexing. Salton's recommendations [Sal89], as well as the Chrysalis implementation, follow.

1. Identify the individual words occurring either in the documents or in the document excerpts (titles/abstracts).

   Chrysalis: Uses the term identification technique described in Section 2.1.7 on the complete document contents. In the case of non-text documents (i.e. images, audio, etc.), perform term identification on the document description text.

2. Use a stop list of common function words to delete them from the documents because they are insufficiently specific for representing the document contents.

   Chrysalis: Uses the stop list technique described in Section 2.1.8.

3. Use a suffix stripping routine to reduce words to their stem form to enhance recall, using a limited number of basic rules.

   Chrysalis: Uses the more complex Porter stemmer algorithm described in Section 2.1.9.

4. For each resulting word stem i occurring in document j, compute a term weighting factor by freq(i,j)*IDF(i). This improves effectiveness by distinguishing the important content terms from the less important ones.

   Chrysalis: Uses just the freq(i,j) component.

5. Represent each document by the chosen set of weighted word stems.

   Chrysalis: This is done.

6. Use a thesaurus to replace terms with low document frequencies (discrimination values near 0) by their corresponding thesaurus class identifications.

   Chrysalis: Not implemented.

7. Use a phrase-formation process to generate term phrases that incorporate terms with high document frequencies (negative discrimination values) based on term co-occurrences in the document.

   Chrysalis: Not implemented.

8. Compute a combined term weight for assigned thesaurus classes and term phrases, and represent each document by the corresponding sets of weighted single terms, term phrases, and thesaurus classes.

   Chrysalis: Not implemented.

9. Perform repeated relevancy feedback cycles.

   Chrysalis: Not implemented.

2.1.14 Information Retrieval Conclusions

Information retrieval has historically been an important topic, and its importance is continuing to grow. Technological revolutions like the Internet, political hype like the Information Superhighway, and national research initiatives like the Digital Library will demand even more from information retrieval in the future. Although this area is too broad to completely survey in one chapter of a thesis, this chapter has attempted to show a thread of automatic information retrieval design decisions, starting at the manual indexing roots and ending with choices made in an implemented system called Chrysalis.

I conclude with two thoughts. First, it is amazing that after so many years of research, the state of information retrieval performance, measured by precision and recall, is so poor. Perhaps this parallels the struggle in the artificial intelligence community with getting machines to understand the intent of humans. Second, the Computer Science Database community and the Library Science community have both made significant contributions to information retrieval. Future results will likely come from people trained in both disciplines.

2.2 Related OODB/FS Work

This section describes two implementations of information retrieval applications on object-oriented databases, the INQUERY/Mneme project from the University of Massachusetts [Bro94a, Bro94b], and the Rufus project from IBM

Almaden Research Center [Mes91, Sho93]. The Semantic File System (SFS) from MIT [Gif91], the Gold mailer project from Matsushita Information Technology Laboratories (MITL) [Bar93], and the project described by Yan and Annevelink [YA94] also share some common characteristics with Chrysalis. The OdeFS system from AT&T [Geh94] is another system, like SHORE, that supports the dual object-oriented database/file system paradigm. OdeFS is also discussed in this section.

2.2.1 INQUERY/Mneme

In INQUERY/Mneme, document posting lists (referred to as inverted lists in their papers) containing term weighting information are stored in an OODB, while the actual documents and a hash table index mapping words to posting lists are stored in a UNIX file system. These structures are explained more fully in Section 3.3. The Mneme persistent object store supports fixed sized objects for the posting lists. Posting lists start as 16 bytes in size, and grow to 8,192 bytes in multiples of two. After this size is reached, successive 8,192 byte objects are chained together. The application must manually copy object contents from the smaller to the larger when the smaller object becomes full. This technique is a good match for the Zipf distribution [Zip49] characteristics of these posting lists, where there are a small number of very large posting lists, and a large number of very small posting lists. This corresponds to the fact that a small set of words are very common, appearing in most documents and a large set of

words are not very common, and appear in few documents. Chapter 4 shows the result of applying this technique to Chrysalis.

An important contribution of this work was the demonstration that the cache management provided by the OODB (Mneme) gave better performance than the previous version of INQUERY, which used a non-caching B-tree implementation in UNIX files instead of an OODB. The goal of the work was to show that an "off-the-shelf" database is well suited for supporting information retrieval applications. They also showed that incremental indexing could be done efficiently, without requiring complete index rebuilding when new documents are added to a collection.

2.2.2 Rufus

In the Rufus information retrieval system from IBM Almaden, the meta data is stored in a database and the actual documents are stored in a file system. Rufus models information with an extensible object-oriented class hierarchy and provides automatic document type classification. Class-specific methods perform basic document functions, like initialization, displaying, and indexing. Rufus can also store relationships between various information objects and handle schema evolution.

The Rufus classifier looks at keywords, file names, and bit patterns near the beginning of a file and applies an evaluation function on the results. The evaluation function returns the document class. They achieved a 90% success

rate in classification, but the classifier became unstable as more class types were added. To address this problem, the Rufus team is moving to a cosine coefficient similarity measure technique where each document class is represented by a feature vector. The dot products of these feature vectors with document feature vectors yield a scalar. The document class corresponding to the largest scalar is the closest match.

Rufus creates an object identifier from type-specific attributes of the document. For example, message identifiers are used for mail messages, and inode/device numbers are used for plain UNIX files. The latter approach fails if a file is moved in the file system.

The Rufus class hierarchy first separates `binary` and `text` objects. `Programs, documents, and mail` are subtypes of text, while `C` and `FORTRAN` are subtypes of `programs`. Rufus is moving to a "conformity model" to address schema evolution by distinguishing an inheritance hierarchy (used for implementation reuse) from an interface set (used to collect object types with like attributes). Old object instances use an old class, new object instances use a new class definition, and accesses are done via the interface set (which includes both old and new objects).

Rufus uses a B-tree index for words. This index points to posting list objects, much like INQUERY/Mneme. Stop list processing is performed. Rufus tries to uses small blocks for infrequently occuring words with small posting lists and larger, fixed-sized larger blocks for the more frequently occuring words with

larger posting lists.

2.2.3 Other systems

The Semantic File System (SFS) goal is to provide associative access to a UNIX file system. It extracts attributes from files and indexes them automatically, but does not use an OODB. It provides a query language that is well integrated with the UNIX file manipulation commands. File system mutation operations trigger incremental indexing.

The Gold mailer is a mail facility designed to inter-operate with unstructured and semi-structured data in a more natural way than previous relational systems. Messages are stored in the file system, while the meta data is stored in a relational index. The schema consists of a keyword, a document identifier, a keyword position, and a header name. This allows a query language to support both full-text and attribute queries by using a common proximity search mechanism.

The system does not support automatic object cache management like SHORE, so the application has to manually manage a cache of posting lists, called an overflow list, to help reduced disk I/O. While transactions are not supported, there is a checkpoint log that can be used for index recovery if a failure occurs during processing. This system, like Chrysalis, uses a level of indirection between the posting lists and the document file names to allow fast document deletion. The Gold mailer uses a hashed file to implement the full-text index.

Yan and Annevelink describe a system that couples an information retrieval

system (Alliance Technologies TextMachine) with an OODBMS (HP OpenODB). The Chrysalis effort by contrast involves using an OODBMS to implement the entire information retrieval system. Their system has a document class hierarchy similar to Rufus, with extensions to support finer granularity parts of documents such as paragraphs and sentences. They concluded that the tight integration of query using a standard database query language with metadata in an OODBMS allows both better modeling of complex text structure and easier access to them than previous text retrieval systems. The conclusions by Yan and Annevelink on the utility of an object-query interface to an information retrieval system also apply to Chrysalis.

2.2.4 OdeFS

OdeFS is another system, like SHORE, that supports the combined object-oriented database/file system paradigm. All documents stored in a UNIX-like subdirectory are of the same type (more restrictive than SHORE). A programmer can override the object read and write methods to perform type-specific handling of files. For example, one could override the read method of a mail object to add the word "Subject: " before the `subject` attribute text, helping users view the multiple attributes of an object as a single UNIX file. Similarly, an overridden write method could parse the "Subject: " field and store it in the object's `subject` attribute. The same functions are performed in Chrysalis/SHORE with parsing during document object initialization and display. The UNIX view in

OdeFS can be dynamically generated, while in SHORE the UNIX view must first be generated and stored in a `text` attribute.

OdeFS uses an NFS protocol for client/server communication. This has the advantage of supporting a variety of computer systems, although has some performance disadvantages and did not allow sufficient cache management control in cases where they wanted to control write caching and NFS always writed cached data. OdeFS does not support transactions, but does support a simple, single-object query facility. Inter-object references are not swizzled to direct memory addresses.

I believe much of the Chrysalis function could be ported to OdeFS, but suspect that the direct RPC communication, page and object caching, and pointer swizzling provided by SHORE would result in better performance. To my knowledge, no commercial OODBMS currently supports this dual OODB/file system functionality, although there is no fundamental reason why this functionality could not be added.

Chapter 3

Chrysalis

This chapter describes Chrysalis, an information retrieval application for SHORE used to explore the issues in this thesis. This chapter focuses on functional and implementation issues. Chapter 4 focuses on performance issues.

3.1 Overview

When building an information retrieval application, the implementor is immediately faced with a choice of technologies for storing persistent data. Perhaps the most common choice is a file system. A file system has the advantage that a huge variety of existing programs can be used to process and view the documents stored in the system (for example, text editors, word processors, typesetting programs, spreadsheets, compilers, image viewers, and movie and audio players). However, this approach has two main drawbacks:

1. A file system is a poor choice for storing the persistent complex data structures required by an information retrieval application.

2. Most file systems provide only rudimentary support for maintaining data consistency and safety in a dynamic environment — in particular, they do not support high concurrency access with transaction semantics.

Moving to a relational database system solves the second problem, since relational databases provide excellent support for maintaining transaction semantics in conjunction with high concurrency. However, current relational systems are perhaps worse than a file system for storing complex data structures, and cannot directly support legacy applications that expect files as their input. There are also examples of file systems that store objects, such as IBM's SOM [Lau94] and Microsoft's OLE [Bro95], but these implementations do not support features like transactions and query.

Object-oriented database systems are much better at solving both the "storing complex data structures" and "provide transaction semantics" problems raised above. However, with few exceptions, they can not support legacy applications that expect to run off of files. This means that, to run such an application, the OODBMS-based system must either (1) constantly export data to files before these applications run, and import data every time such an application modifies the data, or (2) store the meta data in the OODBMS, but keep the bulk of the data externally in a file system. The first alternative is inefficient, while the second removes the guarantees of transaction consistency from most of the data

in the information retrieval application.

A main goal of the SHORE project at the University of Wisconsin is to provide a better alternative for the construction of data servers. The SHORE system provides both OODBMS and file system access. This is described more fully in Section 3.2, but briefly, SHORE can store persistent objects like any other OODBMS, but can also export a UNIX file system interface to an attribute of the objects it stores.

Chrysalis was built to help evaluate how well this dual OODBMS/File System paradigm supports the construction of data servers. This experience showed that the Chrysalis/SHORE approach is an improvement over other approaches in several respects. Like previous information retrieval applications, Chrysalis uses a persistent object store (SHORE) to store meta data. But, unlike previous systems, the documents themselves are also stored as objects. The SHORE approach allows the use of all existing UNIX applications and tools, and also has important impacts on system integrity and ease of programming. SHORE provides many services that the application layer of previous information retrieval systems either did not have, or else had to build themselves, including concurrency control (locking and transaction semantics), recovery, interprocess communication, and cache management. The SHORE support for object-oriented implementation is critical in allowing Chrysalis to easily support extensions to different document types.

An extensible object-oriented framework for information retrieval was developed for Chrysalis This framework is an abstraction of the key interfaces and objects appropriate for a family of implementations. This framework not only allowed easy modification for the various versions of Chrysalis used in the experiments in this thesis, but can also serve as a base design for other research and commercial purposes.

The rest of this chapter is organized as follows. Section 3.2 describes how the information objects corresponding to Chrysalis documents are represented within SHORE. Section 3.3 gives some design details for the implementation and describes some benefits derived from using SHORE. Finally, while in general SHORE's functionality provides excellent support for the implementation of information retrieval systems, there are aspects of SHORE that could be improved for this application. These and other functionality conclusions appear in Section 3.4.

3.2 Information Objects

SHORE provides storage for objects of user-defined types. The type interface specification is via a language called SDL, which is closely related to the ODMG IDL language [Cat94]. SHORE automatically stores the object type along with the object. Method interfaces are also specified in SDL, and their implementation will eventually be supported by several language bindings (initially C++). As an example, Figure 7 shows how a BLOB (Binary Large OBject) and a Document

```
interface Blob {
public:
  void init( in lref<char> fileName );
  ref<Blob> getLink() const;
protected:
  attribute text contents;
}

interface Document : public Blob {
public:
  void initAttribute();
  void display() const;
  void getTitle( in lref<char> buffer ) const;
  string getText const;
  attribute string author;
  attribute long   version;
};
```

Figure 7: SDL Definitions of Blob and Document Objects.

object are specified in SDL. All objects have a globally unique, never reused, 16-byte object identifier (OID). This OID consists of an 8-byte volume identifier and an 8-byte serial number.

In Figure 7, the base class called BLOB defines a special character string of type `text`, called contents, that holds the actual document text (which could be ASCII text or binary data, so text is perhaps a poor name choice– but it is dictated by SHORE). The contents attribute is initialized from an existing UNIX file with an `init` member function. The contents of a `text` field are visible to UNIX applications via the SHORE NFS mount facility.

The class Document is derived from BLOB, inheriting the contents attribute.

There are two additional attributes that describe the document (character string author and an integer that indicates a version number). There are also four member functions, one to initialize the object attributes from a UNIX file, one to display the document, one to return a document title, and one to return the document contents. The initAttribute method looks for a UNIX file ending with a ".key" corresponding to the file name used to initialize the contents. If such a file is found, its contents are parsed to initialize the author and version fields. These fields are set to defaults if no ".key" file is found. If necessary, a SHORE program can be written to subsequently modify these fields. During document retrieval, the UNIX filename "text" (could be any name, I chose text because it reflects the ASCII contents and is easy to remember) is dynamically bound to the retrieved document object. This allows the user to reference the object using this name in UNIX commands (i.e. "cat text") using the NFS mount interface to SHORE, without needing to copy data into a UNIX file.

The SDL interface specification looks like C++, except for a few changes and additions. The term `interface` is used instead of class. Data items are labeled with the keyword `attribute`. Method parameters are labeled as input (`in`), output (`out`), or input/output (`inout`). Pointers to objects and pointers to memory within an object are specified differently. Pointers to persistent SHORE objects (i.e., OIDs) use a template function called `ref`, so `ref<Blob>` is a pointer to a Blob object. Pointers to memory within an object use a template function called `lref`, so `lref<char>` is essentially equivalent to `char*`. The

template syntax provides a level of indirection for implementing pointers to persistent objects. The ref actually occupies only 4-bytes when stored in an object. SHORE uses this value as a pointer to a table whose entry is either a local volume object location or a remote system volume identifier/serial number pair (OID). Thus, object location is transparent to an application. The term OID refers to both this 4-byte ref pointer and the 16-byte volume identifier/serial number in this chapter, since the indirect lookup mechanism is hidden from the application. Pointer swizzling techniques are used to speed object access.

