The Rufus System: Information Organization for Semi-Structured Data

K. Shoens, A. Luniewski, P. Schwarz, J. Stamos, J. Thomas
IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120

Abstract

While database systems provide good function for writing applications on structured data, computer system users are inundated with a flood of semi-structured information, such as documents, electronic mail, programs, and images. Today, this information is typically stored in filesystems that provide limited support for organizing, searching, and operating upon this data. Current database systems are inappropriate for semi-structured information because they require that the data be translated to their data model, breaking all current applications that use the data. Although research in database systems has concentrated on extending them to handle more varieties of fully structured data, database systems provide important function that could help users of semi-structured information.

The Rufus system attacks the problems of semi-structured data. It provides searching, organizing, and browsing for the semi-structured information commonly stored in computer systems. Rufus models information with an extensible object-oriented class hierarchy and provides automatic classification of user data within that hierarchy. Query access is provided to help users search for needed information. Various ways of structuring user information are provided to help users browse. Methods associated with Rufus classes encapsulate actions that users can take on the data. These capabilities are packaged in a framework for use by applications. We have built two demonstration applications using this framework: a generic search and browse application called *xrufus* and an extension to the Usenet news reading program *trn*. These applications are in daily use at our research laboratory.

This paper describes the design and implementation of our framework, our experiences using it, and their influence on the next version of Rufus

**Proceedings of the 19th VLDB Conference
Dublin, Ireland 1993**

1 INTRODUCTION

The volume and diversity of the information stored on computer systems have grown with the systems themselves. Current workstation users store gigabytes of information locally and have access to far more over local area networks. While some of this information is highly structured and stored in databases, most of it is stored in ordinary files arranged in a directory tree.

It is difficult for people to make effective use of the information that's available to them. The large amount of data makes it difficult to find things when they are needed, while the diversity of information makes it difficult to use the data when it is found. Since computer systems offer little help locating and using data, users are compelled to memorize the location of data and procedures for using it. The tools provided by current computer systems are crude and do not scale as needed.

For example, consider the plight of an organization with hundreds of internetworked workstations. Users of these workstations write documents using any of a dozen word processing systems. Although the workstations make use of shared file systems like NFS [17], users must still locate documents of interest by filename. The system might offer a brute force application for searching through files, but the applications tend to be slow and to make it difficult to find what a user needs. Once a document is found, the user needs to remember how to browse or print the document using the application specific for its type. While this example uses word processor documents, the same situation holds for computer programs, images, electronic mail, configuration files, and so on.

Ideally, computer systems would provide significantly better tools for users to manage huge amounts of data. This ideal system would know what each piece of data is and how to use it. In the document example above, the system would know what application(s) apply to each file and how to run them. The ideal system would also know what's inside each piece of data to allow users to search for information about a particular subject. The searching should adapt to the data type, with different techniques available for searching for text, images, and coded data. The ideal system would use indexing so that queries could be answered quickly. Finally, the ideal system would know the relationships between various pieces of data. For example, in a document that includes figures, the system should understand the inclusion relationship.

In contrast to the ideal system, today's users must choose between storing their data in traditional filesystems or in database systems. For various reasons, filesystems have little or no semantics attached to stored files. An attempt to add these semantics to an existing system would likely break all existing applications and creating a new system from scratch, with all new applications, is unthinkably expensive.

Alternatively, users could store their data in a database. Unfortunately, database systems are unprepared to store the semi-structured information inundating users. Instead, database systems are oriented towards providing high integrity storage for structured data. Database research has concentrated on supporting the same type of data more efficiently, with better concurrency, and with better integrity. Efforts to extend the scope of data that database systems can handle have succeeding in capturing more applications with fully structured data, but still do not support semi-structured data.

There are two reasons why current database systems are inadequate for storing semi-structured data. The biggest inhibitor is that database systems insist on "owning" the data. When you decide to use a database system, you convert your data into its format and access the data exclusively through the database system. Moving semi-structured information into a database abandons all the applications that were written against the data's original format.

Another problem is that semi-structured data is imperfect—computer programs may have syntax errors or be incomplete, documents may not format correctly, and electronic mail may be damaged by the delivery system. A database solution designed to store this information must be able to represent imperfections. Database systems are instead oriented towards storing perfect information and for providing facilities for keeping it perfect. This need to cope with imperfection motivates filesystems to maintain unintrusive byte-stream models.

In summary, given the choice between byte-stream filesystems and structured databases, users have chosen filesystems for storing their semi-structured data. This is an unfortunate choice, because database systems offer many features that could help users cope with information overload. Database systems need to step up to the problems of semi-structured data to make these features available.

The Rufus project brings features traditionally belonging to database systems to bear on semi-structured information. An object-oriented database is used to store descriptive information about file system objects. To preserve existing applications, Rufus does not modify the file system objects themselves. An import process automatically categorizes each piece of user data into one of the Rufus classes and creates an object instance to represent the data. The underlying database supports fast querying and object access. Rufus provides various ways of structuring the objects to support browsing. This object infrastructure is made available through a client-server interface. We have built two applications to demonstrate the value of our infrastructure.

