Customized Information Extraction as a Basis for Resource Discovery

Darren R. Hardy

Michael F. Schwartz

Department of Computer Science
University of Colorado
Boulder, Colorado 80309-0430
+1 303 492 3902
{hardy,schwartz}@cs.colorado.edu

March 1994; Revised February 1995

Abstract

Indexing file contents is a powerful means of helping users locate documents, software, and other types of data among large repositories. In environments that contain many different types of data, content indexing requires type-specific processing to extract information effectively. In this paper we present a model for type-specific, user-customizable information extraction, and a system implementation called *Essence*. This software structure allows users to associate specialized extraction methods with ordinary files, providing the illusion of an object-oriented file system that encapsulates indexing methods within files. By exploiting semantics of common file types, Essence generates compact yet representative file summaries that can be used to improve both browsing and indexing in resource discovery systems. Essence can extract information from most of the types of files found in common file systems, including files with nested structure (such as compressed ‘‘tar’’ files). Essence interoperates with a number of different search/index systems (such as WAIS and Glimpse), as part of the Harvest system.

1. Introduction

Publishing tools like Gopher and the World Wide Web are contributing to a dramatic increase in the volume of Internet-accessible information. Yet, it is increasingly difficult to locate relevant information among this burgeoning sea of data. Browsing a hierarchy containing millions of directories is infeasible, particularly given the erratic organization that results when the data are created by many different people. Instead, automated search procedures are needed to help users locate relevant information. For efficiency reasons, this *resource discovery* problem requires indexing the available information. In turn, building an effective index requires information extraction methods tailored to each specific environment.

As an example, consider the problem of locating information about networking protocols among the global set of Gopher data. A simple solution would be to index the menu names. Yet, doing this would not work well for standards documents named according to committee designations (such as X.25). Indexing the contents of Gopher documents would provide better support, but would produce much larger indexes and would require data type-specific content extraction procedures. For example, one would not use the same techniques to extract contents from textual documents as from binary executable programs. Furthermore, it would be useful to be able to tell an indexing tool what files *not* to index. For example, it may be unnecessary to index multiple versions of revision-controlled documents. Avoiding extraneous files can make the index more compact, and the results of searches less ‘‘cluttered’’.

The points illustrated by this example can be summarized by one observation: Different applications and different execution environments require customized means of extracting and summarizing relevant information to support resource discovery. Ideally, all data would be represented as objects, with attached content extraction methods. Because a good deal of information currently exists in non-object oriented repositories (like UNIX[1] files), we have developed a system that creates the illusion of typed objects for ordinary, unencapsulated files. To do this, we provide a framework for recognizing file types and applying type specific selection and extraction methods to files.

This paper makes two contributions. First, we decompose the information extraction problem into customizable components that allow a single system to be used in many different situations. Second, we present measurements of a tuned, well-exercised system implementation, which incorporates an understanding of UNIX file semantics and context to generate compact yet representative summaries for general collections of file data. We call our system *Essence* because of its ability to reduce large amounts of data to relatively small content summaries. Essence supports a more flexible collection of information extraction mechanisms than any related system, can handle more of the types of files found in common UNIX file systems, and achieves much more space efficient content summaries than these systems. Essence is one component of the *Harvest* system [Bowman et al. 1994b], which provides a flexible and efficient means of gathering, extracting, indexing, replicating, and caching information around the Internet. During the past two years, people in 10,800 sites in 90 countries have performed searches using Essence summary information.

We present an overview of related resource discovery tools in Section 2. In Section 3 we describe the Essence information extraction model. We discuss how Essence is used in Section 4. We discuss details of our implementation in Section 5, and measurements in Section 6. We offer our conclusions in Section 7.

2. Overview of Related Resource Discovery Systems

Throughout this paper, we use the term *browsing* to indicate the user-guided activity of exploring an information space. We use *summarizing* to indicate the automated process of extracting a representative subset of keywords or other characterizing information from a collection of files.[2] We use *indexing* to indicate the automated process of building an efficiently searchable representation of the information derived from summarizing, such as a sorted list of keywords. Finally, we use *searching* to indicate the automated process of traversing an index to answer a user’s query.

A basic feature provided by Internet information systems is allowing users to *publish* data that others may retrieve. The oldest publishing system is *anonymous FTP*, a convention that allows users to transfer files to and from machines on which they do not have accounts [Postel & Reynolds 1985]. Prospero built on anonymous FTP by allowing users to organize FTP and other files into *views* that can span sites [Neuman 1992]. The Alex file

[1] UNIX is a registered trademark of UNIX System Laboratories.

[2] As discussed in Section 4, Essence content summaries can handle arbitrary binary data. However, because the currently implemented summarizers are keyword-based, for the sake of concreteness the discussion below focuses on keywords.

system [Cate 1992] also built on anonymous FTP, providing an NFS interface to the global collection of anonymous FTP servers.

More recently, the Internet Gopher [McCahill 1992] defined a simple protocol that allows many different information sources to be joined into a relatively seamless information space. The World Wide Web (WWW) [Berners-Lee et al. 1992] also provides a publishing environment, supporting hypertext links and multimedia data. Associated with these systems are an increasing number of graphical interface clients (or *browsers*), notably Mosaic [Andreessen 1993] and its commercial successors.

With continued rapid growth in the amount of Internet information, there is increasing interest in Internet indexing tools. The earliest Internet indexing systems were Archie [Emtage & Deutsch 1992] and the Wide Area Information Servers (WAIS) system. Archie periodically gathers anonymous FTP directory listings from each of approximately 1,100 UNIX anonymous FTP archives worldwide, builds and replicates an index, and uses the Prospero protocol to provide a search interface to the index servers. Because Archie indexes only file names, a single index can hold information about many resources, but the index only supports name-based (as opposed to content-based) queries. Similar multi-site indexes have been built for Gopher [Foster 1992] and the World Wide Web [McBryan 1994].

WAIS generates indexes containing every word that appears in a set of textual documents, and provides a Z39.50-based [ANSI 1991] search and retrieval interface to these data. For non-textual documents, WAIS builds indexes based on file names. For certain types of data (e.g., various bibliographic database formats), WAIS extracts keywords based on knowledge of the particular document type. Because of the popularity of graphical WWW browsers, people increasingly often use WAIS to provide an index of documents at a WWW server, and use the WWW forms interface to allow users to interact with the index (rather than using the Z39.50 protocol).

The WHOIS++ service gathers templates describing information content and administrative information about each participating site [Weider, Fullton & Spero 1992]. WHOIS++ defines a means of interconnecting index servers into meshes based on duplicate-eliminated lists of keywords from all documents on a server.

The system most closely related to Essence is the MIT Semantic File System (SFS) [Gifford et al. 1991]. SFS exploits knowledge of file naming conventions and content to determine file types, and then runs type-specific *transducers* to extract keywords for an index. SFS defines a *virtual directory* abstraction, interpreting directory names as queries against the index, and providing query results as files in virtual directories. In contrast, Essence provides a content extraction engine that is independent of how the data are stored and the higher level abstractions through which they are exported to users. Essence can therefore be used in different systems (such as an Internet retrieval engine or database system) more naturally. For example, the easiest way to provide a WWW interface to SFS would be to co-locate the SFS name space with the part of the name tree operated on by the HTTP server process (the "httpd root"). Doing this would allow users to perform SFS-based queries through the usual SFS virtual directory mechanism. On the other hand, to make use of the familiar WWW forms interface, it would be necessary to build an "application-level" gateway to translate between the virtual directory interface and the WWW forms interface. In contrast, we created a WWW interface for Essence, but without such a gateway. Instead, we simply export the content summaries into a subsystem that provides a forms interface to indexed content summaries, as part of our Harvest system. Beyond these differences, Essence supports more of the file types found in common file systems, and generates more compact summaries than SFS does.

The Nebula system also gathers keywords from file system data, but uses an explicit typing mechanism rather than heuristic techniques for recognizing files [Bowman et al. 1994c]. Nebula uses a set of grammars to extract the data. Essence allows arbitrary programs to extract data, including programs that use grammars to describe the extraction algorithms.

More recently, Gifford's group built a system called the Content Router [Sheldon et al. 1994]. This system extracts brief descriptions from from WAIS source and catalog files, and uses this information to support global queries across approximately 500 WAIS servers.