Member functions labeled with `const` are signals to the SHORE object cache manager that the function will not modify data, so the objects referenced by these functions need not be flushed to disk as they would be if `const` were not specified. Also, recovery logging time and the associated log space are not needed. This gives a performance boost for read-only operations. Unfortunately, this is a compile-time constraint. During Chrysalis development, I found a significant performance benefit in garbage collection (a factor of 6) by having a function that was initially read only (const), but could be upgraded to read/write (non-const) when it found that data needed to be changed. The garbage collector is a routine that looks for document table and posting list entries (described later) that are marked as deleted, and compacts these areas and frees up the unused space. Since this operation might not modify an item, it needed the flexibility and performance benefit of making this upgrade determination during run time. This upgrade capability was subsequently added to SHORE.

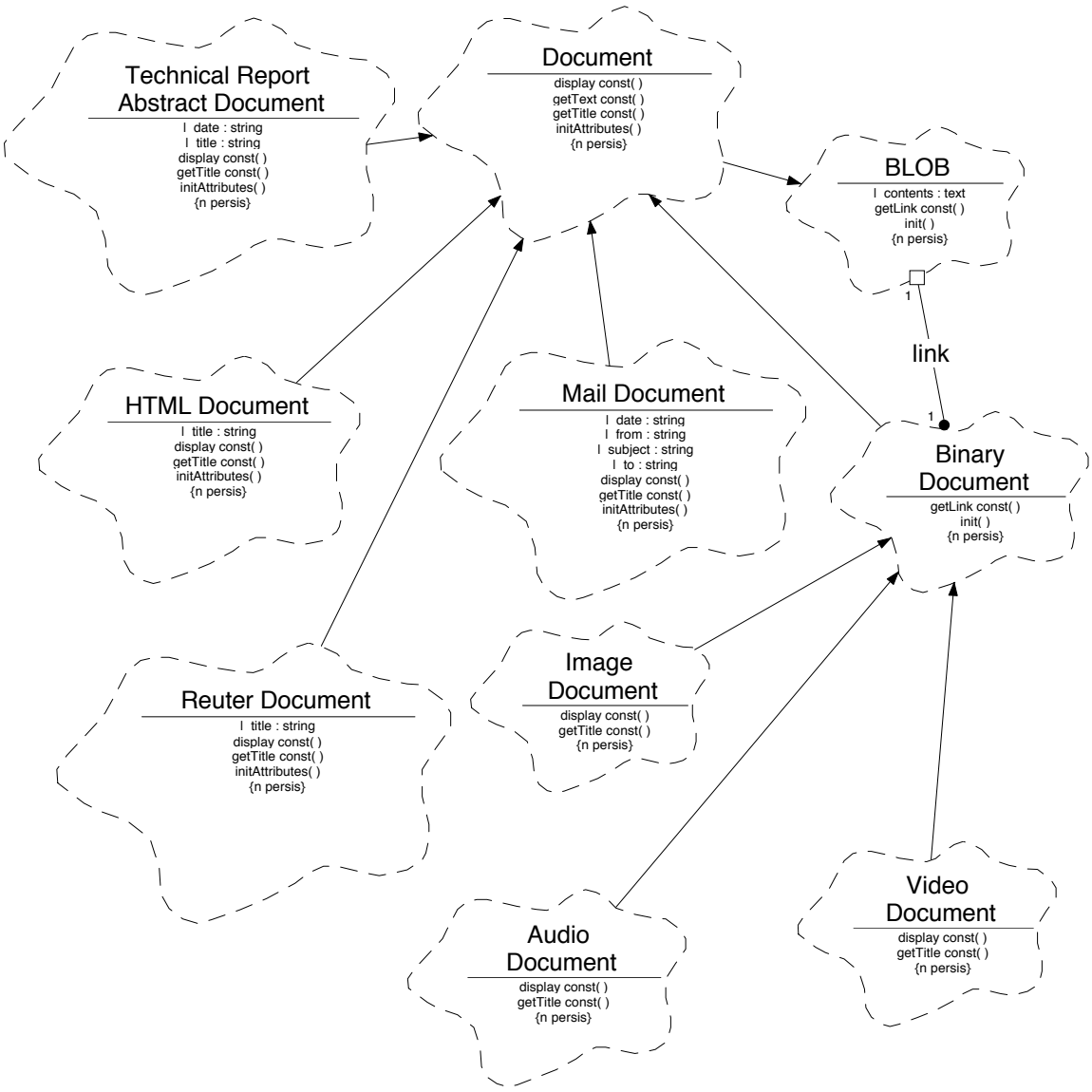

Figure 8: Document Hierarchy.

SHORE supports object inheritance hierarchies. Figure 8 shows the document hierarchy for Chrysalis. The diagrams use Booch notation [Boo94]. Briefly, the notation uses "clouds" to represent classes, "arrows" to represent inheritance, "vertical bars" to represent protected attributes, "n or 1" to represent how many instantiations of the class exist, and "persis" to represent persistent object instantiations. The link relationship has an open square at one end indicating containment by reference with the "1s" indicating cardinality. Each class lists both attributes (names and types) and methods.

Other document types are easily added by writing the interface in SDL and the methods in C++, inheriting from other existing objects where appropriate. Currently, a list of UNIX files tagged with a document type are used as input to a load utility which reads UNIX document files and creates document objects of the appropriate type. Chapter 5 discusses some possible improvements to this using an automatic document classifier (like Rufus uses) and triggers.

For example, a Mail object is defined by the SDL interface definition given in Figure 9. The Mail object inherits from the Document object, so it has the contents, author, and version attributes from its parents. In addition, there are additional attributes that store mail-unique items, like what the subject is, who it came from, to whom it was sent, and its date.

The initialization method (initAttribute) is overridden so that, in addition to initializing the contents attribute to the mail message, it also parses out the subject, from, to, and date fields from the mail message and initializes these

```
interface Mail : public Document {
public:
  override initAttribute;
  override display;
  override getTitle;
protected:
  attribute string subject;
  attribute string from;
  attribute string to;
  attribute string date;
};
```

Figure 9: SDL for Mail Object Definition.

attributes. The overridden display method displays the subject, from, to, and date attributes before the UNIX prompt is presented so that the user can get a better feel for the mail contents before further operating on it. The overridden title method returns the subject attribute contents for the one-line description given by Chrysalis during retrieval ranking. All totaled, the additions for Mail object support required only 30 lines of code and took less than an hour to program. No modifications were necessary to existing code.

Image document type support is a bit more complex. For images, there are actually two objects used to represent the information (see Figure 10). This was done because SHORE does not allow more than one attribute per object to be exposed as a UNIX file and because in some cases it is only desirable to transfer the text annotation and attributes without the much larger BLOB portion. The first object, of type Image, contains user-specified text (initialized

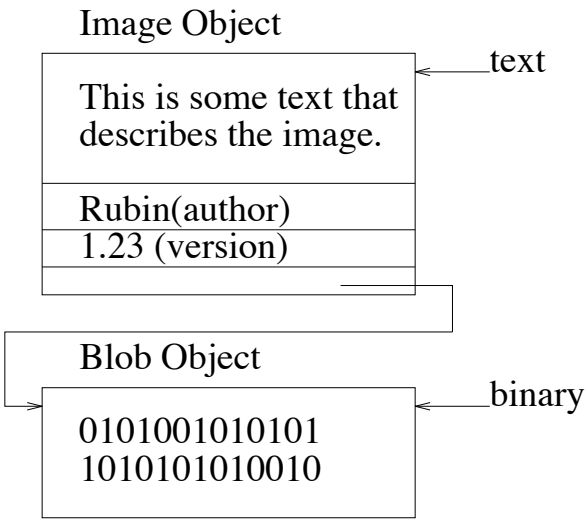

Figure 10: Object Structure for Binary Objects.

with the ".key" technique, defaulting to the filename) that describes the binary data. This part is indexed, and is bound to the filename "text" after retrieval. An additional attribute in the Image (BinaryObj) object, called docLink, points to the second object, of type BLOB. This object contains the binary image data in another text field, so it is also visible to UNIX. During retrieval, the filename `binary` is bound to the binary image data. After retrieval, a user can enter "cat text" at the UNIX shell prompt to display the description, and "xv binary" to display the image (xv is a UNIX utility that displays image files). The SDL for BinaryObj and Image objects are given in Figure 11.

These object structures provide a great deal of power. The UNIX paradigm that a file is a stream of bytes is preserved with the text attribute field, so all of the standard UNIX tools (vi, grep, mail, etc.) can be used on the object, and the objects can be viewed like any other UNIX file (using ls, du, etc.). Chrysalis

```
interface BinaryObj : public Document {
public:
  override initAttr;
  override getLink;
protected:
  attribute ref<Blob> docLink;
};

interface Image : public BinaryObj {
public:
  override display;
  override getTitle;
};
```

Figure 11: SDL for BinaryObj and Image Objects.

provides full text indexing of the text field contents to support vector-space, relevancy-ranked queries. Figure 1 in Chapter 1 showed a sample Chrysalis retrieval session dialog using a full-text query.

In addition, in principal, additional attributes in the document object could be searched with an Object Query Language (OQL). While an OQL query language implementation is not yet completed in SHORE, its syntax might look something like:

```
SELECT *
FROM DocumentCollection
WHERE TYPE = TechReport AND AUTHOR = ´Rubin´
```

The results of one query could provide a result set as input to an other query, so all the technical report documents relative to object oriented databases written

by Rubin could be found. Both Rufus and the Gold mailer allow this type of query.

3.3 Implementation Issues

In this section, I describe some details of the Chrysalis implementation and how this implementation interacts with and relies upon the functionality provided by SHORE. Chapter 2 discussed the information retrieval aspects and related work in greater depth.

3.3.1 Persistent object programming environment

Chrysalis uses the object store for most of its data structures (see Figure 12). In the following, the term "index" refers to the full-text, word index structures. I explored three different implementations of this index, the first using a hash table object, the second using a B+ tree of objects, and the third using a SHORE built-in B+ tree. The first configuration is shown in the figure.

Objects in SHORE have two areas, a "core" and a "heap". The core stores attribute data whose size is known at compile time. The heap stores data of dynamic size (like sequences, strings, and text). Pointers to heap items are stored in the object core. SHORE automatically adjusts the disk space for an object when it is flushed back out to disk. This makes the programming task easier than dealing with the pre-allocated, fixed-size objects used for posting

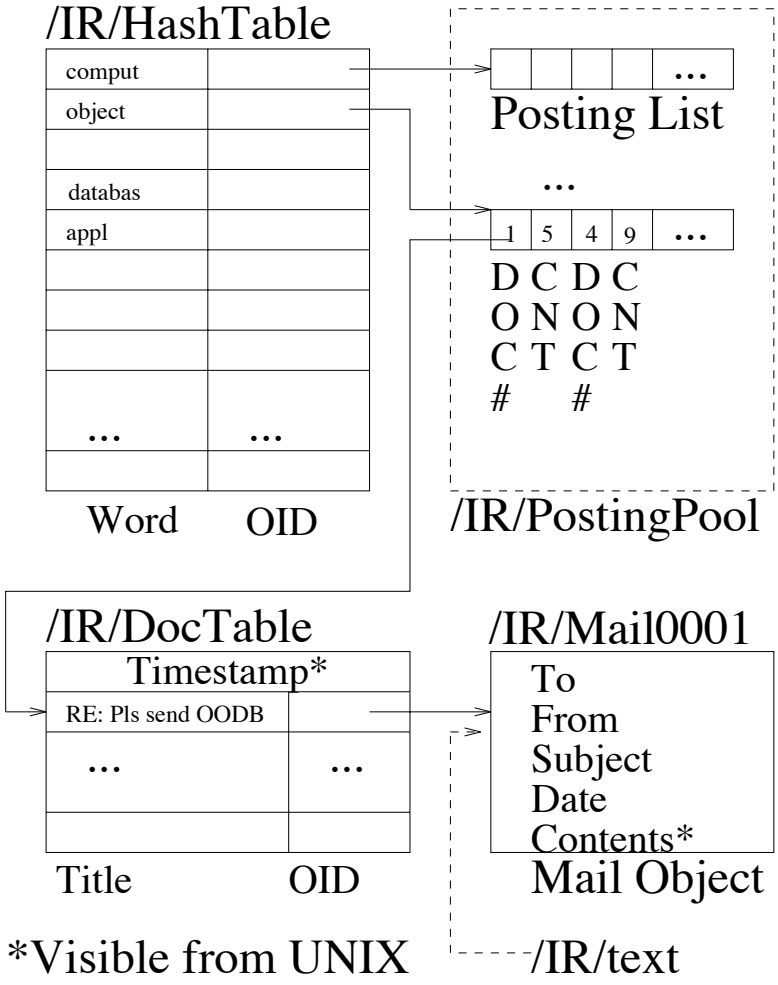

Figure 12: Chrysalis Object Structure.

lists in Rufus and INQUERY/Mneme. In these systems, one must manually select the next largest or smallest fixed-size objects, doing data copy and old object deletion at the application level when list sizes change.

In the object index version, there is an object called HashTable that has the path name /IR/HashTable in the SHORE name space. It stores the significant document words. Words are considered significant after passing through a stop list test and a Porter conflation algorithm. HashTable is implemented as an SDL sequence. Each sequence element consists of a word field that contains a 4-byte pointer to a word character string, which also resides in the object's heap area, and a pointer (OID) to a corresponding posting list object. These posting lists, also SDL sequences, have elements that contain a word count and a document number for each document where the hash table word appears.

Since there are many posting list objects, and there is no need to provide a name in the name space for these objects, I make use of the SHORE pool facility. The pool construct allows creation of a named object that contains a number of unnamed objects, making name space management easier and providing a clustering mechanism for related items on disk. One named (registered) pool object, called /IR/PostingPool, contains all of the unnamed (anonymous) posting list objects. Document handles (2-bytes) are used in the posting list objects as a level of indirection for OIDs (4-bytes) for storage efficiency and for implementing lazy deletion. There is another object, called DocTable, that maps these document numbers to OIDs. The DocTable also contains a Title

field, set by a method for each object type, that contains a character string to be displayed when showing query results. This gives the user some feel for the object contents without the overhead of touching the actual document object. This title caching feature was not, however, used in the experiments described in Chapter 4, so the impact of touching the document objects when the title is retrieved simulates a user not only retrieving a list of documents, but retrieving several documents of interest as well. A timestamp field records the time of the most recent index entry for determining if there are new documents that require indexing.

Document objects are stored as named (registered) objects. This allows both associative access to the documents, via full-text queries, as well as browsing of the document name space. Access via path names is essential for supporting formats like HTML, because HTML hyperlinks (URLs) contain path name references.

The persistent programming environment provided by SHORE considerably eases the tasks of index creation and update. Pointers in data structures are automatically swizzled by SHORE and automatically moved between disk and memory on demand, so code to flatten/unflatten data structures is not needed. For example, adding a document to an existing index begins with adding the document to the DocTable object. Next, each document word is looked up in the HashTable object. The OID pointer in the hash table entry points to the posting list object for a word (assuming it already exists), and it is updated

with the new count/document handle information. All needed structures are demand paged from the SHORE server disk to a client object cache and then, if modified, forced back to disk at end of the transaction. SHORE supports both automatic object cache space management as well as a degree of manual cache control (via commands like fetch and flush).

3.3.2 Name space/object location transparency

Traditional file-based document indices store filenames in the index. This presents a problem when the file is renamed, because the index now points to a file that no longer exists; the renamed file is treated as a new file and is re-indexed, even though the contents did not change. In object-oriented terminology, we need object identity in order to deal with this problem.

