Haystack:

A Personal, Intelligent, Indexing System

Eytan Adar

Project Haystack
Laboratory for Computer Science and Artificial Intelligence Laboratory,
Massachusetts Institute of Technology
Cambridge, MA 02139
eytan@mit.edu

Abstract

This paper is a technical discussion of the work done on Project Haystack. Haystack provides the software mechanisms necessary for a user to index relevant information on their personal computer (or workspace). The intent of this system was to provide a means by which a user could easily and intelligently access information stored on their local system as well as remote servers. Through Haystack a user can also annotate archived data with their own descriptions and comments. Haystack is currently in its first semi-public release for Unix workstations. The work done on Haystack was the result of the effort of a number of people at the LCS and AI labs. This paper describes the work done on Haystack as a whole, as well as my personal involvement in the project.

I. Introduction

The driving force behind the Haystack system was the idea of providing the average user with a means to organize the massive amounts of information they accumulate over time. This information could be stored locally on the user's hard drive (local), somewhere on a server (i.e the WWW), or even as a paper copy. Haystack would provide the mechanisms necessary to retrieve this information from wherever it was stored and index it locally within the user's Haystack workspace.

Additionally, it is our intention that Haystack provide a means by which a user can describe documents they encounter. Through Haystack's system of *description files*, a user can store whatever associations they find valuable as meta-information. For example, a user can indicate that a certain USENET article was "interesting." Subsequently, the user can do searches for news articles from the news group foo (foo, being something Haystack determined because it understood the format of USENET), that are also labeled as interesting.

In the future we intend to provide Haystack with two other powerful features. The first, is the ability to "learn." That is, as Haystack analyzes a user's queries over time, we would like the system to respond to user queries in a more intelligent fashion. For example, if Haystack notices that all user queries are oriented towards a specific directory or set of files, we might wish to bias the responses in that direction. This is a very simple example, and we hope that Haystack will be capable of much more.

The second feature is "collaborative computing." We would like to provide a network architecture for Haystack that will allow different users to interact with each other's stored documents and descriptions. It will potentially be of benefit to a given user to not be restricted to asking their own Haystacks what they previously labeled interesting, but ask the Haystack of their colleagues the same question.

The Haystack system is basically a generic abstraction barrier written in Perl that implements a useful API and user interface to any arbitrary information retrieval system as well as providing an annotation mechanism for objects. Perl is a powerful scripting language that

allowed us to quickly prototype the Haystack system and provided the communication mechanism between the user and the information retrieval or database system. We have recently issued the first alpha release of Haystack for Unix based machines.

II. Previous Work

A number of tools currently exist to allow for indexing of data. Some, like the Cornell initiated SMART system, are specifically intended for IR research. Unfortunately, this makes the system very complicated and very oriented towards academia. Other tools, such as Harvest [BOW94] and Excite allow for the indexing of a fixed corpus and are aimed at the web server market. These tools do not give the user the ability to annotate or describe the data they have archived or easily change individual objects stored in their corpus. The Content Routing project [SHE95] provide users with a mechanism to transfer queries between each other. We see this feature as providing users with "community interaction" or "collaborative computing" mechanisms described above. That is, it should be possible for users to interact with each other's Haystacks in much the same fashion as Content Routing.

Haystack provides a common abstraction barrier that will sit above an already established IR system. Effectively, this means that a user can select any IR package that meets their needs and easily plug it in to the backend of the Haystack system. This approach differs from products such as the Alta Vista Personal Extension that includes a hard coded IR system, and an un-modifiable API. Other similar projects include the Lifestreams [FRE95] project at Yale. Both of these projects provide a higher level means of indexing and easily accessing information stored on a PC. In addition, there are a number of lower level alternatives including IR systems

and more standard databases. IR systems tend to be complex and unmanageable, and databases tend to be too strict in their definitions of objects and have unfriendly user interfaces. These problems are addressed by Haystack.

The systems described above provide a subset of what we would like Haystack to do. We would like Haystack to fill the missing gaps in functionality. However, we would also like to be able to use Haystack on top of these systems to take advantage of their capabilities, but make things simple enough for a user not versed in IR algorithms. Haystack was created with all this functionality in mind. Some functionality such as "intelligence" have not been added to the system as of this first release. However, due to its modular nature it will be relatively easy to add more pieces. We would like for the user to easily change and exchange pieces in the system to satisfy their needs.

