Essence: A Resource Discovery System Based on Semantic File Indexing

Darren R. Hardy & Michael F. Schwartz – University of Colorado, Boulder

ABSTRACT

Discovering different types of file resources (such as documentation, programs, and images) in the vast amount of data contained within network file systems is useful for both users and system administrators. In this paper we discuss the *Essence* resource discovery system, which exploits file semantics to index both textual and binary files. By exploiting semantics, Essence extracts keywords that summarize a file, and generates a compact yet representative index. Essence understands nested file structures (such as uuencoded, compressed, ''tar'' files), and recursively unravels such files to generate summaries for them. These features allow Essence to be used in a number of useful settings, such as anonymous FTP archives. We present measurements of our prototype and compare them to related projects, such as the Wide Area Information Servers (WAIS) system and the MIT Semantic File System (SFS). We demonstrate that Essence can index more data types, generate smaller indexes, and in some cases index data faster than these systems. Our prototype generates WAIS-compatible indexes, allowing WAIS users to take advantage of the Essence indexing methods.

Introduction

In the past two years, a number of resource discovery tools have been introduced to help users locate and use the massive amount of information available in the Internet [Schwartz et al. 1992b]. As disks have become larger, cheaper, and more plentiful, resource discovery has also become a problem in general purpose file systems, such as the Sun Network File System (NFS). Yet, the current set of Internet discovery tools do not apply well to such an environment, for three reasons.

First, information in general file systems is typically very irregularly organized. Most Internet data is explicitly intended for sharing, and hence people often put some effort into organizing the information into a coherent whole (e.g., placing an entire file system into a meaningful hierarchical directory in an *anonymous* FTP site). In contrast, most general file system data are organized according to the individual whims of many people. Therefore, resource discovery systems that depend heavily on users to organize and browse through data (such as Prospero, [Neuman 1992], WorldWideWeb [Berners-Lee et al. 1992], or Gopher [McCahill 1992]) do not work well for general purpose file system data. Instead, automated search procedures are needed. This typically means generating some type of index of the available information [Salton & McGill 1983].

Second, general file systems contain a range of different types of data, from unstructured text to structured data. Systems that use a generic indexing procedure (such as archie [Emtage & Deutsch 1992] or WAIS [Kahle & Medlar 1991]) produce larger or less useful indexes under these circumstances. For example, WAIS is most effective when used on ASCII text files. Using WAIS to index executables and other files found in general file systems is not very effective. The indexes tend to do a poor job of locating information, and tend to be quite large.

Third, Internet discovery tools typically focus on information known to be of reasonably broad interest. For example, anonymous FTP archives typically contain popular documents and software packages, which exhibit heavy sharing [Schwartz et al. 1992a]. In contrast, general-purpose file systems typically contain mostly user-specific data that exhibit relatively little sharing [Muntz & Honeyman 1992, Ousterhout et al. 1985]. Current Internet resource discovery tools have difficulties with such low sharing-value data. For example, WAIS's full-text indexing mechanism may locate many uninteresting files if applied to an entire general file system, since keywords will be generated from files that are of interest to few users.

In this paper we present a system for supporting resource discovery in general purpose file systems. The system addresses the above problems by generating indexes based on an understanding of the semantics of the files it indexes. This technique supports compact yet representative summaries for general collections of data. In addition to supporting file indexes, summaries can be browsed to help decide whether to retrieve a file across a slow network. We call our system *Essence* because of its ability to summarize large amounts of data with relatively small indexes.

We begin with a discussion of indexing techniques. We then survey previous work related to

semantic indexing. We discuss how Essence accomplishes semantic indexing and uses it as a basis for resource discovery. Finally, we discuss the details of our prototype, and present some measurements that compare Essence with WAIS and SFS.

Full Text vs. Filename vs. Semantic Indexing

WAIS supports fine-grained information access by building full-text indexes, in which every keyword from a textual document appears in the index. As indicated above, this approach is primarily useful for purely textual, widely popular data. Moreover, WAIS has large space requirements: its indexes are comparable in size to the data files they represent. Because of these space requirements, WAIS distributes the indexes among the hosts that provide data.

A less space intensive indexing approach is used by archie, in which anonymous FTP files are summarized by name only (i.e., archie indexes contain no information derived from file content). This approach produces indexes that are roughly one thousandth the size of the data that they represent. In turn, this compact representation allows a great deal of index information to be collected onto a single machine, supporting far-reaching searches (currently reaching over 1000 archive sites). Yet, because archie indexes contain only filenames, they support only name-based searches. Searches based on more conceptual descriptions of resources are not possible, except when the filenames happen to reflect some of these conceptual descriptions.

The range of structure and the low overall sharing value in general purpose file systems (as discussed in the introduction), coupled with the need for conceptual descriptions and the need for compact indexes (motivated above), all suggest the use of a different means of indexing data. That means is *semantic* indexing.

Semantic indexing involves analyzing the structure of file data in different ways, depending on file type. For example, UNIX manual page files are broken into structured sections from which it is possible to extract information about a program's name and description, a usage synopsis, related programs or files, and author information. By generating information for different types of files in different manners, semantic indexing can generate representative keywords without including every word from a file. In addition to saving space, this technique can avoid including keywords that might muddle the quality of an index. For example, it makes little sense to include C language constructs like ''struct'' when indexing C source code, since these keywords do not distinguish the conceptual content of different C programs.

Semantic indexing involves two stages. The *classification* stage identifies promising files to index within a file system,[1] as well as type information for each identified file. The *summarizing* stage applies an appropriate indexing procedure (called a *summarizer*, to emphasize the space reduction characteristic) to each identified file, based on the type information uncovered in the classification stage.

Since summarizers understand file types, they can extract keyword information for both textual and binary files. For example, many binary executables have related textual documents describing their usage, from which keyword information can be extracted.

Since keyword information is extracted based on knowledge of where high-quality information might be located, semantic indexing extracts fewer keywords than full-text indexing, and thus generates smaller indexes. Yet, it retains the fine-grained, associative access capability of full-text indexes.

The Essence System

Essence provides an integrated system for classifying files, defining summarizer mechanisms, applying appropriate summarizers to each file, and traversing a portion of a file system to produce an index of its contents.

