[Ter90] D. B. Terry, *7 Steps to a Better Mail System*, Xerox Palo Alto Research Center Technical Report CSL-90-12, September, 1990.

[Unix-a] *Grep*, in The Unix Programmer's Manual.

[Unix-b] *Magic*, in the Unix Programmer's Manual.

[Unix-c] *Shmap*, in the Unix Programmer's Manual.

[Vis89] *Looking Glass: The Desktop Manager for Unix Workstations*, Visix Software, Inc., Arlington, VA 1989.

[YHM88] N. Yankelovich, B. Haan, N. Meyrowitz and S. Drucker, *Intermedia: The Concept and the Construction of a Seamless Information Environment*, IEEE Computer 21, 1 (January 1988).

[Wur89] R. S. Wurman, *Information Anxiety*, Doubleday, New York City, February, 1989

[IBM] *The ST AIRS General Information Manual*, IBM Publication GH12-5107.

[IXI89] *X.desktop Overview*, IXI Limited, Cambridge, England, 1989.

[JaK84] M. Jarke and J. Koch, *Query Optimization in Database Systems*, ACM Computing Surveys 16,2 (June 1984).

[KaM91] B. Kahle and A. Medlar, *An Information System for Corporate Users: Wide Area Information Servers*, Thinking Machines Corporation Technical Report TMC-199, Cambridge, MA, 1991.

[KeR78] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1978.

[MGT87] T. W. Malone, K. R. Grant, F. A. Turbak, S. A. Brobst and M. D. Cohen, *Intelligent Information-Sharing Systems*, Communications of the ACM 30, 5 (May, 1987), 390-402.

[MTD85] B. P. McCune, R. M. Tong, J. S. Dean and D. G. Shapiro, *RUBRIC: A System for Rule-Based Information Retrieval*, IEEE Transactions on Software Engineering SE-1 1, 9 (September, 1985), 939-945.

[Obj91] *ObjectStore User Guide*, Object Design, Inc., Burlington, MA, 1991.

[Oss76] J. S. Ossanna, *NROFF/TROFF User's Manual*, Computer Science Technical Report 54, Bell Laboratories, Murray Hill, N, October 1976.

[Pee91] J. Peek, *MH & xmh: E-mail for Users & Programmers*, O'Reilly & Associates, Inc., January, 1991.

[REB87] J. Rosenberg, C. F. Everhart and N. S. Borenstein, *An Overview of the Andrew Message System*, Proceedings of the SIGCOMM '87 Workshop: Frontiers in Computer Communications Technology, Stowe, VT, August, 1987.

[RiS91] J. Richardson and P. Schwarz, *MDM: An Object-Oriented Data Model* (Extended Abstract), Procedings of the 3rd International Workshop on Database Programming Languages, Nafplion, Greece, August, 1991.

[Ros85] C. Rose, et. al, *Inside Macintosh, Vol. III*, Addison-Wesley Publishing Company, Reading, MA, 1985.

[Sal89] G. Salton, *Automatic Text Processing*, Addison-Wesley Publishing Company, Reading, MA, 1989.

[ScG86] R.W. Scheifler and J. Gettys, *The X Window System*, ACM Transactions on Graphics 5,2 (April, 1986), 79-109.

[Sey88] P. B. Seybold, *Metaphor Computer Systems: A Quiet Revolution*, Patricia Seybold's Office Computing Report 11, 8 (August, 1988).

[Shn87] B. Shneiderman, *User Interface Design for the Hyperties Electronic Encyclopedia*, Proceedings of Hypertext '87, University of North Carolina, Raleigh, NC, 1987.

[SmS77] J. M. Smith and D. C. P. Smith, *Database Abstractions: Aggregation and Generalization*, ACM Transactions on Database Systems 2, 2 (June, 1977), 105-133.

[SWK76] M. Stonebraker, E. Wong and P. Kreps, *The Design and Implementation of INGRES*, ACM Transactions on Database Systems 1, 3 (September, 1976), 189-222.

[Tel90] *TeleUSE Reference Manual*, Telesoft AB, Sweden, 1990.

[Tem90] B. Templeton, *ClariNet Electronic Newspaper Manual*, ClariNet Communications Corp., Waterloo, Ontario, Canada, 1990.

[Ter89] D. B. Terry, *Weaving an Information Tapestry to Replace Electronic Mail*, Xerox Palo Alto Research Center, July, 1989.

Rufus provides an explicit framework in which the structure and semantics of a user 's data is described. Formerly , this knowledge was either buried in application programs or kept in the user 's mind. The description of a user 's file types includes syntactic conventions (e.g., how to locate and parse the dimensions of an image file), as well as actions that can be invoked on the data (e.g., how to format and print various types of documents). Rufus represents this knowledge in a common, object-oriented data model, making it available for both users and applications.

Rufus also provides the means for representing the meta-structure and organization of data. These means include the classification of objects by type, collections of objects, inter -object links, and large objects built up from smaller objects. Rufus provides the tools for building these organizations plus the flexibility to view objects in multiple organizations.

The Rufus prototype includes a general purpose X-windows user interface that demonstrates these concepts. It includes windows to classify and import new data, browse data, submit queries, and browse the results. Future work on Rufus will focus on more sophisticated querying, data modeling, and organizing; streamlined extensibility; active agents, and multi-user sharing of information.

Bibliography

[ANS89] *Information Retrieval Service and Protocol* , ANSI/NISO Z39.50-1988, T ransaction Publishers, New Brunswick, NJ, 1989.

[BCG87] J. Banerjee, H. Chou, J. Garza, W . Kim, D. W oelk, N. Ballou and H. Kim, *Data Model Issues for Object-Oriented Applications* , ACM T ransactions on Office Information Systems 5, 1 (January 1987), 3-26.

[Bol90] *BBN/Slate System T opics Manual* , Bolt Beranek and Newman, Inc., Cambridge, MA, 1990.

[CHR90] L. Cabrera, L. Haas, J. Richardson, P . Schwarz and J. Stamos, *The Melampus Project: T oward an Omniscient Computing System* , IBM Research Report RJ7515, June, 1990.

[Cod91] W. Cody , *The QBIC Project* , IBM Almaden Research Center , San Jose, CA, 1991.

[Con87] J. Conklin, *Hypertext: An Introduction and Survey* , IEEE Computer 20, 9 (September , 1987), 17.41.

[Cro82] D. H. Crocker , *Standard for the Format of ARP A Internet T ext Messages* , RFC #822, August, 1982.

[CYD88] S. K. Chang, C. W . Yan, D. C. Dimitrof and T . Arndt, *Intelligent Image Database System* , IEEE T ransactions on Software Engineering 14 (May 1988), 681-688.

[Fuk90] K. Fukunaga, *Introduction to Statistical Pattern Recognition* , Academic Press, Inc., San Diego, CA, 1990.

[Gol84] A. Goldberg, *Smalltalk-80: The Interactive Programming Environment* , Addison-W esley Publishing Company , Reading, MA, 1984.

[Gro84] W. I. Grosky , *Toward a Data Model For Integrated Pictorial Databases* , Computer V ision, Graphics and Image Processing 25 (1984), 371-382.

[GWJ91] A. Gupta, T . E. W eymouth and R. Jain, *Semantic Queries W ith Pictures: The VIMSYS Model* , Procedings of the Conference on VLDB, Barcelona, Spain, September , 1991.