III. Information Retrieval: An Overview and the Problem

An IR system has the following functionality: given a set of documents consisting of text (a corpus), an IR system will convert the terms in the input set into a data structure that allows for easy lookup of terms and their associated documents. An IR system will also, by means of a query system (which depends on the style of indexing), provide a mechanism for retrieval. Indexing is the first function executed by an IR system, and provides a way of mapping text or partial text in the documents into an easily searchable data structure. This can be anything from a hash table to B-trees. After the IR system indexes the text, it is now possible to quickly find pointers to documents containing a given string.

An IR system can do many things to the input documents in order to make the index more useful. For example, a process known as stemming extends the index by building up words based on the root of an input root. More specifically, if the word "looks" is contained in an input document a stemming IR system will also index the words looks, looked, looking etc. and associate them with the same document. Another optimization is to throw away "common words." Words such as "and" do not always need to be indexed as they will produce a large number of hits. These words are known as stopwords and are either dynamically calculated (throw away all words occurring more than 10,000 times) or "hard coded" into the system.

A query is effectively a question postulated by the user. For example, if the IR system responds to boolean queries, the query "foo $\wedge$ bar," will return all documents (or pointers to documents) that include both the words foo and bar. Depending on the complexity and sophistication of the IR system, queries vary in complexity and utility. A vector based IR system will project all corpus documents into a vector space (based on the occurrence and frequency of words). Given a query, such a system will project the query document into vector space and return the documents whose vectors have the smallest angle to the query vector. Another popular IR optimization is to provide a thesaurus that maps query words into other likely queries. For example, if the query is the word "feline" the IR system might also check for the existence of the word "cat."

Even though such powerful IR tools exist, they are mostly unusable by the average user, either because they are too complex, too unreliable, or perhaps the biggest barrier, have an unfriendly or insufficient user interface. It is the goal of Haystack to provide a standard, but

customizable, interface to IR systems with extended functionality such as annotation, intelligence, and "community interaction."

IV. System Structure and Organization.

A. The Documents

The Haystack system allows a user to index "documents," or more generally, the digital representations of some object. There are many problems with classifying what exactly we mean when we say the word document. For example, we can break apart a document into chapters, and further into pages, and further still into lines. We can claim that each of these sub units is an "intellectual" work on its own and can therefore be classified as a document. We can extend this to say that a document can be a mere word, a letter, or every individual bit of the digital encoding of the document.

However, it is notable that the assumption that each document can be broken down into another sub-documents with no overlapping structures is erroneous. This is a major problem affecting classification and naming of "documents." For example, a document can exist in terms of itself, say as a section in some writing. However, it can also exist as a separate file in a directory. Through Haystack, we would like to make it possible to easily define what constitutes a document. To this end we have created a variety of text parsers which are discussed below that allow for the formation of "objects." It is therefore necessary to discuss our set of documents (the corpus), as a hierarchical system. Documents can be parents and/or children. So a chapter can be the parent of a sub-heading document, and can be the child of a book. The book can in turn exist in a directory on a file system, which is also a "document," in our system. One

possible document hierarchy is presented in figure 1. The document, my_thesis.doc is contained in two different directories /home/eytan/documents, and /home/eytan/thesis. Within my_thesis.doc we find a child, chapter 1. Chapter 1 appears twice because we have two versions, each with a different introduction. However, both copies of chapter 1 have a the same single copy of section 1 as a child. For the purposes of Haystack, we define a document in a much broader sense than the word is traditionally used. It is also notable that in our system of documents, a child can have multiple parents. This leads to many difficulties when designing an archiving system, which will be addressed below.