The reader interested in an overview of resource discovery systems and their approaches is referred to [Schwartz et al. 1992]. For a motivation of the research problems addressed by Harvest (of which Essence is one component), the reader is referred to [Bowman et al. 1994a].

3. Information Extraction as a Basis for Resource Discovery

Essence decomposes the information extraction problem into components that are independent of how the data are stored, updated, or exported. Decoupling the extraction process from the storage system, update mechanism, and export interface provides a flexible substrate on top of which to build resource discovery systems, allowing the extraction mechanisms to be tuned without changing the rest of the system, or to be moved to a new system. Well tuned extraction methods can improve browsing by reducing the amount of information that users must peruse. Tuning can also make extracted data more compact by excluding unimportant keywords, such as common programming language constructs in source code files. A well-tuned extraction engine can also make searching more precise, by biasing the weights of keywords that are known to be important (such as those extracted from document titles and author lists).

The main premise behind Essence is that information extraction is most effective when exploiting the semantics of particular types of files and particular execution environments. For example, by recognizing a file as a document from a particular word processing system, Essence can apply knowledge of that system's syntax to extract the document's authors, title, and abstract. Moreover, by exploiting knowledge of the organization of a file system, Essence can avoid extracting information from files known not to be of widespread interest (e.g., files of interest only to an administrative part of a company).

To exploit these types of semantics, Essence breaks information extraction into the four steps illustrated in Figure 1. The *type recognition* step uses various methods to determine a file's type. While performing this step, Essence sometimes encounters files that are encoded according to some presentation layer [Tanenbaum 1988] format (such as compression or multi-file ‘‘bundling’’). The *presentation unnesting* step transforms such files into an unnested format. Each file's name and typing information are then examined by the *candidate selection* step, to select which objects are to be summarized. Finally, the *summarizing* step applies a type-specific extraction procedure to each selected object, and places the resulting data into a attribute-value stream format (discussed in Section 5).

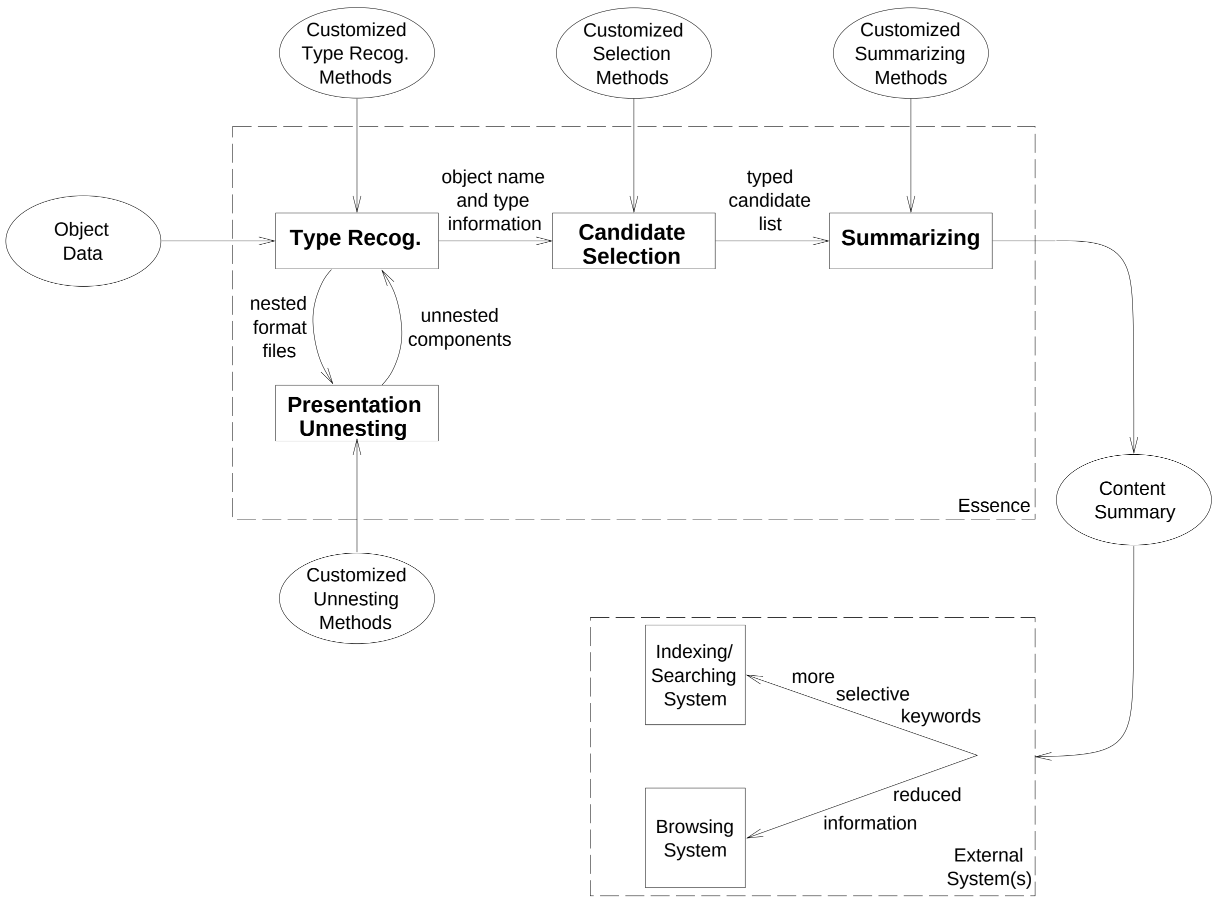


Figure 1: Information Extraction Steps

Each of the steps in Figure 1 can be customized. This allows tailoring to suit particular environments, but also means that the semantics of searches depend on the particular algorithms used. Our implementation provides a set of commonly useful defaults, as documented in Table 3 and in the Harvest User's Manual [Hardy & Schwartz 1995].

Note that because Essence defines only the information extraction interface, it can be run from any system. For example, one could run Essence from a system that invokes summarizers in real time as files are updated. One can also summarize a single file several times using different summarizers.

We consider the steps in more detail below.

Type Recognition

In an explicitly typed file system (such as a PC containing OLE [Microsoft Inc. 1993] documents), the type recognition step can simply extract type information from the file system itself. For untyped file systems (such as UNIX), type recognition must use a variety of heuristics to recognize files. For example, this step can exploit file naming conventions (e.g., noting that PostScript files are often named with the extension ‘‘.ps’’). Implicit typing can also be done by inspecting data within a file (e.g., noting that PostScript files begin with the string ‘‘%!’’).

Presentation Unnesting

Along the way to typing files, the type recognition step sometimes encounters files encoded according to one or more presentation layer data transformations. These transformations arise because of heterogeneity and other complexities in a distributed environment, and include operations such as compression and ASCII-encoding. Table 1 lists a variety of presentation formats that arise in common file system environments.

Format	Typical Purpose	Example Software
Compression	Storage and transmission efficiency	UNIX *compress* [Welch 1984] GNU *gzip*
Encryption	Privacy and security	UNIX *crypt* [NBS 1977] PGP [Zimmermann 1993]
Standardized data representation	Accommodating heterogeneous hardware-level data representations	Sun XDR [Sun 1987] ISO ASN.1 [ISO 1987]
ASCII-encoding	Transmitting binary data via electronic mail	UNIX *uuencode* Macintosh *BinHex*
File ‘‘bundling’’	Grouping related files for distribution	UNIX *tar* UNIX *shar* PC *ZIP* Macintosh *StuffIt*
	Grouping related executable library code for link editing	UNIX *ar*
Compound document structuring	Structured, type-encapsulated documents	MIME [Borenstein & Freed 1993] OLE [Microsoft Inc. 1993]

Table 1: Popular Presentation Nesting Formats

Unlike the *types* expected by the candidate selection step, presentation *formats* are semantics-independent, and hence imply nothing about how best to select or summarize data. For the purposes of information extraction, presentation layer transformations must simply be ‘‘unraveled’’ to expose the underlying files, whose types *do* imply semantics that can be exploited during candidate selection and summarizing.

When a presentation-nested file is encountered, it is unnested into one or more result files. The result files themselves can also be nested. For example, uuencoded, compressed tar files are commonly used for transmitting entire UNIX directories through electronic mail. As illustrated in Figure 1, Essence handles this case by iterating the type recognition and presentation unnesting steps. The final result of this iteration is a set of typed objects.