While designing the Rufus system, four particularly interesting problems were addressed. The first problem is the automatic classification of a file into one of the Rufus classes. Such a classifier must be fast, accurate, and easily extended with new classes. The second problem is correlating file system objects with Rufus objects. The most obvious approach, using file names, fails when files are renamed but users expect object identity to be maintained for the new file name. The third problem is the ability to add and delete classes from the class hierarchy of an existing database. With traditional approaches, such schema changes break the class hierarchy, due to the class relationships established by inheritance. The fourth problem is that of maintaining a dynamically-extendible text index with concurrent readers. This problem is further complicated by the need to be able to scale text indices to hundreds of thousands or millions of objects.

This paper describes the design and implementation of the Rufus system. Particular attention is focused on how the system addresses the four key problems listed above.

2 RELATED WORK

While existing data management systems do not support semi-structured information, research in the areas of object-oriented database systems, information retrieval, classification, hypertext, and some specific applications contribute useful techniques.

Semantic file systems [11] (SFS) provide *transducers* that extract attributes from files and provide them to an indexing system. Queries against a semantic file system are issued via extensions to the file naming syntax of the UNIX file system and are presently limited to conjunctive equality tests with string prefix matching. Rufus and SFS share the goal of raising the level of abstraction of the file system interface. Embedding the query language in the file naming mechanism provides access to SFS facilities without changing applications, but can be unnatural for some queries. As currently described, SFS does not associate actions with data, nor does it represent inter-file relationships. Users need richer data modeling and query capabilities to cope with the millions of files available to them.

Intelligent mail filtering capabilities, such as those found in BBN/Slate [5], the Andrew Message System [25], the Information Lens [18] and Tapestry [13] give users a large measure of control over their incoming mail. The term "mail" is used rather loosely here, as these systems purposely blur the distinction between traditional point-to-point electronic mail and point-to-many bulletin-board message systems. Tapestry, in particular, advocates replacing the notion of sender directed mail by recipient directed content-based retrieval. Thus the receiver, rather than the sender, controls what the receiver sees. Tapestry takes an active approach by providing user-defined intelligent agents to forage through various mail/message databases for items of interest. Collaborative retrieval is supported in Tapestry by associating user annotations with messages. These annotations can be used by others to select messages to read.

In many ways Tapestry is an information retrieval system applied to a limited interactive mail/message domain. In contrast, full-text information retrieval systems such as RUBRIC [19] and WAIS [15] provide users with the ability to retrieve files as the result of queries posed against the document text. To facilitate retrieval across a wide variety of document formats, these systems treat their data as unformatted text. Information retrieval systems and specialized systems such as mail handlers can be thought of as at opposite ends of the "domain specificity" spectrum. Rufus supports both kinds of use. A general purpose application can provide access across all data types, while special purpose applications can be written to exploit the semantics of specific data types. We describe both kinds of applications later in this paper.

Object-oriented database systems, such as ObjectStore [21] and O_2 [8], provide explicit frameworks for describing the structure of data types. These systems provide powerful query languages that allow users to express retrievals based on the structures defined in the schema. Object-oriented systems also provide a simple framework for encapsulating an object's structure together with its semantics, or behavior. As with other database systems, use of an OODBMS requires that a user's data reside in the database.

Database systems do not really concern themselves with modeling and importing existing file types and files. Mechanisms are usually provided for the one-time import of users' files, but the expectation is that they will then "live" in the database world: applications that were used to manipulate the original data are not applicable to the proprietary, internal database formats. Additionally, little or no support is provided for refreshing the imported data from native files that may have changed (through the use of external applications). Users either step fully into the database world or are left to manage the consistency of the two worlds on their own.

Hypermedia systems [6], which are based on a browsing metaphor rather than one of retrieval, also use proprietary internal data formats. Systems such as Intermedia [29] provide no avenues for integrating existing structured data into a hypermedia document other than as flat text. Once a hypermedia document or web is created, the systems offer only limited access paths to the underlying data. For example, although Intermedia is built on top of a relational database, the relational query capabilities are not available to Intermedia users. Hypermedia systems require that the data they operate upon be brought into the system, as is the case with database systems. Most of the value is derived from careful construction of links, which must be added by hand. Finally, hypermedia systems do not encode information about how to operate on data once it is found.

3 RUFUS

This section describes the Rufus approach to supporting semi-structured information. We describe how each aspect of the design is implemented in our current prototype, our experiences with the design choices, and modifications we are making to Rufus based on these experiences. To avoid confusion, "Rufus" refers to our general approach, "Rufus 1" refers to our current prototype, and "Rufus 2" refers to the version that we are in the process of building based on our experiences.

Rufus augments the file system representation of user data with persistent objects that retain information extracted from the original data. The original data is not modified and remains the authoritative copy so that existing applications are not affected. Rufus provides a set of classes that describe types of user files; examples include mail messages, C language source code, and various image file formats. Rufus includes a classifier that automatically determines the Rufus class of a file.