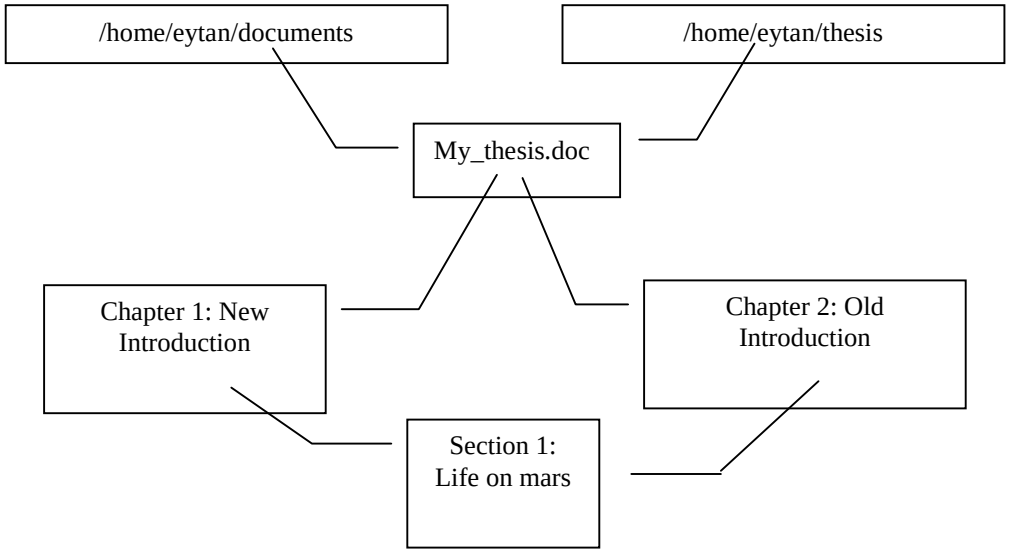

Figure 1

We face an additional problem due to the existence of different digital representations of the same piece of "intellectual" effort. That is, a document can exist as a text file as well as a postscript file. Another example is that of a postscript document that has been converted to individual page images (each a tiff). While these two may exist as distinct objects, they are also associated. We would like for the Haystack system to be able to detect such relationships, but it is not always possible. However, through the use of description files (DFs), which are elaborated upon below, Haystack provides an easy mechanism for the user to draw their own connections.

It should be easy for the user to do things such as label a poem by Dickinson and a poem by Whitman as associated because they both use the same variable beat. Haystack is not currently, and may never be able to draw such arbitrary associations, so it is necessary to allow the user such functionality by other means. It is difficult for a computer to determine semantic meaning, or predict with absolute certainty the intention of a human user. Haystack will determine basic information about any given document being indexed. Potentially, in the future more sophisticated methods will be implemented as part of Haystack that will allow for intelligent determinations of the intent of the user. However, until computers are able to read the minds of users, a flexible system for annotation of indexed documents is vital and is included in the current release of Haystack.

B. Description Files

We would like to find a way in which we can attach arbitrary information about a document to that document. For example, we can take the checksum (or other unique identifier) of the document and include that information in an auxiliary database. Just as important as the checksum, is the ability to attach other information that has been extracted, or derived, or even associated by the user. This information is popularly called meta-data. When a user indexes a file into haystack a description file (DF) is created. The DF contains both control information as well as annotation information that the user can append. An example DF follows:

```
@!df-haystack
Haystack Version 0.1
df-version 3
mime_type : text/plain
ec_locked  [ac3412b6b05a09102692465ea525d78f]:
head : \
\
\
\
\
\
Network Working Group                                          C. Graff\
Request for Comments: 1770                                  US Army CECOM\
```

```
Category: Informational                                    March 1995\
\
\
       IPv4 Option for Sender Directed Multi-Destination Delivery\
\
Status of this Memo\
\
This memo provides information for the Internet community.  This memo\
does not specify an Internet standard of any kind.  Distribution of\
this memo is unlimited.\
\
Abstract\
```

```
type : text/plain
fetch  [type = Default]: hs_fetch %s
url : file://localhost/home/c1/eytan/rfc/rfc1770.txt
processing  [Archive control field]: 0
comment : this is a useful document on Internet stuff
```

All description files contain the header "@!df-haystack" as well as a version number. Versioning will allow us to easily convert between versions of description and to maintain an archive of changes. Each line contains a triplet of field name, field information, and field value. Values that continue onto another line are tagged with a backslash, "\" to indicate this. The DF also includes a df-version field. This is used to perform simple atomic transactions to prevent multiple users/processes inadvertently overwriting DFs. Mime-type contains what Haystack determined to be the MIME type of the document, in this case text/plain. Additionally, we include MD5 checksum information, as well as the "head" of the document. It was deemed useful to be able to quickly provide the user with some indication as to what the document contained when queries were performed. This was done to reduce the amount of disk space necessary to maintain text copies of an entire document, and also reduced the calculation time (i.e. eliminate the need for re-extraction of the text from the document every time). A user can also use this feature to modify the information they see when Haystack responds to queries.

