

Using the SHORE Object-Oriented Database/File System Paradigm for Information Retrieval

Bradley S. Rubin* Jeffrey F. Naughton†

Abstract

The SHORE (Scaleable Heterogeneous Object REpository) persistent object store under development at the University of Wisconsin-Madison is a hybrid of 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 paper, we evaluate the suitability of the SHORE system for supporting the implementation of an information retrieval server. Our experience with our implementation of a prototype multimedia information server, Chrysalis, demonstrates that overall the SHORE persistent object store functionality is better suited to supporting the construction of an information retrieval server than the traditional alternatives of file systems, relational database systems, or even current OODBMS. Chrysalis is the first major SHORE application, so this paper also serves as an experience guide for other potential SHORE applications.

1 Introduction

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 the storing persistent complex data structures that an information retrieval application requires.
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.

*International Business Machines, 3605 Highway 52 North, Rochester, MN, 55901; rubin@cs.wisc.edu. Supported by the IBM Resident Study Program.

†Department of Computer Sciences, University of Wisconsin, Madison, WI, 53706; naughton@cs.wisc.edu. Supported in part by NSF grant IRI-9157357.

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.

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; we will describe this more fully in Section 2, but briefly, SHORE can store persistent objects like any other OODBMS, but can also export a UNIX file-system interface to some of the fields of the objects it stores. (The ODE system at Bell Labs is another system that supports this dual paradigm. To our knowledge, no commercial OODBMS currently supports this functionality, although there is no fundamental reason why this functionality couldn't be added.) As a step toward evaluating how well this dual OODBMS/File System paradigm supports the construction of data servers, we built Chrysalis, a prototype information retrieval application, within SHORE.

Our experience, detailed in this paper, shows that the Chrysalis/SHORE approach is an improvement over other approaches in several respects. Like previous information retrieval applications, Chrysalis uses the persistent object store (SHORE) to store metadata. But, unlike previous systems, the documents (here “documents” includes a variety of types of data, including text, video, and image) themselves are also stored as objects. This 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 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.

The rest of this paper is organized as follows. Section 2 gives some information about the SHORE system and about the information retrieval functionality provided by Chrysalis. In Section 3 we describe how the information objects corresponding to Chrysalis documents are represented within SHORE. Section 4 gives some details of our decisions within the implementation of Chrysalis, and describes some benefits we 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. Most notably, if SHORE provided a trigger mechanism keyed off of events occurring through the SHORE UNIX file system interface, then updating indices upon new document insertion in Chrysalis would have been much cleaner than it is in our implementation. We discuss this in more detail, and add our general conclusions and plans for future work, in Section 5.

2 Background and Related Work

2.1 SHORE and Chrysalis

SHORE [CDF⁺94] 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 namespace, 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 also be supported.

The SHORE hierarchical namespace looks like a UNIX namespace, except there are typed objects at the leaves, instead of untyped files. The SHORE filesystem can be Network File System (NFS) mounted into a UNIX file system, and the contents of an object attribute field called `text` appear as a byte-stream file to UNIX, allowing compatibility with existing UNIX applications and tools. Another example of this combined OODB/Distributed File System paradigm is OdeFS from AT&T [Geh94].

Chrysalis is an information retrieval application implementing the vector space model of retrieval [FBY92, Sal89]. We developed Chrysalis to allow us to explore the implications of building information retrieval systems using the SHORE paradigm. Chrysalis allows a user to enter a natural language-like query and to receive a list of potential matching information objects (generically referred to as documents in this paper), listed in relevancy order. The system uses a full-text index of the document contents. This index consists of a hash table object, which is used to store a representation of each significant word (after stoplist and conflation processing [FBY92]), with a pointer to a posting list object. The posting list contains a count of how many times the word appears in each document. The words in a query are also stoplist and conflation processed, and a logical dot product of the query vector (where each element corresponds to a word in the query) with each document vector (where each element corresponds to the number of times a word appears in the document) is performed for each document. The resulting scalar value is used to rank the documents in relevancy order. When a user selects a document for manipulation, they are given a UNIX shell prompt and can execute any UNIX command or application program against the document.

2.2 Related Work

In this paper, we refer to two other 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,

In INQUERY/Mneme, document posting lists are stored in an OODB, while the actual documents and the hash table index are stored in a UNIX file system. 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.

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

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.

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 metadata is stored in a relational index. The query language allows both full text and attribute queries by using a common proximity search mechanism.

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 an information retrieval system. The conclusions by Yan and Annevelink on the utility of an object-query interface to an information retrieval system also apply to Chrysalis.