[HCL90] L. M. Haas, W . Chang, G. M. Lohman, J. McPherson, P . F. Wilms, G. Lapis, B. Lindsay , H. Pirahesh, M. Carey , and E. Shekita, *Starburst Mid-Flight: As the Dust Clears* , IEEE T ransactions on Knowledge and Data Engineering, March 1990, 143-160. Also available as IBM Research Report RJ7278, San Jose, CA, Jan. 1990.

[HaJ84] D. Haderle and R. Jackson, *Introduction to IBM Database 2* , IBM Systems Journal 23, 2, 1984.

fleshing out the behavior of a class is a combination of creating links to existing application programs (e.g., the *reply* method for Mail messages invokes the existing *mh* mail system [Pee91]) and creating new methods from scratch (e.g., the hexadecimal previewer for Binary files).

Extensibility is an issue with the system's applications as well. The prototype contains a single application, built on top of Rufus-system primitives. No attempt has been made to provide a usable interface to the underlying system functions. Since one of the goals of the system is to provide a centralized understanding of a user's data for application use, a more friendly application toolkit needs to be developed on top of the core system.

Graphical interfaces will be needed at all levels of the system. At the base level, modification of the class hierarchy should be presented to a user as simply as possible. A multi-window text-based system such as used in Smalltalk [Gol84] would be a good start. At the user level, a graphical interface for posing queries (as well as manipulating the results), such as found in the Metaphor system [Sey88], would make the system more usable to non-Computer Scientists.

Implementation and Performance

The performance of the prototype is disappointing. While the speed of the interactive portions of the system (e.g., query, display, navigation) is adequate, the rate of importation of new files is not. Sample measurements show importation of new files at a rate of 2 files/second. Of this time, 75% goes to text index maintenance, 20% to object creation and attribute index maintenance, and 5% to file parsing. The performance problems are primarily due to implementation shortcuts, and could be readily improved by more careful coding.

Using a more fully-featured, performance-oriented database system (DBMS) such as Starburst [HCL90] or ObjectStore [Obj91] could also improve the system's performance. This would also provide the traditional benefits of a DBMS such as recovery, multi-user function (including concurrency controls), and more powerful queries (e.g., "joins"). Moving Rufus to a new DBMS will require either that the new system provides a superset of the functionality implemented in the Rufus object-oriented database (i.e., the class hierarchy, the object store, the query engine, etc.), or that the DBMS has an open architecture in which this functionality could be integrated.

Further Enhancements

While the prototype browsing application is useful for user-directed searches, "active" applications that operate in the user's stead would be valuable in coping with massive data flows (e.g., news, mail, etc.). An organizational assistant, akin to the intelligent agent of Terry's *Tapestry* system, might automatically organize a user's data, in addition to partitioning and prioritizing his mail. A subsystem to automatically alert a user as to changes in the underlying data (e.g., new mail) or timely events (e.g., meetings gleaned from the user's on-line calendar) would also be very useful.

Looking further ahead, issues of a distributed Rufus system are intriguing. Users will desire access to both local data, on their workstation or local-area network, as well as data that is completely outside of their administrative control. This latter category includes data available from other users' workstations, inter-enterprise data, and data available only through an operative interface (e.g., such as that found in the CompuServe[10] system).

7. Conclusions

Rufus helps users organize and manage existing data. Rufus focuses on existing data because it is impractical to require users and information providers to convert data into a new "normal form." At best, this data conversion requires extensive modification of user's current applications; at worst it renders the applications unusable.

---

10. CompuServe ™ is a trademark of Compuserve Information Services, Inc.

While the system can model composite objects (i.e., objects composed of subobjects such as a Tele directory composed of Employee subobjects), the prototype is not particularly helpful in handling them. The query language contains no provisions for posing queries against subobjects of a composite object, and the display system is not particularly graceful at showing them.

In use, it is clear that the depth with which the class attributes and methods model the semantics of the underlying data contributes greatly to the overall usability of the system. Classes that are more fully articulated in the schema make the system as a whole "feel" better .

Browsing and Organizing Data

As expected, the introduction of multiple access paths to a user 's data is very valuable. Combining query access, class extents (which automatically group all objects of a particular class) and user - defined collections provides a user with a number of ways to locate particular files. Producing collections as the result of queries easily allows users to refine the result in one or more ways. On the negative side, the collections themselves can quickly inundate a user in a flood of new data. In general, Rufus needs to be more able to "browse itself." For example, browsing and query mechanisms should be able to operate on the underlying Rufus metadata (e.g., "Which classes support a method call named *print* ?").

The lack of organizational tools and, to a lesser degree, the prototype' s handling of user specified collections, are a problem. The single-threadedness of the prototype is also a concern. Currently users can only browse a single collection and object at a time. This is at odds with the prevailing multi-processing multi-window environment in which Rufus operates.

Also missing from the prototype is a Link class for representing hypermedia links. A paper design for non-intrusive links (i.e., links that do not alter the underlying data) has been completed and its implementation is planned. Once the Link class has been implemented, the Rufus data import methods will be enhanced to perform some automatic link discovery . For example, the import method for Unix manual pages will automatically link the UnixMan object to those of related manual pages. Automatic discovery of hypermedia links in existing data is an area for future research.

Extensibility

Another shortcoming of the prototype is the lack of user extensibility . While "system programmers" can extend the system with relative ease, no user -level tools are provided for augmenting or adding a new class. This can be particularly frustrating to users who find a necessary behavior (e.g., a method for formatting and directing a document to a printer) missing.

Fleshing out the class hierarchy would be an easier task if a more easily programmed front-end were developed. Rather than requiring the skills of a Rufus systems programmer to include a new file type, users would be able to add them themselves. This will be especially important as the tool is distributed to users in different workstation environments. As the hierarchy is enlarged, the poverty of the prototype' s data model will become more apparent. Basing the system on a more fully developed data model, such as that being developed for the Melampus project [RiS91] will be necessary .

The bulk of the effort of adding a new data type is concentrated in three areas: the classifier , the import method, and type-specific methods. The current form of the classifier often requires the implementor to have a deep understanding of a file type' s peculiarities to successfully integrate it into the existing classification scheme. Adding a file type with very small differences from an existing type (e.g., a new mail format) can be quite challenging. This problem will compound as the hierarchy grows. Other classification techniques, particularly those based on trainable, statistic-based architectures are likely to prove more workable in the long run [Fuk90].

Writing an import method that extracts attributes from a native data file requires a deep understanding of the file type' s structure. In some cases (e.g., Vnet mail), the structure is not wholly defined, and the import method must be reworked as new twists on the format are discovered. Finally ,

6. Experience and Future work

The prototype system and application have been in use for several months, both for demonstrations as well as personal use by the authors and their colleagues. Overall the sample system has proven very useful, even considering the limited nature of the implementation. In this section the positive and negative aspects of the prototype are explored from both users' and implementors' perspectives.

Considering the small number of data types that are implemented and the limited detail for each one the prototype is surprisingly useful. The small type set is partially offset by the inclusion of very general types such as File. Thus even file types that aren't recognized explicitly can be manipulated in basic ways. For example, text files can be searched-for , displayed, edited and printed.