Although the type field in this case contains the same information as the mime-type field, it is important to realize that this is not always true. It was decided early on in our attempt to classify documents that MIME would be an unsuitable format for object classification as it was highly limiting. For this reason, the "type" field was created. It is easy to imagine that the type field may include special details indicating the encoding the user has chosen for their type field.

The fetch and URL fields are "control" fields, and are therefore inaccessible to the user. The fetch field indicates the series of commands or actions necessary to retrieve the real copy of the document. Currently, hs_fetch ("hs" standing for Haystack) is the name of the command line program that obtains the file from a local system.

DFs are currently accessed only through a library called hs_df. This library acts as an abstraction barrier to DFs. The original reasoning of creating unique files, that were human-readable, was to give the user more power over the data stored in the DFs. However, this approach causes many concurrence problems when archiving as it provides no warning to the user that multiple processes are accessing the df simultaneously. Hs_df provides a mechanism for parsing DFs into memory for easy access by the calling code.

C. The Code

The Haystack system includes a set of command line utilities, which are based on Perl library code, and server code (also based on Perl). Perl was chosen as the development language for Haystack as it allowed for quick prototyping and easy debugging. Perl is an interpreted language and is therefore a little slower than a compiled language such as C or C++. However, the benefits of using Perl include ease of programming, portability, and its text processing

power. Text processing is provided through a highly powerful set of regular expression parsers and converters. Throughout the course of this project we have switched between Perl 4.036 and Perl 5.00x for a variety of reasons, finally settling on 5.00x as it provided a number useful properties and is currently more popular than 4.036.

One of the goals of creating Haystack was to allow for a way to easily create new top level interfaces and abstraction layers based on the base Haystack libraries. To this end, the command line programs that are a part of haystack are small shells that pass arguments to library functions. This allows for the inclusion of Haystack procedures into other Perl code without forcing I/O interaction (i.e. without forcing the use of Unix pipes).

What follows is a complete discussion of some of the major components in the Haystack system.

The IR Interface: hs_call_ir_system

Haystack is not an IR system. Rather we assume that there is some underlying IR system that is interfaceable by some method (either an API, or Unix I/O pipes). For the purpose of a first release we chose to use "Managing Gigabytes," (MG) [WIT94]. Although we are not completely satisfied with the speed of (MG), it was publicly accessible and more importantly, it was (mostly) functional.

The IR interface, hs_call_ir_system is a library of primitive functions that form an abstraction layer for access to the IR system. The primitive functions include, initialization,

build, merge, query, as well as the ability to determine which primitives are defined. The only primitives that are required are, init, build, query, and return all primitives. Haystack can take advantage other methods of access but they are not required for the system to function correctly.

Additionally, hs_call_ir_system is written with full understanding of the properties of the IR system so that optimizations can be made that allow for speedier indexing, queries, and parsing. For example, in the case of MG, we discovered early on that indexing by merging (i.e. adding to the index) files serially was taking an excessively long time. So we optimized hs_call_ir_system in the following way. Merges are no longer used. Rather we force a build every time new files need to be added. While builds are being done, new files that need to be indexed are enqueued in the background and will be added by the next build sequence. The benefits of this are that the more files one has to index, the more this optimization helps as the build of many files simultaneously is cheaper then merging each file individually. In the next release we would like to address more speed issues by intelligently determining where a merge would be most useful and cost effective, by for example, maintaining a high water mark in terms of the bits of enqueued data.

All text that is input into the IR system for indexing is maintained in .ix files (index files). These files contain the text of the DFs appended to the extracted text of the document. By indexing DFs with their associated documents it is then possible to query on annotations or field names that are available only through the DF. So when we classify an object as type postscript, we can subsequently do all queries for documents that contain the word “foo” and are of type postscript.

The query operation uses features of the IR system to produce useful/usable results. With MG, we simply feed in the query terms, which are evaluated with a boolean AND in the IR system. MG returns the "head" of the index files that contains the query terms. We then process the information returned by the query by processing the heads, and determine the ID of the documents and return that for further processing. Each object has a unique ID in haystack. Currently, this is just a number that is uniquely assigned at the time of archiving.