3 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 can be in several languages (initially C++). 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. As an example, Figure 1 shows how a UnixFile and

```

interface UnixFile {

public:
    void init( in lref<char> fileName );

protected:
    attribute text contents;
}

interface TextObj : public UnixFile {

public:
    void init( in lref<char> fileName );
    void display() const;
    void index( in ref<HashTbl> h,
                in long handle ) const;
    lref<char> title() const;

protected:
    attribute string author;
    attribute long   version;
};

```

Figure 1: SDL definitions of UnixFile and TextObj Objects.

a TextObj object are specified in SDL.

In this code, the base class called UnixFile 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). The contents field 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 TextObj is derived from UnixFile, 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, one to display it, one to index it, and one to return a 60 character document title. The init method, in addition to initializing the contents field, 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 (can be any name, we 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.

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 use a template function called `lref`, so `lref<char>` is equivalent to `char*`. Pointers to objects (or OIDs) use a template function called `ref`, so `ref<HashTbl>` is a pointer to a HashTbl object. The `ref` syntax provides a level of indirection for implementing pointers to persistent objects. The `ref` requires 8-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. We refer to both this 8-byte `ref` pointer and the 16-byte volume identifier/serial number as OIDs in this paper, 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 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, we 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 file 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, we needed the flexibility and performance benefit of making this upgrade determination during run time. This upgrade capability was subsequently added to SHORE.

SHORE supports object inheritance hierarchies (see Figure 2). Currently, Chrysalis supports ASCII text, mail, net news, Wisconsin technical reports, images, audio, and MPEG movies. Other document types are easily added by writing the interface in SDL and the methods in C++, inheriting from other existing objects where appropriate. Yan and Annevelink [YA94] describe a similar hierarchy, with extensions to finer granularity parts of documents, like paragraphs and sentences.

For example, a Mail object is defined by the SDL given in Figure 3. The Mail object inherits from the TextObj 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 (`init`) is overridden so in addition to initializing the contents attribute

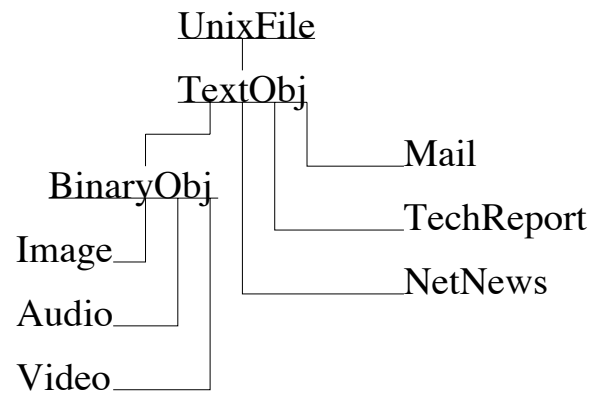


Figure 2: The Chrysalis Object Hierarchy.

```

interface Mail : public TextObj {

public:
    void init( in lref<char> fileName );
    void display() const;
    lref<char> getTitle() const;

private:
    attribute string subject;
    attribute string from;
    attribute string to;
    attribute string date;
};
  
```

Figure 3: SDL for Mail object definition.

```

interface BinaryObj : public TextObj {

public:
    ref<UnixFile> getLink() const;

protected:
    attribute ref<UnixFile> docLink;
};

interface Image : public BinaryObj {

public:
    void display() const;
    lref<char> getTitle() const;
};

```

Figure 4: SDL for BinaryObj and ImageObj objects.

to the mail message, it also parses out the subject, from, to, and date fields of the mail message and initializes these attributes. The display method displays the subject, from, to, and date attributes before the UNIX prompt is presented so the user can get a better feel for the mail contents before further operating on it. The title method returns the subject attribute contents for the one-line description given during retrieval ranking. The index method can be used as-is from the TextObj object, giving full text indexing of the mail contents. All totaled, the additions for Mail object support required only 60 lines of code and less than an hour to program. Also, 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 5.) The first object, of type Image, contains user-specified text (initialized 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 BinaryObj object, called docLink, points to the second object, of type UnixFile. 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 4.

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,

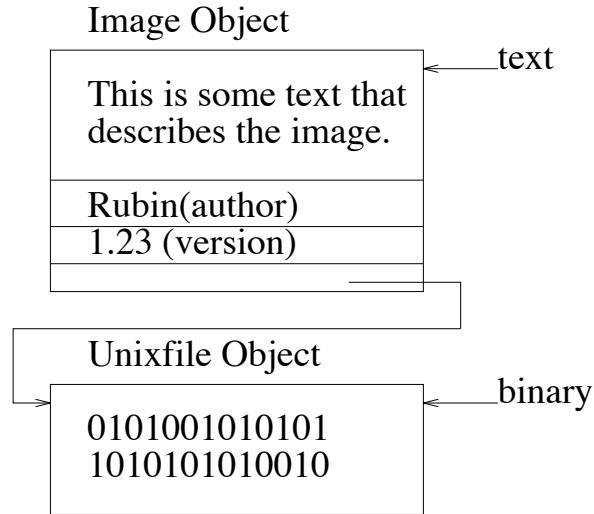


Figure 5: Object Structure for Binary Objects.

grep, mail, etc.) can be used on the object, and the objects can be viewed like any other UNIX file (ls, du, etc.). Chrysalis can provide full text indexing of the text field contents to support vector-space, relevancy ranked, queries. Figure 6 gives an example Chrysalis dialog showing the user view of a retrieval session using a full-text query.