Users new to the workstation environment (and hence unfamiliar with the specific file types) find the data abstractions and behavioral encapsulations to be especially helpful. Familiar entities such as Document or Mail provide an understandable abstraction of more specific (and arcane) classes such as TeX or RFC822. Similarly , the coalescing of behaviors for various classes (e.g., the *x-display* methods for TeX documents, GIF images, UnixMan manual pages, etc.) behind a single label permits users to perform basic operations on objects without having to understand the underlying details of each data type.

The Query Interface

As with other text retrieval systems, the combination of a text index across a broad range of user data and simple single-variable queries provides a useful facility for locating files. With the addition of data-specific attributes, integrated across data types, the query interface becomes much more powerful. For example, the *from* attribute of the Mail class allow users to search across a range of mail formats (i.e., RFC822, Vnet, Usenet) for a semantic entity (e.g., the sender of the mail) without having to understand the particular syntax of each mail format.

While single variable queries prove quite useful, it is readily apparent that more complicated multivariate and recursive queries are required. Data model features such as sequenced attributes (e.g., a single attribute of Troff document objects contains a sequence of pre-processors used for formatting) amplify the power of single-variable queries, but do not obviate the need for powerful query mechanisms. Conceptually simple queries such as "Find all object modules whose corresponding source module has been modified since the object module was last compiled" cannot be framed with a single-variable query . Similarly , queries such as "Find all employees who work directly or indirectly for Mr . Dithers" cannot be posed without a recursive query facility .

It might also be useful to be able to pose less specific queries such as "Find all Mail objects with any attribute of type *date* equal to July 13, 1936 [9]," without having to specify the particular attribute to match. Queries across attribute types, such as "Find all Mail objects with an attribute containing July 13, 1936" might match *date* attributes as well as *text* attributes that contain the given string.

Data Modeling

The simplicity of the Rufus data model has turned out to be a mixed blessing. On the positive side the model is easily learned by new implementors. Adding a new class to the schema is a straightforward operation, and the description of a particular class is relatively easy to read and understand. On the negative side, the model is sometimes strained in trying to capture the detail of a user 's data. The single-inheritance structure of the hierarchy is unable to model certain data types cleanly . Additionally , the system is not adept at handling multi-file objects such as C Programs or MH mail folders. The classifier has a few patches on it that permit it to recognize multi-file semantic entities, but this is not a long-term solution.

9. July 13, 1936 is the date on which Dagwood Bumstead first crashed into Beasley the postman.

5.4. A Sample Rufus Application

A general application was written to help users classify, search, browse and act upon their files. The application is an X-windows client, and was implemented with the TeleUse user-interface development system. The application is designed to be left on a user's X-window desktop at all times, being picked up to search for files or browse collections when needed. As with the rest of the Rufus system, the application does not interfere with a user's use of their regular tools.

The application's central "control window" allows a user to open and close the four central function windows:

1. **Classify and Import.** A user selects files either by typing their names or selecting them from a menu. The system then classifies the selected files and imports them into the Rufus object store. Directory objects are automatically explored recursively, classified and imported.
2. **Text Search.** A user specifies a text search with a predicate (see section 5.2) and a collection to be searched. Additionally, a user may specify what *context* the text should be found in, whether the search is case-sensitive, and how close the adjacency operator should measure. The objects satisfying a query are placed in a collection that can be browsed or used as the source collection of further queries.
3. **Context and Attribute Search.** A user creates a *filter* through which a selected collection is passed. A filter is a conjunction of predicates on attributes of a selected class. Each base type (e.g., Integer, Date, etc.) allows a type-specific set of comparison operators that can be used to construct the individual predicates.
4. **Collection and Object Browse.** A user scrolls through a list of collections currently in the object repository. This includes class-extents, user-defined collections and collections produced as the results of queries. Once a collection has been selected, a user may browse any of the objects within.

   The object browser provides a rudimentary display of the object and its attributes and allows a user to invoke class-specific methods on the object. For example, when browsing a Mail object, a user can invoke *reply*.

The system is usually initialized with a user's home directory in its database. For performance reasons this is done off-line with a batch-oriented tool. The batch tool is also used for refreshing (i.e., updating) the database. Refreshing of the database is typically performed automatically overnight. During the course of the day the user brings additional files and directories into the system via the classify and import window.

Once data has been imported into the system, a user can scroll through a list of collections, select collections for browsing, and select specific objects within a collection for browsing. From the object browser, a user will either scroll through the text of an object or invoke specific actions (e.g., document objects might be processed for printing on a laser printer). Additionally a user may trace backwards from an object to its location in the underlying directory structure.

Searches are typically conducted as a series of refinements, using the output of one search as the input domain of the next. The application makes this easy by forming the output of a search as a collection and allowing any collection (e.g., the output of a search, a type extent, a Directory collection, etc.) to serve as the input domain of a search.

The prototype application was designed to expose a maximal amount of the underlying Rufus core functionality in a user friendly way. It succeeds in demonstrating the system's ability to identify, import and search across a broad array of user data types. The application provides a powerful, yet easy-to-use interface to the underlying query and browsing mechanisms.

File metadata, such as a filename extension (e.g., ".c" on a C-language source module), is also evaluated in the recognition functions.

The second step recursively invokes the classifier on each of the subclasses of the current class. The subclass that returns the maximal positive weight is saved along with the weight for the final computation.

The third step matches the abstract's hash table of scanned tokens against a pre-computed table of class-specific keywords. The tables are constructed by a Rufus tool that searches a set of sample files for common keywords. These sets are then manually reduced to keywords that are uncommon among the other types in the system. This is a simple version of the sort of single-term indices used in information retrieval systems [Sal89]. The output of this phase is a weight relating the percentage of keywords in the table that are found in the document token sample.

Once the three steps have been completed, the classifier computes a weighted average of the non-zero weights and returns it, along with a class identifier, to the caller. The class identifier is either a subclass (if there are any, and if any produced a suitably large weight in their identification), or the current class. This process terminates when the recursive traversal returns a weight and class pointer to the classifier's caller.

Thresholds are used to short-circuit traversal of portions of the hierarchy that are unnecessary. For example the recognition functions can unilaterally reject an abstract, causing the subtree rooted at the current class to be skipped. This is useful, for example, in avoiding the evaluation of binary file abstracts against text-file classes. Other thresholds allow for the unilateral identification of an abstract based on the values returned by a class's children or based on a high-degree of matching in the keyword test.

There are many subtleties to the classification system that are not evident from this cursory overview. Most notable is the difficulty of devising weights, especially in the recognition functions, that provide meaningful relationships across classes. For example, recognition functions for different classes might use vastly different weights within the (0, 1) range to indicate "probable" identification. Because of the ad hoc nature of the recognition functions, the weights often must be balanced through extensive testing and readjustment.

Another subtlety is the need for a *conditional acceptance*. This is used when a recognition function has reason to reject an abstract, but might find it contradicted by subclass or keyword evidence. Rather than rejecting the file unilaterally (thus abandoning the entire subtree rooted at this class), a class may continue processing it, rejecting it only if none of the subclasses identify it. For example, in the case of the Binary class, a file that is textual cannot be rejected unilaterally, as the PostScript subclass might identify it.