Finally, to produce reliable operation, hs_call_ir_system will duplicate the index created by the IR system before completely rebuilding it. This allows for a simple method of error recovery if something happens during indexing. We would like for the user to be comfortable enough with the stability of Haystack that they would trust that it will not corrupt vital information.

Archive: hs_archive and hs_index

Archiving is the method by which new documents are entered into the user's Haystack (or as we call the storage directory, the Hayloft). Archiving involves a number of steps: fetching, extracting, field finding, "shelving," and indexing. Let's say we have a document sitting on our hard drive which is an email document. A simplified version of the archiving process (into which we'll go into more detail below) for this email document is as follows. The document is "fetched" and a copy is placed in the user's Haystack working directory. This copy is the archived copy, and is labeled as haystack_id.ar (the haystack ID being the unique number generated for this object, and ar signifying archived). This process is called shelving.

If the object is judged to be unique (as described below), Haystack will attempt to determine the type of the file by means of a number of heuristics. Upon determination of this, Haystack will extract the text of the document by invoking the appropriate extraction method as dictated by the type. In this case, we determine the type to be email, so we can use hs_extract_email[1]. The extraction also performs the task of field finding. That is, we can determine information about a given document when we know it's type and have some heuristic for extracting useful data. In the current release of Haystack we have a number of field extractors and textifiers including, email (or more specifically Rmail), news, and postscript. All the information we extract from the file is then stored in the description file for that object. We will even create DFs for directories, graphics, and other binary files. This allows us to describe everything we have in our workspace. In the future we will be able to run OCR or image detection algorithms on graphics, or be able to extract procedure names from compiled code, and this information will be indexed as well. Haystack was designed with the addition of these new modules in mind. We argue that it should be easy for the user to plug in new modules that understand new files, or supercede our extraction methods.

With the extracted fields and possibly text in hand, Haystack creates the .ix file by merging the DF (which contains the extracted information) for the current object and the text. Haystack then issues the command to hs_call_ir_system for the indexing of the document.

The description above is a very simplified version of the decision tree used by hs_archive to determine what exactly it should do with files. In the current system, things are indexed in

two ways. The first is the instance in which the user asks to archive one file directly (i.e. by calling hs_archive on a single file). The second is the batch build process, in which Haystack has to guess what the user wants. Both methods follow the decision tree in figure 2 (see page 17).

While this flow chart looks intimidating at first, it is fairly simple. When a user archives, they should have a high degree of control over how the process is accomplished. Each decision has an associated flag for the command line argument that allows complete specification of what to do with the file being archived. What happens is the following. When hs_archive receives an archive request it will first test to see if the URL is in a URL database that Haystack maintains (1). If it is in the database we know that we have already indexed something with that URL, the default of this decision is to abort. The user can continue on to step 7. Here we calculate the checksum of the object and determine if that is already in "already archived" checksum database (similar to the one for URLs). For this example, let us suppose it is. In this case we have the option of creating a new or superseded Haystack object. There are three different states in which we can end (plus abort/exit). Here is a brief description:

- New: This creates a brand new DF, and indexes the document into the system. No associations are drawn between the new DF and any other object (even if there were matching URLs or checksums).
- Merge: This case deals with moved files. For example, if I move a file from one place to another in my file system, the URL will change, but the checksum will not. We can detect this, and repoint the URL field in the original DF to the new location.

[1] This doesn't actually exist in Haystack, rather we understand Rmail and other specific formats of email.

- Supersede: If we determine that an object is a newer version of something we have seen before we can create a new DF for the object, and indicate in the old DF that it is superseded by the new one. This allows us to actually maintain an archive and to track changes. We see potential in Haystack to act as a "revision control system" for objects. We can therefore track what changed between versions of DFs and allow a user to revert to an older copy.

The first end state (new) will not allow us to maintain any of the annotations added to associated DFs. On the other hand both merge and supersede will. Merge keeps one DF, and we can simply walk through the DF superseded chain picking up annotations as we go along.