In addition to unnesting the input files, the presentation unnesting step also keeps a record of the nested origin of each unnested file, for use by the candidate selection and summarizing steps. For example, after the constituent files have been unnested from a directory, the candidate selector can note that the files came from a directory that contained both the source (‘‘.c’’) and object (‘‘.o’’) files for a program, and choose to exclude the object files. The summarizer can use the file-nesting linkage information to decide that an entire directory should be summarized as one unit, so that searches against any of the extracted keywords will match the whole directory, rather than one file in the directory. Doing so reduces the amount of result ‘‘clutter’’ at search time.

Presentation-nested files are prevalent in anonymous FTP file systems, multimedia documents, and PC word processing documents. While nested FTP files arise mainly because of deficiencies in FTP for transmitting entire directories, compound documents are a useful and powerful paradigm. They represent a step towards object-oriented data encapsulation. We believe extracting information from presentation-nested files will be increasingly important in the future.

Candidate Selection

Given a set of typed objects, the candidate selection step chooses objects to summarize. This step allows the system to remove files that would contribute unnecessary information to an index, cluttering the results of searches. Candidate selection is similar to the notion of *stop lists* introduced in the information retrieval literature [Salton 1986]. However, our model allows for more general selection conditions, such as regular expressions for filtering names and types. Our model also allows more complex selection procedures, that attempt to eliminate redundancy among related files. For example, we have implemented a selector that attempts to eliminate object code that can be derived from available source code files. In such a case, the candidate selector gives preference to types that can be more effectively summarized (source code in this case).

Pruning based on name can be useful for eliminating certain files known to contain unneeded information, such as old versions of files signified by extensions ending in ‘‘.bak’’. Pruning based on type can eliminate files for which summarizing procedures are not capable of extracting much useful information. This is most commonly the case for files whose types were not recognized during the type recognition step.[3]

Another aspect of tuning candidate selection is customizing the choices based on the environment being summarized. For example, anonymous FTP archives typically contain popular documents and software packages, which exhibit heavy sharing [Danzig, Hall & Schwartz 1993]. In contrast, general-purpose file systems typically contain mostly user-specific data that exhibit relatively little sharing [Muntz & Honeyman 1992, Ousterhout et al. 1985]. The candidate selection procedures can prune the name space based on such source-specific selection criteria. This is important, because summarizing and indexing narrowly shared data will mean that search results are ‘‘cluttered’’ by uninteresting matches.

Summarizing

The summarizing step applies an extraction procedure (called a *summarizer*) to each selected object, based on the type information uncovered in the type recognition step. For example, a summarizer for UNIX manual pages understands the troff syntax of these documents, as well as 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.

The summarizing step also extracts some information independent of file type, including owner, group, and full file name. Moreover, it lets users add keywords manually if desired. These type-independent processing steps allow some summarizing information to be included even for files of unrecognized type.

Summarizers can extract keyword information from both textual and binary files. For example, many binary executables have related textual documents describing their usage, from which keyword information can be extracted. The current implementation only extracts textual keywords found within the binary files, but the model would permit more complex summarizers that, for example, used speech recognition algorithms to index audio data.

If the type recognizer provides relationships between file types, summarizers can share code with one another using a multiple inheritance scheme. For example, our implementation recognizes UNIX manual pages as a special case of troff formatting source, and inherits extraction support from the formatting source summarizer.

Example File Processing

Figure 2 illustrates an example of the set of steps a compressed ‘‘tar’’ file might go through on the way to being summarized. After unnesting the compression and tar formats, a set of extracted files are typed and passed to the Candidate Selection step. Recognizing that both the source (‘‘Essence.ms’’) and formatted (‘‘Essence.ps’’) versions of a paper are available, the selector extracts information only from the source version, since that version contains more effectively summarizable information. In the example the source code is also excluded from

[3] It may still be useful to ‘‘summarize’’ unrecognized files using a type-independent summarizing mechanism − for example, extracting the file’s name.

summarizing. Finally, keywords are extracted from the selected files and placed into *SOIF* format (described in Section 4), based on knowledge of the semantics of each file type and local site preferences about what data should be extracted from each file type.

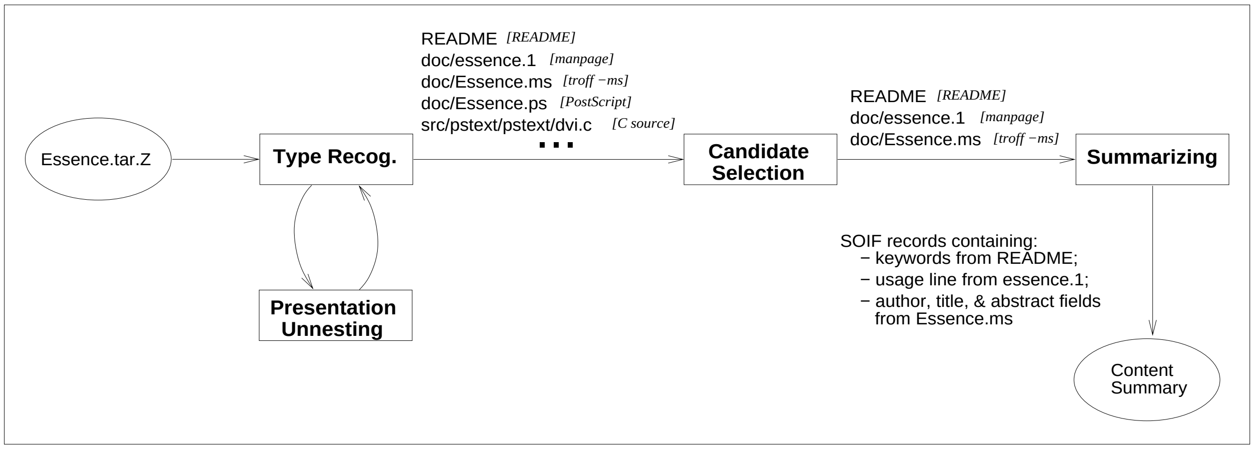


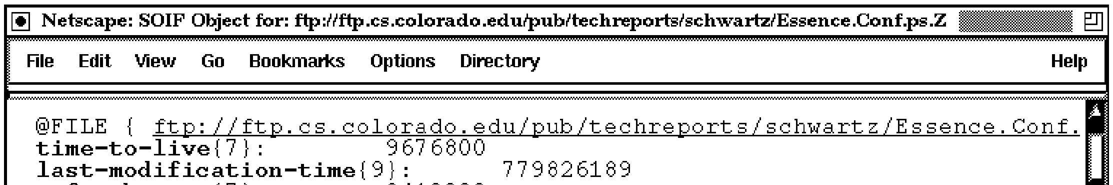
Figure 2: Example File Processing

4. Using Essence

In our initial implementation [Hardy & Schwartz 1993], the system administrator would specify a list of files and directories to be summarized, and Essence would recursively descend into any directories that were specified, generate content summaries, and pass these summaries to a version of WAIS modified to understand the Essence summary file format. In 1994 we separated the directory enumeration functionality and WAIS dependencies from the core content extraction engine, and incorporated Essence into the *Gatherer* subsystem of Harvest [Bowman et al. 1994b]. The Gatherer allows users to specify FTP, Gopher, HTTP, news, and local file system Uniform Resource Locators (URLs), pointing either to individual documents or to collections to be enumerated (e.g., by recursively listing FTP directories or by following WWW hypertext links). After enumerating and retrieving the specified documents, the Gatherer runs Essence and exports the Essence content summaries via a TCP/IP server interface. Harvest *Brokers* retrieve the content summaries, build indexes using a variety of different index/search engines (including WAIS, Glimpse [Manber & Wu 1994], and Nebula [Bowman et al. 1994c]), and provide WWW query interfaces to the indexes. Each Broker is intended to focus on a particular topic or community of interest.

Essence represents content summaries in an attribute-value stream format called the *Summary Object Interchange Format (SOIF)*. This format preserves file structure information, for example by indicating that a particular keyword came from a document title. SOIF provides a means of bracketing collections of summary objects, allowing Harvest Brokers to retrieve SOIF content summaries from a Gatherer for many objects in a single, efficient compressed stream.