Essence determines file types by exploiting file naming conventions (such as common filename extensions like ''.c''), and locating identifying data or common structures within files (such as UNIX ''magic numbers''). Once Essence determines a file's type, it executes the appropriate summarizer to extract keywords from the file. Among other types, Essence understands *nested* file structures, such as compressed, uuencoded ''tar'' files. It recursively extracts files *hidden* within a nested file, and indexes them.

As a design goal, we wanted to allow summarizers to be constructed quickly and easily, so that Essence could be made to understand many different file types, and so individual sites could customize their summarizers. To accomplish this goal, we allow summarizers to be as simple as a one line script (perhaps containing a ''grep'' command).

Essence indexes can allow users to locate needed data. Moreover, Essence produces *summaries* of file data, which allow quick perusal of potentially large files.

Essence has many practical resource discovery applications:

- Systems administrators and users can use Essence to locate and learn about resources

[1] For example, this procedure might embody site-specific knowledge that certain parts of the file tree contain uninteresting administrative information, and hence needn't be indexed. Our current prototype does not select file system subsets - it simply indexes whatever file trees are specified.

contained within their file systems without understanding the details of their local environment. This is particularly helpful in environments where mount points are ''hidden'' by the *amd* auto-mount system.
- Public archive administrators can use Essence to index archive contents, providing compact yet representative descriptions of files, including compressed archives. These indexes allow users to search for information more effectively, and examine summaries about files before retrieving them.
- People who wish to index data and search it using WAIS can use Essence to index more file types than WAIS itself currently supports, and to produce more space efficient indexes.

Once Essence generates an index for a portion of a file system, it exports its indexes via WAIS's search and retrieval interface. This allows our indexes to be used within the context of a well established, easy to use information system.

Related Work

Identifying and Locating File Resources

Semantic indexing depends on successfully determining file types. Furthermore, Essence uses semantic indexing to locate file resources. Many systems can either determine file types or locate file resources, but Essence integrates both aspects into a single system.

- The Modules system is a sophisticated administrative approach to locating file resources associated with specific applications [Furlani 1991]. Applications are associated with a particular *module*, which can be easily incorporated or removed from a user's environment. Both the location and identification of the applications and their file resources are explicitly supplied by an administrator, and are hidden from the user.
- The NeXT file system browser determines common file types by exploiting filename extensions [NeXT 1991]. It then displays an icon representative of the file's type. Users can *launch* a specific application by supplying only a filename, as the application that is launched is determined by the file's type. Locating file resources is accomplished by browsing a UNIX file system hierarchy.
- The UNIX *file* command attempts to determine various file types based on file contents, but provides no mechanism for locating files [USENIX 1986].
- The UNIX *find* command locates files using an exhaustive search of a portion of a file system. It allows predicates to be specified concerning which files to locate. Among other things, these predicates can specify location based on the file types understood by the UNIX file system (such as ordinary file, directory, or symbolic link) [Leffler, et al. 1989, USENIX 1986]. Higher-level types (such as image, script, or C source code) cannot be specified.
- Many programs use file naming conventions to infer file types. C compilers, for example, assume that a filename ending in ''.c'' is a C source file, while a file ending in ''.o'' is a relocatable object file. Similarly, *make* has various implicit rules based on file naming conventions.

Exploiting File Semantics

Semantic indexing also depends on the ability to extract good keyword information from files based on their file types. A number of UNIX commands can extract information with varying degrees of quality from files based on their file types [USENIX 1986].

- *ctags* extracts procedure, macro, and variable names from C source and header files. Some versions of *ctags* understand other programming languages, such as Lisp, Pascal, and C++.
- *strings* extracts embedded ASCII text strings from binary files.
- *deroff*, *detex*, and *ps2txt* extract ASCII text from troff, TeX, and PostScript files, respectively.
- *what* extracts embedded Source Code Control System (SCCS) information from files.

Essence provides a single cohesive system that integrates determining file types, locating file resources, and exploiting file semantics to extract good keyword information from files.

MIT Semantic File System

The MIT Semantic File System (SFS) uses semantic file indexing to provide a more effective storage abstraction than traditional hierarchal file systems [Gifford et al. 1991]. SFS exploits filename extensions to determine file types, and then runs *transducers* on files to extract keyword information for building an index. SFS provides a *virtual directory* interface to search the resulting index and to access files. Virtual directory names are interpreted as queries against the index, and the contents of virtual directories are the results from these queries. Therefore, users perceive a search-based interface to explore file systems, rather than the more traditional hierarchical file system interface.

Although Essence and SFS use similar semantic indexing techniques, they differ in orientation, summarizer breadth, and space efficiency.

Orientation

SFS emphasizes semantic indexing as a storage abstraction. In contrast, Essence emphasizes semantic indexing as a basis for resource discovery. Concretely, while both systems support flexible associative access to file data, they export the data

differently. Essence exports the data through a search and retrieval interface, while SFS exports the data through a file system interface. The advantage of the SFS approach is that it reuses an existing and familiar storage abstraction. The disadvantage is that doing so leads to undefined semantics. For example, if a user tries to copy data into a virtual directory (created as a result of an SFS query), the semantics are undefined.

Summarizer Breadth

Essence summarizers are autonomous UNIX programs, which are easy to implement, integrate, and maintain. The Essence prototype implements summarizers for many more file types than SFS does. Essence can index a wide variety of textual and binary data common in network file systems.

Space Efficiency

The Essence prototype provides better index compression than the SFS prototype. Comparative measurements appear later in this paper.