The structured objects that Rufus extracts are used for querying, organizing, and operating upon the data. In addition to extracting structured information about user data, Rufus indexes the data's contents. Rufus 1 supports a full-text index for textual data; other types of indexing could be added for non-textual data. Queries on Rufus data combine search operators on the contents of data (full-text in Rufus 1) with predicates on the extracted objects. Rufus uses collections, sub-classing, composite objects, and hypertext links to represent structure between objects. Some examples: 1) a mail folder is modeled as a collection of mail message objects; 2) a C program is modeled as a composite object including collections of source and object code, compilation instructions, and documentation; and 3) the structure of questions and answers in bulletin board articles is represented with hypertext links.

Rufus 1 is implemented on a client/server model to mimic the location transparency provided by distributed file systems. Rufus applications are written using a client library that provides program access to queries and Rufus data. Rufus 1 includes a catalog server to locate active Rufus servers. We have written two applications, one general purpose and one data-specific, as Rufus clients. These applications are in daily use at our laboratory.

Figure 1 shows the general structure of the Rufus system.

3.1 Classifier

The classifier examines a file and guesses what its Rufus class is, providing the first piece of information that a user needs about a piece of data. Given the volumes of semi-structured data, it is unreasonable to expect people to classify information manually. Thus, automatic classification is needed. Successful classification permits Rufus to dispatch the correct import method to extract attributes from user data.

The classifier uses the presence of keywords, file name patterns and file type (directory or normal file), and the presence of constant bit patterns near the beginning of the file ("magic numbers"). For efficiency, the classifier scans the file once to prepare an abstract of sampled keywords from the beginning, middle, and end of the file. The keyword samples include the token to the left of the keyword to pick up punctuation in examples like `\section` in LaTeX.

Each class supplies an evaluation function that returns a weight from 0 to 10 according to how likely the data is a member of the class. For sufficiently nondescript data, the classifier will likely return TEXT (plain ordinary text) for files that have mostly printable characters or BINARY for anything else. In case of error, the user may manually reclassify an object.

For the set of classes we have defined, the classifier is reasonably accurate and fast. To test it, we classified 847 examples of various file types. 90% were classified correctly, 8% were editor backup files that are given unusual names and were classified as "Text" instead of their actual type (mostly C language source code), and 2% were more significant errors (most were telephone directories classified as "Text"). A similar test on 100 randomly-selected user files revealed 4 significant misclassifications. Two were text formatter documents generalized to "Text," one was an extremely short text editor command script misclassified as "Binary," and the last was a command script misclassified as a specific kind of script. On these tests, the classifier averaged 55 milliseconds of CPU time per file on an IBM RISC System/6000, model 350.

The larger problem is that the classifier must carefully balance the weights returned by the evaluation functions to make the right decision in most cases. It would be difficult or impossible to add many more classes without upsetting this balance.

To address these limitations, we are building a new classifier based on a different model. In this new classifier, the programmer describes salient features of a new type, such as binary numbers that should appear near the beginning or regular expressions that should be found in textual formats and provides examples of objects of the given type. A classifier training program collects all the unique features into a global *feature vector* and computes the centroids of the feature vector for each type for which samples are available.

The actual classification of an object is performed by constructing its feature vector and matching it to the class with the nearest centroid. We are currently using the cosine coefficient similarity measure, a dot-product of two feature vectors nor-

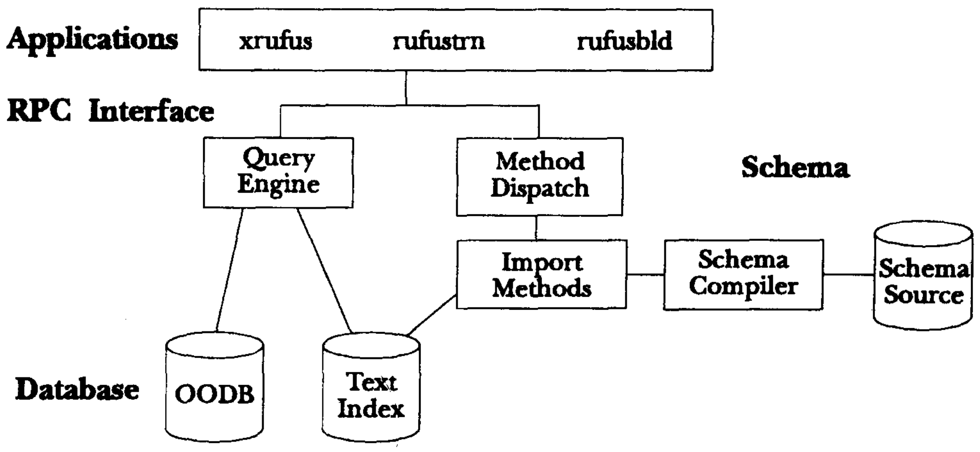

Figure 1: Structure of Rufus System

malized to remove biases towards classes with many features.