When a user queries a Harvest Broker, the result contains a list of matching documents, including hypertext pointers to each source document and optionally to the matching lines from each content summary and to the content summaries themselves. As an example, we issued the query "content AND indexing" at a Broker we built that focuses on software and documents concerning Networked Information Discovery and Retrieval (NIDR). This Broker indexes over 1,500 objects located at 80 sites. The result included 33 matches in a variety of formats (compressed tar distributions, PostScript files, etc.), locating many of the widely known content indexing systems (WHOIS++, Essence, PH, SOLO, etc.). Figure 3 shows the SOIF-formatted content summary reached via a hypertext link from the query result corresponding to our earlier Essence conference paper [Hardy & Schwartz 1993]. The partial-text field includes the keywords extracted from the uncompressed PostScript file. Note also that Essence generates a number of fields that are used by other parts of Harvest. For example, Brokers use the MD5 cryptographic checksum to perform duplicate suppression.


Netscape: SOIF Object for: ftp://ftp.cs.colorado.edu/pub/techreports/schwartz/Essence.Conf.ps.Z
File
Edit
View
Go
Bookmarks
Options
Directory
Help

```
@FILE { ftp://ftp.cs.colorado.edu/pub/techreports/schwartz/Essence.Conf.
time-to-live{7}:          9676800
last-modification-time{9}:       779826189
```

Figure 3: Example Essence-Extracted Content Summary

The Harvest User's Manual [Hardy & Schwartz 1995] describes the Gatherer and Broker components in more depth and provides a formal description of SOIF, a list of common SOIF attribute names and meanings, and documentation about the SOIF processing routines.

5. Implementation

The Essence implementation currently focuses primarily on files found in the UNIX file system environment. The software consists of approximately 33,700 lines of mostly C and Perl code, including a number of summarizers adopted with minor modifications from other software packages.

Below we describe the implementation of the steps outlined in Section 3, including a number of performance optimizations and other improvements we made in response to users' suggestions and our own experiences with the system.

5.1. Type Recognition

Essence recognizes file types using a combination of file and site naming conventions, content testing, and user-defined methods. It can also use explicit file typing information, for environments that contain such information.

Naming conventions often correctly identify file types. For example, file names with a ''.ps'' extension are typically PostScript files, while files whose names contain the string ''README'' typically hold information about an entire directory or source distribution. Figure 4 shows some entries from the Essence configuration file that specifies naming conventions for type recognition. In this file, the first field is the file type, and the second field is a regular expression for the corresponding file naming convention.

```
Compressed                        .*\.Z
Makefile                          .*[mM]akefile.*
PostScript                        .*\.(ps|eps)
Revision Control System (RCS)     .*,v
TarArchive                        .*\.tar
```

Figure 4: Example Configuration File for Naming Convention-Based Type Recognition

In addition to using naming conventions, Essence examines file contents to try to determine file types. In particular, many UNIX files contain an identifying number (called a *magic number*) at the beginning of the file. For example, Sun SPARC binary executables start with the hexadecimal number *0x8103*, while Sun Pixrect images start with the hexadecimal number *0x59a66a95*. Furthermore, particular strings within a file may indicate the file type. For example, PostScript images start with the string ''%!'', while electronic mail files contain lines that begin with header field strings such as ''From:'' and ''To:''.

Essence uses code from the UNIX *file* command [USENIX Association 1986] (but integrated into the main Essence process to avoid excess forking) to support this form of type recognition. As with exploiting file naming conventions, locating identifying data is a rule-based technique, expressed with regular expressions. To modify the rules, users modify the *file* command's *magic* configuration file. Figure 5 shows some sample entries from this file, where the first field is the byte offset of the identifying data, the second field is the type of this field, the third field is the identifying data itself, and the last field is a description of the corresponding file type.

```
0   string   /*               C program text
0   string   \037\235         Compressed data
0   long     0x8103           Sun SPARC binary executable
0   string   #!.*/bin/perl    Perl program
0   string   %!               PostScript image
```

Figure 5: Example Configuration File for Data Content-Based Type Recognition

Creating a suitable *magic* file is not trivial, because the identifying data 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 is common for binary formats, which usually depend on a single magic number. We built the Essence *magic* file through experimentation.

Some file types are difficult to identify using only simple naming conventions and content testing. Therefore, Essence lets users provide their own customized stand-alone programs to identify file types. These programs can use a variety of techniques. For example, we implemented a program that uses simple heuristics to identify software distribution directories (noting that they contain C source code, Makefiles, and README files), for use with one of the candidate selectors we implemented.

As an optimization, Essence applies file type recognition methods in increasing order of expense: file and site naming conventions followed (if necessary) by the ‘‘file’’ command followed (if necessary) by customized recognizer programs. Also, Essence caches file information from the *stat* system call, for reuse in the presentation layer unnesting step.

5.2. Presentation Layer Unnesting

Table 2 indicates the seven presentation unnesting formats that Essence currently supports, along with the corresponding unnesting methods. As an optimization, Essence unnests certain common combinations (such as compressed tar files) as a single ‘‘pipelined’’ step. This avoids the disk I/O overhead of creating intermediate files. As a second optimization, we perform presentation unnesting, candidate selection, and summarizing in a single pass for the *Archive* file format. Doing so reduces disk I/O and processing costs. This optimization is possible because extracting and combining the symbol tables from the unnested object files is equivalent to extracting the symbol table directly from the archive file. We are able to exploit this fact because of the use of customized information extraction.

Nesting Format	Unnesting Description
Archive	Extract archived binary relocatable object files
Directory	Extract individual files
Compressed	Uncompress
GNUCompressed	Uncompress GNU-format file
ShellArchive	Extract contained files
Tar	Extract archived files
Uuencoded	Decode ASCII-encoded contents

Table 2: Essence Unnesting Actions

5.3. Candidate Selection

Essence lets users prune the set of summarized files based on lists of file types as well as regular expressions for filtering based on file name. Candidate selection can also perform more sophisticated processing. For example, if a set of files was unnested from a single directory that contained only software and documentation files, the candidate selector recognizes the collection as a *SourceDistribution*. In this case, it invokes special rules for excluding extraneous files − for example, excluding PostScript files for which the system has already processed compressed versions of the same PostScript files. This can be useful when processing data from archive sites that provide data in both compressed and uncompressed form.

While candidate selection is conceptually one step, for performance reasons it is broken into two parts. Some files are rejected based on a name-based stop list before type recognition, avoiding the overhead of identifying a file’s type. Files that pass this filter are then typed and passed through a second filter, which selects files based on their types.

5.4. Summarizing

Essence's summarizers include both stand-alone UNIX programs and regular expression-based rules, which are easy to write and integrate into the system. Each summarizer is associated with a specific file type and understands the type well enough to extract summary information from the file. Currently, Essence supports summarizers for twenty-seven file types, as listed in Table 3. This table defines the default actions; individual actions can be changed and new types supported through appropriate customizations.

File Type	Summarizer Description
Audio	Extract file name
Bibliographic	Extract author and titles
Binary	Extract meaningful strings and manual page summary
C, CHeader	Extract procedure names, included file names, and comments
Dvi	Invoke the RawText summarizer on extracted ASCII text
FAQ, README	Extract all words in file
Framemaker	Extract text and pass through RawText summarizer
Font	Extract comments
HTML	Extract anchors, hypertext links, and selected fields
Mail	Extract certain header fields
Makefile	Extract comments and target names
ManPage	Extract synopsis, author, title, etc., based on ‘‘-man’’ macros
News	Extract certain header fields
Object	Extract symbol table
Patch	Extract patched file names
Perl	Extract procedure names and comments
PostScript	Extract text in word processor-specific fashion, and pass through RawText summarizer
RawText	Extract first 100 lines plus first sentence of each remaining paragraph
RCS, SCCS	Extract revision control summary
ShellScript	Extract comments
SourceDistribution	Extract full text of README file and comments from Makefile and source code files, and summarize any manual pages
SymbolicLink	Extract file name, owner, and date created
TeX	Invoke the RawText summarizer on extracted ASCII text
Troff	Extract author, title, etc., based on ‘‘-man’’, ‘‘-ms’’, ‘‘-me’’ macro packages, or extract section headers and topic sentences.
Unrecognized	Extract file name, owner, and date created.