Some systems, like Rufus, create an object identifier from the document file contents to get object identity capability. This allows Rufus to identify an object and to avoid re-indexing it even when it is moved in the name space. There are cases, however, where there is no intrinsic unique identifier, so Rufus uses the UNIX filename for object identification. In these cases, filename identity and object identity must remain in sync, or else renaming a document file will cause the index and the document to become inconsistent.

The SHORE name space/OID correspondence provides advantages for information location transparency. None of the Chrysalis data structures contain path names of the information objects, as they contain only the OIDs. Since

each object has a unique OID, which does not change as files are renamed, the index always points to the object. Chrysalis uses the SHORE cross reference facility to temporarily bind a cross reference path name (e.g., "text") to an OID so that it can be used by UNIX tools via the NFS mount facility. Since OIDs are never reused, if an OID in the index is dangling, the application will be able to discover that the object has been deleted.

3.3.3 Transactions and concurrency control

During document indexing, it is very important to ensure consistency between all of the index structures. This is accomplished with the SHORE transaction facility. A set of documents is processed in a single transaction. If an error occurs at any point during indexing, the transaction can be aborted and the previous index structures will remain intact. The transaction will commit only when all processing is successful. So, documents are either fully indexed or left un-indexed. Neither Rufus nor INQUERY/Mneme provide transaction semantics for both the document files and the meta data, so the two can get out of sync in the event of failure.

SHORE provides implicit locking, allowing multiple users to access the same data structures without data integrity problems. This allows multiple users to read the index concurrently with a shared read lock; an index update process will exclusive write lock the index during modification, causing readers to wait until the write lock is released.

SHORE does not support inter-transaction caching of objects in the object cache, but objects in the server page cache are maintained across transactions. This means that all objects in the object cache are invalidated at the end of a transaction. Thus, even if the object is not needed by another user, and is not modified, it will be re-read into the object cache by a new transaction that needs it. Chapter 4 discusses the performance impacts of this aspect of SHORE. Some other OODBMS's do implement inter-transaction caching. ObjectStore [Lam91] implements this using a callback locking consistency protocol, which lets the server recall page locks associated with an object if another process requests the object for modification. The virtual memory page of stale objects is marked as protected so attempted accesses will be detected by the memory management hardware and software. Objectivity supports inter-transaction caching using a locking technique and a check on access scheme based on timestamps to maintain consistency. The timestamp operations are piggy-backed on lock/unlock requests, but these lock requests may be as expensive as re-fetching the objects. Chapter 5 discusses a future SHORE solution for inter-transaction using a SHORE server and associated page cache in a client.

SHORE shares a restriction on cache architectures with other OODBMS's. A private client object cache is provided for every transaction, so only a single transaction at a time can run against this cache; objects are replicated in SHORE object caches when referenced by multiple transactions. I have not found an example of a system that allows multiple transactions to concurrently access a

single client object cache. Chapter 5 discusses the implications of providing such a function.

3.3.4 Indexing algorithms

This section describes how indexing of documents is performed. New, changed, and deleted objects are always detected, and index updates are incremental and atomic.

Users occasionally generate a file that contains a list of UNIX files that they want indexed, along with a numeric type identifier that specifies an object type for each file. This is usually done with a shell script that processes specific file types (see Chapter 5 for an improvement on this). A load utility processes this list by creating SHORE objects of the appropriate type. SHORE automatically assigns the unique OID. The object field contents and any type-specific attributes are also initialized at this time using the object specific initialization method.

In addition to user-requested document indexing, Chrysalis uses SHORE's UNIX-like system calls to periodically generate a list of SHORE objects and their modification times. It then compares these modification times with a timestamp that is stored in the DocTable structure. The timestamp represents the youngest indexed document. This results in a list of new objects that may need to be indexed (or re-indexed). Chrysalis next makes sure that each new object is of an indexable type by using a SHORE "isa" function to see if it is in the Document inheritance hierarchy. If it is a new, indexable object, it is added to the index.

All of the new object processing occurs in a transaction, so if a failure occurs, the timestamp is restored to its old value and the indexing structures are restored to their previous, consistent state. Thus, index updates are atomic.

In the Gold mailer system, the designers implemented a lazy insertion technique where insertions that update the disk data structures are first inserted into a memory structure called an overflow list. The disk operations for an overflow list are batched together to minimize disk I/O. SHORE provides a similar performance gain with its built-in object cache— but without requiring explicit application level code— while also automatically handling the concurrency control and recovery requirements.

One further step is added to the above algorithm to detect document modifications. Before a document is indexed, its OID is checked against all of the document OIDs in the DocTable data structure. If the OID already exists, then the new document is actually a modification of an existing document. In this case, the old index information for the document is deleted by marking its DocTable entry as invalid, which allows immediate deletion with lazy garbage collection of the posting list and DocTable entries (done with a separate application program at a convenient time). This lazy deletion scheme allows both faster processing of modified files and greater efficiency because the garbage collector need only make a single pass through all posting list objects, instead of the multiple passes that would be needed if each deletion were handled in real time. The document can now be re-indexed.

If a document is deleted from SHORE, its index information will point to an object that no longer exists. This can be detected with a SHORE function called "valid" that determines if an associated object exists for a given OID. This operation does not require the object itself to be retrieved. If a user requests a document that no longer exists, they are told so, and the index entry is then deleted. During garbage collection (an application that can be run at any time), all the OIDs are checked for validity and cleaned if needed. Systems without unique, never-reused identifiers, like Rufus, cannot detect deleted documents until the corresponding file retrieval is attempted.

3.3.5 Hash table vs. B+ tree for word index

There are three word index implementations in Chrysalis. The first uses a hash table (implemented as a SHORE registered object), the second uses a B+ tree of SHORE objects, and the third uses the SHORE server's B+ tree support. Chapter 1 discussed the architectural aspects of the client versus server choice. This section discusses the implementation details and characteristics of each choice, and Chapter 4 will discuss the performance tradeoffs associated with each choice.

The SHORE server provides support for a B+ tree [Com79] index. This B+ tree is used by Chrysalis to map words to posting lists. In this application, the word is the index key and the index leaves contain OIDs that point to posting list objects. Each page in this B+ tree index is 8,192 bytes. Subtracting header

space, there are 8,100 bytes of usable space. When used to support the word index in Chrysalis, each entry for the internal nodes requires (physically) 4 bytes for slot information, 10 bytes (average) for the word key, 4 bytes for a pointer to the page at the next lower level of the tree, and 2 bytes for alignment, giving a total of 20 bytes per entry. Thus, the internal nodes can support 406 keys and 407 pointers when full. Each leaf node requires 4 bytes for slot information, 2 bytes for key size, 2 bytes for the value (i.e. OID) size, 10 bytes (average) for the key, 4 bytes for the OID value itself (a pointer to the anonymous object posting list), and 2 bytes for alignment, giving a total of 24 bytes per entry. This allows 337 keys per leaf node when full. Using a 69% average efficiency assumption [Knu73], this changes to 233 values on average. In the technical reports collection to be described in Chapter 4, the 512 documents contain 4,177 unique words after parsing, stop list processing, and conflation. This requires a B+ tree of 19 pages, or 155,648 bytes (18 leaf nodes, 1 root node). In actuality, SHORE implements suffix compression on keys in internal nodes, but this does not affect this calculation because there is only one internal node (the root) and it is not full.

As an alternative to using a SHORE B+ tree index, the full-text index can also be implemented as a regular SHORE object. When choosing an internal data structure for the index implementation in this case, several factors were considered. First, the number of unique words in a document collection follows a distribution where in the early stages of indexing, many words are found that

have not been indexed. This increase in new words then slows, so in the latter stages of indexing, fewer new words are found; the rate of growth for the index thus also slows. Figure 20 shows the growth rate for one of the document collections used in the experiments of Chapter 4. Second, there is a general ceiling on the number of words in a document collection. For example, most dictionaries contain on the order of 100,000 entries. Since Chrysalis indexes stems of words, this bound is somewhat smaller. Whereas a fixed sized data structure is generally not appropriate for indices, these distribution characteristics of document collections make it more suitable. I chose a hash table, using double hashing [Knu73]. Double hashing uses a table that has a prime number of slots. A first hash is generated from the key (word in this case), normalized to the number of slots in the hash table. If this slot is not the desired one (for searching) or not empty (for insertion), a second normalized hash is generated and successively added to the slot pointer (modulo the table size) until the correct slot is found. In addition, to allow data structure growth, the hash table doubles in size (and increased by one in size until it contains a prime number of slots) when the load factor exceeds 70%, so the load factor varies between 28% and 70%. Each hash table entry contains a 4-byte pointer to the word in the object's heap, a 4-byte word length, and a 4-byte OID. For the 512 document workload discussed in Chapter 4, the hash table has 6421 slots, yielding 77,052 bytes. In addition, the words in the heap occupy an additional 41,760 bytes for the workload with 512 documents, giving a total of 118,712 bytes.

I also experimented with a B+ tree application-level SHORE object index. I started with a code base from Jannink [Jan95] and added support for string keys and OID values, along with converting the B+ tree class definition to IDL. Each node is represented by an anonymous SHORE object. The node size was set to about one page (8,192 bytes). Each B+ tree node consists of a set of key/value pairs. The key consists of a 4-byte pointer to the word in the object's heap, a 4-byte word length, and a 10-byte (average) word. The value is a 4-byte OID. So, each leaf node contains 8,192/(4+4+10+4) = 372 keys. With a 69% efficiency assumption, this changes to 256 keys. This requires a B+ tree of 18 pages (17 leaf nodes, 1 root node), or 147,456 bytes for the technical reports document collection. Chapter 4 discusses the performance differences between this B+ tree object index and the hash table object index.

3.4 Functionality Conclusions

When implementing an information retrieval system, one must make decisions about where to put the document data and where to put the meta data. Object-oriented databases are attractive for the meta data because they can model the rich semantics of different document types and their interrelationships. File systems are attractive for the document data because of the need to preserve the unstructured byte stream semantics of documents for use by existing UNIX applications. SHORE combines these two concepts, and Chrysalis uses this

paradigm to store both the document data and the meta data in the same repository.

This merged file system object-oriented database capability provides clean solutions to problems that are awkwardly addressed by previous information retrieval systems. The Chrysalis prototype demonstrates the ease of creating SHORE applications that are both functional and robust. SHORE provides many built-in facilities that previous information retrieval applications either did not support or had to build themselves; it provides an integrated UNIX-like file system, persistent objects, transactions, concurrency control, and object caching. While we have seen that SHORE provides a good match to information retrieval application requirements, there are several functional and performance shortcomings. These will be discussed in Chapters 4 and 5.

Chapter 4

Experiments and Results

This chapter discusses the experimental setup, tests performed, and results obtained that highlight tradeoffs in both index placement and information retrieval function placement. Three different workloads and two different document collections are used to study the performance of several different Chrysalis system configurations. In this chapter, the information retrieval performance metric is response time, not the other information retrieval performance metrics described in Section 2.1.3.

4.1 Experimental Setup

All experiments were performed on a Sun Sparc 10/40 with 32 Megabytes of main memory running SUN/OS 4.1.3. The SHORE code used was a post-beta release (without debug code) of 9/15/95. All of the SHORE default parameters

were used except for two items. First, the SHORE server page cache size was increased from 320 Kilobytes to 1 Megabyte so more of the objects referenced in a transaction could remain resident in server memory. Second, the pre-fetching of anonymous objects was turned off for the "Query" workload. When an anonymous object is transfered from server to client, the entire page is transfered and held in a buffer in the client. With pre-fetching, the object cache is filled with any other objects on that page. Without pre-fetching, the object cache is not populated by these adjacent objects, although if one is referenced, it is retrieved from this client buffer. In Chrysalis, adjacent posting list objects have no correlation to each other, so turning off this feature saves unneeded object fetches into the object cache. The object cache size was 2 Megabytes. The shared memory size, used for the transfer of data between client and server, was 128 Kilobytes. Although the caches may seem small, the memory load is substantial with multiple clients and a server on the same machine. Experiments were also performed with larger object cache and page cache sizes.

The workstation had three disks, each using a Unix file system (not a raw device). The SHORE data volume resided on the first, the SHORE log resided on the second, and the executables and UNIX source files resided on the third. These three classes of I/O were targeted to separate disks to not only allow for I/O parallelism, but also to separate the random pattern of data volume I/O from the sequential pattern of the log I/O.

One test document collection consisted of 512 technical report abstracts from

Parameter	*Value*
Number of documents	512
Total bytes	762,387
Average bytes/document	1,489
Standard deviation	645
Smallest document size	245
Largest document size	4,144

Table 1: Technical Reports Document Collection Statistics

Parameter	*Value*
Number of documents	16,384
Total bytes	16,452,483
Average bytes/document	1,004
Standard deviation	944
Smallest document size	26
Largest document size	13,772

Table 2: Reuters Document Collection Statistics

the University of Wisconsin-Madison computer science technical report FTP site. Table 1 characterizes this workload. The second test document collection consisted of a standard collection of 22,173 Reuters news articles from 1987. This collection, known as the Reuters-22173 text categorization test collection, is about 20 Megabytes of text data and is available via anonymous FTP from ciir-ftp-cs.umass.edu/pub/reuters1. Table 2 shows the statistics for the first 16,384 documents in this collection. To help limit the log size, an intermediate checkpoint using the SHORE transaction chaining feature was taken after every 512 documents were processed during document loading and index building.

There is an interaction between the operating system disk cache and my application. For example, a UNIX "update" process runs every 30 seconds to

flush the cache to disk, and this causes a burst of real I/O. To compensate for these effects, I used SHORE statistics for RPC counts as well as data volume read/write I/O and log I/O counts in the experimental results. Based on some previous performance analysis work, RPC counts are a good indicator of server CPU load and data volume and log I/O counts are a good indicator of server I/O load. The RPCs counted are actual ones. These are, in general, fewer than the number of RPC requests made because SHORE performs batching of RPCs. Some examples of RPCs in this environment include "mkregistered" (registers a registered object in the name space), "mkanonymous" (writes out a new anonymous object), "update" (updates an existing object), and "read" (reads an existing object).

The number of I/O bytes come from two main areas. SHORE performs data volume writes when the server page cache becomes full, and data volume reads at initial reference and whenever some of the data that was removed from the cache (either written or tossed) is re-referenced. If the SHORE page cache is large enough, there may not be any need for the re-referenced type of volume reads. SHORE also performs logging and keeps track of the log bytes written. The number of I/O bytes counted in these experiments is the sum of data volume writes, data volume reads, and log writes. Operating system disk caching may reduce some of the physical disk delays from these reads and writes.

I made several runs using a Unix raw disk device to verify that the SHORE I/O statistics reflected the actual I/O generated by SHORE. I also found that

the UNIX iostats function did not yield a reliable, repeatable picture of my application I/O activity and found the documentation for what this function really measures to be very poor, so I used the SHORE statistics exclusively.

Having both client and server execute on a single workstation provided a constant resource environment, so implementation tradeoffs could be better compared. The single workstation environment also allowed me to explore cases where real I/O must be done. For example, the multi-user query workload using the Reuters collection had a memory demand that exceeded available memory, so the impact of real I/O could be observed.

All of the execution time tests were performed enough times to give a confidence interval of at least $\pm$ 5% at a 95% confidence level, except for the multiple user query results which are within $\pm$ 10%. Variations occured because of other miscellaneous tasks running on the workstation. The number of documents were increased by powers of two. The number of concurrent querying users was varied from 1 to 10 in increments of one.