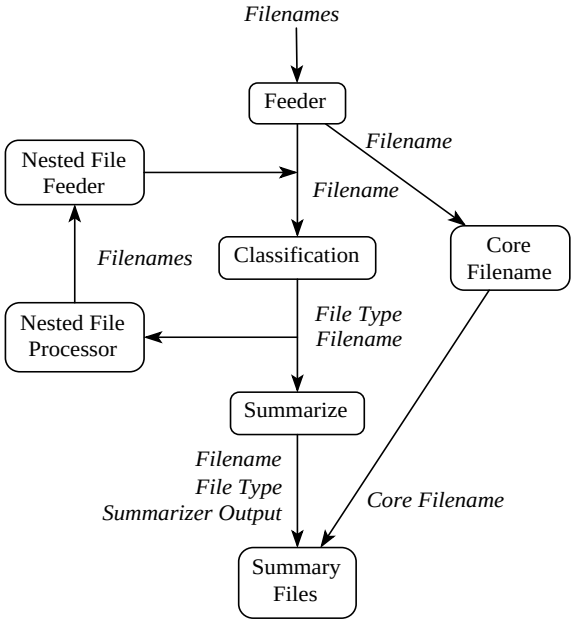

Figure 1: Organization of the Essence System

The Design of Essence

Figure 1 shows how Essence is organized. Essence operates as follows:

- Users supply Essence with the filenames from a select portion of a file system that they wish to index.
- The *Feeder* module iteratively passes each of these filenames to the *Classification* module, which determines the file's type.
- The *Summarize* module chooses an appropriate summarizer based on the file's type. It then runs this summarizer on the file to extract keywords for the *Summary Files*.

The three modules, *Core Filename, Nested File Processor,* and *Nested File Feeder,* allow Essence to support nested files.

- Essence saves the initially supplied filename as the *Core Filename*.
- If the *Classification* modules determines that the file has a nested file type (such as a compressed file), it passes the file to the *Nested File Processor*.
- The *Nested File Processor* extracts the hidden files from the nested file structure and passes the extracted files to the *Nested File Feeder*.
- The *Nested File Feeder* module performs the identical function as the *Feeder* but bypasses the *Core Filename* module.

Determining File Types

Essence determines file types using a combination of exploiting file naming conventions and heuristically locating identified data and common structures within files.

Exploiting File Naming Conventions

Observing even simple conventions in file naming can determine file types with fairly high certainty. The most basic file naming convention is filename extensions. For example, filenames with a ''.c'' extension are typically C source code files; filenames with a ''.ps'' extension are typically PostScript image files; and filenames with a ''.txt'' extension are typically ASCII text files. File naming conventions also include using specific words within a filename. For example, information about an entire source distribution or application is often found in files whose name contains the string ''README''. Files named ''Makefile'' are typically associated with the UNIX *make* command [USENIX 1986].

In Essence, file naming conventions are represented as regular expressions. For example, **.ps* or **[Mm]akefile** represent the PostScript and Makefile file types, respectively. Expressing file naming conventions as regular expressions allows sites to easily integrate their local semantics into Essence.

Locating Identifying Data and Common Structures

In addition to using naming conventions, Essence examines file contents to try to determine file types. In particular, many files have an identifying *magic* number associated with them. For example, NeXT binary executables start with the hexadecimal number *0xfeedface*, and Sun Pixrect images start with the hexadecimal number *0x59a66a95*. Furthermore, common structures within a file may determine its file type. For example, PostScript images start with the string ''%!''; UNIX shell programs start with the string ''#!''; C source code files typically have comments denoted with the ''/* */'' delimiters; electronic mail files have distinctive header tags, such as *From:*, *Received by:*, and

Sender:; and USENET news articles also have distinctive header tags, such as *Newsgroups:*, *Distribution:*, and *Path:*.

As with exploiting file naming conventions, locating identifying data and common structures within a file is a rule-based technique expressed with regular expressions. Sites can easily integrate their local semantics into the discovery process by modifying these rules.

Nested File Structure

Nested files contain *hidden* files. Examples include compressed files, tar files, uuencoded files, ZIP files, and shell archive files. Furthermore, files can be arbitrarily nested within these file types. For example, compressed tar files or uuencoded compressed files are common. Understanding nested file structures is useful in file system environments (such as anonymous FTP file systems) in which the vast majority of files have nested structure.

When Essence determines that a file has nested structure, it extracts the hidden files, determines the resulting files' types, and summarizes them. This process continues recursively, until no more nesting is found. Extracting hidden files from a nested file is accomplished by running a corresponding extraction program, such as the UNIX *uncompress* command for compressed files, the UNIX *tar* command with the 'x' flag for tar files, or the UNIX *uudecode* command for uuencoded files.

Summarizers

Essence's summarizers are simple stand-alone UNIX programs that are easy to write and integrate into the system. This design provides a powerful paradigm for exploiting file semantics. Each summarizer is associated with a specific file type, and understands the file's format well enough to extract summary information from the file. For example, the summarizer for a UNIX troff-based manual page understands the troff syntax and the conventions used to describe UNIX programs. It uses this understanding to extract summary information, such as the title of a program, related programs and files, the author(s) of the program, and a brief description of the program. Similar techniques can be used on many other moderately structured file types, such as source code. However, some file types do not easily lend themselves to automated interpretation. For example, plain ASCII text files typically contain unstructured data that is difficult to exploit effectively. Similarly, the UNIX *ps2txt* program can extract ASCII text from PostScript images, but the resulting information is unstructured text.

Essence Prototype

In this section, we describe the techniques used by the Essence prototype to determine file types and exploit file semantics with summarizers. We also discuss how we integrated Essence with WAIS.

Determining File Types

As described earlier, file types are determined by understanding naming conventions and locating identifying data and common structures within a file. In the prototype, naming conventions are expressed with case-insensitive regular expressions. The following example shows some entries from the configuration file that holds the expressions. In this file, the first field is the file type, and the second field is a case-insensitive regular expression for the corresponding file naming convention.

```
Compressed   .*\.Z
ManPage      .*\.[12345678]
PostScript   .*\.(ps|eps)
README       .*readme.*
SCCS         s\..*
```

The prototype also uses the UNIX *file* command to determine file types, based on identifying data and common structures within a file [USENIX 1986]. *file* uses the */etc/magic* file to specify recognizable file types. The following list shows some sample entries from */etc/magic*, where the first field is the offset of the identifying data or common structure, the second field is the type this data, the third field is the identifying data or common structure itself, and the last field is the corresponding file type.

```
0  string  /*           C program text
0  string  \037\235     Compressed data
0  long    0xfeedface   NeXT binary pgm
0  string  #!/bin/perl  Perl program
0  string  %!           PostScript image
```

Creating a suitable *magic* file is not trivial, because the identifying data or common structures must be distinctive. For example, the ''/*'' delimiter for C programming language comments is not sufficiently distinctive, and will likely appear in a variety of types of files. A lack of distinctive identifying data or common structures is common for binary formats, which usually depend on a single magic number. Although distinctive *magic* entries are difficult to formulate, careful selection of a *magic* file allows *file* to accurately identify file types. In Essence, building the *magic* file was accomplished through experimentation with various entries.