Experience with this classifier has shown it to be a reasonable system for a relatively small, stable set of types. It is not, however, amenable to extension. The type-independent scanner can drop critical information, which must be found with the class-specific identification functions. As more classes are added the classifier becomes more dependent on these functions and less on the more general keyword mechanism. This is lucky, as calculating keyword sets becomes something of an art when relatively similar classes (e.g., different mail formats) are added to the hierarchy.

An even more obvious problem is the lack of coherency in the weighting scheme. Adding a new class with an identification function requires the implementor to analyze the existing functions, pick prospective weights, and then iteratively tune the weights (both new and old) on a variety of sample data. Adding a new class often causes previously balanced identification functions to be overruled, sometimes erroneously.

As discussed further in section 6, a more objective statistical approach to classification might provide a classifier that is more amenable to extension.

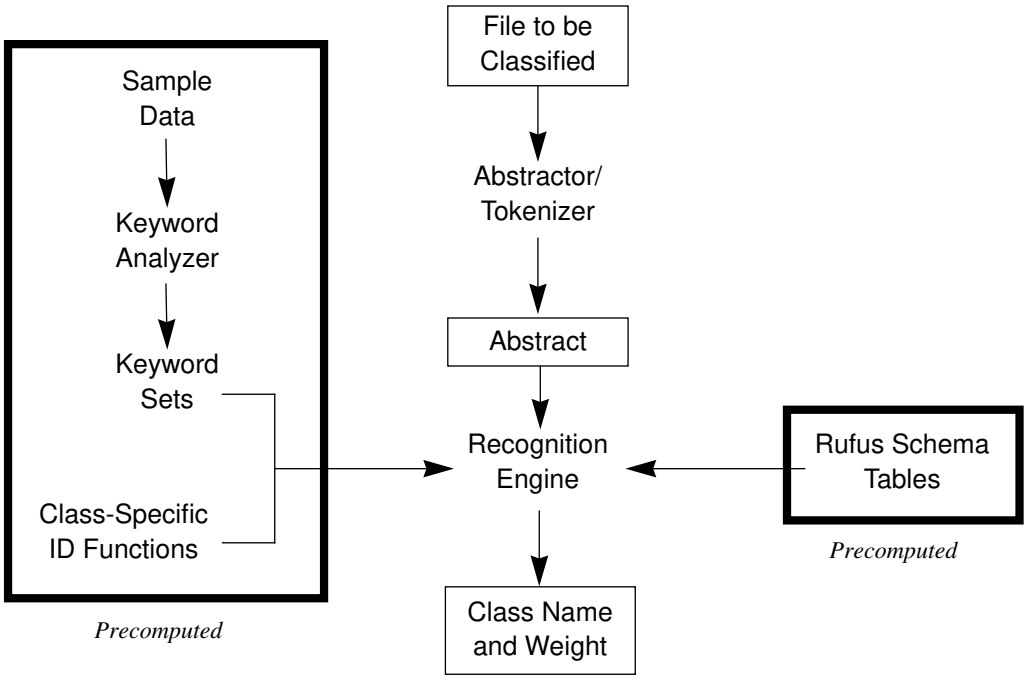

Figure 4 -- Block Structure of the Rufus Classifier

The prototype creates abstracts of textual files that include both metadata about the file (e.g., the filename, system permissions, etc.) and a hash table of tokens sampled from the native file. For binary files only the metadata is included. Initial versions of the classifier tokenized text files from beginning to end, but it quickly became apparent that this was both too slow for long files, and unnecessary . Due to the repetitive nature of most file types, a token sampling scheme was introduced.

The scanner samples tokens (delimited by a fixed punctuation and white-space set) from the beginning, middle and end of the file. This process gathers representative tokens without scanning the entire file. The beginning and end often contain identifiable declaratory tokens, e.g., “ `\begin`” “`\end`” token pairs for TEX documents. Tokens extracted from the middle are useful for identifying files such as source code that have type-unspecific English language comments at the beginning and context-ending punctuation (e.g., “ `}`”) at the end.

Once the file has been scanned, the abstract is passed to the recognition engine. The engine classifies the abstract by recursively traversing the Rufus class tree in a depth first search. The search begins with the File class. At each node in the tree, the classifier gathers three types of evidence in order:

1. The weight returned by an optional class-specific recognition function.
2. The weights returned by the children of the current class.
3. The results of a class-specific keyword search.

In the first step, the recognition functions look for information lost in the scanning phase as well as for file metadata. For example, the opening “ `%!`” character sequence of a PostScript[8] file is lost in the scanner as a string of punctuation. Similarly , the “magic strings” of many Unix binary file formats [Unix-b] are not scanned at all and are instead located directly in the file by the recognition functions.

8. PostScript® is a registered trademark of Adobe Systems, Inc.

AIX file mapping [6] and the B-tree package used by Rufus, a conventional buffer -pool is used to cache index pages.

A full text index is built automatically for all *string* attributes and *contexts* that are defined by an object's class. [7] Additionally, class definitions may specify that an index be maintained for any or all of the class's other attributes. These indices are used by the query engine to optimize the execution of queries posed against the database. Objects may be fetched either by query access paths, or directly by object-id.

A query is specified in three parts: a *source collection* to search, an optional *search class*, and a *predicate string*. The result of executing the query is the collection of all objects in the source collection that satisfy the predicate string and are of the specified search class or one of its subclasses. The default search class is the nearest ancestor class to the classes represented in the source collection. That is, the default is the most specific class for which each class represented in the source collection is a subclass.

The predicates are connected with the logical operators *and*, *or*, and *not*, and optionally grouped with parentheses. The individual predicates are simple conditionals on individual object attributes: *attrib relop value*. The relational operator may be any of the standard operators (e.g., = $\neq$ < >) as well the text-search operators for word ordering and proximity. To execute a query, an application makes a Rufus system call that compiles the textual specification of the query into an internal form. In addition to the three parts of the query described above, the internal form also contains the current state of the search within the specified collection. This internal query form is passed, usually inside an iterative loop, to a Rufus system function that generates the next object in the result set.

The query compilation process translates the predicate string into disjunctive-normal form (DNF). Given the DNF for a query, such as

$$(A \vee B \vee C) = (a_1 \wedge a_2) \vee (b_1 \wedge b_2 \wedge b_3) \vee (c_1)$$

the system solves each individual disjunct (i.e., A, B and C) and then merges the results. Each disjunct is calculated by iteratively testing the objects from an *initial collection* against the individual predicates (i.e., the a_i's, b_i's and c_i's). The disjunct computations are optimized by the system in two ways. First, the initial collection for each disjunct is automatically selected from a number of alternatives (e.g., the specified source collection, the result of a text search predicate within the particular disjunct, the type extent of the search class, etc.). Second, the order in which the individual predicates are evaluated is selected using standard cost-based optimization methods [JaK84].

5.3. Classification and Extraction of a User's Data Files

At the front end of the Rufus prototype is a subsystem that automatically identifies the class of a user's file. Given a file name, the classifier returns a pointer to the class in the File subtree (see Figure 2) whose characteristics most closely match that of the candidate file. Figure 4 provides an overview of the classification system.

Since the classifier is often dispatched non-interactively on a user's data (i.e., for the initial importation of all of a user's files), accuracy is its primary goal. Additionally, since the classifier can also be operated interactively by a user, it must be able to classify individual files nearly instantaneously.