To date, we have tried this new classifier technique on a set of about 45 types, including those that Rufus 1 supports. The results are encouraging: the new classifier is about as accurate as the previous version and the process of adding new types is simple to it is simple. We now need to improve the performance of the new classifier (it can take it several seconds to classify a file) by combining the regular expression features into a single finite automaton with an algorithm like that suggested in [1] and to build the feature vectors for each class in a single pass through the file.

3.2 Importing Data

Once a piece of user data has been classified as class C, it can be imported into Rufus. Importing simply means that attributes are extracted from the underlying data and stored as an object. If the underlying data is not perfectly formatted, values may be left out of the extracted object. If the class of the object is text-oriented, the textual contents of the data are added to the text index.

Each Rufus class provides an import method that's responsible for performing extraction. Although the writers of classes are free to choose any formalism they wish to analyze the underlying data, Rufus neither supplies nor dictates the use of any such formalism. We have used plain C code for extraction in the classes we have implemented so far.

When a file is imported into Rufus for the first time, a new object identifier is created for it. If the file is modified and re-imported, its object identity is retained. When files are renamed, it can be difficult to maintain object identity. To cope with this problem, Rufus uses a type-specific unique identifier to track file identity, rather than the file name. For example, mail messages have unique "message identifiers" associated with them. Plain Unix files can be identified by their "inode" and "device" numbers. When a file is imported, its unique identifier is discovered by its class's constructor. A persistent mapping from unique identifier to object identifier is consulted; if the unique identifier is already known to Rufus, then the existing object identifier is reused. Rufus converts the varying-length unique identifiers to fixed-size object identifiers for convenience.

The Rufus strategy for unique identifiers works well for data that has an intrinsic unique identifier. For cases where Rufus must rely on the Unix file identification, our scheme works as long as file identity and object identity remain in sync.

Rufus includes a utility for importing data called *rufusbld*. Rufusbld reads a user-written specification that describes the files to be imported, classifies the files, and imports them into Rufus. Rufusbld does little work for files that are already "current" in the Rufus database, so an affordable way to keep Rufus current is to periodically run rufusbld. We decided against a strategy of hooking Rufus into the operating system's file system interface to avoid non-portable, system-dependent programming.

We tested rufusbld on a sample of 1,000 USENET articles. On our IBM RISC/System 6000 model 350, it takes about 130 milliseconds of CPU time per article imported, exclusive of classification. The real time to import is about 2.5 times the CPU time, due to waiting for database disk I/O's.

3.3 Data Model

The Rufus data model represents structured information observed about user data. The structured view describes what the data is, interesting values determined about the data, and what operations can be performed on it. We chose an object-oriented (OO) model [12]. The classes in the hierarchy are recognizable to users as types of information that they use. Rufus creates an object to represent each piece of information that a user might think of as distinct. For example, Rufus creates an object for each file of C source code, as well as an object for the **makefile** (compilation instructions) and an encompassing object for the entire program that refers to the constituent source code, **makefile**, documentation, etc.

A Rufus class is defined by a set of *attributes* associated with each instance of the class and a set of *methods* that can be applied to any instance of the class. Rufus supports substitutability, where instances of a class can be used in places that expect instances of a superclass. This capability allows users to take a specific or general view of a piece of data. For instance, one might seek a document that contains a particular phrase, without regard to the type of formatter used to compose it.

Attributes of type *context* define parts of the underlying real data for text indexing, similar to location restrictions in information retrieval systems like STAIRS [14]. For example, a document class might define the context *abstract* to refer to the words in the up-front summary of a paper. Contexts allow users to restrict the locations that words or phrases must appear in so that more accurate results are possible. For convenience, a class may indicate that particular string-valued attributes are to be indexed as contexts.

Every Rufus class includes a set of standard attributes and methods. The standard attributes include the object identifier, the document identifier (used to cope with new versions of the object in the text index), a unique identifier derived from the underlying object for correlation, the object's class, the date the object was last refreshed, and a string description of the object for browsing. Standard methods *display* an object, *import* an object into Rufus, and *print* an object. The standard methods provide the set of basic services that any Rufus object is expected to support.

Although methods defined for a Rufus class may choose to modify the underlying data, Rufus provides no built-in mechanism for mapping modifications to Rufus objects to the underlying data. Such a mapping would be difficult to provide, given that Rufus objects do not typically model all the information in the underlying data.

Our prototype currently has 34 classes, including a few formats of electronic mail, several formats of documents, C language source code, a few image types, bibtex citations [16], and employee records from IBM telephone books. Figure 2 shows a subset of the supported classes.