There were three main Chrysalis functions used in these experiments. The program "Load" reads the input documents from UNIX files and creates registered objects from them. Index structures are also created, but no indexing is performed. The program "Build" builds an index from the registered object documents. The program "Query" queries the document index and returns a relevancy-ranked list of document objects. Some documents are then selected for retrieval.

To automate the query process and to simulate a multi-user environment, a query script was used by all query users (see Figure 13). A common query script was used for all users with the Technical Reports collection, while both a common query script as well as a unique query script per user were used for the Reuters collection. The script has no think time because the goal was to aggressively drive the query function to uncover its behavior, not to find real limits to the number of concurrent users. These two types of scripts explored the extremes of all users querying using the same words and retrieving the same documents, and each user querying using different words and retrieving different documents. A query test driver program forks off a process for each simulated concurrent user. All documents were loaded in a single transaction. All documents were also indexed in a single transaction. The query session for a given user was done using both a single transaction (one for all queries) and multiple transactions (one for each query).

Two different document index architectures were used in these tests. For the first architecture, two implementations were explored. For the first implementation, the index is implemented as a hash table (using double hashing) where the hash table is a registered object that resides in the object cache. The hash table starts out with 1597 slots (enough for the first 41 documents in the workload). To double this in size, the entries are rehashed into a transient hash table object which then replaces the old persistent hash table when the rehashing is complete. In addition to the hash table index, most experiments were also performed with

```
database  // look for ``database´´
0         // select document 0
1         // select document 1
2         // select document 2
3         // select document 3
4         // select document 4
q         // new query
object    // look for ``object´´
0
1
2
3
4
q
object oriented  // look for ``object oriented´´
0
1
2
3
4
q
object oriented database // look for ``object oriented database´´
0
1
2
3
4
x         // quit query session
```

Figure 13: Query Script

a B+ tree object index as well. This tree used 8,192-byte anonymous objects as nodes. The second architecture used the built-in SHORE B+ tree index. This index resides in the server page cache. All of these indices were described in greater detail in Section 3.3.5.

Two different versions of SHORE were used in these tests. In the first, the default, or Generic Server (GS) was used. In the second, a custom Information Retrieval Server (IRS) was used. The IRS server was built by taking the Chrysalis client code and moving it into the server, which proved to be a very easy process. The main program was changed to a procedure, and a server command was added to execute this procedure. The multi-threaded query program required special care to eliminate global variables.

The IRS object caches, one for each thread of execution, also reside in the server. Each object cache is, like the GS version, 2 MBytes. There is also, like the GS version, one object cache per thread of execution. For the Load and Build workloads, there is one active object cache because there is one thread of execution. For the Query workload, there is only one object cache per concurrent query user. Making this shift in function removes the communication overhead of the RPCs and reduces the number of processes from two to one during most of the activity.

In summary, I explored twenty-four base configurations. Three different functions were used (Load, Build, Query), two different index architectures were used (object index, built-in server index), two different server architectures were

used (GS, IRS), and two different document collections were used (Technical Reports and Reuters). In addition, two different Object Index implementations were explored (B+ tree and hash table) and several posting list implementations were studied.

In the graphs, the following abbreviations are used:

- GS-OI Generic Server, Object Index
- GS-SI Generic Server, Server Index
- IRS-OI Information Retrieval Server, Object Index
- IRS-SI Information Retrieval Server, Server Index

The hash table and B+ tree Object Index cases are not distinguished in the graphs because there is no statistically significant difference in execution time. This occurs because there are only slightly more RPCs needed for the B+ tree (because multiple node objects must be retrieved) and the sizes of the two indices are comparable, so data volume and log I/O are also comparable. The other components of the Build workload as well as SHORE overhead dominate over the index choice. Unless otherwise noted, Object Index (OI) refers to both the hash table and B+ tree implementations.

4.2 Experimental Results-Load

Figure 14 shows how long (real time) the Load function took to complete for a varying number of documents in the four different environments for the Technical Reports collection. For the Generic Server case, the execution time is linear with the number of documents in all cases. Two main results are apparent. With

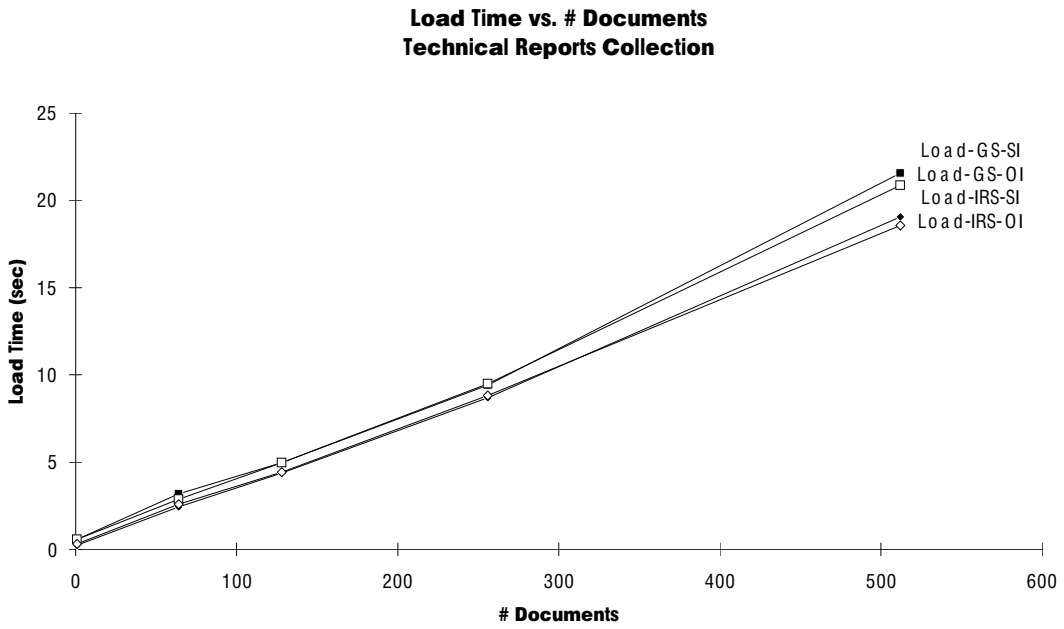

Figure 14: Load Time-Technical Reports

the index type held constant, the Information Retrieval Server always runs Load faster than the Generic Server implementation.

Figure 15 shows a comparison of the number of RPCs needed in the Generic Server implementation for both the Object Index and Server Index cases. The number of RPCs are virtually identical. There are 17 RPCs required when loading one document. This grows approximately linearly to 533 RPCs when loading 512 documents because each additional document means an additional non-batched "mkregistered" RPC, which registers the document's registered object in the SHORE name space. The Server Index version, when compared with the Object Index version, requires only an additional "addindex" RPC, which creates the server B+ tree and eliminates a "mkregistered" RPC for the

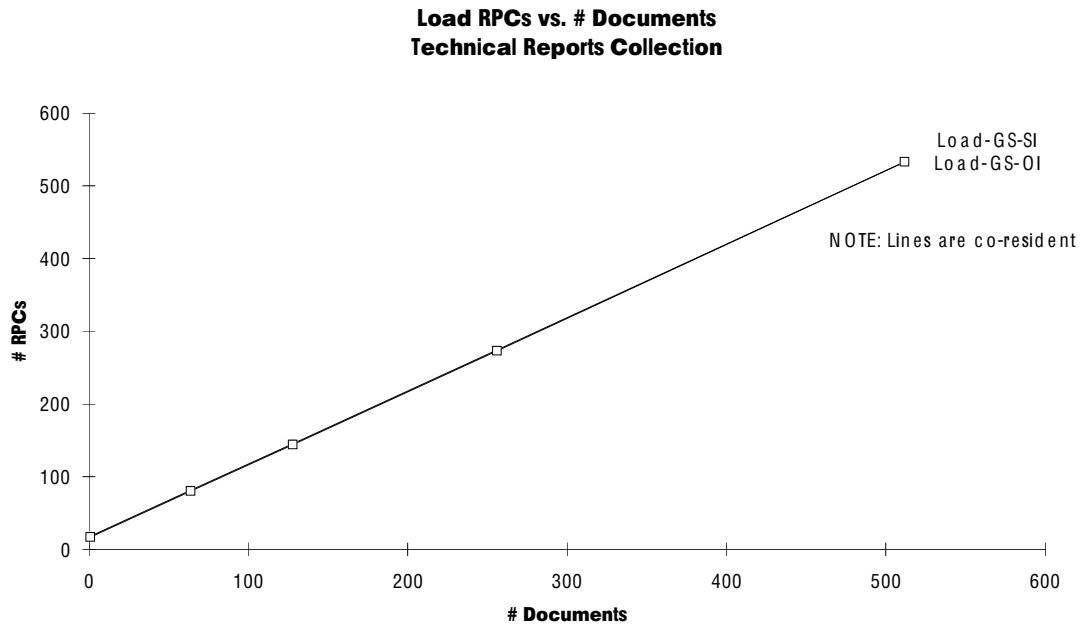

Figure 15: Load RPCs-Technical Reports

hash table registered object (or a "mkanonymous" RPC for the B+ tree root anonymous object). RPC counts are not shown for the Information Retrieval Server because the SHORE RPC communication overhead is eliminated and the RPCs are transformed into regular procedure calls.

Figure 16 shows a comparison of the number of I/O bytes generated for the Generic Server Implementation for both the Object Index and Server Index cases. The curves are almost identical because the same document objects are being created in both cases. The indices, although different, are only being initialized and not loaded. The IRS Client and Server Index cases are not shown because they are identical to their corresponding GS Client and Server Index versions. The same changes are being made to the data, so the logging and volume activity

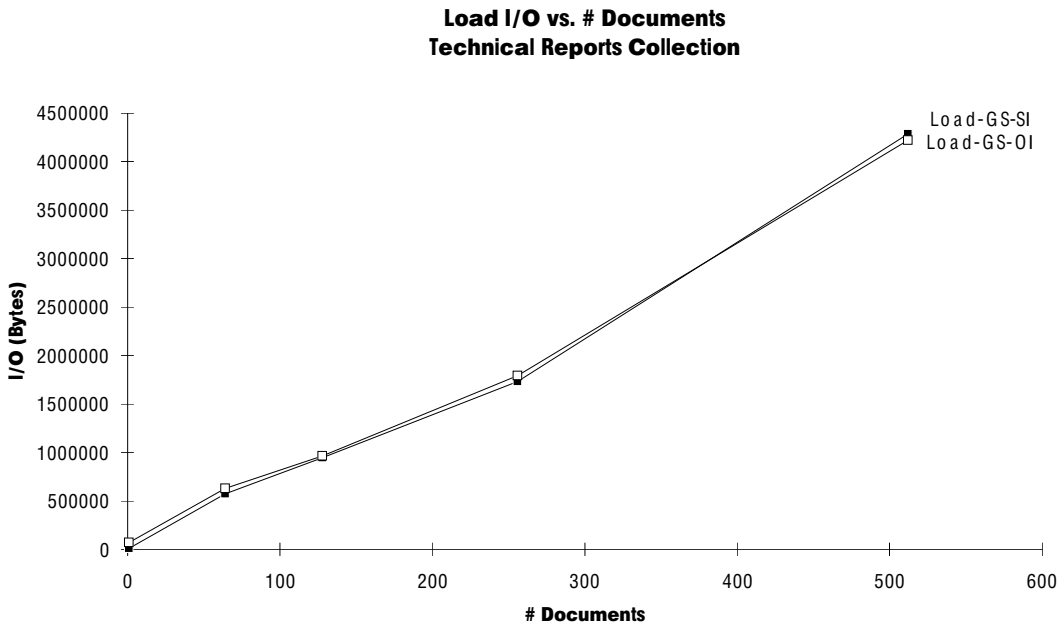

Figure 16: Load I/O-Technical Reports

is the same. Only the RPC overhead is eliminated.

Figure 17 shows that Load, for the Reuters collection, is linear with the number of documents, as the previous experiments showed. So, Load scales well.

4.3 Conclusion-Load

For the Generic Server version, the Server Index and Object Index versions result in similar execution times. The number of RPCs increases linearly with the number of loaded documents and the I/O increases linearly with the size of the document collection (which, in these collections, is roughly linear in the number of loaded documents), so the net performance curve is about linear in

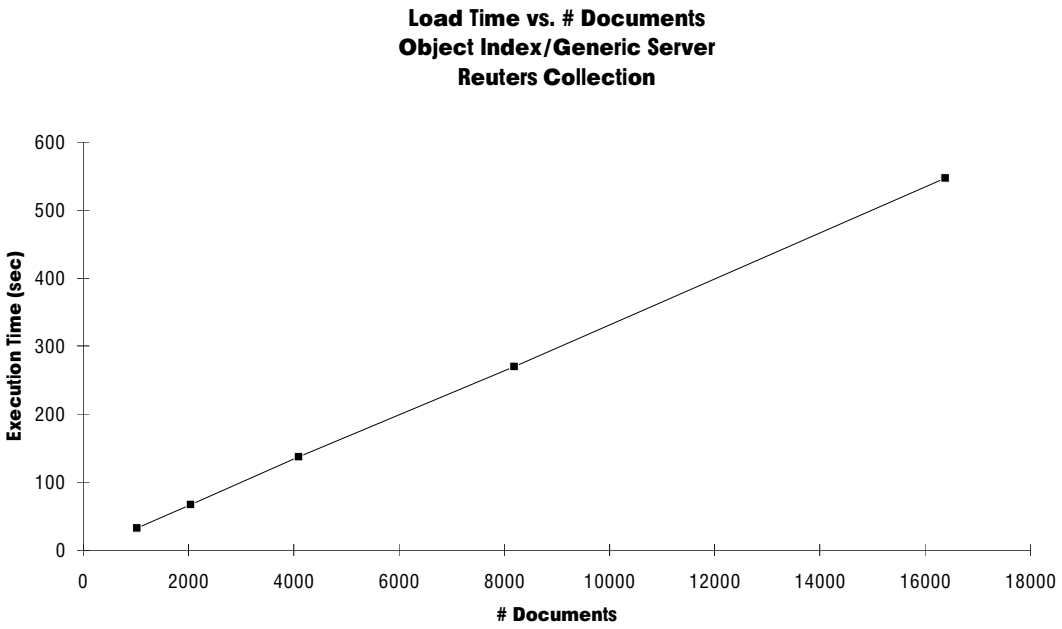

Figure 17: Load Time-Reuters

the number of documents. For the Information Retrieval Server version, there are no RPCs required at all. This decreases the CPU demand, resulting in a lower execution time for both index configurations. Load scales well too. Note that a real-world version of Load might run with logging turned off, since the source data (UNIX files) would remain intact in case the load operation failed and had to be restarted. This restart could also be done incrementally.