Summarizers

In the prototype, summarizers are simple UNIX programs that extract keyword information through understanding the syntax and semantics of a specific file type. Currently, the prototype supports summarizers for twenty-one file types and four nested file types. Table 1 describes these file types, their frequencies of occurrence by number of files, average file size, and which systems support them in two file system environments: an NFS file system that contains commonly shared data and programs in our local environment, and a fairly popular anonymous FTP file system (ftp.cs.colorado.edu). The most frequent file types in the NFS file system were *Text*,

CHeader, and *ManPage*. In the anonymous FTP file system the most frequent file types were *CHeader*, *C*, and *Text*.

Essence supports more of the file types found in common NFS and anonymous FTP file systems than either WAIS or SFS, as shown in Table 1. Although WAIS and SFS support most of the frequently occurring file types (such as *Text*, *C*, and *CHeader*), Essence is the only system that supports the file types that contribute most to overall data size (such as *Binary*, *Tar*, and *Archive*). Occurrence frequencies will be used in our measurements, later in this paper. Note that Table 1 does not list specialized file types supported by WAIS or SFS that are not supported by Essence, because those types do not occur in common NFS and anonymous FTP file systems (and hence we have no measurements for them). Examples include MedLine and New York Times formats. There are 12 such formats understood by WAIS, and 4 understood by SFS. Also, as indicated in the table, SFS indexes Unknown file types. It does so by including the standard UNIX attributes in the index, such as owner, directory, and group.

Table 2 briefly describes the techniques used by the Essence summarizers for the supported file types, other than nested types (the techniques for which were already discussed, in the "Nested File Structure" section). Many other potential summarizers are possible. For example, writing summarizers for other types of source code (such as Lisp or Pascal) would be an easy extension of the prototype's source code summarizers. However, writing summarizers for audio or image formats would be difficult.[2]

The following sections describe some of the techniques used in various summarizers, representative of Essence's supported file types.

[2]One possibility would be to sample a bitmap file down to an icon. While this does not easily support indexing, it could be used to support quick browsing before retrieving an entire image across a slow network.

File Type	File Type Description	File Types Supported By			Frequency by Number of Files		Average File Size	
		Essence	**WAIS**	**SFS**	NFS	AFTP	NFS	AFTP
Archive	Library archives	×		×	0.36	0.12	626.31	47.52
Binary	Binary Executable	×		×	5.06	0.02	145.74	112.00
C	C source code	×	×	×	1.27	19.33	3.87	28.36
CHeader	C header files	×	×	×	14.73	22.42	4.29	2.40
Command	UNIX shell scripts	×	×		1.78	3.06	2.75	1.55
Compressed	Compressed file	×			0.69	11.81	114.98	73.29
Directory	Directory	×		×	4.87	5.05	0.81	0.50
Dvi	Device-indep. TeX output	×	×		0.03	0.87	33.32	59.18
Mail	Electronic mail	×	×	×	0.02	0.17	1.79	35.30
Makefile	UNIX makefiles	×	×	×	0.26	3.87	0.85	3.04
ManPage	UNIX manual pages	×	×	×	13.78	0.70	6.76	295.92
News	USENET news articles	×	×	×	0.01	0.04	21.96	1.75
Object	Relocatable object file	×		×	0.00	1.12	0.00	28.11
Patch	File difference listing	×	×		0.02	0.00	1.88	0.00
Perl	Perl script	×	×		0.00	0.02	0.00	3.62
PostScript	PostScript images	×	×		1.42	3.31	64.64	194.45
RCS	RCS version control files	×	×		0.00	1.41	0.00	8.41
README	High-quality information	×	×	×	0.38	1.32	1.95	2.88
SCCS	SCCS version control files	×	×		0.00	0.00	0.00	0.00
ShellArchive	Bourne shell archive	×			0.00	0.10	0.00	486.75
Tar	Tar archive	×			0.00	0.81	0.00	1734.21
Tex	TeX source docs	×	×	×	0.67	0.23	17.79	34.17
Text	Unstructured ASCII text	×	×	×	21.64	19.73	7.87	31.11
Troff	Troff source docs	×	×		0.03	0.25	9.21	9.08
Unknown	Unknown file type			×	32.96	4.26	44.02	16.10

Table 1: Supported Common File Types and their Frequency and Average Size in Measured File Systems

Directory

Obtaining a listing of the files in a directory is an obvious method for a *directory summarizer*. However, Essence strives to obtain a higher-level understanding of a directory's contents. Therefore, the prototype attempts to extract copyright information from files, in addition to the directory listing. Copyright information typically contains project, application, or author names. Keyword information from README files is also included in the directory summarizer, since these files contain high-quality information about the directory's contents.

File Type	Summarizer Description
Archive	Extract symbol table
Binary	Extract meaningful strings, and manual page summary
C	Extract procedure names, #include'd filenames, and comments
CHeader	Extract procedure names, included filenames, and comments
Command	Extract comments
Directory	Extract directory listings, copyright information, and README files
Dvi	Convert to ASCII text
Mail	Extract select header fields
Makefile	Extract comments
ManPage	Extract author, title, etc., based on ''-man'' macros
News	Extract select header fields
Object	Extract symbol table
Patch	Extract filenames
Perl	Extract procedure names and comments
PostScript	Convert to ASCII text
RCS	Extract RCS-supplied summary
README	Use entire file
SCCS	Extract SCCS-supplied summary
Tex	Convert to ASCII text
Text	Extract first 100 lines
Troff	Extract author, title, etc., based on ''-man'', ''-ms'', ''-me'' macro packages, or extract section headers and topic sentences.