The speed of the classifier is improved by having it classify a system generated reduced, abstract version of the input file rather than the full text of the original file. By reducing the file to its essentials, the later stages of the classifier have fewer bytes to process. The use of abstracts introduces a tension between speed and accuracy: a smaller abstract is analyzed more quickly, but retains less information about the original file.

---

6. AIX 3.1 limits a process to 13 memory mapped files.

7. The index is roughly the same size as the text being indexed. Further performance figures are given in section 5.4.

```
Mail {
      parent Text
      comment                                  "General Mail Message Class"
      attributes {
            to            string sequence     "Who the message is to"
            from          string              "Who the message is from"
            subject       string              "Subject of message"
            posted        date                "When it was mailed"
            headers       context             "Headers of the message"
            body          context             "Body of mail message"
            quoted        context             "Included (quoted) text"
      }
      classmethods {
            parse         Mailparse
      }
      methods {
            display       Maildisplay
            reply         Mailreply
            forward       Mailforward
      }
}
```

Figure 3 -- Sample Rufus Class Definition

made for pragmatic reasons. Object-oriented data model research is still considering how to best handle such considerations.

An object is created by specifying a class and an initial set of attribute values. A unique object-id is assigned at object creation and is associated with the object forever. The set of attribute values is referred to as the object's *instance record.* The instance record can be fetched from secondary storage, examined, modified, and stored back as needed.

A Collection object represents a set of Rufus object id's. A collection may be heterogeneous and non-exclusive. That is, a collection may contain objects of any number of different classes, and a single object may reside in any number of different collections. Rufus automatically creates a *type-extent* [BCG87] for each class, which is a Collection that contains all instances of that class.

5.2. The Rufus Object-Oriented Store and Query System

The Rufus objected-oriented store is a disk-based system for storing and retrieving Rufus objects. The store maintains indices to expedite value-based object access. Rufus objects are stored on disk as flattened versions of the in-memory instance records. Attributes of variable-length types (e.g., text strings) that are stored in a run-time heap are appended to the instance record for disk storage. These attributes are unpacked into heap storage when the object is retrieved.

Rufus does not explicitly cache object instances. Object caching is handled by the underlying AIX virtual memory system, while index caching is handled internally by the Rufus buffer pool. The difference is due partly to system design and partly to AIX system limitations. The object store uses an AIX memory mapped file structure [Unix-c] that allows for efficient system caching. Due to limitations in

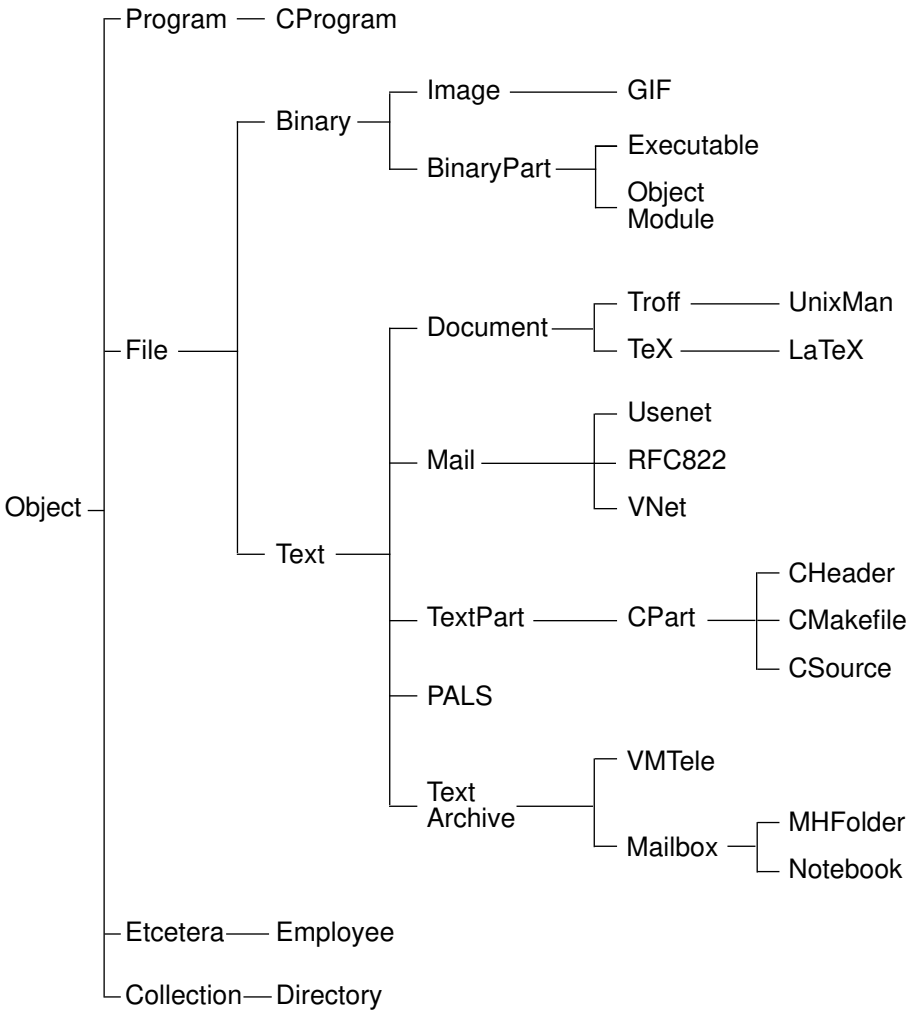

Figure 2 -- The Rufus Prototype's Class Hierarchy

native data itself is not copied into the object. The text pointed to by Context attributes is automatically indexed by the Rufus storage system. Rufus provides functions to the user for posing queries against the underlying native text. Contexts differ from String attributes primarily in the former 's ability to represent multiple, discontiguous entities. For example, a source module of a program usually has several non-contiguous comment blocks in it; these combined blocks constitute a *Comment Context* . Contexts are particularly useful for framing queries on textual data, as discussed further in section 5.4.

Methods can be declared either simply or as class methods. The latter are invisible outside of the class, and are for private, class-internal use only . An example of this would be utility routines related to the parsing of a file during importation.

In addition to those attributes and methods declared by the class, a class inherits the attributes and methods of the class' s parents. A class definition may optionally redefine an inherited method, but may not redefine an inherited attribute. The prototype' s type system does not do any type checking on the parameter or return value types of redefined methods. These simplifying assumptions were

