Haystack: Per-User Information Environments

Eytan Adar
Xerox Palo Alto Research Center
Palo Alto, CA 94304
E-mail: `adar@parc.xerox.com`.

David Karger
MIT Laboratory for Computer Science
Cambridge, MA 02139
E-mail: `karger@lcs.mit.edu`.

Lynn Andrea Stein
MIT Laboratory for Computer Science and
Artificial Intelligence Laboratory
Cambridge, MA 02139
E-mail: `las@ai.mit.edu`.

Abstract

Traditional Information Retrieval (IR) systems are designed to provide uniform access to centralized corpora by large numbers of people. The Haystack project emphasizes the relationship between a particular individual and his corpus. An individual's own haystack priviliges information with which that user interacts, gathers data about those interactions, and uses this metadata to further personalize the retrieval process. This paper describes the prototype Haystack system.

1 Introduction

Current large-scale IR systems are in many ways very similar to libraries: they manage massive corpora for anonymous individuals using a fixed organizational schema. Taxonomies (like Yahoo) emulate the Dewey decimal system. Search engines (like Alta Vista) mimic the card catalog. But in a library, a user cannot ask for "the fat book about computers I skimmed last month," and current IR systems do not support queries for "the email about reinforcement learning that I forwarded to Terry last week." A search for "apples" will yield the same results to a computer shopper or a farmer. A user who "throws out" certain search results will find them coming back again the next time he performs the same search.

Libraries are huge, filled with masses of data irrelevant to any query. They are impersonal, presenting every user with the same information regardless of their background and interests. And a previous successful search facilitates future searches only by training the user; the library itself does not adapt to the needs of individual patrons. In a traditional library, these problems are ameliorated by professional reference librarians. In automated IR systems, no such resource currently exists.

But even in the paper world, libraries are typically the last place we turn in seeking information. When a person looks for information, he will often start with his own bookshelf. This "personal repository" contains a collection of information, built up over time, that reflects the needs and knowledge of its owner. This makes it different, in crucial ways, from the library. For example, all the content was actively placed there by the user, who is familiar with it and believes it to be useful. In the user's area of expertise, it is often more up to date than the library: its owner, who is actively seeking information in his area of interest, often finds new information before the library gets around to it. Overall, a person's bookshelf contains the bulk of the information that he considers most valuable.

An individual's bookshelf is also organized in an idiosyncratic fashion. While library materials are arranged according to a standardized classification scheme, individuals have been known to arrange their books by topic, chronology, usage pattern, or even size and color. Even users who make no active attempt to organize their books find them structured in some kind of most-recently-used hierarchy. Individuals exploit their idiosyncratic organization when searching for information: they may look for a blue book, or a book on the bottom shelf, or a book next to another book. At a library, users are limited to searching the standard classification.[1]

[1] Even libraries are not immune to this wish to personalize information outside the traditional schema. Recently, the New York Public Library announced that it would be disposing of its card catalogue, as it had been superseded by more modern search tools. This was reported in the New York Times and was newsworthy because many people objected to the elimination of this old-fashioned search tool. It turned out that over the years, a great deal of information had been penciled on to the cards in the catalogue by its users, to the

1.1 A Digital Bookshelf

The Haystack project aims to make a digital IR system that is less like a library and more like a personal bookshelf. Fundamentally, this means building a system that adapts to its user, instead of forcing its user to adapt to the limitations of the system. A haystack provides automated data gathering (through active observation of user activity), customized information collection, and adaptation to individual query needs.

A critical step in our design is *information maximization:* gathering, representing, exposing, and using *all* the information about a corpus, its user, and their relationship that might help information seeking. Our early research on the Haystack project has focussed primarily on the gathering and representation of such information, as this must precede any effect use of it.