4.4 Experimental Results-Build

Figure 18 shows how long the Build function took to complete for a varying number of documents in the four different environments with the Technical Reports document collection. The execution time is linear in the number of documents

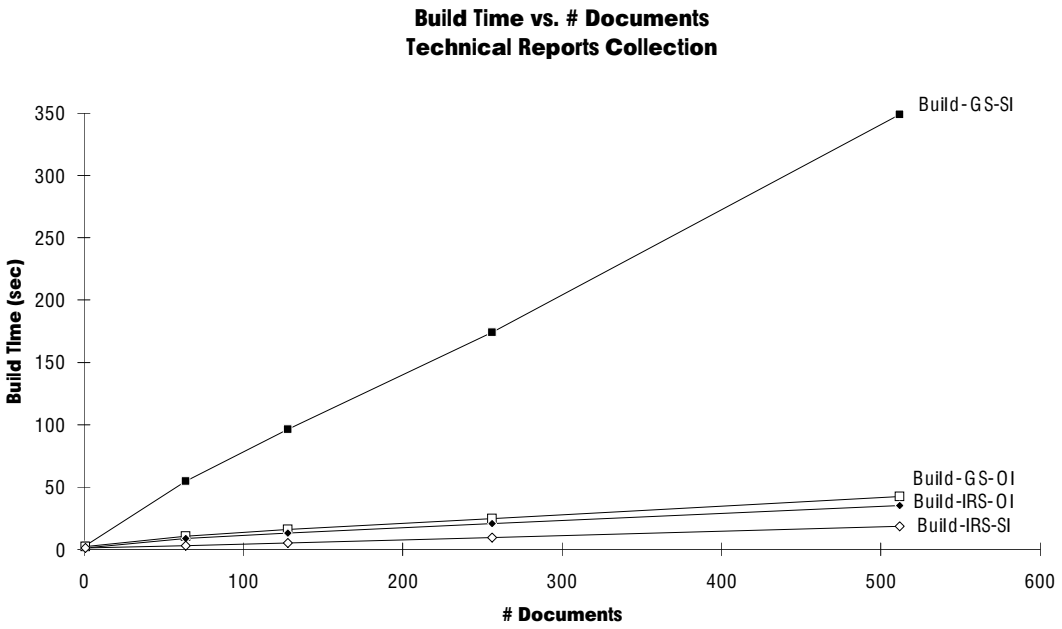

Figure 18: Build Time-Technical Reports

for all cases. The Server Index with the Generic Server configuration has a much worse execution time than the other cases. With the index type held constant, the Information Retrieval Server always runs Build faster than the Generic Server implementation. The Object Index performance is about the same whether in the client or in the server.

Figure 19 shows a comparison of the number of RPCs needed in the Generic Server implementation for both the Object Index and Server Index cases. The number of RPCs required differs dramatically between the two. Going from indexing one to 512 documents in the Object Index case requires an increase in RPCs from 30 to 2,086, while the Server Index case requires an increase in RPCs from 191 to 64,059! There are a few reasons for the majority of the

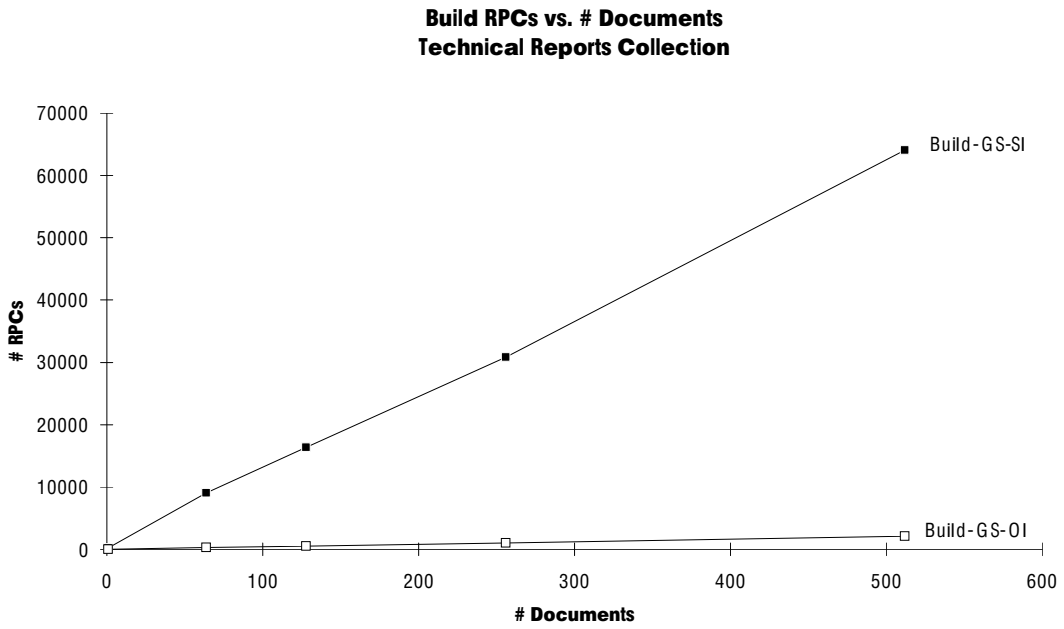

Figure 19: Build RPCs-Technical Reports

difference. In the Object Index case, indexing involves looking up each word in the object structure (one object for the hash table case, several for the B+ tree case) residing in the object cache. There is no RPC overhead for doing this lookup. The Server Index case, however, requires a "find" RPC for each word indexed. Furthermore, if a word needs to be added to the index, an additional "insert" RPC is needed in the Server Index case, while this is an in-memory operation in the Object Index case.

The number of RPCs grows about linearly with the number of documents in the Object Index case, while it grows about linearly with the number of words in the Server Index case (which is about equivalent to a high-constant times the number of documents with this collection). This assumes that after a number of

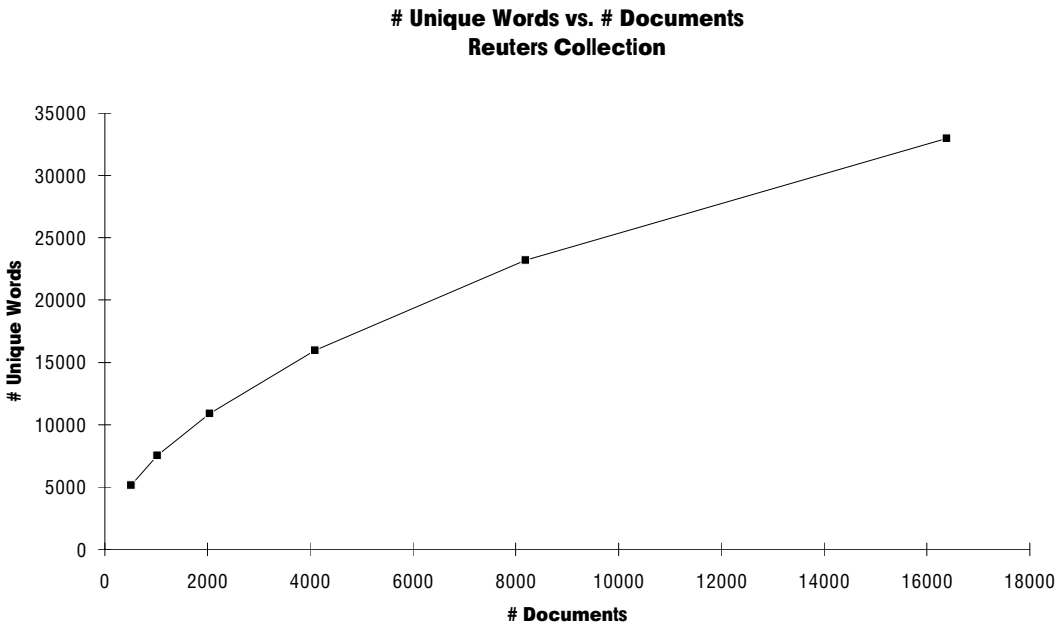

Figure 20: New Word Growth

documents have been indexed, the number of new words grows slower than the number of already indexed words, so fewer and fewer "insert" RPCs are needed. Figure 20 shows the number of unique words, for varying numbers of documents, with the Reuters document collection. It illustrates this slowing growth of new words as the number of documents is increased. So, for the Server Index case, the "find" RPCs dominate over the "insert" RPCs for a reasonably large document collection, so the RPCs are about proportional to the number of words in the document collection. RPC counts are not shown for the Information Retrieval Server because the RPC overhead is eliminated (although the same functions must still be performed).