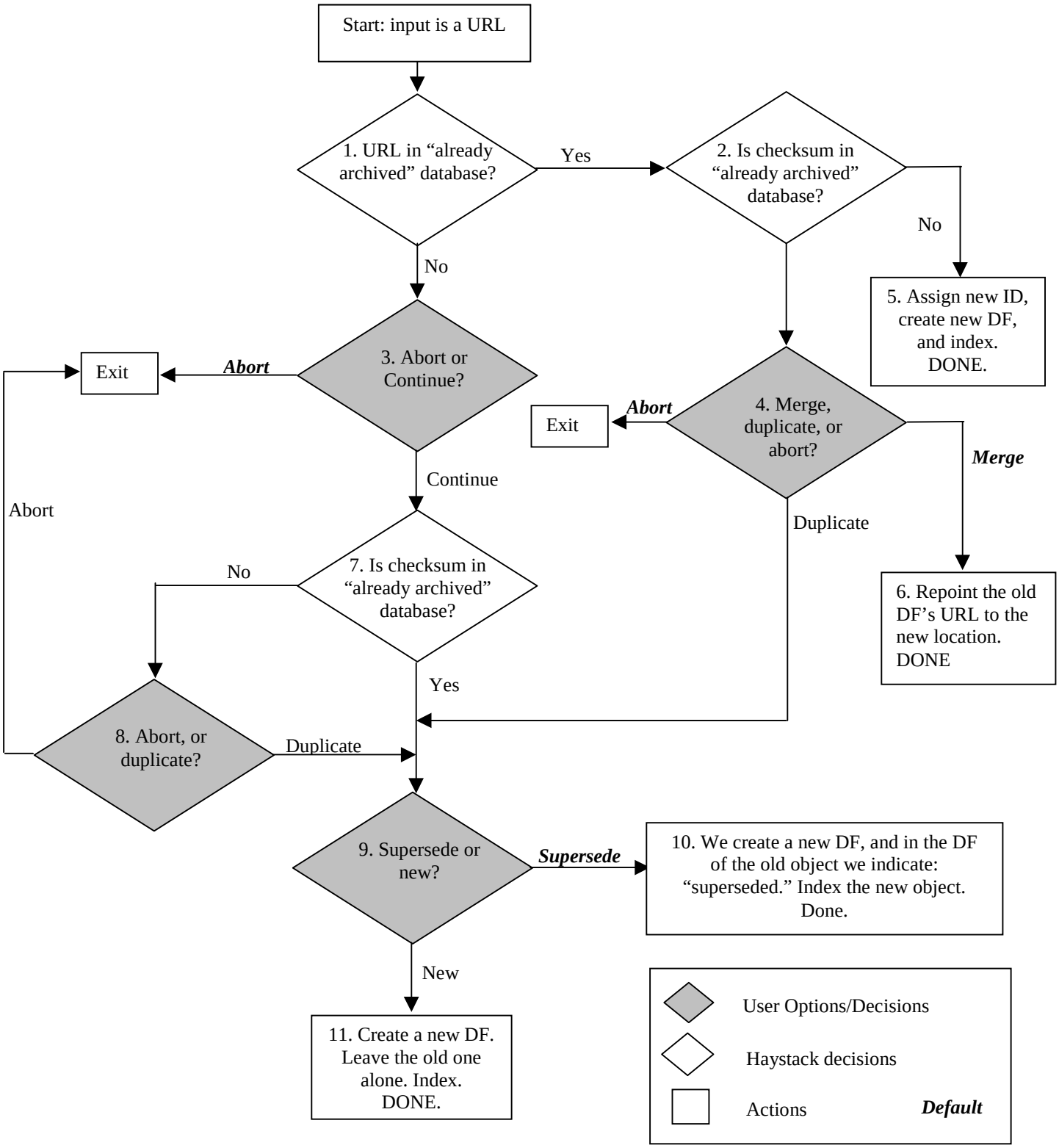
Start: input is a URL
1. URL in "already archived" database?
Yes
2. Is checksum in "already archived" database?
No
5. Assign new ID, create new DF, and index. DONE.
No
3. Abort or Continue?
Abort
Exit
4. Merge, duplicate, or abort?
Abort
Exit
Merge
6. Repoint the old DF's URL to the new location. DONE
Duplicate
Continue
Abort
7. Is checksum in "already archived" database?
No
Yes
8. Abort, or duplicate?
Duplicate
9. Supersede or new?
Supersede
10. We create a new DF, and in the DF of the old object we indicate: "superseded." Index the new object. Done.
New
11. Create a new DF. Leave the old one alone. Index. DONE.
User Options/Decisions
Haystack decisions
Actions
Default