Table 3: Essence Default Summarizer Actions

The processing for these types fits into a multiple inheritance hierarchy, corresponding to the type structure illustrated in Figure 6. All types inherit type-independent processing steps, which include the file's name, owner, and group, and also let users add keywords manually. The text formatting summarizers inherit code to deal with textual extraction. The PostScript summarizer extracts text in different ways depending on what typesetting system generated the PostScript. This significantly reduced the occurrence of cases where words were split apart, and represents another example of the advantages of customized extraction. The source distribution summarizer makes the most involved use of the hierarchy, inheriting methods from the textual extractors, source code and Makefile extractors, and ManPage extractor. As discussed below, the binary extractor inherits methods from the *ManPage* extractor, so it can incorporate the keywords from the manual page for the given binary.

The following subsections provide more detail about the techniques used in some of the summarizers, representative of Essence's supported file types.

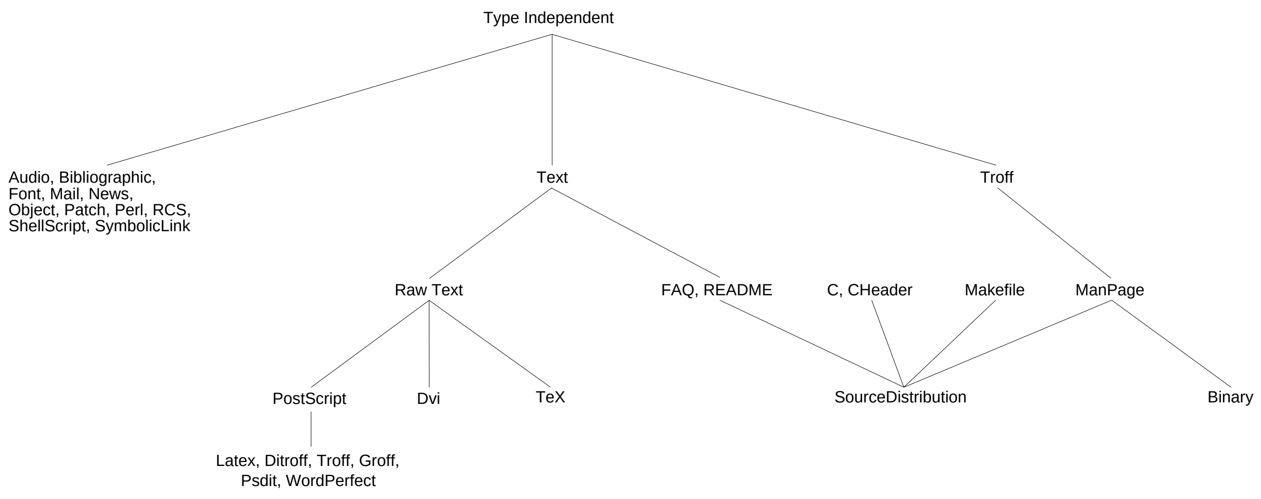

Type Independent
Audio, Bibliographic, Font, Mail, News, Object, Patch, Perl, RCS, ShellScript, SymbolicLink
Text
Troff
Raw Text
FAQ, README
C, CHeader
Makefile
ManPage
PostScript
Dvi
TeX
SourceDistribution
Binary
Latex, Ditroff, Troff, Groff, Psdit, WordPerfect

Figure 6: Multiple-Inheritance Type Hierarchy Basis for Summarizer Code Sharing

Binary

An obvious method for a *Binary* summarizer is to extract ASCII strings from the binary file, using the UNIX *strings* command. Because these strings typically contain a good deal of unimportant information, Essence filters them using heuristics that only retain strings that convey the program's purpose – searching for strings such as ‘‘usage’’, ‘‘version’’, or ‘‘copyright’’. Essence also ‘‘cross-references’’ binary executables with associated manual pages by running the *ManPage* extractor on the manual page for the given binary and incorporating those keywords into the content summary for the binary.

Raw Text, FAQ, and README Files

Raw text is difficult to summarize because it is unstructured. Rather than attempting a complex natural language processing approach, Essence assumes that the most useful keywords in raw text files lie near the beginnings of files. This heuristic works well, for example, with paper abstracts or tables of contents. For this purpose, the *RawText* summarizer extracts all of the words from the first one hundred lines of each file. Because many users of our initial prototype reported that this technique did not capture enough of the content of raw text files, we added the capability to extract the first sentence of every paragraph after the first one hundred lines.

Some unstructured text, like FAQ (Frequently Asked Questions) and README files, typically contain relevant, concise information about a software distribution or application. The *README* summarizer extracts every word from its input files, typically providing useful keywords without consuming too much space.

Text Formatting Source

Although text formatting source (such as TeX, Troff, or Word Perfect) is more structured than raw text, effectively summarizing these files is difficult unless semantic information is also available. For example, plain Troff files or Troff files using the ‘‘-me’’ macro package are difficult to exploit semantically, because their syntax is associated with formatting commands (such as font size or line spacing), rather than more conceptual commands (such as macros to indicate an author's name or a paper's title). Troff files using the ‘‘-ms’’ or ‘‘-man’’ macros are much easier to summarize, because they use such conceptual macros.

Essence supports a sophisticated summarizer for Troff using any of the ‘‘-me’’, ‘‘-ms’’, and ‘‘-man’’ macro packages. The *TeX* summarizer only extracts ASCII text from TeX (or LaTeX) files using the UNIX *detex* program. Exploiting TeX semantics would be a straightforward extension of the methods used in our *Troff* summarizer.

The *Dvi* and *PostScript* summarizers extract keywords by first converting the file to its corresponding text, and then running the *RawText* summarizer on the extracted ASCII text. In our original implementation [Hardy & Schwartz 1993], dvi files were first converted to PostScript using *dvips*, and then processed using the *PostScript* processor. After experimentation, we changed this summarizer to use the *dvitty* program instead, because it does a better job of extracting the text from the page positioning codes. We also adopted the improved *PostScript* summarizer illustrated in Figure 6.

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 file names, and comments from a C source code file. The *Object* summarizer extracts the symbol table from an object file.

Some extra semantics can be exploited in the case where source code is included in part of a directory recognized as a *SourceDistribution*. First, the summarizer uses the file-nesting linkage information recorded at presentation unnesting time to decide that the entire directory should be summarized as one unit. Second, the summarizer attempts to extract copyright information in the source or object files. This information typically contains project, application, or author names. The summarizer also extracts keyword information from README files, because these files often contain useful information about the directory's contents.

Other Possible Summarizers

Many other summarizers are possible. For example, Lisp or Pascal summarizers could be implemented with straightforward extensions to the existing source code summarizers. In contrast, audio or image summarizers would be difficult to implement, because summarizers are currently limited to keywords. If they were not limited to keywords, one possibility would be to sample a bitmap file down to an icon. While this would not easily support indexing, it could be used to support quick browsing before retrieving an entire image across a slow network link, similar to what is done by the Dienst system [Davis 1994].

6. Evaluation

In this section we evaluate four aspects of the Essence implementation: keyword quality, space and time efficiency, the overall costs of each of the model steps, and the completeness of the supported presentation unnesting and summarizer methods. Since these measurements depend on the particular summarizers being used, the current measurements should be interpreted as an *example* of how well Essence works. It is possible to improve the performance of our default summarizers by customizing them for a particular environment. For examples of more highly customized content extraction, the user is referred to the demonstration Harvest Brokers accessible via the WWW at *http://harvest.cs.colorado.edu/harvest/demobrokers.html*.

6.1. Keyword Quality

An important issue in the evaluation of any keyword-based discovery system is the quality of keywords that it supports. The usual approach to measuring keyword quality is to compute *precision* and *recall* values for a set of user queries, against a given set of documents (called a *reference set*). Intuitively, recall is the probability that all of the relevant documents will be retrieved, while precision is the probability that all of the retrieved documents will be relevant [Blair & Maron 1985]. More precisely,

$$Precision = \frac{Number\ of\ Relevant\ \text{and}\ Retrieved\ Documents}{Total\ Number\ of\ Retrieved\ Documents}$$

$$Recall = \frac{Number\ of\ Relevant\ \text{and}\ Retrieved\ Documents}{Total\ Number\ of\ Relevant\ Documents}$$

Obtaining meaningful precision and recall measurements is difficult, because they require manual analysis of what constitutes the relevant set of documents in response to a particular set of keywords. Moreover, because the notion of relevance is subjective, to be truly representative this analysis would have to be repeated separately for each of a large number of users. Doing so becomes impractical with large reference sets.