Table 2: Essence Summarizer Techniques

Binary

An obvious method for a *binary summarizer* is to extract ASCII strings from the binary file, using the UNIX *strings* command. However, Essence filters these extracted ASCII strings using heuristics that only keep strings that convey the binary's purpose, such as usage, version, or copyright information. Essence also uses cross references to obtain high-quality summary information from binary executables. For example, the binary summarizer looks for associated manual pages for the given binary executable, and generates keywords using the *manual page summarizer* on it.

Formatted Text

Although formatted text (such as TeX, Troff, or Word Perfect) has structured syntax, effectively summarizing these files is difficult unless semantic information is also available [Knuth 1984, Lamport 1986, USENIX 1986]. For example, plain Troff files or Troff files using the ''-me'' macros are difficult to exploit semantically, since their syntax is associated with formatting commands (such as font size or line spacing), rather than more conceptual commands (such commands to indicate an author's name or paper title). Troff files using the ''-ms'' or ''-man'' macros are much easier to summarize, since they contain conceptual commands (such as delimiting an abstract, author, and title).

Essence supports a sophisticated summarizer for Troff and the ''-me'', ''-ms'', and ''-man'' Troff macros. The *TeX summarizer* only extracts ASCII text from TeX files using *detex*, but exploiting TeX semantics would be a trivial extension of the methods used in our *Troff summarizer*.

Simple Text

Simple text is difficult to summarize because it is unstructured. Essence assumes that the highest quality information in most unstructured text files is near the beginning of the file, as is common with paper abstracts or tables of contents. Therefore, the *text file summarizer* extracts keywords from the first one hundred lines of the text file. However, README files typically contain crucial, concise information about a distribution or application. Using a full-text index of README files provides high-quality keywords without occupying too much space. Therefore, the *README summarizer* uses the entire file to generate keywords.

The *Dvi*, *PostScript*, and *Tex summarizers* extract keywords from all of the ASCII text extracted from these files. Essence assumes that these file types contain generally useful information, and hence generates full text indexes for them.

Source and Object Code

Both source and object code are highly structured, and contain easily exploited semantic information. The *C summarizer* extracts procedure names, header filenames, and comments from a C source code file. Similarly, the *object summarizer* extracts the symbol table from an object file.

WAIS Interface

Essence exports its indexes through WAIS's search and retrieval interface, allowing users to use tools such as *waissearch* and the X Windows-based graphical user interface *xwais*. In order to generate

WAIS-compatible indexes, Essence uses WAIS's indexing software to index the Essence summary files. This mechanism generates full-text WAIS indexes from the Essence summary files.

We modified the WAIS indexing mechanism to understand the format of the Essence summary files, so that it generates meaningful WAIS *headlines*. These headlines provide users with a short description of a single file, usually a filename. With Essence, headlines represent a file's core filename, its actual filename, and its file type.

To support additional file types, WAIS must be recompiled with new procedures that understand these file types. With Essence, one need only write a new summarizer, add its name to a configuration file, and add new heuristics for identifying the file type; no recompilation is necessary. In this sense, Essence modularizes the typed-file indexing extensions that WAIS can use, because it removes the keyword extraction process from WAIS and places it instead in Essence. Essence is better suited to incorporating new file types, and can be quickly adapted to become a *comprehensive* indexing system.

Figure 2 shows an example search of an index generated by Essence of the ftp.cs.colorado.edu anonymous FTP file system. It shows an ordered list of the ten files that best match the keyword *netfind*.[3] The headlines have up to three fields representing the matching file: the core filename, the filename (if different from the core filename), and the file type.

[3] *Netfind* is an Internet user directory service [Schwartz & Tsirigotis 1991].

Consider the effectiveness of the example search in Figure 2. The best match is a PostScript paper that discusses a number of techniques for distributed information systems, with particular emphasis on techniques demonstrated by Netfind; the second match is the same file, but found in the compressed tar distribution *ALL.PS.tar.Z*. The third match is the C source code for the interactive user interface to Netfind. The fourth match is the README file found in the Netfind distribution directory; the fifth match is the same file, but found in the compressed tar distribution *netfind.3.10.tar.Z*. The sixth match is the UNIX manual page for Netfind. The remaining matches are PostScript papers in which Netfind is discussed.

In WAIS, a user retrieves files by selecting a matching headline. With Essence, if the headline represents a file hidden within a nested file (such as the first headline in Figure 2), the summary file is retrieved, instead of retrieving the hidden file itself. If the headline represents a plain file (such as the fourth headline in Figure 2), the file is retrieved. This functionality requires allocating storage for both the required summary files and the index. However, it allows users to browse through remote file systems by retrieving and viewing small summary files without having to retrieve complete files. This is useful when trying to decide whether to transfer large files across a slow network.

Evaluation and Measurements

In this section, we present measurements of indexing speed and space efficiency, for Essence, WAIS, and SFS. We also discuss the usefulness and