Figure 21 shows a comparison of the number of I/O bytes generated for the

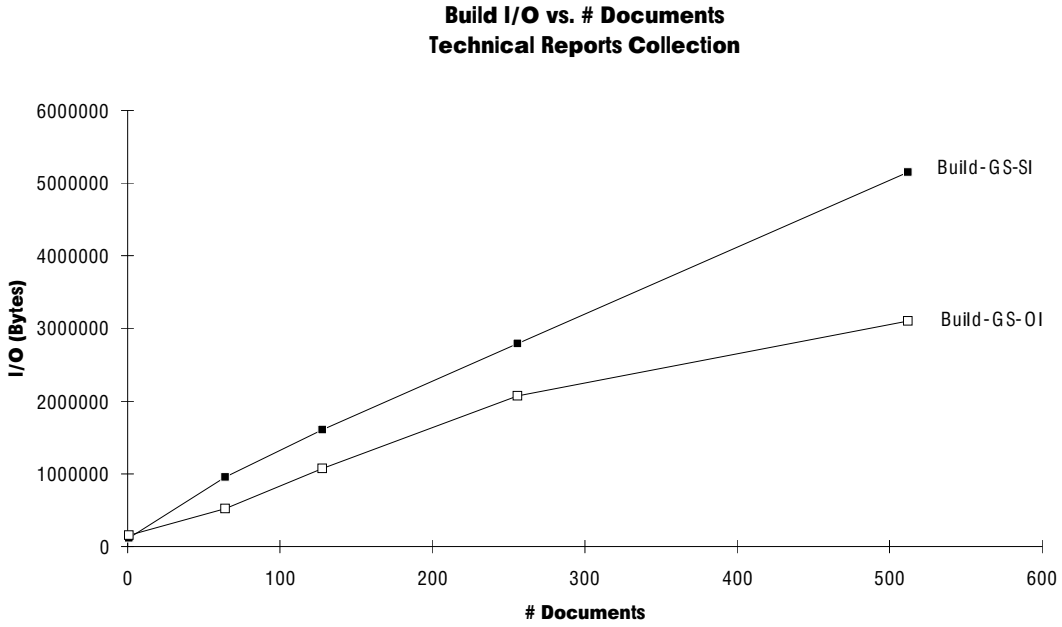

Figure 21: Build I/O-Technical Reports

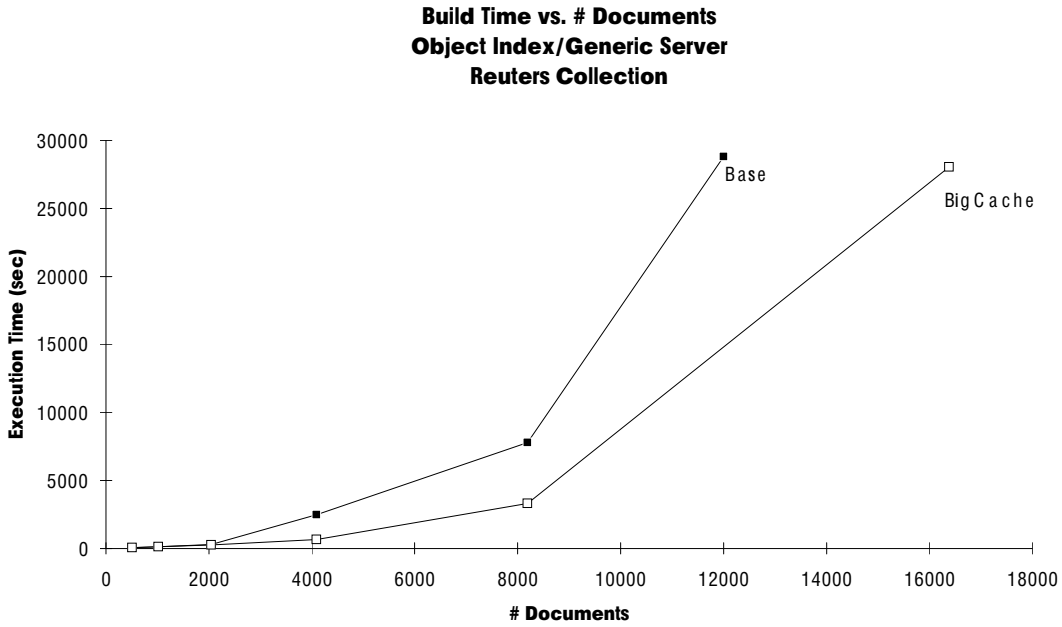

Figure 22: Build Execution Time-Reuters

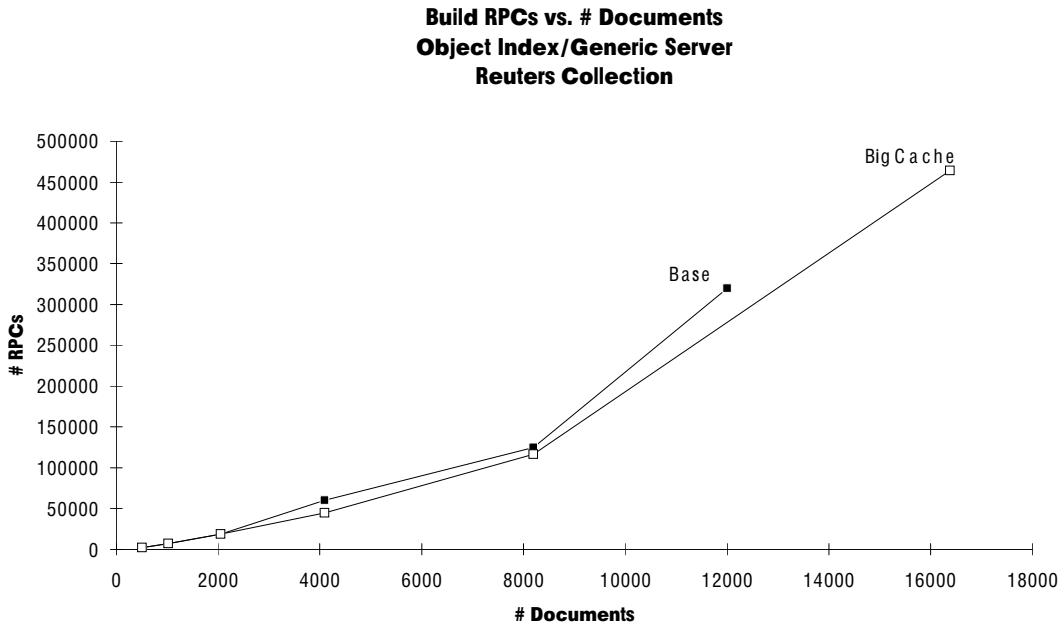

Figure 23: Build RPCs-Reuters

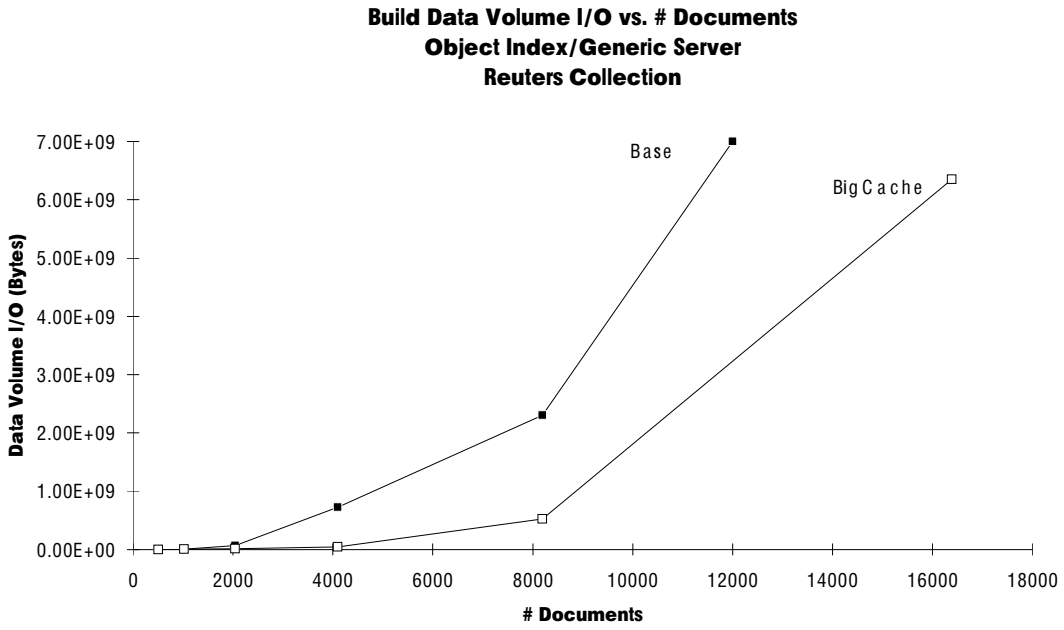

Figure 24: Build Volume I/O-Reuters

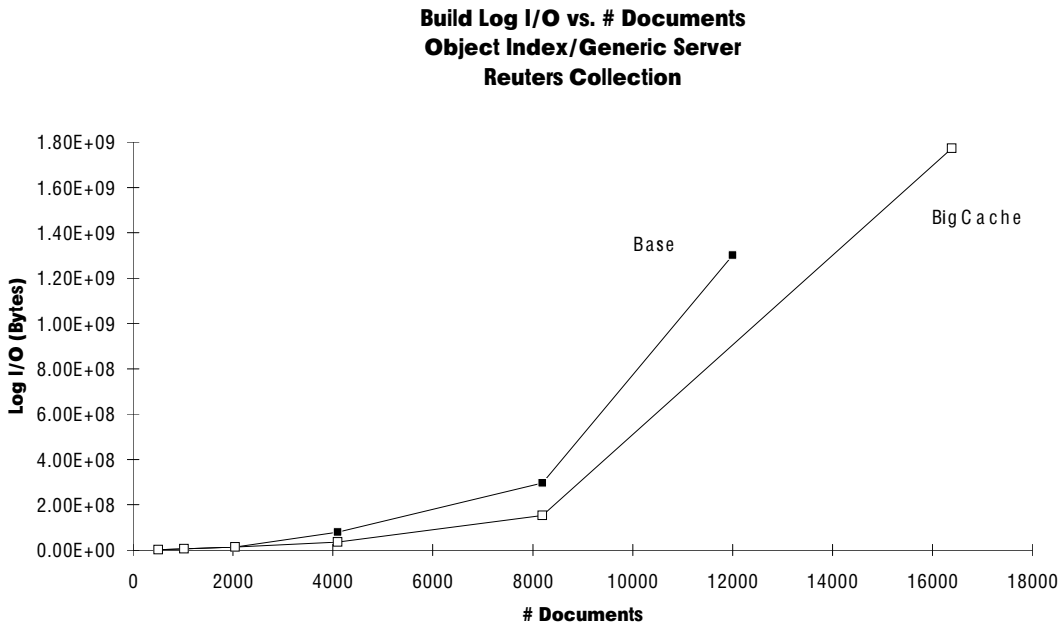

Figure 25: Build Log I/O-Reuters

Generic Server implementation for both the Object Index and Server Index cases. In the Object Index case, dirty data volume pages are only dirtied at the end of the Build transaction, so are only available for writes at that time. At that time, the object (or objects for the B+ tree case) is moved from the object cache to the page cache and the dirty pages can be written to disk. In contrast, the Server Index B+ tree is constantly locked and unlocked, and since it resides in the page cache, is available for dirty page volume writes many times during the transaction because SHORE uses a no force/steal [HR83] buffer management policy. Since the process of building a B+ tree dirties pages multiple times, multiple volume writes occur because SHORE greedily writes dirty pages during free I/O cycles. This increase in volume writes as the number of indexed documents is increased

results in a higher I/O amount for the Server Index case. However, since this extra I/O is only performed by the SHORE server when there are free I/O cycles, so this extra I/O has no effect on the net performance. The slowing growth in Object Index I/O corresponds to slowing growth in the index size because of slowing growth in new words. The IRS Client and Server Index cases are not shown in Figure 21 because they are identical to their corresponding GS Client and Server Index versions. The same changes are being made to the data, so the logging and volume activity is the same. Only the RPC overhead is eliminated.

A series of experiments performed with the Reuters collection. The "Base" Build case used a one Megabyte page cache and a two Megabyte object cache. The "BigCache" case used a four Megabyte page cache and a four Megabyte object cache. In another case, which used the same cache sizes as "BigCache" but used an algorithm similar to INQUERY/Mneme (see Section 2.2), where if the space for a posting list object is exceeded, the posting list is grown by a factor of two. The final case also used the same cache sizes as "BigCache", but uses a linked list of single-element posting elements. All test were limited to 8 hours of execution time.

Using the Reuters collection and first discussing the Base case, Figure 22 shows that Build is super-linear with the number of documents for a Generic Server version with an Object Index. The previous experiments did not reveal this non-linearity. Figures 23, 24, and 25 show that the RPCs, data volume I/O, and log I/O are also super-linear with the number of documents. This

occurs because as more documents are indexed, the posting list objects grow increasingly larger. This puts an increasing memory demand on the object cache and eventually causes thrashing. Thrashing results in significant I/O activity because these growing posting list objects are written, re-referenced, and read.

In an effort to reduce the thrashing, I increased the page cache and the object cache to four Megabytes each. Figures 22, 23, 24, and 25 show this "BigCache" case and the resulting execution time, RPC count, volume I/O, and log I/O. The execution time is still super-linear, but the execution time is significantly improved. This was the only case that finished in under 8 hours at the 16,384 size document collection. The RPCs are reduced because there is less object cache paging. The volume read/write I/O and log I/O are also improved. The two other approaches (the size-doubling and linked-list schemes) showed significantly poorer results than their "BigCache" baseline.

4.5 Conclusion-Build

For the Generic Server version, there are a large number of RPCs required in the Server Index case (one "lookup" for each word indexed, and an additional "insert" for each new word indexed) that are not needed in the Object Index case, where these operations are all local to the object cache. I/O is also larger for the Server Index case, but this is not a factor in the net execution time. The net performance result is that the Object Index approach (using the Generic Server) is much faster than the Server Index approach.

There is an index bulk loading facility in SHORE, but it is currently implemented with only a server interface, so the Chrysalis client program can not use this function. If it were available, Chrysalis could maintain a cache of key/value pairs. When this cache is full, this cache could be sorted and sent to the server with a "bulk insert" RPC, which would not only lessen the RPC demands, but also load the B+ tree more efficiently.

For the Information Retrieval Server version, there is no RPC communication overhead required at all. This decreases the CPU load, resulting in a lower execution time for both index configurations, most dramatically for the Server Index case because it uses the most RPCs. The net result is better performance for both index approaches when using the Information Retrieval Server, but the Server Index approach is superior.

The super-linear execution time apparent with the larger Reuters collection occurs because posting lists are growing in size, which increases object cache memory demand. The linked-list and space-doubling scheme only aggravated this problem with the additional posting list size overhead. One possible solution would be to have a mechanism that read and wrote smaller portions of large objects and logged only the changes to this smaller portion as well. This would also be useful in multimedia editing applications where one might deal with a small portion of the object data. The storage manager in EXODUS [CD+88] provides this capability. While INQUERY/Mneme was forced into a space doubling scheme because it only had fixed-sized objects, these results show that a

better scheme is to only allocate an object size matching the actual requirements. The dynamic object sizing capabilities of SHORE were essential for implementing this approach.

4.6 Experimental Results-Query

Figure 26 shows how the average response time varies with the number of users in four different environments, all using the Generic Server version and the Technical Reports collection. Both the Object and Server Index versions were studied, along with performing the queries in the query script in either a single transaction (one for all queries) or using multiple transactions (one for each query). All the results were obtained with the Technical Reports collection of 512 indexed objects. The Server Index implementation has a lower average response time than the Object Index case. Also, the single transaction case outperforms the multiple transaction case for both index types.

Table 3 shows a comparison of the number of RPCs needed in the Generic Server implementation for both the Object Index (hash table version) and Server Index cases. This was obtained by running the query script, with one user and with ten users. It shows a smaller number of RPCs for the Object Index case when compared to the Server Index case, and both cases increase linearly with the number of users. There are seven more RPCs needed for the Server Index case; each query word lookup requires a "find" RPC (and there are seven words in the query script), while this is a local object cache operation for the Object

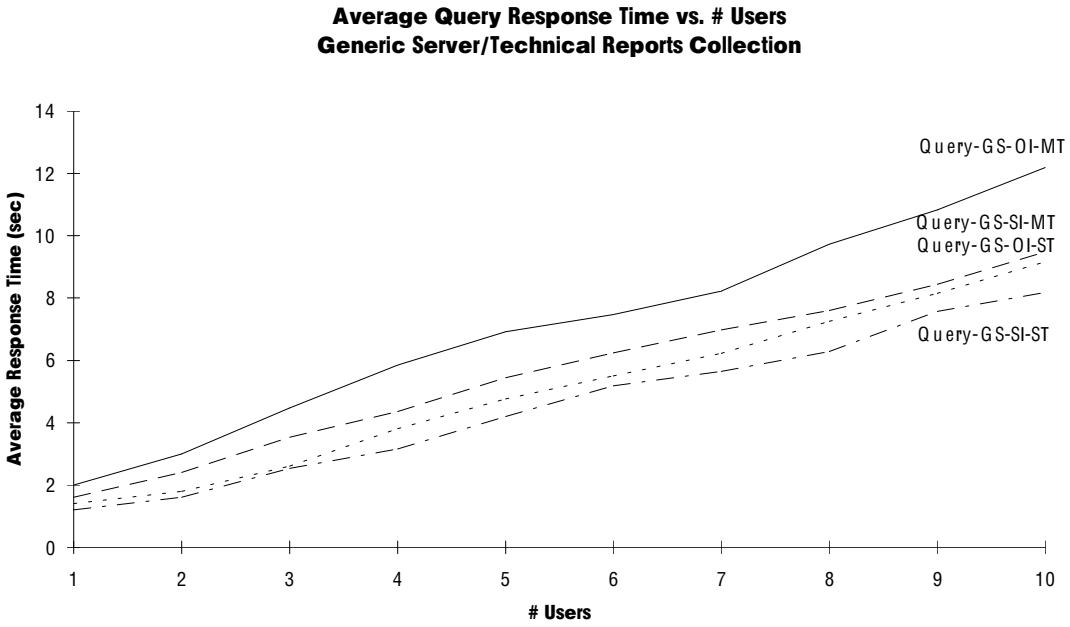

Figure 26: Query Response Time (GS)-Technical Reports

Environment	*RPCs (1 user)*	*Total RPCs (10 users)*
Query-GS-OI-ST	81	840
Query-GS-SI-ST	88	910

Table 3: Query RPCs-Technical Reports

Index case. If the query script were longer, the Server Index RPC difference would be even greater.

Table 4 shows a comparison of the number of I/O bytes generated for the Generic Server implementation for both the Object Index (hash table version) and Server Index cases with both one and ten users. It shows that the Object Index generates about 30% more I/O than the Server Index. This difference occurs because the entire index object must be read, while only the referenced B+ tree pages are read for the Server Index case. A longer Query script will

Environment	*I/O Bytes (1 user)*	*Total I/O Bytes (10 users)*
Query-GS-OI-ST	385,024	385,024
Query-GS-SI-ST	270,336	270,336

Table 4: Query I/O-Technical Reports

reference more of the B+ tree pages and the I/O counts will be more comparable. Although the Object Index must be read into each object cache (once for each query user), this does not produce a direct I/O penalty because these multiple reads are sourced from the page cache after the first one is read from disk, so there is no difference between the I/O counts for one and ten users. The Server Index is also only read into the server page cache once and can then service all users, again making the I/O between one and ten users the same.

Figure 27 shows the impact of object cache size on the number of RPCs generated by the query workload for a single user using the Object Index (hash table version) and Generic Server configuration. As the object cache size is reduced, the object cache reaches a point where it must toss out objects to make room for referenced objects. If the tossed object is again referenced, like the Object Index, additional read RPCs are generated. Since these objects are read-only, there are no write object RPCs generated, only read RPCs. The graph shows the importance of having an object cache at least as large as the working set of the workload. In this case, it is key to have at least enough space to hold the Object Index hash table so it is not thrashed. In this run, the volume read I/O is constant regardless of object cache size because the page cache is large enough to retain the working set even though the object cache cannot.

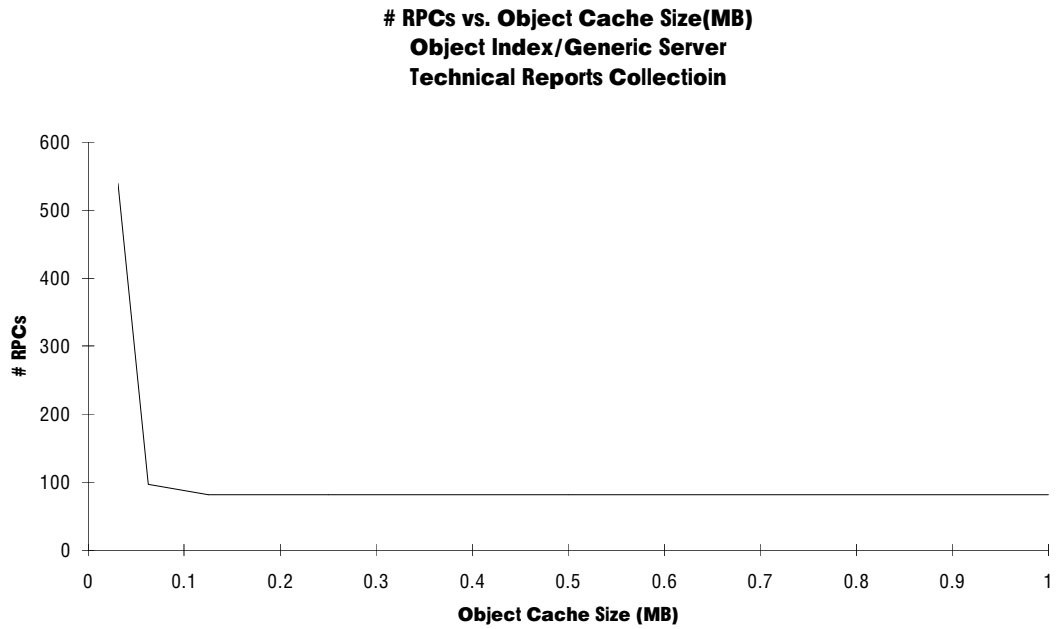

Figure 27: Query RPCs for Varied Object Cache Size

Figure 28 shows the result of an IRS implementation that supports multi-threaded queries using the Technical Reports collection. It uses multiple object caches, each capable of supporting an individual transaction thread. It shows improved performance due to RPC reduction for all environments, but most dramatically for the Server Index version. For the Server Index version, the B+ tree index is read into the page cache only once. It can then service each client. No index pages are read into the object cache, unlike the Object Index version. Like the Object Index version, the posting list objects and the documents themselves are read into the object cache, but these are much smaller than the Object Index. There is no RPC overhead, and the result is an efficient implementation. A disadvantage of this configuration is that multiple users mean multiple object