Because of these difficulties, we took a different approach to measuring keyword quality for Essence. Observing that WAIS has become a widely accepted ‘‘industry standard’’ for information retrieval, we chose to measure how closely Essence could retrieve exactly the documents that the WAIS logs indicated were relevant, for each logged WAIS search. In other words, we measured precision and recall *relative* to WAIS.

We began with logs for a popular WAIS database containing three years of full-text CACM articles, handled by a server run by Thinking Machines, Inc. WAIS logs a great deal of information, from which we extracted the keywords used for each search, the list of matching documents, which documents users retrieved,[4] and the list of

[4] When a WAIS match occurs, it is presented to the user as a headline. Users must explicitly select documents to be retrieved, based on these

documents used as relevance feedback input.[5] For each search we defined the *Logged Relevant Document Set (LRDS)* to be the set of documents that the user either retrieved or selected for relevance feedback (based on the CACM WAIS search logs). We then ran each of the logged searches against an Essence-generated index, and computed precision and recall values based on this LRDS. More precisely, for each search we make the following definitions:

$$\textit{WAIS Result Set (WRS)} = \textit{the set of documents matched by WAIS}$$

$$\textit{Essence Result Set (ERS)} = \textit{the set of documents matched by Essence}$$

$$\textit{Relevant} \text{ and } \textit{Retrieved WAIS Set (RRWS)} = WRS \cap LRDS$$

$$\textit{Relevant} \text{ and } \textit{Retrieved Essence Set (RRES)} = ERS \cap LRDS$$

Finally, we define the relative precision as the ratio of Essence precision to WAIS precision, and we define the relative recall as the ratio of Essence recall to WAIS recall:

$$\textit{Relative Essence Precision} = \frac{|RRES|}{|ERS|} \Big/ \frac{|RRWS|}{|WRS|}$$

$$\textit{Relative Essence Recall} = \frac{|RRES|}{|LRDS|} \Big/ \frac{|RRWS|}{|LRDS|} = \frac{|RRES|}{|RRWS|}$$

Note that this approach does not provide or require precision and recall measurements for WAIS itself -- such measurements would require the subjective relevance analysis discussed earlier. It simply measures how closely Essence is able to match all and only the documents that WAIS users indicated might be relevant.

Figure 7 presents the results of this experiment, for a range of result set sizes. These results were based on a set of 653 searches for which users either retrieved documents or specified relevance feedback, between October 1 and November 30, 1993.

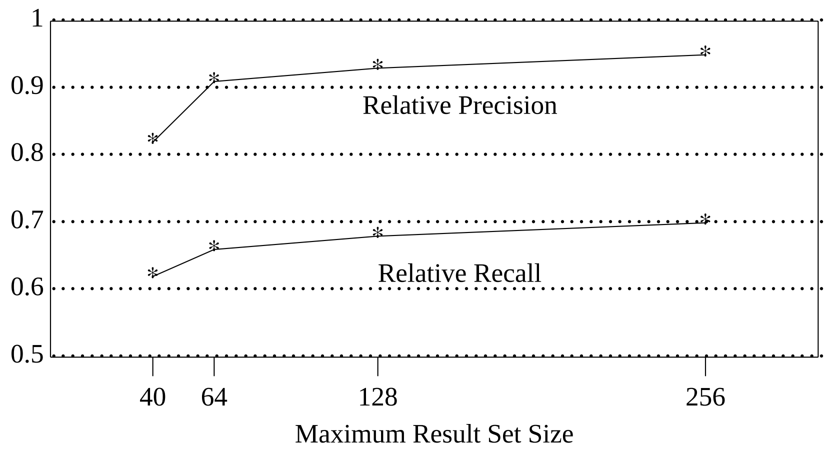


Figure 7: Relative Precision and Precision Measurements for Essence

As Figure 7 shows, Essence's precision approaches that of WAIS as the result set sizes increase. We did not attempt a larger size than 256 because users are unlikely to browse the result headlines for large result set sizes. One might expect Essence's relative precision to be greater than 1, because Essence content summaries contain fewer keywords than WAIS full text indexes do, and hence Essence-based searches might match fewer documents per search than WAIS does. Essence's relative precision is less than 1 because the server truncates result sets to a (small) user-defined size,[6] so there are cases where Essence filled the result buffer and truncated matches that WAIS

headlines.

[5] The WAIS relevance feedback mechanism lets users select a particular set of documents from the results of a query for use in recomputing weights for future queries – in effect, asking the system to locate other documents ''like these documents''.

[6] 40 is the default size using the xwais client.

found.

Relative recall is less than 1 because Essence omits some of the keywords that WAIS indexes include. In a few cases relative recall could have been improved if the CACM articles had more structure. This is because the missing keywords appeared in portions that Essence ignored, such as the reference lists. Note that this measurement provides a *lower bound* on Essence's relative recall, as it is likely that the logged WAIS searches did not locate some documents that users would have deemed relevant, which Essence could have located.[7] These measurements only highlight the times when Essence was unable to locate documents that WAIS located, but not vice versa. Moreover, because Essence can extract data from binary file types that WAIS does not understand, there are cases in which Essence would achieve higher recall than WAIS. Also, because Essence is easily customizable, its precision and recall can be improved for specific environments. Finally, users can ask Essence to perform full text indexing if they feel the added indexing space requirements are warranted.

6.2. Space and Time Efficiency

Because it only extracts keywords from selected parts of files, Essence can generate more compact indexes than a full-text system like WAIS, and can generate these indexes more quickly as well. For example, Essence's *RawText* summarizer skips much of the data. In this section we evaluate the space and time efficiency gains made possible by this approach, comparing the costs of extracting and indexing file contents for each of Essence, SFS, and WAIS. Space efficiency is important, because a single space-efficient index can represent information about many resources. Time efficiency is less of an issue because information extraction can be parallelized or performed incrementally.[8] Even so, time efficient extraction is useful, for example for supporting on-the-fly extraction and searching [Manber & Wu 1994].

We computed average space and time costs by measuring space and time requirements for each non-nested file type supported by Essence, and then weighting these measurements according to typical file type occurrence frequencies and sizes. (We discuss the costs of presentation unnesting in Section 6.4.) For this purpose we define *indexing rate* as the speed of content extraction plus indexing, and *summarizing ratio* as the size of the source data divided by the size of the generated index. This latter measure highlights the space savings provided by selective indexing compared with full text indexing. To obtain realistic mixes of file types, we performed these measurements in two environments: a departmental file system that contains commonly shared data and tools in the University of Colorado Computer Science Department, and the department's anonymous FTP file system. We chose these two environments because they typify two different common uses of file systems. Finally, we computed average time and space measurements for each system using the formulae:

$$\text{Average Summarizing Ratio} = \sum_{i=1}^{n} f_i s_i S_i$$

$$\text{Average Indexing Rate} = \sum_{i=1}^{n} f_i s_i I_i$$

Here f_i is the frequency associated with file type i, s_i is the average file size associated with file type i (in KB), S_i is the summarizing ratio associated with file type i, I_i is the indexing rate associated with file type i, and n is the number of non-presentation-nested file types supported by the system. The results of these measurements are presented in Table 4. We performed these measurements on a Sun 4/280 server running SunOS 4.1.1, with local SMD disks and 64 megabytes of main memory.

Essence achieves higher indexing rates for file types where the overhead of customized information extraction is overshadowed by the cost savings of running WAIS indexing against fewer input keywords. For example, this is true for the *RawText* summarizer, because it excludes many words from the indexing step.

While detailed measurements were not available, using numbers from [Gifford et al. 1991] we estimate that SFS processes data at 712 KB/min. However, because the SFS measurements were performed on a Microvax-3 (which is approximately one-third as fast as the Sun 4/280), SFS's indexing speed appears to be comparable with that of Essence.

The summarizing ratio measurements given in Table 4 show that Essence summaries are only 3-11% as large as WAIS indexes for the file types it supports, in the measured file systems. Again using numbers from [Gifford et al.

[7] Although WAIS provides full-text indexing, in most cases result set size limitations cause the returned documents to constitute only an essentially random subset of the actual matching documents. Therefore, it is possible that Essence can return a relevant document that WAIS missed.

[8] Harvest supports incremental updates.