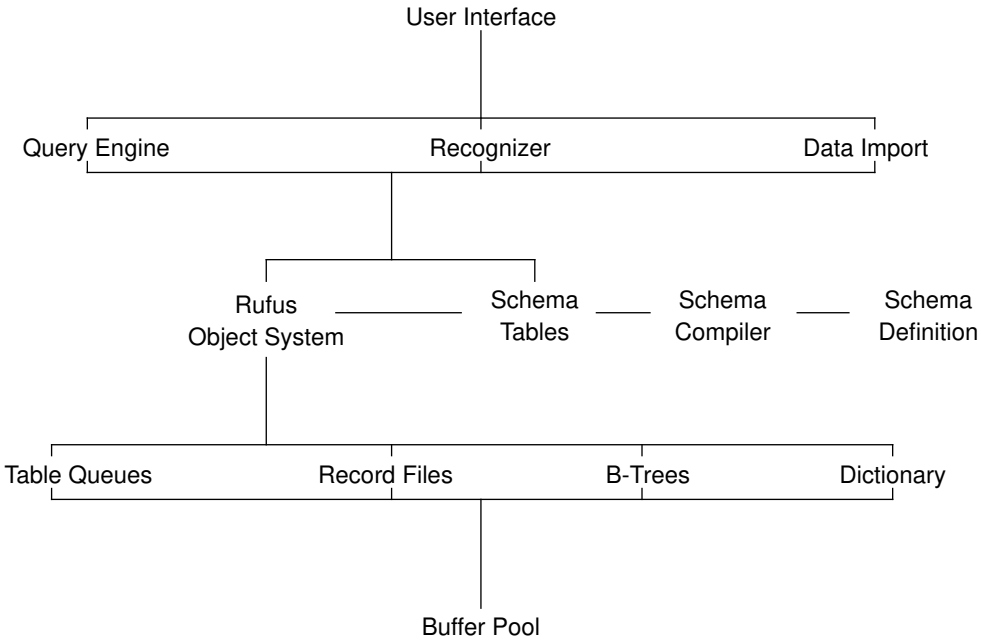

Figure 1 -- System Structure of the Rufus Prototype

Module	**Lines of Code**
Type System	2400
OO Storage System	11000
Query Engine	4800
Classifier	1800
Import Methods	4300
User Interface	12200 †
Total	36,500

† The user interface consists of 3900 lines of C, 5200 lines of D [T el90] and 3100 lines of code produced by a graphical user -interface builder .

Table 1 -- Code Size of the Rufus Prototype

5.1. The Rufus Data Model

The Rufus system is built on top of a simple single-inheritance object-oriented data model. The prototype's class hierarchy is shown in Figure 2. The class schema is defined statically in a C-like data definition language. A sample class definition is shown in Figure 3. Each instance of a class has a set of attributes and methods. Attributes are statically typed as either Integer , Real, String, Date, Object-ID or Context. An attribute can also be declared to be a *sequence* of any one of the base types except Context. Any attribute tagged with "indexed" prompts the storage system to keep an index for this attribute, as discussed below in section 5.2.

While attributes of most types (e.g. Integer , Object-ID) hold actual data (e.g., an integer , an object identifier), Context attributes are pointers to (possibly noncontiguous) areas of the native data; the

2. **Automatic type classification** . Rufus provides a mechanism for automatically determining the types of a user 's files. Once the type of a file has been determined, Rufus automatically extracts the necessary attributes for storage in the database.
3. **Multiple access paths to a user 's data** . Users may search and browse their data with a variety of text search methods such as those found in systems like ST AIRS [IBM] and RUBRIC [MTD85]. These operate on both the underlying free text of the native data as well as in the more constrained environment of the imported attributes. Users may navigate through their data via hypermedia links, or browse through a variety of organizational mechanisms such as collections.
4. **Associating data with its semantics** . The Rufus class hierarchy specifies both the structural attributes (as discussed above) and semantic attributes of a user 's file types. Methods for performing meaningful operations on a particular file type are encapsulated in the hierarchy with the type itself. E.g., Rufus knows what applications to invoke to compile a CSourceCode file [KeR78], format a Troff document file [Oss76], etc.
5. **Flexible mechanisms for associating and collecting data objects.** Besides the collections supplied by the base operating system (e.g., a Unix mailbox or directory), Rufus allows users to group objects manually or automatically as the result of a search, filter , or query operation.

The initial system includes a generic data search and manipulation application based on these principles. A toolkit with which new applications can be built is also planned.

5. The Rufus Prototype

A prototype of the Rufus system was developed in the first quarter of 1991 at the IBM Almaden Research Center . The prototype includes an object-oriented data model and database, a class hierarchy modeling the structure and semantics of 20 user data types, an automatic file classifier and associated import functions, and an X-windows application for browsing, searching, organizing and acting upon a user 's data. The prototype runs on an IBM RISC System/6000 workstation running on the AIX operating system.

For expediency , several functional short-cuts were taken in the design of the prototype. As discussed more fully below , the current system has limitations such as a static (i.e., compiled-in) type system, a single-variable query system (i.e., no *join* operation), and so on. Further , several decisions in the design of both the data model and query system (e.g., the details of the object-oriented query language) were made simply to move the project forward, rather than push the boundaries of these particular areas. Although limited, the prototype still demonstrates the advantages of a system that intelligently manages existing data.

The structure of the prototype is displayed in Figure 1. The type system, discussed in detail in section 5.1, is composed of the three schema modules. The object-oriented storage system, discussed in section 5.2, includes the table queue, record file, B-tree, dictionary and buffer pool modules. Section 5.2 also describes the query engine and query language. The user interface, presented in section 5.4, sits at the top of the prototype and interacts with the storage system through the query engine, recognizer and data import methods. These latter two pieces of the system are discussed in section 5.3.

The majority of the prototype system is written in the C programming language and constitutes approximately 36,500 lines of code, divided as shown in Figure 2. The user interface was built with the T eleUSE user -interface management system [T el90] and is written partially in C and partially in the T eleUSE language D. The schema definition of the underlying object system is written in a home-grown C-like language and compiled into C by the Rufus schema compiler .

and functions for coping with existing data, the Melampus project is designing their data model and system on a clean slate.

Rather than retrieval, a number of systems use a *browsing* metaphor for data access. Hypermedia systems [Con87] support linked *webs* of information that are traversed dynamically by a user. Instead of specifying a single query to retrieve a large volume of data, a user makes a sequence of incremental decisions, traversing hypermedia *links* from one object to another. The Hyperties system [Shn87] provides simple navigational aids, while Intermedia [YHM88] provides maps of the hyperweb as well as a complete history of a user's movements within the web.

A weakness of many hypermedia systems is the proprietary architecture of their data stores. Hypermedia systems generally do not concern themselves with existing data, preferring to require webs to be created from scratch. Once a web is created, hypermedia systems offer limited access paths to the underlying data. For example, although Intermedia is built on top of a relational database, none of the relational query capabilities are available to Intermedia users. Instead the browsing metaphor is the sole access path available to the data.

Finally, several systems allow for explicit links between data and applications. The Apple Macintosh [3] for example, allows a single application to be bound to a data file through the concept of a *creator* [Ros85]. When a data file's icon is double-clicked, the associated application is automatically started. Systems such as the Looking Glass [4] [Vis89] and X.desktop [5] [IXI89] graphical Unix interfaces provide similar abilities to link actions (e.g., particular application programs) to files through a series of rules files. The links are based primarily on the file name, owner, and access permissions, and not on the actual underlying data.

4. The Rufus Solution

The Rufus project has created a system that has an understanding of a user's data and integrates it with the functionality of systems such as object-oriented databases and information retrieval systems. The system operates on a user's existing data, providing additional access paths to the data and its semantics without invalidating previously existing access and applications. Attributes are extracted from a user's data into a standard form (e.g., into a database) without affecting the native data files. The system is extensible and customizable so that users can add new data types as needed. As types are added to the system, their attributes and semantics are integrated with those of previously defined types. Tools are provided for searching and organizing across these extracted structures, and connections are formed between these extracted objects, the original data files, and the applications that operate upon the data.