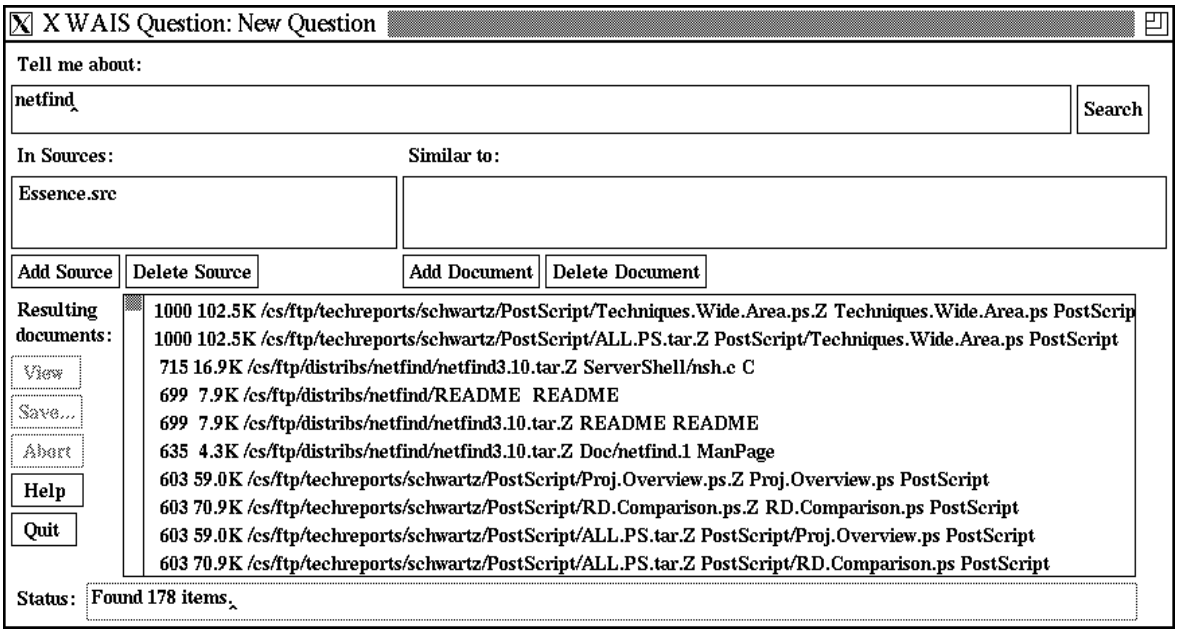

Figure 2: Example WAIS Search Using Essence-based Index

overhead of indexing nested files. Finally, we discuss the difficulties in evaluating keyword quality.

Before presenting measurements of the various systems, we note that it is difficult to interpret time and space efficiency measurements of the systems being compared, for two reasons. First, indexing speed and compression are highly dependent on indexing techniques. For example, an indexer that skips most of the data (such as our Text summarizer) will achieve much higher indexing speeds and compression factors than one that uses all of these data (such as the Text indexers used by SFS and WAIS). In this case, the salient issue becomes recall/precision effectiveness of the generated indices (which is difficult to quantify). For example, a small, quickly generated index would not be a reasonable tradeoff if one could not use this index to locate desired data. Second, aggregate measurements (as given in Table 4) are affected by the distribution of different file types in the sample file systems. Ideally, we would have measured each indexing system against the same file system data. We did this for WAIS and Essence, but the SFS code was not available at the time we made these measurements. Instead, we attempted to interpret the measurements given in [Gifford et al. 1991]. Notwithstanding these difficulties in interpreting the measurements, we feel it is worthwhile to present quantitative comparisons of these systems.

Table 3 presents the space and time measurements for Essence and WAIS, based on file type. We do not show measurements for nested file types

File Type	Indexing Rate (KB/min)		Compression Factor vs. Index		Semantic Exploitation
	Essence	**WAIS**	**Essence**	**WAIS**	Overhead
Archive	3289.18	-	10.89	-	*low*
Binary	563.40	-	21.15	-	*high*
C	357.84	593.15	2.46	1.45	*high*
CHeader	164.87	342.93	1.33	0.65	*high*
Command	168.20	277.30	1.23	0.43	*high*
Dvi	278.71	2160.00	0.79	1.76	*high*
Mail	3718.12	1071.89	9.44	0.74	*low*
Makefile	421.05	648.65	2.33	0.86	*high*
ManPage	165.67	661.59	3.34	1.18	*high*
News	1913.29	329.24	20.82	0.63	*low*
Object	2588.75	-	15.93	-	*low*
Patch	7218.00	993.30	80.20	2.00	*low*
Perl	282.50	713.68	2.05	0.88	*high*
PostScript	1151.19	765.60	4.56	1.67	*low*
RCS	1293.16	614.25	5.91	1.27	*low*
README	390.00	400.83	0.68	0.65	*high*
SCCS	315.79	1500.00	3.12	3.52	*high*
Tex	598.93	385.52	2.09	0.92	*low*
Text	4699.59	1346.67	8.13	1.37	*low*
Troff	703.01	2036.92	7.04	1.97	*high*

Table 3: Essence and WAIS Time and Space Measurements Based on File Type.

File System	Indexing Rate (KB/min)			Compression Factor vs. Index		
	Essence	**WAIS**	**SFS**	**Essence**	**WAIS**	**SFS**
NFS	1489.58	891.40	712.00	14.40	1.27	6.82
Anonymous FTP	1826.17	897.01	712.00	4.93	1.44	6.82
Average	1657.88	894.21	712.00	9.67	1.35	6.82

Table 4: Weighted Time and Space Averages Based on File Type Frequencies

here. Those measurements are discussed in Table 6. Nor do we show measurements for SFS in this table, because transducer-specific information was not available. Also, note that the indexing costs shown for Essence include the time and space needed to indices - not just the summaries that are produced as an intermediate step. As indicated in Table 1 and with a '-' in Table 3, WAIS and SFS cannot index all of the file types that Essence can. Table 3 shows that because there is a high amount of overhead associated with interpreting the semantics of a file type, Essence indexes slower than WAIS for some file types. Essence indexing is faster than WAIS for file types that have a low amount of semantic interpretation overhead.

Table 4 presents weighted averages of the space and time measurements in Table 3, based on the file type frequencies and average file sizes (as measured in Table 1). The weighted averages were computed using the formula:

$$\sum_{i=0}^{n}(f_i a_i)v_i,$$

where f_i is the frequency associated with file type i, a_i is the average file size associated with file type i, v_i is the the indexing rate or the compression factor value from Table 3 associated with file type i, and n is the number of file types supported by the system. $f_i a_i$ is used to normalize the measurements, to reflect only the system's supported file types. In particular, only non-nested file types are included in the aggregate measurements for WAIS and SFS (since those systems do not support nested files), while all types (including nested files) are included in the Essence measurements. We discuss the ''unraveling'' costs of dealing with nested file structures in Table 6.

The Essence and WAIS measurements were performed on a Sun Microsystems 4/280 server running SunOS 4.1.1, with a local SMD disk. The SFS measurements were performed on a Microvax-3 running UNIX version 4.3BSD [Gifford et al. 1991]. This machine is approximately one-third as fast as the Sun 4/280.

Table 4 shows that Essence can index data faster than WAIS. Taking into account the slower machine on which SFS was measured, SFS appears to index data somewhat faster than Essence does.