File System	Indexing Rate (KB/min)		Summarizing Ratio (source size / index size)	
	Essence	**WAIS**	**Essence**	**WAIS**
Dept. Shared File System	2,589.67	1,422.89	38.81	1.16
Anonymous FTP Server	1,961.69	2,073.01	21.93	2.33

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

1991], we estimate SFS's summarizing ratio to be 6.8 -- which is intermediate between Essence and WAIS.

Filename-based indexers achieve much higher summarizing ratios than those reported above. For example, Archie's summarizing ratio is 765 [Emtage & Deutsch 1992]. The tradeoff, of course, is that filename-based indexes support less powerful searches. Because Essence is customizable, users can implement an Archie-like system by telling Essence to index only file names. For example, we are currently looking into building a new version of the Veronica search service (which provides Archie-like functionality for Gopher menus) using Harvest.

6.3. Content Capturing Efficiency

We can compute a simple measure of content capturing efficiency by combining the measurements from Sections 6.1 and 6.2. In particular, if we multiply relative precision or recall by the summarizing ratio for each system, we measure how much precision or recall is attained per unit of index space. Table 5 shows the measurements for the anonymous FTP file system. These figures show that Essence attains much more precision and recall per byte of index than WAIS does. Customized information extraction provides a much more space efficient way of achieving a given level of precision or recall than full text indexing does.

	Relative Precision * Summarizing Ratio	Relative Recall * Summarizing Ratio
Essence	20.83	15.35
WAIS	2.33	2.33

Table 5: Content Capturing Efficiency

6.4. Model Component Costs

Table 6 indicates overall costs for each of the model steps illustrated in Figure 1. These measurements were performed on the departmental anonymous FTP data, on a DECstation 5000/125 running Ultrix 4.2, with 32MB of RAM and a local SCSI disk. Presentation unnesting is quite costly because of its inherent I/O intensity, and the need to touch file data once for every unnesting operation. Interestingly, these costs roughly mirror the corresponding presentation layer costs of data transmission, which are predicted to become the dominant cost of networking in this decade [Clark & Tennenhouse 1990].

Model Step	% Time
Type Recognition	23.8
Presentation Unnesting	36.6
Candidate Selection	1.1
Summarizing	38.5
Total	100.0

Table 6: Model Component Costs

Much of Essence is implemented as interpreted scripts (in awk, sed or Perl). This choice supports easy customization and rapid prototyping, but also impacts performance. Process creation also affects performance, because many of the system components execute as subprocesses (e.g., forking the *file* command to classify files, and various programs to summarize files). As an example, to summarize and index the anonymous FTP file system in our measurements, approximately 15% of the time was spent forking UNIX processes and loading executable images. This overhead could be reduced by moving to a single shared address space environment. We believe the overhead is warranted, given the low ratio of indexing to searching operations in most resource discovery systems, and the ease that users have of creating new customized components as separate scripts. WAIS took the opposite choice, and the need to modify the source for each new file type is a frequently cited critique of the system.

The fact that Essence reduces file data to small summaries means indexing costs are also quite small. For example, using the Table 6 workload we found that, of the total time to run Essence plus WAIS indexing, only 3% of the time was spent indexing.

Table 7 shows how much space overhead Essence incurs when unnesting presentation-nested files in the measured anonymous FTP file system. In this table, *Original Data* refers to the data that reside in the anonymous FTP file system. *Processed Data* refers to the data that Essence processes while summarizing the *Original Data*, including all of the original (nested) files plus each file extracted during unnesting. *Summarized Data* refers to the data on which summarizers are run – i.e., all of the ‘‘bottom level’’ files after presentation unnesting.[9] There are many more processed than summarized data files because of the prevalence of nested file formats in the measured data. For example, a compressed PostScript file is counted as both a compressed file and a PostScript file for the count of processed data files, but only as a PostScript file in the count of summarized data files. *Summary Data* refers to the resulting content summary files.

	Total Number of Files	Total Size (in MB)
Original Data	996	92.58
Processed Data	14,348	431.69
Summarized Data	3,152	261.76
Summary Data	3,152	9.79
Index	2	10.47

Table 7: Presentation Unnesting Space Measurements

The ratio between original data and index size (92.58/10.47 = 8.8) actually understates Essence’s space efficiency compared with WAIS, because of Essence’s support for nested data. WAIS would need to keep the unnested data around, and then would generate a relatively large index. Essence can operate on nested data, and generates a smaller index. For the case of Table 7, WAIS would require 262 MB for the unnested data plus 112 MB for the index (using the 2.33 ratio from Table 4), totaling 374 MB. In contrast, Essence requires only 92.58 + 9.79 + 10.47 = 113 MB. This represents a 70% space savings over WAIS. Clearly, it is worthwhile to support presentation unnesting. WAIS could benefit, for example, by modifications that at least let it work with compressed files. While it currently allows compressed files to be retrieved, it cannot index compressed files.[10]

Essence’s space efficiency is enhanced significantly by the fact that it does not require indexes to reside on the same machine as the indexed data, like WAIS does. Instead, Harvest Brokers index Essence content summaries of remotely gathered data. As an example of the space savings from this approach, the above measurements indicate that Essence requires only 9.79 + 10.47 = 20.26 MB at the index server, a 94.6% savings over WAIS. Essence can therefore handle a much larger collection of data than is possible with WAIS. For example, we used Essence to build a content index of over 24,000 Computer Science technical reports from 300 sites around the world, using 275 MB. In contrast, WAIS would have required 9 GB -- over 50 times as much space.

[9] Note: for the purposes of these measurements, we did not eliminate any files with the candidate selection step.

[10] The ‘‘freeWAIS’’ system being supported by the Clearinghouse for Networked Information Discovery and Retrieval (CNIDR) can index and retrieve compressed files. Also, CNIDR has recently incorporated Essence into the freeWAIS software distribution.

6.5. Completeness of Presentation Unnesting and Summarizing Methods

Table 8 reports the percentage of data in the measured file systems that Essence, WAIS, and SFS were successfully able to handle. These measurements reflect the fact that 85% of the files in the anonymous FTP file system had nested structure, while only 2% percent of the files in the shared departmental file system had nested structure. The prevalence of nested files in the anonymous FTP file system caused WAIS and SFS to fare poorly in that environment. While the departmental file system contained relatively few nested files, it contained many specialized file formats that the systems were unable to interpret.

	Essence	**WAIS**		**SFS**	
		All Files	Non-Nested Files	All Files	Non-Nested Files
Anonymous FTP Server	97.50	10.40	69.31	11.39	75.94
Dept. Shared File Server	71.15	39.11	39.91	67.99	69.38

Table 8: Percentage of Interpretable Data for Essence, WAIS, and SFS

Independent of presentation layer nesting, Essence supports more of the file types found in common departmental file systems and anonymous FTP file systems than either WAIS or SFS. Although WAIS and SFS support most of the frequently occurring file types (such as *RawText* and *CHeader*), Essence supports the file types that contribute most to overall data size (such as *Archive* and *Binary*). WAIS and SFS each support various types of files that Essence does not support. Examples include MedLine and New York Times formats. There are 8 such formats understood by WAIS, and 14 understood by SFS.

Essence, WAIS, and SFS each support about 30 (partially overlapping) types of files. Among the three systems, 52 different file types are supported. 17 of these types are supported by all three systems, while 23 are supported by a single system. This large overlap and the need for custom file type support argues for a single extensible system like Essence. This would avoid the need to reimplement extraction procedures for each system, and would permit easy sharing of extraction code between the systems. Moreover, Essence is more flexible than these other systems. For example, rather than automatically recognizing file types, WAIS depends on users' explicitly specifying types for each set of files they index. Moreover, WAIS does not support any notion of partial text extraction – it either includes all keywords in a textual document or just a file name for non-textual documents.

7. Conclusions

Content indexing provides a powerful means of helping users locate relevant information among large repositories. To be most effective, content indexing must exploit the semantics of the different types of data and different execution environments in which it is used. In this paper we presented a model for customized information extraction and an implementation of this model in the Essence system. Essence allows users to associate specialized extraction methods with ordinary files, providing the illusion of an object-oriented file system that encapsulates specialized indexing methods within files.

Essence supports a more flexible collection of mechanisms for recognizing, selecting, and extracting information than either WAIS or SFS. Moreover, because Essence supports presentation unnesting, it can extract information from many more files than either WAIS or SFS for various common settings, such as anonymous FTP archives, CD-ROM collections, and local source trees.