Rufus amplifies the power of traditional data manipulation techniques, such as information retrieval, by integrating a deeper understanding of the data structure into the operation. Rather than limiting a user to a simple text search, the Rufus system provides the ability to search *con*textually. A Rufus user, for example, can retrieve documents based on the values of specific attributes (e.g., "Author" as discussed above) rather than based on simple pattern matching.

Rufus is built upon five key components:

1. **A unified model of a user's data**. A user's data types (e.g., Mail, Documents, etc.) are defined to the Rufus System in the form of an object-oriented class hierarchy. Rufus extracts attributes (e.g., Sender, Author, etc.) from a user's data and stores them in a database. Rufus presents the user with a virtual card catalog model that integrates similar attributes across different data types. Through this catalogue, a user has common access to and across a broad array of data types.

---

3. Apple ® and Macintosh ® are registered trademarks of Apple Computer, Inc.
4. Looking Glass ® is a registered trademark of Visix Software Inc.
5. X.Desktop™ is a trademark of IXI Limited.

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.

All of these systems can automatically filter and file incoming mail according to a user 's preferences. The Information Lens augments its filtering abilities with a hierarchy of pre-defined mail *templates* . These templates provide pre-filled fields that receivers can depend on in constructing their filters. Both the Information Lens and T apestry provide an "anyone" specification that allows a message to be received by anyone whose filters accept it. T apestry takes an even more active approach by providing an architecture that allows user -defined intelligent agents to forage through various mail/message databases for items of interest.

In many ways, T apestry is an *information retrieval system* applied to a limited interactive mail/message domain. Full-text information retrieval systems such as ST AIRS [IBM], RUBRIC [MTD85] and W AIS [KaM91] provide a user with the ability to retrieve documents as the result of queries posed against the document text. ST AIRS, for example, provides a number of text-matching operators (e.g., string searching, nearness operators, etc.), and a language for expressing retrieval queries against a pre-indexed universe of documents. RUBRIC extends the power of these text operators by adding a concept-matching hierarchy that weights different patterns of matching individually .

Systems such as the Information Lens and ST AIRS are at opposite ends of the "domain specificity" spectrum. The Information Lens stresses a deep understanding of its limited data domain, while STAIRS may process any type of data as text. W AIS treads more towards the middle of the spectrum by dividing the retrieval function among a number of *information servers* . Users pose queries to a common front end, which then passes the query (in parallel) to applicable information servers. These servers, on local and wide-area networks, then process the query according to their own domain-specific algorithms. Each returns a weighted list of documents, which are then merged at the front-end.

While W AIS provides some measure of data semantics at the back end, its prototype client at the front end does not. For example, while individual W AIS servers might be able to locate "author" attributes in their document bases, there is no obvious way to specify (or even know of) the attribute at the front end. This is apparently more a limitation of the prototype client application than the WAIS approach, as the underlying query protocol (ANSI/NISO Z39.50-1988 [ANS89]) does accommodate the tagging of search terms with attributes.

Relational database systems such as DB2 [HaJ84] and Ingres [SWK76], and object-oriented systems such as ObjectStore [Obj91], provide explicit frameworks for describing the structure of data types. Relational systems provide limited support for attribute integration, either explicitly in the schema itself (e.g., with generalization [SmS77]) or implicitly using union operators within the queries. Object-oriented systems take a more direct approach to attribute integration through their class hierarchies and inheritance mechanisms. Object-oriented and some newer relational systems allow the schema designer to introduce a new data type into the system and integrate its attributes with those of the extant types. Database systems provide powerful query languages that allow a user to express retrievals based on the structures defined in the schema.

In addition to an object-oriented database' s facilities for describing data structure, these systems also provide a simple framework for encapsulating an object' s structure together with its semantics, or behavior . As noted previously in section 2.2, one of the problems users face is the separation between their data and the application programs that operate upon it. Using an object-oriented data model, one can connect an object and its behavior through encapsulation.

An interesting application of an object-oriented data model is the Melampus system [CHR90]. Melampus is investigating the possibility of representing *all* data within a system with an object-oriented database. Melampus is intended to manage *all* of a user 's data as well as *all* of the underlying system structures. Everything from a user 's random scribbling to the system process tables will be contained in the Melampus object repository . In contrast to Rufus' approach of devising a data model

mail based on another attribute (e.g., mail routed through a particular host), they must often resort to the formal specification of the format (e.g., [Cro82]) and a search on the raw file text. This problem is compounded if the user wishes to search across multiple mail formats, each specifying the route differently . The problem is further complicated when users wish to search on attributes that span semantic types, e.g., finding all documents (mail, papers, programs, etc.) authored by a particular person.

By making data structure implicit in application programs, it becomes unavailable to future application programmers. With the definitions of data structures and semantics scattered among a series of applications, reusing or refining the structure often requires cannibalizing existing applications. For example, a list of names, addresses and phone numbers might have different applications to browse, search and cross-index. Building a new application, such as an address label formatter , requires the programmer to rediscover the data format by reading the source code for an existing application. If an *address* were an object known to the system, rather than to specific applications, application builders could reuse the structural definition more easily .

2.4. New Types of Data

In addition to current data types, it is quickly becoming practical to store new kinds of information on-line. Temporal data such as a developing story on a newswire feed or the output of a stock ticker present new problems for a user . These *data feeds:* (1) contain more data than a single person can or would care to assimilate; (2) become stale rapidly; (3) drift at various rates among different topics; (4) are typically retained on-line for a limited time to limit their size; (5) are loosely or poorly organized.

Data feeds introduce problems beyond the volume of data they represent. In particular , selection of an interesting subset of the feed is important. The selection process is complicated by the poor classification of items, the drift on topics in the feed, the drift of a user 's interests, and the varying amounts of time a user wishes to spend reading. Some types of feeds have complex structures that represent conversations. Exploitation of these structures is necessary for readers to cope with the volume. For example, users need to be able to drop out of conversations as they lose interest.

While access to textual data feeds is quickly becoming common (e.g., the Clarinews service that provides access to the UPI wire service [Tem90]), computer storage of continuous audio, high resolution images and full-motion video is still in its infancy . As these technologies mature, they will require new storage, browsing and search techniques. Current data browsers and search engines are geared almost uniformly towards textual data, necessitating new methods of specifying and matching patterns in audio, image and video data. Several projects are studying algorithms for modeling and indexing image data [Gro84][CYD88] as well as methods for querying images by semantic content [Cod91][GWJ91]. Any information management system designed to integrate new data types should necessarily be able to accommodate advancements in audio and video.

3. Related Work

Many systems exist that address aspects of information anxiety . Mail systems, information retrieval systems, hypermedia browsers, relational and object-oriented databases, and integrated desktop applications each provide a user with sophisticated data-handling abilities, but each only in limited domains.

Intelligent mail filtering capabilities, such as found in BBN/Slate [Bol90], the Andrew Message System [REB87], the Information Lens [MGT87] and Tapestry [Ter89] give users a large measure of control over their incoming mail. BBN/Slate, Andrew and Tapestry support multimedia mail and provide recipient-specified translation (i.e., the systems automatically remove multimedia objects when compound documents are displayed on a device that does not support a particular medium such as graphics or audio).

2. The Problems with Existing Data

Today's computer users have ready access to data of many different origins and types. A user's local data might include computer programs and documentation, data test sets, documents, calendars, and so on. Additionally a user might passively receive data via networks, including electronic mail, broadcast messages, and newswire feeds. Users might also actively reach out across a network or through a modem to access remote information servers. Given this access, users find their present tools for searching, filtering, organizing and processing quickly outstripped by the volume, diversity and complexity of the available data.