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

```
SELECT *
FROM INFOBASE
WHERE TYPE = TechReport
AND AUTHOR = 'Rubin'
```

The results of one query could provide a result set as input to the other query, so all the technical report documents relative to object oriented databases written by Rubin could be found. We have not yet explored a query syntax that combines the full text queries with the attribute queries. Both Rufus and the Gold mailer allow this type of combined query.

4 Implementation Issues and the SHORE Functionality

In this section we describe some details of our implementation, and how our implementation interacted with and relied upon the functionality provided by SHORE.

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 6: Sample Chrysalis user dialog.

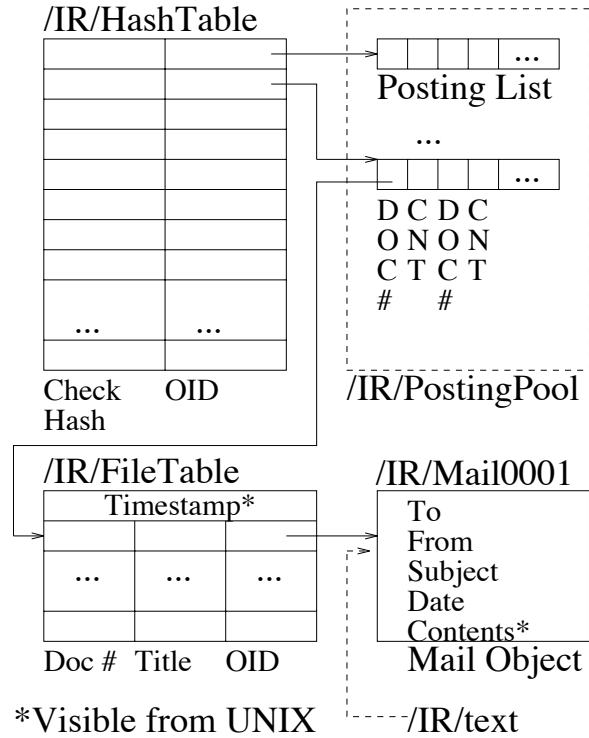


Figure 7: Chrysalis Object Structure.

4.1 Persistent Object Programming Environment

Chrysalis uses the object store for most of its data structures (see Figure 7). In the following, we use the term index to refer to the application layer data structures, not the indices that SHORE provides for set operations. There is an object called HashTable which stores a hashed form of the significant document words. It has the pathname /IR/HashTable in the SHORE namespace. Words are considered significant after passing through a stoplist test and a Porter conflation algorithm [FBY92, Por80]. The conflation process reduces related words to common stem (i.e.. computes, computer, computation all conflate to “comput”). The hash table entries do not contain the word itself, but a hashed, 16-bit version of the word. This check hash technique is described by [Dod82]. There is potential for false hits with this technique, but previous analysis showed that 16-bits yield an error rate of only 0.01% with 70% hash table loading.

The hash table contains pointers (OIDs) to corresponding posting list objects. These posting lists contain a word count and a document number for each document where the hash table word appears. This data is stored in a string attribute, which takes advantage of the automatic dynamic storage extension provided by SHORE as new documents are added to the index. One can just append bytes to or delete bytes from the string using standard memory-based programming techniques like `strcat()`. String attributes can contain both binary and null-terminated strings of bytes. 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 ob-

jects used for posting 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 data sizes change.

Since there are many posting list objects, and there is no need to provide a name in the namespace for these objects, we make use of the SHORE pool facility. The pool construct allows creation of a named object that contains a number of unnamed objects, making namespace 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 numbers (2-bytes) are used in the posting list objects as a level of indirection for OIDs (8-bytes) for storage efficiency. There is another object, called FileTable, that maps these document numbers to OIDs. The FileTable 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. A timestamp field, visible from UNIX, records the time of the most recent index entry.

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

The persistent programming environment considerably eases the tasks of index creation and updating. Pointers in data structures are automatically swizzled by SHORE and automatically moved between disk and memory on demand, so no code to flatten/unflatten data structures is needed. For example, adding a document to an existing index begins with adding the document to the FileTable 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 by concatenating the count/document number bytes to the posting list string. All needed structures are demand paged from the SHORE server disk to a client object cache, and forced back to disk at end of transaction. SHORE supports both automatic object cache space management as well as manual cache control, via commands like fetch and flush. Avoiding the impedance mismatch between in memory data structure formats and their flat disk counterparts also saves application code.