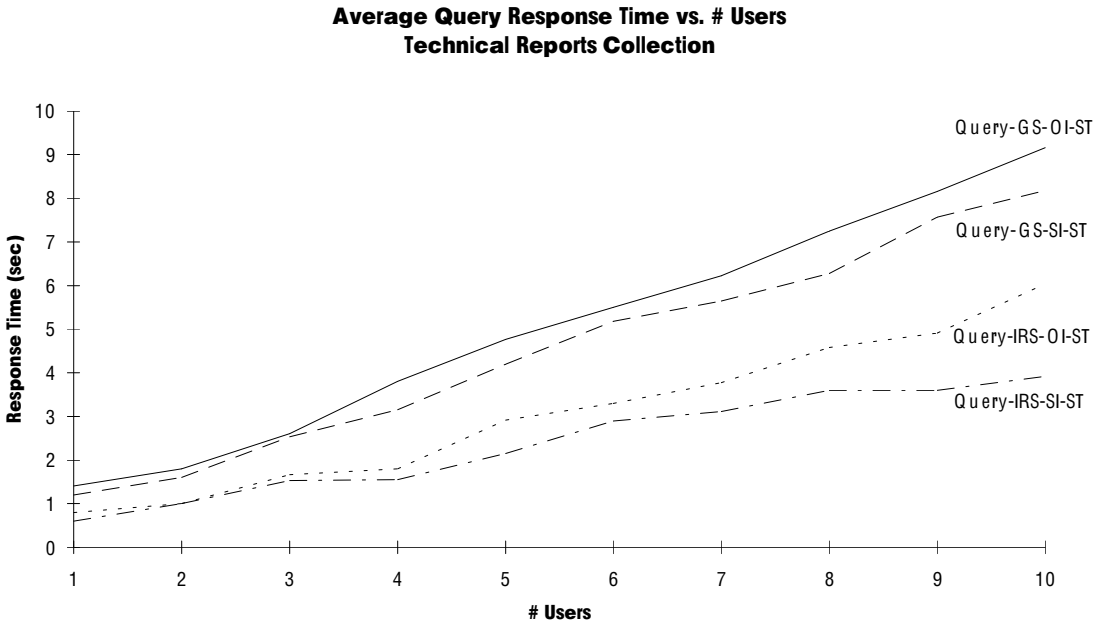

Figure 28: Query Response Time (IRS)-Technical Reports

caches consuming resources in the server. Chapter 5 discusses the implications of this configuration further.

I performed a series of tests to show the impact of the larger Reuters document collection using 16,384 documents on multi-user query response time and to explore the differences between all users running a common query script and each user having a different query script. In addition, I examined both the hash table and B+ tree versions of the Object Index. All of these experiments were performed using the Object Index and Generic Server configuration with a four Megabyte page cache. The Generic Server with Server Index configuration is not show because the Build time was too great to build a database to compare queries.

Figure 29 shows the results. The larger database has a larger object index (944,748 bytes vs. 118,388 for the Technical Reports collection) as well as larger posting list objects. The average document size is about the same in each workload. Comparing the B+ tree and hash table Object Index types, the graph shows a significant improvement in response time with the B+ tree index. This occurs because in the hash table case, the entire object must be read for the query, while in the B+ tree case, only the nodes referenced by the query must be read. Table 6 shows this I/O difference. There are slightly more RPCs needed in the B+ tree case because multiple B+ tree node objects are referenced. This is shown in Table 5. The I/O difference dominates over the RPC difference. The differences between 1 and 10 users follows the same pattern as the Technical Reports experiment discussed earlier.

Another series of runs were made where each user ran a different query script. The same basic script template (see Figure 13) was used, except random words were added to the script, and each user used a script with different random words. Since each user accesses a different set of words, there is a corresponding greater demand for posting list objects, which increases the memory demands and results in response time increasing more severely than the common script case. Tables 5 and 6 show a comparison of the RPCs and volume reads for both 1 and 10 users in the common and multiple script cases, for both index types. It shows the increase in read I/O for the multiple script case because of more posting list references. The B+ tree I/O increases more significantly, because

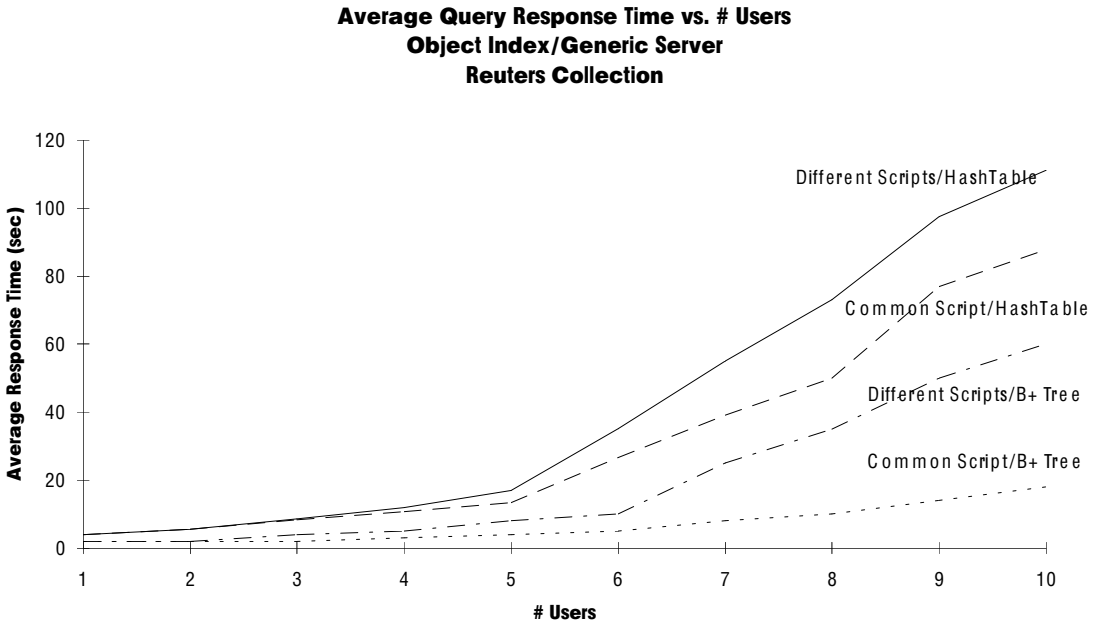

Figure 29: Average Query Response Time (GS)-Reuters

in addition to the additional positing lists, more B+ tree node objects are also referenced with the multiple scripts.

Figure 30 shows a comparison of both Object Index types with the Server Index, using the IRS configuration with multiple scripts. The figure shows that while the B+ tree Object Index outperforms the hash table Object Index, the Server Index performs best of all. No object cache space is needed for the Server Index, and no CPU time is needed for the data transfer from page cache to object cache. Figures 29 and 30 also shows the advantage of eliminating RPC overhead by using the IRS configuration.

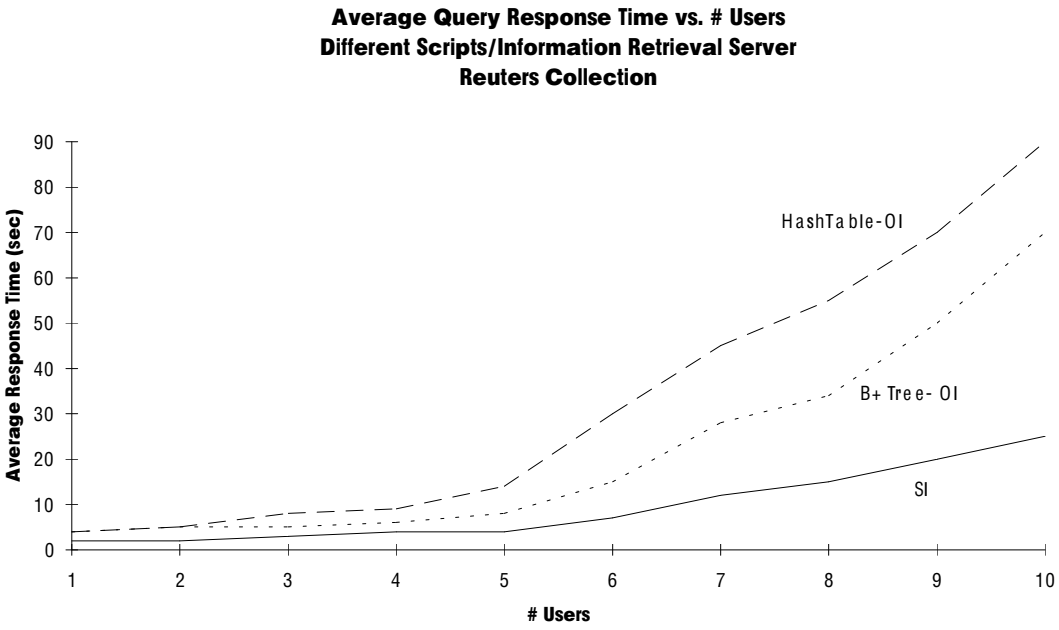

Figure 30: Average Query Response Time (IRS)-Reuters

Environment	*RPCs (1 user)*	*RPCs (10 users)*
Hash Table/Common Script	100	1000
Hash Table/Multiple Scripts	100	874
B+ Tree/Common Script	108	1080
B+ Tree/Multiple Scripts	108	954

Table 5: Query RPCs-Reuters

Environment	*I/O Bytes (1 user)*	*I/O Bytes (10 users)*
Hash Table/Common Script	1,572,864	1,572,864
Hash Table/Multiple Scripts	1,572,864	4,368,992
B+ Tree/Common Script	425,984	425,984
B+ Tree/Multiple Scripts	425,984	2,154,496

Table 6: Query I/O-Reuters

4.7 Conclusion-Query

Of the three benchmarks, Query is perhaps the most important because in a real-world system, querying is performed much more often than loading documents or building an index. The results show that for the Generic Server case, the Server Index approach is the least sensitive to the transaction packaging of the queries and is superior for the multiple transaction case, but is more comparable to the Object Index case when using single transaction packaging. The Server Index requires RPCs for each query word lookup, compared to local object cache operations in the Object Index case. The I/O amount is somewhat greater when the query script is short, but more comparable with a longer query script.

The B+ tree version of the Object Index performs significantly better than the hash table version because the former only reads index nodes referenced by the query while the latter always retrieves the entire index object.

Both index cases could be improved with inter-transaction caching in SHORE. This feature would be beneficial in several scenarios. In a real information retrieval environment, multiple users would query the index concurrently, while an occasional index update would be required. There are two options for transaction packaging. If each query is a separate transaction, then for the Object Index case, where the index object is invalidated after each query, and the next query would cause the index object to be reread into the object cache even though it was not changed. Inter-transaction caching would allow this index object to be more quickly accessible unless it were needed by an updating process, like

the index builder. If the entire query session is packaged as a transaction, then for the Object Index case, the index object will remain in the object cache for successive queries, but any updating process, like the index builder, must wait until all outstanding long running query transactions have completed.

The IRS version of the Server Index implementation provides the best solution for the Query workload. RPC overhead is eliminated, the B+ tree server index pages are read into the page cache only once (and only the referenced pages are read), and only those posting list objects and document objects that are pertinent to a given user's query are read into the user's object cache. Since query is the most important workload, the custom server support combined with a server index is optimal for information retrieval using SHORE.

4.8 Registered object I/O

Recall that registered objects are those that have a name in a name space. In an effort to understand the large I/O for the Load workload, which creates registered document objects, I performed an additional experiment using a fixed-size hash table registered object with 32,707 slots. There are two different I/O impacts from registered objects. The hash table registered object has 392,492 bytes (12 bytes per hash table slot times 32,707 plus 8 bytes for the timestamp). This object is logged during the "mkregistered" RPC processing (when it is first registered in the name space), even though its contents have not yet been set at this point. The log space required for this is a minimum of 392,492 bytes. The

object is again logged after its contents have been set, at commit time. Because a previous log entry exists, this second logging action logs both the old and the new data (even though the old data is garbage). This logging requires an additional 2 * 392,492 = 784,984 bytes of log overhead. The total log bytes for this object, therefore, has a lower bound of 784,984 + 392,492 = 1,177,476 bytes (there are additional log bytes needed for overhead). This calculated log space difference compares with a measured value of 1,190,952. Section 5.3.6 discusses some possible ways to reduce the amount of logging required here.

Chapter 5

Conclusions

This chapter discusses the functional, architectural, and performance characteristics that I have found to be important for information retrieval applications. I also highlight SHORE's strengths or shortcomings in these areas. Finally, I conclude with a list of areas for future work.

5.1 SHORE Functional Issues

5.1.1 Objects for indices and documents

Objects are useful for representing both the index structures that help locate relevant documents for full-text query processing and the documents themselves. All the pieces of a retrieval system then exist in the same repository and transaction semantics can be applied to all operations, leading to a robust implementation. There were many occasions during the development of Chrysalis

where an error occured during various phases of processing, but the index and the documents were always in sync, allowing another processing attempt from a consistent base.

Objects are a useful representation for documents. An attribute can be used to hold the document contents, whether text or binary data. An especially useful feature, and perhaps the biggest contribution of SHORE, is to have this attribute addressable as a UNIX file in a name space. This allows the vast array of existing UNIX programs and tools to be used with the objectified document. A full-text index can also be built based on the contents of this attribute. Other attributes can be used to characterize the document, although SHORE's lack of an object query language limited their exploitation in my prototype. I believe the combination of full-text queries of document context contents used in conjunction with queries upon the document's attributes would yield a powerful document search tool.

The document inheritance hierarchy developed in this work was able to handle a broad range of document types while taking advantage of code reuse. New document types are easily created and require little extra code. The dual object implementation for binary documents allowed full-text indexing on the descriptive text as well as minimizing I/O by only retrieving a small object header, and not the potentially large BLOB object with the binary data itself unless it was necessary for display or manipulation purposes.

5.1.2 Persistent object programming environment

Programming with persistent objects that look much like C++ objects without having to worry about a separate database language, object flattening and unflattening, schema mapping, or cache management yielded a highly productive programming environment for writing this application. Having transaction semantics eliminated much consistency checking and correcting code. Chrysalis was both relatively easy to build and to maintain as a result.

5.1.3 Concurrency control

It was useful to separate methods into three sets, those that modified data, those that did not, and those that could be upgraded at run time to allow modification. Functions like query used only methods that did not modify data, which eliminated the need for flushes from objects in the object cache to the server as well as the associated logging. This also allowed for maximum concurrency. There are, of course, cases where data must be modified. There were cases, however, where it was advantageous to assume that an object would not be modified, but to later change that assumption if needed. SHORE provided support for both const (read only) and non-const (read/write) methods, as well as an ability to upgrade a const method to non-const at run time.

5.1.4 Named and unnamed objects

I found it useful to distinguish two types of objects, those that need a name in a name space and those that do not. Some objects, like documents, need a name because they are accessed by UNIX programs. Other objects do not need a name because they are accessed by navigation from a named object. The posting list objects are an example of this latter type of object. SHORE supports both types of objects (named objects are called registered and unnamed objects are called anonymous). The development of a World Wide Web document server that used a standard browser as a user interface illustrated the utility of these named document objects.

It was also useful to be able to temporarily bind a name visible in the name space to different objects. For example, when an object is selected for viewing, it could be made visible via a well-known name. SHORE provided a cross reference operation that supported this capability.