Figure 2

The final relevant attribute of the Haystack archiving process is the way in which we handle hierarchical archiving. If a user asks Haystack to archive a directory a DF will be created that represents the directory, and all children that are generated within that directory archive are noted in the directory DF. In this case, the children are merely the files contained within the directory. When Haystack receives instructions to archive something that it understands to be a top level view (as in the directory, or an Rmail file), it will recursively break the upper level object down into its constituent parts, and will only index the text of the leaves. It is unnecessary to re-archive the text at every level as combining the leaves can form the text. That is, we should not archive the entire Rmail file and the text of all the individual mail messages. Although there are potentially cases in which this would be valuable, and hs_archive should be forced to archive the "whole" object, we do not currently provide this functionality. For the most part it is our suspicion that archiving not just leaves, but merged leaves would result in wasted disk space.

Archiving in Haystack is a very tricky topic and we are constantly introducing new and optimized methods for achieving our goal of intelligently catering to the users needs. We discovered as we designed the archiving system more and more complicated uses and expectations, and we have slowly gained a deep understanding of our problem domain, which we wish to more fully address in subsequent releases of Haystack.

User Interface: hs_www_server and the Command Line

Currently Haystack has two methods by which a user can interact with the system. The first is the command line approach, and the second is a customized World Wide Web (WWW) server.

Through the command line, a user can perform all functionality from initializing (hs_init_haystack) to archiving (hs_archive) to querying (hs_query). By designing Haystack as a library of Perl functions and single function scripts it is easy for a user to build on this and create new interfaces to Haystack. For example, we would like to be able to create an Emacs interface to Haystack. We would like to provide as many different interfaces as possible, but we would also like to ensure that users can easily create new interfaces that fit the way they work.

The first release of Haystack also includes a WWW server written specifically for optimized Haystack operation. All relevant Perl libraries are pre-loaded into the Haystack server so that it is unnecessary to re-execute Perl to handle Common Gateway Interface (CGI) scripts. CGI scripts are programs that reside on the server side that are executed in response to client input (for example to handle the input from a form). To ensure compatibility with other WWW servers, the CGI scripts are portable. That is, a standard WWW server can still execute the CGI scripts through standard shell commands. This is necessary if a user has to either run a different server, or can only use a server catering to many users (at the loss of security). However, by designing the WWW server to be lightweight, we believe that it is possible for each user to run their own server. Nonetheless, we realize that this may be unreasonable for a machine that hosts many users simultaneously, so a subsequent version of Haystack may include a "multiplexing" server. Doing this, unfortunately, raises security concerns for our system.

When designing Haystack we intended to have the user freely index anything on their file system, even personal files. The difficulty in providing a WWW server is that this data can be

obtained illicitly by simply connecting to the server. Additionally, we provide system-affecting CGI scripts through the WWW interface. We would not like any arbitrary user to touch our own Haystack by either adding to it, or looking at what's there. To solve this problem, we created a first line of defense by means of standard WWW realm security measures. The server requests a username/password from the client (and hence the user) with every transaction to the server. The client maintains a copy of this username/password so that the user doesn't have to type it in for every request to the WWW server (the password gets wiped from the clients memory when the client is terminated). On the server side, initialization of the server involves creating a password file. This file is the hashed version of the clear text password the user enters, so that the file is useless to anyone other than the server. The server in turn does not keep a copy of the password, either encoded or decoded. When the server receives a request (which will always have the password attached), it encodes the password using the same hash function and compares it the stored file. This is not completely fail-safe, but neither are standard Internet operations such as telnet or FTP. We merely wish to provide the same degree of security as rlogin and assurance to the users of Haystack.

The Haystack WWW server provides the ability to query and archive by means of a graphical interface. A directory browser allows the user to easily traverse their file system space to determine which files/directories they would like archived, and allows for easy setting archive flags. Querying is also optimized for the WWW. For example, it is possible to highlight query terms in the returned document. It is also possible to easily edit the DFs for any given Haystack object. A form based interface allows quick viewing and editing of the information. This is a

feature that the command line interface lacks. Through the command line, users are only able to edit/annotate the DFs by means of a text editor.