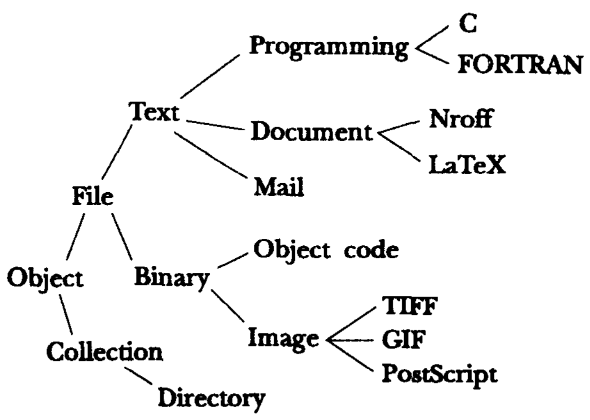

Figure 2: Subset of Rufus class hierarchy

We chose the object-oriented (OO) model because its features closely follow a user's mental model of data. For example, OO data models feature a strong notion of object identity, while other data models are oriented around values. Object identity gives us an easy way to refer to objects in different contexts; in particular, it allows us to organize the same objects in different ways. Object identity is also useful for modeling complex objects made up of simpler ones. Rufus uses the attributes of objects to describe features extracted from the underlying data (e.g., author, title, and date written). Rufus uses the methods of an object to describe both user-visible operations that can be performed, as well as operations needed by the Rufus infrastructure, such as *display* and *import*.

Effective exploitation of the class system requires high quality, detailed class definitions. For example, Rufus 1 extracts the sender, newsgroup, subject, line count, summary, and organization fields from USENET articles, as well as creating links that show the "question and answer" relationship between articles. In contrast, the C source code class does not provide the same level of detail: it only distinguishes between *string* and *comment* contexts and does not distinguish between function definitions and references. As a result, Rufus 1 is more helpful for manipulating USENET articles than for C source.

While the object-oriented model has served our purposes well, some things are difficult to describe in a single-inheritance hierarchy. For example, in Rufus, embedded PostScript is a sub-class of IMAGE, which in turn is a subclass of BINARY. PostScript is textual rather than binary, though. In addition, while most people probably think of embedded PostScript as an image format, others see it as a programming language, requiring different treatment.

Schema changes in our prototype are discouraged because they invalidate existing databases. We note that schema evolution is a common problem in object-oriented database systems.

We are adopting a significantly different OO data model to address the above problems. Our new system uses the conformity data model of Emerald [3] and Melampus [22]. In this data model, only the methods of an object are visible outside its class definition. For convenience, an attribute can be marked so a method will be generated to return its value. In the conformity model, the suitability of an object for a purpose is dictated by whether it implements the necessary method names with the right parameter lists. For example, a user might be looking for objects that have an *Author* and a *Title*. In the system, objects of type LaTeX, TROFF -MM, and SGML might conform to this specification by implementing these two methods.

In the conformity model, inheritance is decoupled from subtyping. Schema evolution is simplified by the resulting independence between class definitions. Inheritance is not used to structure the class hierarchy, but rather as a modularity and reuse aid. When a class definition changes, the old definition of the class will be retained as long as there are object instances of the old class. All new object instances will use the new class definition.

In the new data model, class definitions are machine independent, so that client applications can retrieve class definitions from servers to interpret objects, even on different architecture hosts.

We have a working class compiler for the new data model and modifications to Rufus that fetch and store the new types of objects, keep track of the types of collections, dispatch methods on objects, and execute simple queries. We have so far implemented a few types in the conformity model, including FILE, TEXT, and RFC822 (mail messages).

3.4 Example of a Class Definition

This section presents an example of a Rufus class definition. The definition of the RFC822 class (electronic mail as passed over the Internet) is used. RFC822 is a subclass of MAIL. The table below lists some of the attributes extracted from RFC822 format methods:

Attribute	Data Type	Meaning
length	integer	Length of message
filename	string	File message stored in
messageid	string	Unique identifier
posted	date	Date written
subject	string	Subject of message
to	string list	List of recipients
from	string	Message sender

In addition, RFC822 supports contexts that contain the header fields of the message, the "Subject:" field of the message, the sender of the message, and the body of the message.

The RFC822 class supports several methods, among them:

Method	Meaning
display	Format message for display
edit	Edit the file containing the message
reply	Compose a reply to the message
forward	Forward message to someone else

The "reply" and "forward" methods bring up parts of a pre-existing application to perform these tasks. Users like to locate a message with Rufus, then apply their usual mail-reading tools to it.

In the original Rufus classifier, RFC822 format messages were recognized by the existence of "Received" and "To" fields in the header of the message. In the new classifier, RFC822 format messages are required to have a line beginning with "Received:" or "Delivery-Date:"; other features indicative of the format are lines beginning with "From:" "To:" and "Message-Id." We currently use about 30 samples of RFC822 format messages to train the classifier.

3.5 Structuring Concepts

While it is important for users to be able to understand facts about an individual object, it is also important to understand the relationships between objects. By making these relationships explicit, users are freed from having to know or discover them. Structure information is particularly helpful to applications that support browsing.

Rufus provides collections, object composition, subclassing, and hypertext links [6] to represent inter-object structure. Collections are sets of objects. An object can be in several collections at once. Collections themselves are objects and can be stored in collections as well. For each class, Rufus maintains a collection of all instances of the class, called the *class extent*. Collections are also used to store the results of queries. The objects in a collection can be from arbitrary classes.