5.1.5 Schema migration

In this prototype, if there is any change to an object definition, including changes that do not affect the storage layout of the object (like adding a method), the object definition file must be recompiled. This invalidates existing objects with the old definitions. Other systems provide techniques that allow changes to objects that preserve the previous storage layout, allowing the ability to access these old objects. Currently, there is no strategy for object schema migration in

SHORE. One possible solution, used in IBM's System Object Model (SOM), is described in [Lau94].

5.1.6 Triggers

It would be useful to automatically trigger operations when certain events (like the creation/modification of an object) occur. SHORE could then trigger an indexing operation when a file is added into the NFS view of SHORE.

5.1.7 Command shell

There are two command shells for objects in SHORE. One is the UNIX command shell that is used with the NFS view, and the other is a native SHORE command shell. Some operations can only be performed by one and not the other, like object deletion (the SHORE shell must be used in this case). This can be viewed as a feature, because special SHORE programs could be written to provide "smart deletion" of objects by knowing their structure and by properly handling objects they might point to. But today, it is awkward to manipulate objects in both environments. A unified shell capability would be a useful improvement.

5.1.8 Tracing and statistics

Tracing and statistics are essential for performance analysis. While SHORE contains a wealth of statistics and trace facilities, they need to be more oriented towards the needs of the application writer. This dissertation has highlighted

the importance of reducing RPC calls and logging, yet there is no easy way to correlate application program actions with generated RPCs. In a similar vein, there is no easy way to correlate log volume or volume read/write activity with application program actions. The page-based logging scheme makes this a more difficult problem because in addition to tracking page logging activity, one must also relate the involved pages to an object or objects to present useful information to a user.

5.2 Architectural Issues

5.2.1 Object vs. server B+ tree index

There were two main architectural issues explored in this dissertation. The first involves how to implement the full-text index— the options considered were using application level index objects versus a server B+ tree index. When the index is implemented as an object, it must be present in the object cache of each user, which means replicated data, but this allows index operations to be performed in-memory within the cache. To get the index object into the object cache at the beginning of a transaction, there must be I/O performed as well as RPCs to bring the object across the shared workstation memory. The cost of these RPCs and I/O must be compared to the alternative when the index is implemented as a SHORE server index. In this latter case, an RPC is required for each index operation, which makes initially invoking an operation much more

costly than the equivalent operations for the object index (which are just local object cache operations). The net result of this index tradeoff is that there is a penalty for using an object index when reading the index into the object cache and writing it out at transaction commit time, but a major benefit accrues when the index is being built.

These differences are highlighted in two workloads, with the Object Index case running faster than the Server Index case for the Build workload because in-memory object cache index operations are much faster than RPCs to the server. The Server Index case, in contrast, performs better than the Object Index case with the Query workload for the Information Retrieval Server case (and the Generic Server case for the smaller collection tested). The differences are less apparent on the single transaction Query workload, but more pronounced on the multiple transaction Query workload, where the lack of inter-transaction caching in SHORE causes a larger difference. The IRS implementation supports multi-user query best of all with the Server Index. This configuration attains the minimum I/O for the index, requires object cache filling for only the pertinent posting list objects and document objects, and also has the advantage of some RPC overhead elimination.

5.2.2 Client vs. server function placement

The second major architectural tradeoff examined is where to put the information retrieval function, with the choices being in a client or in the SHORE server

itself. The latter approach eliminates the RPC overhead, significantly speeding up both Object and Server Index implementations. Furthermore, the process for migrating the Chrysalis application into the SHORE server proved to be very simple, suggesting that custom function servers might be appropriate and very feasible for other special purpose applications as well. The Build and Query workloads both highlighted this tradeoff.

5.2.3 Multi-threaded object cache

The Query workload, like any multi-user workload, needs an object cache for each thread of execution. This means significant additional resources are consumed for each user in the IRS implementations, especially for the Object Index case. It would be interesting to explore an architecture that allows a single object cache that supports multiple threads of execution. There are several major challenges to implementing this, including resource allocation and security.

5.2.4 Extensible object-oriented design

During the studies, many modifications were made to the Chrysalis code base. Over time, a pattern emerged for the application that was common for the implementations and provided good isolation of changes. An extensible object-oriented design was eventually developed, allowing easy modifications of information retrieval algorithms, index types, and document types.

5.3 Performance Issues

5.3.1 Scalability

The major area for disappointment with the Chrysalis implementation is in its scalability. When single objects are used for word posting lists, the overhead of reading and writing these lists becomes increasingly great as the number of indexed documents increases. On the other hand, when using a linked list of objects for word posting lists, the overhead of handling so many small objects, as well as the overhead for object headers and storing pointers, is even greater. The space doubling scheme used in INQUERY/Mneme does not perform as well as keeping the posting lists only as large as necessary. A possible solution is for SHORE to support reading, writing, and logging of small portions of large objects (ala EXODUS [CD+88]).

5.3.2 Manipulating small portions of large objects

The growing posting list problem discussed in this thesis points to a need to efficiently handle small changes to large objects without incurring the overhead of reading, writing, and logging whole objects when only a small portion is changed.

For the Query workload, the whole posting list object must be retrieved anyway, so this function would have no benefit. For the Build workload, the only portion of the posting list of interest is the last entry, so this function could

provide substantial benefit. This would not only be useful for the posting lists in Chrysalis, but might be applicable to large multi-media object handling where only a small portion is changed (like audio or video editing). This could be implemented by having the user specify a hint that they are only interested in retrieving the specific data portions referenced in a read or write method, and not the entire object. In addition, offset and length based retrievals could be performed on sequence or large BLOB attributes. SHORE, unlike its predecessor EXODUS, does not yet support this finer granularity access.

5.3.3 Inter-transaction object caching

Inter-transaction caching would allow the index object(s) to be quickly re-accessed after a query transaction so that the next transaction does not pay the full penalty of re-reading the object(s) again. In the cases examined here, this index is unchanged, so the same object state is re-read. The single versus multiple-transaction Query experiments show a significant potential gain from inter-transaction caching.

As discussed in Chapter 3, callback locking and timestamp cache coherency algorithms have been implemented by several OODBMS companies to address this problem, but these solutions have architectural and performance issues (especially with high concurrency). SHORE does not currently implement inter-transaction caching, but there are plans to address this issue with a SHORE server page cache that resides in a client, as previously shown in Figure 6.

5.3.4 B+ tree vs. hash table object indices

There is no difference in Build performance between the two Object Index types, but there is a significant difference for Query. This occurs because with the B+ tree, only referenced index node objects are retrieved. The hash table case requires the entire index object to be read for each query. The B+ tree also provides deletion capability as well as ordering, if these functions are required.

5.3.5 B+ tree index bulk loading

As discussed in Chapter 4, if SHORE extended the B+ tree index bulk loading facility so a client application could make use of it, most of the individual index insertion RPCs could be combined with a single bulk insertion RPC, using a sorted list of key/value pairs. This should improve the Generic Server configuration that uses the Server Index with the Build workload.

5.3.6 Registered object performance

Registered objects in SHORE have two implementation problems. First, in order to preserve the UNIX semantics that say that a file name must appear in the name space upon file creation, a new registered object is logged three times. One logging event occurs after a "mkregistered" RPC, even though the object contents have not yet been set at this point. At commit time, the object is essentially logged two times (old and new object contents) even though the old object contents are uninitialized. This problem could be resolved by loosening

the semantic constraint, so that the object could be created in the name space at commit time, thus resulting in only one log event. Another solution would be to allow the name to be registered but the object to be uninitialized, with an appropriate error if an attempt is made to use it between registration and commit time.

The second problem is different, although related to the first. When a series of registered document objects are created, the initial "mkregistered" RPCs for the objects create them contiguously on SHORE disk pages. At this point, however, these objects only have a core area (non-string/text attributes). At commit time, these objects also contain a heap area (for the string/text attributes). At commit time, the larger versions of these objects cannot be placed at the same place where they previously resided, because there is no room at the end of the object to grow. The objects must therefore be moved to another page and then grown. This results in a domino effect of object expansions with corresponding CPU overhead as well as possible I/O overhead. If UNIX semantics weakening is not acceptable, there could be a "hint" parameter added to the "mkregistered" command so that an appropriate estimated amount of space could be reserved on the page for the heap area which would then be filled at commit time.

One could use the SHORE "xref" facility to bind a name to an anonymous object and avoid the registered object overhead, as was done in Chrysalis (see Section 3.3).

5.4 Chrysalis Future Work

Finally, in addition to consoling the previous suggestions for SHORE modifications, there are several other areas that would be interesting to explore further. These include:

- All the studies described in this dissertation were performed on a single workstation, so the client(s) and server competed for the same memory, and client/server communication used shared memory instead of a communications mechanism. If the client(s) and server were on separate workstations, virtual memory thrashing would be lessened and RPCs would have an even greater impact on performance. These configurations should be further explored.

- The Chrysalis/SHORE application uses a client/server architecture. It would be interesting to explore both the functional and performance implications of building this application with distributed object technology, using an architecture such as CORBA [MZ95].

- As discussed in Chapter 1, SHORE will support multi-server configurations, so it would be interesting to explore the performance characteristics of this configuration as well as explore opportunities for parallelism.

- Some Chrysalis performance profiling revealed that the Porter stemming algorithm accounts for a large portion of the index building and querying

paths in the Chrysalis client code. It would be interesting to explore caching stems in order to speed up the stemming function.

- The minimal finite state machine used for stop list checking was developed before the work on this thesis and was implemented persistently by flattening it into an ASCII file after construction and then unflattening it when needed. This would be a candidate for a persistent object, making for an interesting performance comparison between the two implementations.

- Chrysalis could benefit from an automatic document type classifier such as the one used in Rufus, instead of the manual tagging approach now used.

- As discussed in Chapter 2, there are better term weighting schemes as well as relevance feedback techniques that would enhance the information retrieval quality aspects of Chrysalis.

- SHORE supports a security model similar to the UNIX file system. While Chrysalis does not make use of this feature, it should be fairly easy to provide protected document support.

Bibliography

[Bar93] D. Barbara, et al. The Gold mailer. In *Proceedings of the International Conference on Data Engineering*, 1993.

[BC87] N. Belkin and W. Croft. Retrieval techniques. In *Annual Review of Information Science and Technology*, pages 109-145, 1987.

[BM85] D. Blair and M. Maron. An evaluation of retrieval effectiveness for a full-text document-retrieval system. In *Communications of the ACM*, pages 289-299, March 1985.

[Blo92] J. Bloomer. *Power programming with RPC* O'Reilly & Associates, Sebastopol, CA, 1992.

[Boo94] G. Booch. *Object-oriented analysis and design with applications.* Benjamin/Cummings, Redwood City, CA, 1994.

[Bro95] K. Brockschmidt. *Inside OLE* Microsoft Press, Redmond, WA, 1995.

[Bro94a] E. Brown, et al. Fast incremental indexing for full-text information retrieval. In *Proceedings of the 20th Very Large Database Conference*, 1994.

[Bro94b] E. Brown, et al. Supporting full-text information retrieval with a persistent object store. In *Proceedings of the 4th International Conference on*

Extending Database Technology, pages 365–378, 1994.

[Cat94] R. Cattell. *The Object database standard: ODMG-93 (Release 1.1)*. Morgan Kaufmann, San Mateo, CA, 1994.

[CD+88] M. Carey, D. DeWitt, G. Fraefe, D. Haight, J. Richardson, D. Schuh, E. Shekita, and S. Vandenberg. The EXODUS extensible DBMS project: an overview. Technical Report 808, Computer Sciences Department, University of Wisconsin, Madison, WI, November 1988.

[CDF+94] M. Carey, D. DeWitt, M. Franklin, N. Hall, M. McAuliffe, J. Naughton, D. Schuh, M. Solomon, C. Tan, O. Tsatalos, S. White, and M. Zwilling. Shoring up persistent applications. In *Proceedings of the 1994 ACM-SIGMOD Conference on the Management of Data*, Minneapolis, MN, May 1994.

[Com79] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, June 1979.

[Fox90] C. Fox. A stop list for general text. In *SIGIR Forum.*, pages 19-35, January 1990.

[FBY92] W. Frakes and R. Baeza-Yates. *Information retrieval: data structures and algorithms*. Prentice Hall, Englewood Cliffs, NJ, 1992.

[Geh94] N. Gehani, et al. OdeFS: a file system interface to an object-oriented database. In *Proceedings of the 20th Very Large Database Conference*, 1994.

[Gif91] D. Gifford, et al. Semantic file systems. In *ACM SIGOPS Operating Systems Review*, pages 16–25, 1991.

[HR83] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. In *ACM Computing Surveys*, December 1983.

[Jan95] J. Jannink. Implementing deletion in B+-trees. in *ACM SIGMOD Record*, March 1995, pages 33–38.

[Lam91] W. Kim, et al. The ObjectStore database system. In *Communications of the ACM*, pages 50–63, October 1991.

[Knu73] D. Knuth. *The art of computer programming. vol 3: sorting and searching.* Addison-Wesley, Reading, MA, 1973.

[Lan79] W. Lancaster. *Information retrieval systems: characteristics, testing, and evaluation.* John Wiley and Sons, New York, NY, 1979.

[Lau94] C. Lau. *Object-oriented programming Using SOM and DSOM.* Van Nostrand Reinhold, New York, NY, 1994.

[LS90] E. Levy and A. Silberschatz. Distributed file systems: concepts and examples. In *ACM Computing Surveys*, pages 321–374, December 1990.

[LPJ+94] C .Liu, J. Peek, R. Jones, B. Buus, and A. Nye. *Managing Internet information services.* O'Reilly & Associates, Sebastopol, CA, 1994.

[Mes91] E. Messinger, et al. Rufus: the information sponge. Technical Report RJ8294, IBM, August 1991.

[MZ95] T. Mowbrary and R. Zahavi. The essential CORBA: systems integration using distributed objects. Wiley, 1995.

[Por80] M. F. Porter. An algorithm for suffix stripping. *Program*, pages 130–137, July 1980.

[RN95] B. Rubin and J. Naughton. Using the SHORE object-oriented database/file system paradigm for information retrieval. Technical Report 1269,

Computer Sciences Department, University of Wisconsin, Madison, WI, June 1995.

[Sal68] G. Salton. *Automatic information organization and retrieval.* McGraw-Hill, New York, NY, 1968.

[Sal+75] G. Salton, et. al. A vector space model for automatic indexing. In *Communications of the ACM*, pages 613-620, November 1975.

[SM83] G. Salton and M. McGill. *Introduction to modern information retrieval*, McGraw-Hill, New York, NY, 1983.

[Sal86] G. Salton. Another look at automatic text-retrieval systems. In *Communications of the ACM*, pages 648-656, July 1986.

[Sal88] G. Salton and C. Buckley. Parallel text search methods. In *Communications of the ACM*, pages 202-215,. February 1988.

[Sal89] G. Salton. *Automatic text processing: the transformation, analysis, and retrieval of information by computer.* Addison-Wesley, Reading, MA, 1989.

[Sho93] K. Shoens. The Rufus system: information organization for semi-structured data. In *Proceedings of the 19th Very Large Database Conference*, 1993.

[Spa81] K. Spark Jones (ed). *Information retrieval experiment.* Butterworths, London, 1981.

[SK86] C. Stanfill and B. Kahle. Parallel free-text search on the connection machine systems. In *Communications of the ACM.* pages 1229-1239, December 1986.

[YA94] T. Yan and J. Annevelink. Integrating a structured text retrieval system with an object-oriented database system. In *Proceedings of the 20th VLDB Conference*, pages 740–749, 1994.

[Zip49] G. Zipf. *Human behavior and the principle of least effort.* Addison-Wesley, 1949.