In the next release of Haystack we plan to improve the WWW server, as well as provide other means of access to Haystack. Perhaps, through Java or a Tk (windowing toolkit for X), we can provide more robust functionality.

Future Work

Now that we have set up the infrastructure for Haystack and have given it basic functionality we will continue to increase the feature and capability set of the system. Below is a sampling of projects are currently in the works or are proposed for the near future.

- Lore: Lore is a database produced by the University of Stanford to store semi-structured data [QU94], which is effectively the type of information we maintain in Haystack. It has come to our attention that keeping DFs as files is not necessarily the optimal answer. We would rather take the information stored in the DF and keep it within Lore. This would allow us to specifically make queries to Lore as well as to the IR system. Work is currently in progress on the integration of Lore into our system. Lore is itself in the process of development in Stanford and is still not fully functional (in terms of our requirements). Once implemented, however, Lore will give Haystack users very powerful query and annotation facilities.
- Re-indexers: In the current system Haystack has no easy mechanism by which indexes are re-built. We would like to provide a "daemon," that will intelligently determine which files need to be re-archived and/or re-indexed, so that a user will not have to worry about doing it

themselves. That is, if I move a document from one directory to another, Haystack should match that change in its own objects.

- New IR systems: We would like to find new IR systems (or write our own), and interface them to Haystack. A user should be able to select which IR system they would like to use, as well as potentially indexing their files using multiple IR systems. To this end it should be possible to query multiple IR systems through different hs_call_ir_system libraries. Work is currently in progress for simple query systems built on top of Unix grep, as well as one based on Savant, a vector based IR system.
- Proxy services: We also have the base functionality for a proxy server for use by WWW users to archive files they encounter on the web. When a user points their web browser to our proxy server, all documents returned to the user's browsers include a small button at the end that allows them to index the text of what they are currently viewing into Haystack. We would also like to provide the option to index all files viewed on the web. In this way, the user always has a record of what they have seen on the web.
- OCR: We would not only like to provide a means for indexing digital information, but paper information as well. A potential project will involve allowing the user to scan in their paper documents, and through an OCR system allow for the indexing of the content of these documents. While the paperless office may be a ways off, there is no reason a user shouldn't be able to find what they are looking for.
- Other changes: Finally, there are a number of basic infrastructure components with which we are not completely satisfied. They work, but could be improved and optimized.

Conclusions

Haystack promises to be a very interesting project both in terms of research as well as practical application to users. There are a number of questions that remain unanswered in our system. These range from systems issues such as security and transaction processing as well as theoretical questions and artificial intelligence. More importantly, we would like to create a product that addresses the needs of an average user with way too much information on their hands and no way to easily find anything.

Acknowledgments

Haystack is a result of the hard work of many individuals. Professor David Karger, of the LCS, and Professor Lynn Stein, of the AI lab and LCS conceived the project. Michael Coen, and Thomas Lee worked on the first, very basic version of Haystack. The author (Eytan Adar) joined the project in January of 1996. During the summer of 1996, a number of undergraduates joined the project and contributed to the current Haystack code. These include: Mark Asdoorian, Dwaine Clarke, Eric Prebys, Lilian Liu, and Chuck Van Buren.

Bibliography

[BOW94] Bowman, C. Mic, et. al., “Harvest: A Scalable, Customizable Discovery and Access System,” Technical Report CU-CS-732-94, Department of Computer Science, University of Colorado, Boulder, March 1995.

[FRE95] Freeman, Eric and Scott Fertig, “Lifestreams: Organizing your Electronic Life,” AAAI Fall Symposium, AI Applications in Knowledge Navigation and Retrieval, November, 1995, Cambridge, MA.

[QU94] Quass, D. et. al., “Querying Semistructured Heterogeneous Information,” Stanford University, to be published in DOOD ’95.

[SHE95] Sheldon, Mark, “Content Routing: A Scaleable Architecture for Network-Based Information Discovery,” MIT, PhD Thesis, 1995.

[WIT94] Witten, Ian, et. al., *Managing Gigabytes*, Van Nostrand Reinhold, New York, 1994