Essence achieves much more space efficient content summaries than either WAIS or SFS, yet manages to capture most of the important keywords from summarized files. In particular, Essence achieves nearly the same precision and 70% the recall of WAIS, but requires only 3-11% as much index space and 70% as much summarizing and indexing time as WAIS. Moreover, it is much easier to write and integrate new Essence's summarizers than to make the corresponding modifications to WAIS, which requires modifying and recompiling a large piece of software. Essence also provides a more general information extraction mechanism than SFS, because it can be used with any storage system or export interface. SFS works only with a virtual directory file system abstraction.

Because Essence defines a *content summarizing* abstraction that is independent of how the data are stored and the higher level abstractions through which they are exported to users, it can support a wide range of resource discovery applications, from local file system search tools to wide area distributed archives and databases.

The Essence software is freely available as part of the Harvest software distribution. Readers can get information about Harvest (including demonstrations, papers, software, and documentation) via the World Wide Web from *http://harvest.cs.colorado.edu/*.

Acknowledgements

Panagiotis Tsirigotis implemented the source code distribution recognizer and the improved PostScript parsing program discussed in Sections 3 and 5. Mic Bowman, Peter Danzig, Jim Guyton, Jim O'Toole, Mark Sheldon, and David Wood provided helpful comments on this paper. Jonathan Goldman provided us with the WAIS logs needed to compute the measurements in Section 6.1.

This material is based upon work supported in part by the National Science Foundation under grant numbers NCR-9105372 and NCR-9204853, the Advanced Research Projects Agency under contract number DABT63-93-C-0052, and an equipment grant from Sun Microsystems' Collaborative Research Program. The information contained in this paper does not necessarily reflect the position or the policy of the U.S. Government or other sponsors of this research. No official endorsement should be inferred.

8. Bibliography

[ANSI 1991]
ANSI. *ANSI Z39.50.* American National Standards Institute, May 1991. Version 2, 3rd Draft.

[Andreessen 1993]
M. Andreessen. NCSA Mosaic Technical Summary. Tech. Rep., National Center for Supercomputing Applications, May 1993.

[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*, 2(1), pp. 52-58, Meckler Publications, Westport, CT, Spr. 1992.

[Blair & Maron 1985]
D. C. Blair and M. E. Maron. An Evaluation of Retrieval Effectiveness for a Full-Text Document-Retrieval System. *Commun. ACM*, 28(3), pp. 289-299, Mar. 1985.

[Borenstein & Freed 1993]
N. Borenstein and N. Freed. MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies. Req. For Com. 1521, Sep. 1993.

[Bowman et al. 1994a]
C. M. Bowman, P. B. Danzig, U. Manber and M. F. Schwartz. Scalable Internet Resource Discovery: Research Problems and Approaches. *Commun. ACM*, 37(8), pp. 98-114, Aug. 1994.

[Bowman et al. 1994b]
C. M. Bowman, P. B. Danzig, D. R. Hardy, U. Manber and M. F. Schwartz. The Harvest Information Discovery and Access System. *Proc. 2nd Int. World Wide Web Conf.*, pp. 763-771, Chicago, IL, Oct. 1994.

[Bowman et al. 1994c]
C. M. Bowman, C. Dharap, M. Baruah, B. Camargo and S. Potti. A File System for Information Management. *Proc. Conf. on Intelligent Information Management Systems*, Washington, DC, June 1994.

[Cate 1992] V. Cate. Alex - A Global Filesystem. *Proc. Usenix File Systems Workshop*, pp. 1-11, Ann Arbor, MI, May 1992.

[Clark & Tennenhouse 1990]
D. D. Clark and D. L. Tennenhouse. Architectural Considerations for a New Generation of Protocols. *Proc. SIGCOMM Symp.*, pp. 200-208, Philadelphia, PA, Sep. 1990.

[Danzig, Hall & Schwartz 1993]
P. B. Danzig, R. S. Hall and M. F. Schwartz. A Case for Caching File Objects Inside Internetworks. *Proc. SIGCOMM '93*, pp. 239-248, San Francisco, CA, Sep. 1993.

[Davis 1994]
J. R. Davis. *Dienst: A Distributed Interactive Extensible Network Server for Techreports.* Design Research Institute, Cornell Univ., Jan. 1994.

[Emtage & Deutsch 1992]
A. Emtage and P. Deutsch. Archie - An Electronic Directory Service for the Internet. *Proc. USENIX Wint. Conf.*, pp. 93-110, Jan. 1992.

[Foster 1992]
S. Foster. About the Veronica Service. Electronic bulletin board posting on the comp.infosystems.gopher newsgroup, Nov. 1992.

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

[Hardy & Schwartz 1993]
D. Hardy and M. F. Schwartz. Essence: A Resource Discovery System Based on Semantic File Indexing. *Proc. USENIX Wint. Conf.*, pp. 361-374, Jan. 1993.

[Hardy & Schwartz 1995]
D. R. Hardy and M. F. Schwartz. Harvest User's Manual. Tech. Rep. CU-CS-743-94, Feb. 1995. Version 1.1. Available from http://harvest.cs.colorado.edu/harvest/doc.html.

[ISO 1987] ISO. Information Processing Systems - Text Communication - Remote Operations - Part 2: Protocol Specification. Draft International Standard ISO/DIS 9072-2, International Organization for Standardization, 1987.

[Manber & Wu 1994]
U. Manber and S. Wu. GLIMPSE: A Tool to Search Through Entire File Systems. *Proc. USENIX Wint. Conf.*, pp. 23-32, Jan. 1994.

[McBryan 1994]
O. McBryan. GENVL and WWWW: Tools for Taming the Web. *Proc. 1st Int. World Wide Web Conf.*, CERN, Geneva, Switzerland, May 1994.

[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.

[Microsoft Inc. 1993]
Microsoft Inc. *OLE 2.01 Design Specification.* Microsoft OLE2 Design Team, Sep. 1993. Describes the Object Linking & Embedding environment.

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

[NBS 1977] NBS. Data Encryption Standard. *Federal Information Processing Standards Publication 46*, National Bureau of Standards, U.S. Dept. of Commerce, Washington, D.C., Jan. 1977.

[Neuman 1992]
B. C. Neuman. The Prospero File System: A Global File System Based on the Virtual System Model. *Computing Systems*, 5(4), pp. 407-432, Fall 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. *Proc. 10th ACM Symp. Operating Syst. Prin.*, pp. 15-24, Dec. 1985.

[Postel & Reynolds 1985]
J. Postel and J. Reynolds. File Transfer Protocol (FTP). Req. For Com. 959, USC Information Sci. Institute, Oct. 1985.

[Salton 1986]
G. Salton. Another Look at Automatic Text-Retrieval Systems. *Commun. ACM*, 29(7), pp. 648-656, July 1986.

[Schwartz et al. 1992]
M. F. Schwartz, A. Emtage, B. Kahle and B. C. Neuman. A Comparison of Internet Resource Discovery Approaches. *Computing Systems*, 5(4), pp. 461-493, Fall 1992.

[Sheldon et al. 1994]
M. A. Sheldon, A. Duda, R. Weiss, J. W. O'Toole, Jr. and D. K. Gifford. Content Routing for Distributed Information Servers. *Proc. 4th Int. Conf. on Extending Database Technology*, Cambridge, England, Mar. 1994.

[Sun 1987] Sun. XDR: External Data Representation Standard. Req. For Com. 1014, Sun Microsystems, Inc., June 1987.

[Tanenbaum 1988]
A. S. Tanenbaum. *Computer Networks.* Prentice Hall, Englewood Cliffs, NJ, 1988. Second edition.

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

[Weider, Fullton & Spero 1992]
C. Weider, J. Fullton and S. Spero. Architecture of the Whois++ Index Service. Internet Draft, WNILS Working Group, Nov. 1992. Available from ftp://nri.reston.va.us/internet-drafts/draft-ietf-wnils-whois-00.txt.

[Welch 1984]
T. A. Welch. A Technique for High Performance Data Compression. *IEEE Computer*, 17(6), pp. 8-19, June 1984.

[Zimmermann 1993]
P. Zimmermann. Pretty Good Privacy -- Public Key Encryption for the Masses. PGP User's Guide, June 1993.