Rufus uses object composition to represent complex things made up of other objects. For example, a C program is modeled as the composition of a **makefile**, C source code, and documentation. Rufus complex objects are represented by allowing the use of object identifiers as object attributes.

Subclassing is used in Rufus to indicate the specialization of object types. For example, an implementation might model filesystem directories as a subclass of collections. A filesystem directory does everything that a collection does, in addition to which it has file system attributes like filename, owner, modification date, etc.

Finally, hypertext links model system-discovered and user-specified connections between otherwise unrelated objects. For example, entries in the traditional Unix manual refer to other entries. The import method for Unix manual entries can create links to represent the cross-references. Likewise, a user writing a textual annotation of an image might establish a link to represent the relationship. In Rufus, links are separate objects that point to the linked objects. Fine granularity of link endpoints is achieved using type-specific *selection identifiers*, which are fixed length bit strings that an object's class can convert into a specific part of the underlying native data. In contrast with traditional hypermedia systems, Rufus does not modify the original data to represent links.

3.6 Query Language

The Rufus query language extends content-based access to semi-structured data. Rufus queries combine predicates on the objects extracted from the underlying data with predicates on the underlying data content. Rufus 1 supports simple object predicates and text search.

A Rufus 1 query searches a collection or class extent and returns objects that match a predicate. The predicate contains boolean combinations of conditions on the objects' attributes and text search predicates on the underlying real data. Attribute conditions are simply relational tests against constants, such as **posted > date(12/10/92)**.

More powerful query capabilities would be useful. For example, suppose one were looking for a message written during an electronic mail conversation with a colleague. In order to find the set of messages that comprise the conversation, one would like the query language to be able to follow the links established by the "In-Reply-To" fields of the messages and compute the transitive closure. We chose a simple subset to implement for expediency and to capture the most immediate needs. We are considering a new query language based on the set-oriented operators of the Melampus query language [23] to address these needs.

For text predicates, we implemented *near* (words close to each other) and *adjacent* (words close to each other in the right order). The boolean combinations supported by the query language provide the usual *and*, *or*, and *not*. A special optimization is made to execute text predicates like "*phrase1* **and not** *phrase2*" efficiently. The proximity and boolean operators can be nested to pose queries like **near(adj(San Francisco) earthquake)**.

Rufus supports stemming [27] and flexible capitalization. Stemming is based on a dictionary of 10,000 root forms and allowable stems. Users can state that they wish to ignore capitalization, want an exact match, or want at least the first letter capitalized.

Boolean and proximity text search have been thoroughly criticized [4]. We selected them as our initial text search capability because the results are easily explained to users. We are adding approximate searching based on term weighting [24] and relevance feedback [26] ("find me more documents like *these*") to Rufus.

3.7 Text Indexing

Rufus maintains an index on the text content of imported files to support fast searching on their content. We implemented a text index due to prevalence of textual data. For flexibility, we selected inverted files with word locations. Inverted files support both traditional boolean and proximity searching [27] and term-weighted searching [24]. We used fixed-size small blocks to represent the inverted file so that it can be updated incrementally. For our intended applications, we find that most of the objects indexed are unchanged from day to day, so incremental indexing makes refreshing the Rufus database significantly faster than a complete rebuild. The inverted file can also be updated concurrently with query access. A B-tree is used to store the starting block number of the inverted list for each indexed word.

For the sake of experience, we support two large Rufus databases. One covers a week's worth of Usenet articles (about 35,000); the other covers many weeks of IBM internal bulletin board articles (about 130,000). We found that the text index dominated the size of the Rufus data, the time to refresh the Rufus databases, and query performance. We were able to realize significant improvements with some simple modifications. Further improvement is expected when we change our stored structures.

When we measured the Rufus text index, we found that a huge number of words were being indexed. As a test, we selected 20,000 random articles from the Usenet article base. When indexed, they yielded more than 400,000 unique words and more than 8,000,000 word occurrences. We developed a stop list of the 280 most common words that eliminates 44% of the word occurrences. Random sampling of the vocabulary revealed that many time-stamp based identifiers and meaningless words derived from textual encoding of binary data were being indexed. Refinement of the constructor for the USENET class to avoid indexing such material reduced the vocabulary by one half. This example illustrates the advantage of specializing import according to the data's class.

We took a small random sample of the remaining vocabulary and classified each word by hand. Here's what we found:

Category	%
Proper names	20%
Real words	18%
Machine names, userid's	18%
Program symbols	13%
Misspellings	8%
Addresses, ZIP codes, phone numbers	7%
Other junk	5%
Acronyms	4%
Message-ID's	4%
Codes	3%

In our text database, we also found that the storage scheme of using fixed size blocks suffered from internal fragmentation and poor locality. Due to the distribution of word frequencies, many words have few occurrences. These infrequent words take up an entire small block, wasting space. Other words are more common and take up many small blocks, requiring many seeks to resolve a query. We are replacing our fixed block text inverted list implementation with a variable length block scheme such as that described in [9]. Briefly, this scheme uses small initial blocks, then scales up to larger blocks as the list of occurrences for a word grows. Efficient storage is achieved for infrequent words, while longer word lists are clustered better, reducing seeks.