Essence obtains about a 10:1 index compression factor on the file types that it supports, compared to WAIS (1:1), SFS (7:1), and archie (765:1) [Emtage & Deutsch 1992]. These measurements are not perfect, because detailed SFS measurements were not available.

Table 5 shows the percentage of data in the measured file systems that Essence, WAIS, and SFS were successfully able to interpret and index. The NFS file system contained many custom file formats that the indexing systems were unable to interpret. However, the anonymous FTP file system contained many more common file formats. Even though Essence only supports a relatively small number of common file types (21), it can index 75% of the data found in an average file system – far greater than WAIS (33%) or SFS (18%).

We found that seventy-eight percent of the files in our anonymous FTP had nested structure. These measurements indicate that supporting nested file structures is essential for such file systems. In contrast, only one percent of the files in the NFS file system had nested structure. In the future nested file structures may become less common, as they mostly represent inadequacies of current file systems and remote access protocols. For example, tar files are popular in FTP file systems because they make it easier to retrieve an entire directory tree, and FTP does not provide a recursive retrieval mechanism.

	Essence	**WAIS**	**SFS**
Anonymous FTP	98.51	48.47	27.56
NFS	50.70	17.88	8.11
Average	74.61	33.18	17.84

Table 5: Percentage of Interpretable Data

Table 6 shows how much overhead the prototype incurs when indexing nested files in the measured anonymous FTP file system. In this table, the *Original Data* row concerns the data which reside in the anonymous FTP file system. The *Processed Data* row concerns the data that Essence processes while indexing the *Original Data*. These data include all of the original files and each file within a nested file structure. For example, given the file *foo.tar.Z* from the example in the previous Nested File Structure section, *foo.tar.Z*, *foo.tar*, *foo.c*, *foo.h*, *Makefile*, and *README* are all included in the *Processed Data*. The *Summarized Data* row concerns the data on which summarizers are run. For example, *foo.c*, *foo.h*, *Makefile*, and *README* are all included in the *Summarized Data*. The *Summary Output* row concerns the resulting summary files. The resulting index of the summary files consumed 12.94 megabytes.

Note that this compression ratio (60.22/12.94) understates the actual compression, because the indexed data actually consumed 262.03 MB. In particular, indexing systems (like WAIS) that do not support nested structure would have to leave the data uncompressed. Hence, we actually achieve a two-fold space reduction. WAIS would need to keep the uncompressed data around, and then would generate an index whose size was comparable to the uncompressed data. Essence generates a smaller index, and can function with compressed data. Putting the numbers together, WAIS would require approximately 264 MB of space for the uncompressed data plus index (basically, twice the size of the Summarized data), while Essence

requries only 73 MB total – a 72% space savings over WAIS.

Analysis of Keyword Quality

Qualitative analysis of information retrieval systems is difficult. *Recall/precision* measurements are difficult to obtain, since they rely on hand-chosen reference sets [Salton & McGill 1983], and hence do not scale well to measuring large information collections. More effective measurements might be obtained by evaluating the effectiveness of a system from experience with an active user community. We have made the Essence prototype is publically available to allow users to make their own subjective judgements.

	Total Number of Files	Total Size (in MB)
Original Data	1213	60.22
Processed Data	6409	262.03
Summarized Data	5334	132.36
Summary Output	5334	15.87

Table 6: Nested File Structure Overhead

Future Directions

On-the-Fly Nested File Summarizers

The Essence prototype relies heavily on the file system to implement nested file structure interpretation. This implementation degrades performance when indexing files with nested file structures (as shown in Table 6), because it causes a large amount of disk I/O. An in-memory implementation would significantly improve performance, by drastically cutting file system I/O. We are currently considering such an implementation, based on the GNU ''tar'' program, which supports an option to output extracted files to stdout.

Summarizers

The prototype currently supports twenty-one summarizers. Expanding Essence's summarizer suite to support more file types would further increase its effectiveness.

Anonymous FTP Indexing

Currently, the Essence index for the anonymous FTP site at ftp.cs.colorado.edu is available through WAIS using the *aftp-cs-colorado-edu.src* WAIS source. However, we would like to make more anonymous FTP sites available through WAIS, using Essence-based indexes. Using Essence to index public archives allows remote users to search information based on conceptual descriptions and to view summaries before retrieving files. This would help decrease the network traffic of unwanted files.

Record-Level Indexing Support

WAIS supports indexing and retrieving information with record-level granularity (e.g., allowing a file containing many electronic mail messages to be treated as a sequence of mail records). Essence only supports indexing and retrieving information with file-level granularity. A future improvement would be to modify Essence to support record structured files.

File Tree Pruning

The design of Essence's file classification stage includes the ability to identify promising files to index within a file system, in addition to type information for each file. Our current prototype does not select file system subsets – it simply indexes whatever file trees are specified. A future improvement would be to add selection criteria to the prototype (e.g., pruning files from consideration based on their location in the name tree, names/types, or sharing history). This would further refine the quality of indexes, and reduce the space required for indexing an entire file system.

Summary

The increasing abundance of inexpensive local disks creates resource discovery problems even in locally distributed file systems. The Internet resource discovery tools that have achieved popular acceptance over the past two years are not well suited to general purpose file systems, because of the irregular organization, the range of different degrees of information structure, and the generally low sharing value of information in such file systems.

In this paper we presented the Essence system, which generates file summaries based on an understanding of the semantics of the various types of files it indexes. The summaries are useful both for producing searchable indexes, and for allowing users to retrieve and browse small summaries before deciding whether to retrieve a large file across a slow network. Simple techniques to exploit file semantics yield compact yet representative indexes for both textual and binary files. The indexes generated in this fashion are more content-rich than archie's index, yet more space efficient than WAIS indexes.

Essence provides an integrated system for classifying files, defining summarizer mechanisms, applying appropriate summarizers to each file, and traversing a portion of a file system to produce an index of its contents. Importantly, Essence understands nested file structures (such as uuencoded, compressed, ''tar'' files), and recursively unravels such files to generate summaries for them. The ability to index nested files and many other file types allows Essence to be used in a number of useful settings, such as anonymous FTP archives.