4.2 Namespace/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, and 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 desire 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 namespace. 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 namespace/OID correspondence provides advantages for information location transparency. None of the Chrysalis data structures contain namespace path names of the information objects, 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 pathname to an OID, so it can be used by any UNIX tool via the NFS mount facility. Since OIDs are never reused, if an OID in the index is dangling, the application knows the object was deleted.

4.3 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 metadata, so the two can get out of sync.

SHORE provides 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, but an index update process to exclusive write lock the index during modification, causing readers to wait until the write lock is released.

4.4 Indexing Algorithms

This section shows how the indexing is performed. New, changed, and deleted object are always detected, and index updates are incremental and atomic.

The user occasionally generates 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 Section 8 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 init method.

Chrysalis uses 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 FileTable structure. The timestamp represents the youngest indexed document. This results in a list of new objects that may need to be indexed. Chrysalis next makes sure the new object is an indexable type by using a SHORE isa function to see if it is in the TextObj 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 FileTable timestamp is restored to its old value, and the indexing structures are restored to their previous, consistent state. So, 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, 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 modification. Before the document is indexed, its OID is checked against all of the document OIDs in the index. All OIDs of indexed documents are kept in a set (an SDL construct), and membership is checked via an index on this set. If the OID 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 FileTable entry as invalid, which allows immediate deletion with lazy garbage collection of the posting list and FileTable entries. 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 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 deleted. During garbage collection, all the OIDs are checked for validity and cleaned if needed. Systems without unique, never reused identifiers, like Rufus, can not detect deleted documents until the corresponding file retrieval is attempted.

5 Conclusions and Future Work

When implementing an information retrieval system, one must make decisions about where to put the document data and where to put the metadata. Object-oriented databases are attractive for the metadata 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 for existing UNIX applications. SHORE combines these two concepts, and Chrysalis uses this paradigm to store both the document data and metadata in the same repository.

This merged distributed 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.

While SHORE provided a good match to our requirements, there are several functional shortcomings. Currently, we must tag UNIX files with an object type identifier in a file list so Chrysalis knows how to assign the object type when it is imported into SHORE. An automatic classifier, like

Rufus has, would be a useful addition to Chrysalis. Also, we would like the ability to automatically trigger operations when certain events (like the creation/modification of an object) occur. The combination of these two capabilities would allow a UNIX file to be added to SHORE by just copying it into the NFS view. SHORE would then trigger an indexing operation, which would classify the object and index it.

Another problem is that there are two command shells for objects in SHORE. One is the UNIX shell that is used with the NFS view, and the other is a native SHORE shell. Some operations, like object deletion, can only be performed by one and not the other (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 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.

The INQUERY/Mneme project [Bro94a, Bro94b] has demonstrated that storing information retrieval meta-data in an OODB can improve performance over a file system-based implementation, due to more sophisticated buffer management. An important area for future work on Chrysalis is to do a performance study on the implications of storing the data and the meta-data in the OODB.

We are currently building a version of Chrysalis that supports HTML and is accessible via the World Wide Web and a Mosaic client. This will leverage the automatic browser selection feature of Mosaic to invoke the appropriate viewer for each document object type.

Acknowledgments

We would like to thank the SHORE development team (Nancy Hall, Mark McAuliffe, Daniel Schuh, C.K. Tan, and Michael Zwilling) for supporting our development, even though we were right on the heels of their own development, and for entertaining and implementing several unplanned additions to SHORE.

References

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

- [CDF⁺94] Michael J. Carey, David J. DeWitt, Michael J. Franklin, Nancy E. Hall, Mark McAuliffe, Jeffrey F. Naughton, Daniel T. Schuh, Marvin H. Solomon, C. K. Tan, Odysseas Tsatalos, Seth White, and Michael J. Zwilling. Shoring up persistent applications. In *Proceedings of the 1994 ACM-SIGMOD Conference on the Management of Data*, Minneapolis, MN, May 1994.
- [Dod82] D. J. Dodds. Reducing dictionary size by using a hashing technique. *Communications of the ACM*, pages 368–370, June 1982.
- [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.
- [Mes91] E. Messigner, et al. Rufus: the information sponge. Technical Report RJ8294, IBM, August 1991.
- [Por80] M. F. Porter. An algorithm for suffix stripping. *Program*, pages 130–137, July 1980.
- [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.
- [YA94] Tak W. Yan and Jurgen Annevelink. Integrating a structured text retrieval system with an object-oriented database system. In *Proceedings of the 20th VLDB Conference*, pages 740–749, 1994.