The remainder of this section discusses four central aspects of information anxiety: the increasing availability of huge quantities of data (section 2.1), the widening gap between data and its semantics (section 2.2), the inability to operate across data types (section 2.3) and new types of data with which users have no previous experience (section 2.4).

2.1. Data Volume

The primary cause of information anxiety is the sheer volume of data now available to users via computers and computer networks. The authors' workstation (an IBM RISC System/6000 running IBM's Unix product, AIX 3.1) holds over 500 megabytes of local data (of which 85 megabytes are user data) and over 4.5 *giga* bytes of total disk space. An additional 500 megabytes of data is mounted from network filesystems. The authors' mainframe system (an IBM System/390 running VM/CMS) has over 500 megabytes of local data (14 megabytes of user data), and roughly 400 *giga* bytes of total disk space. [2] Pattern matching tools such as *grep* [Unix-a] are useless across such volumes of data.

Users also must contend with an enormous diversity of file types. *File type* is used here loosely to refer to the different file structures and naming conventions used by applications. The authors' mainframe system yielded over 800 distinct file types; their workstation system over 200. The diversity of file formats creates two immediate problems: first, a user is expected to identify the type of a file and provide the names of applications for manipulating it; second, sharing data between different applications is nearly impossible.

Data volume can also be measured by the number of files into which the raw storage is partitioned, the organizations of files into multi-file objects (e.g., multi-module programs), the location of data spread across different platforms and operating systems, and so on.

2.2. The Gulf Between Data and Applications

Once users have actually located files of interest they are often at a loss as to what to do with them. Given a document, how is it processed for printing? Given a program, how is it compiled into an executable object? The affiliations between files and the applications that manipulate them reside primarily in the creator's head. As users acquire access to data created by others (e.g., through shared file systems, electronic mail, etc.) they often find themselves unable to use it. Further, users often forget how to manipulate data of their own creation.

Ideally the system's knowledge of a user's data types should include an understanding of normal data processing. At the very least the system should be able to provide users with a list of applications that are commonly used to manipulate a piece of data (e.g., users looking at a document file should be given a list of options for printing, displaying, spell checking, editing, and so on).

2.3. Interoperating Among Data Types

Users often find limited access to a file's semantics through a single application. For example, a typical mail reading application might locate mail from a particular sender. If users wish to locate

2. AIX ™, System/390 ™ and VM/CMS ™ are registered trademarks of the International Business Machines Corporation.

1. Introduction

The increasing volume and diversity of electronic data hinders a user's ability to locate, organize and process information. Wurman has characterized this loss of control and the resulting user emotion as *information anxiety* [Wur89]. Data volume can be viewed in several dimensions. For example, data can be plentiful in its raw number of bytes, the number of files into which these bytes are divided and the number of application-specific file formats (e.g., mail files, documents, etc.) these files represent. A typical workstation user routinely encounters thousands of files in any of several hundred formats. For each format the user must be cognizant of the applications that operate upon it. The connections between a data file and the applications that operate upon it constitutes yet another dimension of data volume.

The proliferation of data formats not only makes a single piece of data more difficult to use, it also hinders a user's ability to operate *across* data types. For example, searching a variety of files for a particular attribute (e.g., "Author") is only possible if the user is willing to specify how to find the "Author" in each different data format. None of the inherent structure of the individual files (e.g., the "Author" of a mail file is usually identified in a line that begins "From:") is available to search tools because the semantics of the data formats are locked away in individual application programs.

Many systems have been created that address aspects of the data flood. These include electronic mail filters such as the Information Lens [MGT87], text retrieval systems such as STAIRS [IBM], hypermedia browsers such as Intermedia [YHM88], and database systems such as DB2 [HaJ84]. While each of these provides valuable function, they do so either for a limited domain (e.g., mail systems), without any explicit understanding of a user's data (e.g., text retrieval systems), or provide little or no assistance in coping with existing data formats (e.g., hypermedia and database systems). What is needed is a system that integrates the functionality of these previous technologies with a deeper understanding of a user's existing data. Pragmatically, this will need be done as an addition to a user's current applications, rather than a replacement.

The use of existing data is at the core of the Rufus project's vision. In contrast to projects that are developing an entirely new data model to *replace* current file systems (e.g., Melampus [CHR90]), the Rufus project is studying how to cope with existing file systems and formats. The Rufus system is built around a centralized description of a user's existing data types and formats. Rufus extracts important attributes from a user's data and stores them separately, much like the Author, Title and Subject of a book are stored in a library's card catalogue. Rufus *augments* a user's access to data without invalidating the existing access paths or the current suite of application programs.

Rufus's centralized data descriptions support the construction of new applications that operate across a user's data types. By integrating the attributes of a user's data in the central repository, technologies such as information retrieval engines and database management systems are able to operate on a semantic level rather than a purely syntactical (i.e., text-oriented) level.

A prototype Rufus system has been constructed for IBM's RISC System/6000 [1] workstations. The prototype currently understands about 20 user data types, including different document and electronic mail formats, telephone directories, program source and object code. A prototype application has been built for searching, querying and browsing the imported data. The ability of users to search on semantic attributes of their existing data (both homogeneously and heterogeneously) has proved quite valuable.

Section 2 provides a more detailed exploration of the problems created by existing data formats. Section 3 discusses related work, including mail and information retrieval systems, database management systems, and hypermedia browsers. Section 4 provides an idealized view of the Rufus solution, while section 5 presents a more pragmatic view of the initial Rufus prototype. Section 6 details the developer's and user's experiences with the prototype and suggests directions for future research. Section 7 provides some final thoughts.

---

1. IBM ® and RISC System/6000 ™ are registered trademarks of the International Business Machines Corporation.

Rufus: The Information Sponge

Eli Messinger
Kurt Shoens
John Thomas
Allen Luniewski

IBM Almaden Research Center
650 Harry Road, K55/801
San Jose, CA 95120-6099

Abstract

Typical computer users are connected via worldwide networks to an increasing number of data sources and other users. This interconnectivity provides users with previously unknown riches of available knowledge. Unfortunately, it simultaneously drowns them in a flood of unsorted, unclassified and unusable data.

Existing information systems only cure some of information overload's symptoms. File systems, for example, typically treat documents such as electronic mail, project notebooks, and source code as uninterpreted text, with no associated structure or semantics. Retrieval systems search for text patterns rather than semantic attributes (e.g., *author*). This is especially problematic when searching across heterogeneous file types. Database systems focus primarily on their own internal data representations, providing one-way paths for importing a user's native data.

What is needed is an environment in which existing applications co-exist with new applications that can operate across existing file types. A framework needs to be developed in which a user's existing data can be described, in both structure and behavior, together with a toolbox upon which new applications can be built. These new applications will use the centralized descriptions to provide a more integrated view of the user's data, and easier access to the data's behavior.

The Rufus project has developed an extensible object-oriented data model, storage system, and associated search and display methods for a variety of user file types. The system automatically classifies a user's data files and extracts type-specific attributes. Users may then search, browse, filter, link and display the imported data objects. New file types can be added to the system as new applications arise. Library support for more specialized applications is planned.