While textual data is prevalent, indices oriented towards other data types would be useful. For example, the QBIC (Query by Image Content) project [20] at IBM Almaden is working on searching medical images. Their algorithms could be provided in Rufus for image data in addition to the text search capabilities already supported.

3.8 Client/Server

Since users are provided access to much of their data through distributed file systems, the Rufus capabilities described in the foregoing are implemented by servers to provide the same connectivity to data. Each Rufus server mediates access to a single Rufus database. Access to Rufus server functions is provided through a client-callable library of routines. These routines provide the ability to connect to Rufus servers; create, destroy, and modify objects; iterate through collections; invoke methods; and pose queries. In turn, the client library routines invoke functions in the Rufus server using an RPC mechanism.

Rufus servers provide their own concurrency control to allow queries and data import to run in parallel without threatening the physical integrity of the Rufus database. The concurrency control that Rufus wields does not apply to the underlying files. Due to the asynchronous updating of the Rufus database with respect to the underlying data, it is possible to locate files via queries that should no longer match the query predicate. We have considered additional processing to drop query results that should not match due to changes that occurred since import but have not done so. The larger problem is locating query results that now ***should*** be included in the answer but do not due to a stale Rufus database; for that we have no answer without modifying the operating system.

In Rufus 1, client applications can connect to a single Rufus server at a time. This limitation puts the burden on the user to figure out the correct Rufus server to use. We are removing this restriction in Rufus 2 so that clients will be able to connect to several servers at once. Objects in one database will be able to point to objects in other databases. The new system will also be able to access servers supporting other remote protocols, such as WAIS [15] with Z39.50 [2]. Conversely, our system will export a Z39.50 protocol itself, so that its databases can also be searched by WAIS clients. Servers will be able to swap class definitions between themselves, using the same mechanism as is used to inform clients of class definitions.

Servers will be able to publish a summary of the information they store to permit automatic routing of queries to only those servers that might have useful information.

4 Applications

We wrote two applications to demonstrate the Rufus capabilities. **xrufus**, an X-windows [28] application, provides querying, browsing, and operation execution over any of the data types known to Rufus. An object found by querying or browsing is displayed in **xrufus** according to the object's class. In addition, a menu of operations is prepared specific to the type of object. For example, the menu for electronic mail objects contains actions like "reply" and "forward."

Users can create ***buttons*** that represent commonly useful queries. For example, a user might define a "Callup" button that looks up a name in the site telephone book. Then, the user can select a name with the mouse in most X-windows applications, and click the button to run the query.

With more class definitions and integration, **xrufus** could be extended into the "researcher's workbench." Activities like processing mail, reading bulletin boards, program development, document processing, appointment scheduling, and talk preparation could all be provided in a seamless environment. The Rufus infrastructure would help users find and organize their information and to drop into the right applications at each step without having to think about them.

We've also developed an extension of the popular *trn* news reading program [7] called *rufustrn*. Rufustrn works just like trn, in addition to which users can define *virtual newsgroups* that contain all the articles that match a Rufus query. For example, a user might select specific articles from a newsgroup based on content to cut down the number of articles that must be examined. Alternatively, a subject of interest might appear in several newsgroups. This subject can be collected into a single virtual newsgroup for convenience. Rufustrn also allows users to pose queries of "one time" interest and browse the results. A nice feature of rufustrn is that articles are always displayed with the standard trn user interface. The result is a news reader enhanced with query capabilities, rather than a completely new application. The approach of rufustrn differs somewhat from that used in Infoscope [10]. Infoscope defines virtual newsgroups in a DAG structure based on the contents of other virtual newsgroups and of header fields. In contrast, rufustrn defines virtual newsgroups as the result of a query. Rufustrn provides a single mechanism for both one-time queries and for topics of continuing interest.

We envision supporting further applications beyond **rufustrn**. For example, a mixed database of text and multimedia data could allow users to search for film clips by searching through textual descriptions and invoking methods to view related clips. To support such an application, Rufus only needs to have a "film clip" data type added with a method that invokes a video viewer on the user's workstation.

5 CONCLUSIONS

Users are inundated with semi-structured information. Current database systems do not handle such information well. As a result, users are forced to turn to specialized applications that improve access to particular kinds of data. Each specialized application is forced to re-invent and re-implement basic infrastructure to support flexible access. For structured information, database systems provide standard capabilities that make applications easier to write. The same leverage must now be applied to semi-structured information.