Essence can index more data types, index data faster, and generate smaller indexes than WAIS or the MIT Semantic File System. Our prototype generates WAIS-compatible indexes, allowing WAIS users to take advantage of the Essence indexing methods.

Prototype Availability

The Essence prototype, including its source code and WAIS modifications, is publically available by anonymous FTP from ftp.cs.colorado.edu in /pub/cs/distribs/essence. The prototype is written in the C and Perl programming languages [Kernighan & Ritchie 1988, Wall & Schwartz 1991].

Acknowledgements

This material is based upon work supported in part by the National Science Foundation under grant NCR-9105372, and a grant from Sun Microsystems' Collaborative Research Program.

We thank Sean Coleman, Jim O'Toole, David Wood, and the USENIX program committee for their helpful comments on this paper.

References

[Berners-Lee et al. 1992] T. Berners-Lee, R. Cailliau, J. Groff and B. Pollermann, World-Wide Web: The Information Universe, *Electronic Networking: Research, Applications and Policy*, 1(2), Meckler Publications, Westport, CT, Spring 1992.

[Budd & Levin 1982] T. A. Budd and G. M. Levin, A UNIX Bibliographic Database Facility, Tech. Rep. 82-1, Department of Computer Science, University of Arizona, Tucson, AZ, 1982.

[Emtage & Deutsch 1992] A. Emtage and P. Deutsch, Archie - An Electronic Directory Service for the Internet, Froc. USENIX Winter Conf., pp. 93-110, January 1992.

[Furlani 1991] J. L. Furlani, Modules: Providing a Flexible User Environment, Froc. USENIX Large Installation Systems Administration V Conf., October 1991.

[Gifford et al. 1991] D. K. Gifford, P. Jouvelot, M. A. Sheldon, and J. W. O'Toole, Jr., Semantic File Systems, Froc. 13th ACM Symp. Operating Syst. Prin., pp. 16-25, October 1991.

[Kahle & Medlar 1991] B. Kahle and A. Medlar, An Information System for Corporate Users: Wide Area Information Servers, *ConneXions - The Interoperability Report*, 5(11), pp. 2-9, Interop, Inc., November 1991.

[Kernighan & Ritchie 1988] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, 2nd Edition, Prentice Hall, Englewood Cliffs, NJ, 1988.

[Knuth 1984] D. E. Knuth, *The TeXbook*, Addison-Wesley, Reading, MA, 1984.

[Lamport 1986] L. Lamport, *LaTeX: A Document Prepartion System*, Addison-Wesley, Reading, MA, 1986.

[Leffler, et al. 1989] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*, Addison-Wesley, Reading, MA, 1989.

[McCahill 1992] M. McCahill, The Internet Gopher: A Distributed Server Information System, *ConneXions - The Interoperability Report*, 6(7), pp. 10-14, Interop, Inc., July 1992.

[Muntz & Honeyman 1992] D. Muntz and P. Honeyman, Multi-Level Caching in Distributed File Systems — or — Your Cache Ain't Nuthin' But Trash, Froc. of the USENIX Winter Conf., pp. 305-313, San Francisco, CA, January 1992.

[NeXT 1991] NeXT Computer, Inc., *NeXT User's Reference*, NeXT Computer, Inc., Redwood City, CA, 1991.

[Neuman 1992] B. C. Neuman, Prospero: A Tool for Organizing Internet Resources, *Electronic Networking: Research, Applications, and Policy*, 2(1), pp. 30-37, Meckler Publications, Westport, CT, Spring 1992.

[Ousterhout et al. 1985] J. Ousterhout, H. Da Costa, D. Harrison, J. Kunze, M. Kupfer, and J. Thompson, A Trace-Driven Analysis of the UNIX 4.2 BSD File System, Froc. 10th ACM Symp. Operating Syst. Prin., pp. 15-24, December 1985.

[Salton & McGill 1983] G. Salton and M. J. McGill, *Introduction to Modern Information Retrieval*, McGraw-Hill, New York, NY, 1983.

[Schwartz et al. 1992a] M. F. Schwartz, D. J. Ewing, and R. S. Hall, A Measurement Study of Internet File Transfer Traffic, Tech. Rep. CU-CS-571-92, University of Colorado, Boulder, CO, January 1992.

[Schwartz et al. 1992b] M. F. Schwartz, A. Emtage, B. Kahle, and B. C. Neuman, A Comparison of Internet Resource Discovery Approaches, *Computing Systems*, 5(4), University of California Press, Berkeley, CA, Fall 1992.

[Schwartz & Tsirigotis 1991] M. F. Schwartz and P. G. Tsirigotis, Experience with a Semantically Cognizant Internet White Pages Directory Tool, *J. Internetworking: Research and Experience*, 2(1), pp. 23-50, March 1991.

[USENIX 1986] USENIX Association, UNIX Supplementary Documents, 4.3 Berkeley Software Distribution, November 1986.

[WAIS 1992] WAIS server sources, available by anonymous FTP from think.com:/wais/wais-sources.tar.Z.

[Wall & Schwartz 1991] L. Wall and R. L. Schwartz, Frogramming Perl, O'Reilly and Associates, Inc., Sebastopol, CA, 1991.

Author Information

Darren R. Hardy earned a B.S. in Computer Science from the University of Colorado, Boulder, and is currently completing an M.S. in Computer Science. He specializes in network resource discovery, distributed systems, and information retrieval. As a systems engineer at XOR Network Engineering, Inc., he develops network support software and Internet utilities. Hardy can be reached by US Mail at the Computer Science Department, University of Colorado, Boulder, CO 80309-0430; or by electronic mail at hardy@cs.colorado.edu.

Michael F. Schwartz received his PhD in Computer Science from the University of Washington. He is currently an Assistant Professor of Computer Science at the University of Colorado, Boulder. His research focuses on issues raised by international networks and distributed systems, with particular focus on resource discovery and network measurement. Schwartz chairs an Internet Research Task Force Research Group on Resource Discovery and Directory Service, and is on the editorial boards for *IEEE/ACM Transactions on Networking* and for *Internet Society News*. Schwartz can be reached by US Mail at the Computer Science Department, University of Colorado, Boulder, CO 80309-0430; or by electronic mail at schwartz@cs.colorado.edu.