The Rufus project has developed an infrastructure based on object-oriented database and text search principles to support applications using semi-structured information. Applications built with the Rufus infrastructure remember key information that users would otherwise be forced to memorize, such as the relationship between files, how to find them, and what to do with them when you find them. Rufus raises the level of abstraction so that users no longer have to deal with their data as simple sequences of characters. We have built a prototype to demonstrate the Rufus ideas and deployed it for use at Almaden. Early experience with the prototype has been promising and has suggested important areas for further work. While we built our prototype on a UNIX system, we expect the Rufus concepts to be useful in other operating system environments as well.

The extensions being made in our new Rufus prototype will support applications on a significantly larger scale. Improvements in the storage structures will support databases with millions of objects. The work in distributed access will free users from specifying where to search for information and will integrate users' environment with information available in external information servers and libraries. The new conformity data model will allow new classes to be written and refined to support new kinds of data.

Acknowledgements Eli Messinger wrote both versions of the Rufus classifier. This paper benefited from the helpful feedback given by the reviewers.

References

[1] A. Aho and M. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6), 1975.

[2] ANSI/NISO, New Brunswick, NJ. *Information Retrieval Service and Protocol*, 1989.

[3] A. Black, N. Hutchinson, E. Jul, and H. Levy. Object structure in the Emerald system. In *Proceedings of ACM Conference on Object Oriented Programming Systems, Languages and Applications*, September 1986.

[4] D. Blair and M. Maron. An evaluation of retrieval effectiveness for a full-text retrieval system. *Communications of the ACM*, 28(3), March 1985.

[5] Bolt, Beranek, and Newman, Inc., Cambridge, MA. *BBN/Slate Topics Manual*, 1990.

[6] J. Conklin. Hypertext: An introduction and survey. *IEEE Computer*, 20(9), 1987.

[7] W. Davison. *TRN—Threaded Read News Program*, 1992.

[8] O. Deux et al. The O2 system. *Communications of the ACM*, 34(10), October 1991.

[9] C. Faloutsos and H. Jagadish. On B-tree indices for skewed distributions. In *VLDB Conference*, 1992.

[10] G. Fischer and C. Stevens. Information access in complex, poorly structured information spaces. In *Proceedings 1991 CHI Conference*, 1991.

[11] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and Jr. James W. O'Toole. Semantic file systems. In *SOSP91*, October 1991.

[12] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Publishing Company, 1983.

[13] D. Goldberg, D. Nichols, B. Oki, and D. Terry. Using collaborative filtering to weave an information tapestry. *Communications of the ACM*, 35(2), 1992.

[14] IBM. *STAIRS General Information Manual.*

[15] B. Kahle and A. Medlar. An information system for corporate users: Wide area information servers. Technical Report TMC-199, Thinking Machines Corporation, Cambridge, MA, 1991.

[16] L. Lamport. LaTeX*: A Document Preparation System*. Addison-Wesley Publishing Company, 1986.

[17] Eliezer Levy and Abraham Silberschatz. Distributed file systems: Concepts and examples. *ACM Computing Surveys*, 22(4), February 1990.

[18] T. Malone, K. Grant, F. Turbak, S. Brobst, and M. Cohen. Intelligent information-sharing systems. *Communications of the ACM*, 30(5), May 1987.

[19] B. McCune, R. Tong, J. Dean, and D. Shapiro. RUBRIC: A system for rule-based information retrieval. *IEEE Transactions on Software Engineering*, 11(9), September 1985.

[20] W. Niblack, R. Barber, W. Equitz, M. Flickner, E. Glasman, D. Petkovic, P. Yanker, C. Faloutsos, and G. Taubin. The QBIC project: Querying images by content using color, texture, and shape. *SPIE 1993 International Symposium on Electronic Imaging: Science & Technology, Conference 1908, Storage and Retrieval for Image and Video Databases*, February 1993.

[21] Object Design, Inc., Burlington, MA. *ObjectStore User Guide*, 1991.

[22] J. Richardson and P. Schwarz. Aspects: Extending objects to support multiple, independent roles. In *Proceedings of ACM SIGMOD Conference*, 1991.

[23] J. Richardson and P. Schwarz. MDM: An object-oriented data model. In *Proceedings of the Third International Workshop on Database Programming Languages*, August 1991. Also available as IBM Research Report RJ 8228, San Jose, CA, July 1991.

[24] S. Robertson and K. Sparck Jones. Relevance weighting of search terms. *Journal of the American Society for Information Science*, 27(3), 1976.

[25] J. Rosenberg, C. Everhart, and N. Borenstein. An overview of the Andrew Message System. In *Proceedings of SIGCOMM '87 Workshop*, August 1987.

[26] G. Salton. *Relevance Feedback and the Optimization of Retrieval Effectiveness*, chapter 15. Prentice Hall, 1971.

[27] G. Salton. *Automatic Text Processing: The Transformation, Analysis and Retrieval of Information by Computer*. Addison-Wesley, 1989.

[28] R. Scheiffler and J. Gettys. The X window system. *ACM Transactions on Graphics*, 5(2), April 1986.

[29] N. Yankelovich, B. Haan, N. Meyrowitz, and S. Drucker. Intermedia: The concept and construction of a seamless information environment. *IEEE Computer*, 21(1), January 1988.