In Haystack, information maximization is achieved in three steps. First, all information—including meta-information—is stored using an extremely general data model. This means that any information users encounter, as well as any information the system observes, can be stored, indexed, and retrieved uniformly. Second, Haystack gathers as much information as possible about its user and corpus—by analyzing the corpus offline, by observing its use, and by encouraging direct human input. Finally, Haystack modifies its data and its retrieval process based on user interactions, adapting to the behavior of the user and the properties of the collected data.

Haystack is currently implemented as a refined prototype in Java. This paper describes the general features of this prototype. The details of Haystack's implementation are described in earlier publications [ADA98, ASD98]. Although it will not be discussed in this paper, an advanced graphical user interface has recently been added to Haystack and is described in [LOW99].

1.2 Haystack in Action

As an example of some of the elements of the haystack system, consider a user who recalls at one time seeing a document relevant to a latent semantic indexing problem he is trying to solve. The user types in "LSI" as a query to his haystack.[2] LSI does not actually appear in the document in question, but his haystack notices that a previous query he made about principal component analysis (and which the haystack dutifully stored away) contains numerous documents about LSI. So the haystack shows that query and its results to the user. The user, inspecting the result set for that query, discovers a postscript version of the paper he was looking for and asks his haystack to display that paper. In addition, a link from that postscript paper shows that it arrived in an email message from a colleague; a further link from that email message points to a followup message from a second colleague, critiquing some elements of the paper. A separate link points to a copy of the paper's abstract located in a seminar announcement that identifies a "host" who might have other useful information. The user can follow links to find out "what papers have been published by this host?" on the assumption that they may well be relevant to the topic in which they are interested.

This example demonstrates several ways in which Haystack exploits its single-user orientation. Haystack can (though it need not) limit its search to material the user has encountered before. A search which would be unreasonably broad on the web can still make sense on this narrower corpus. Haystack also remembers past queries that the user has made; given a typical user's tendency to be interested more than once in the same information, this provides a leaping off point for future queries. Haystack scans its corpus to make connections between documents with similar content. In addition, by tracking the usage history of objects, Haystack is able to draw connections that might not be visible under a word-based matching rule. Finally, by creating and exploiting metadata links, the haystack data model lets users follow associations to discover additional related material.

1.3 Paper Roadmap

The reminder of this paper begins with a discussion of previous work (Section 2). In answer to our need for a versatile data representation we introduce the graph-based Haystack data model (Section 3). This is followed by a high level disucussion of the Haystack prototype's architecture (Section 4). To address the need for data gathering and maximization we introduce various approaches in Section 5. We conclude with a brief discussion on evaluation (in Section 6, some observations and future work in Section 7.

2 Prior Work

Haystack grows out of two long-established research areas, information retrieval and information filtering.

IR focuses on the question of guiding users to information in a large, fixed corpus. Research is embodied in such systems as SMART [SAB94] and large web search engines (Alta Vista, Excite, Lycos). The typical assumption in these projects is that the corpus is large

---

benefit of subsequent users. The paper catalogue was a useful tool but turned out to be missing valuable information. People modified it over time to make it even more useful.

[2] The user might really have typed "latent semantic indexing", or the user might have typed (and intended) the acronym LSI. A third possibility makes use of Haystack's query tracking feature: the user might have typed LSI, then refined it to "latent semantic indexing." In future queries, his haystack would short-cut this refinement, searching for latent semantic indexing when the user asked for LSI.

and non-personalized. The general metric for quality in these systems is the precision-recall curve which is based on some testing data. Relevance is generally judged in this testing approach by human "experts." Unfortunately, this causes most systems to be trained to the opinions of relevance based on experts. Individual users with differeing opinions and interests may find their experience of the system to be quite different. These measures also ignore the prior history of a user's interaction with the corpus and its effect on retrieval.

A number of commercial vendors have recently begun to create software packages for indexing of small desktop based collections. Largely, these systems include a text indexing mechanism that occasionally includes the ability to query on other document properties (creation/modification dates, file owner, etc.). Two particularly robust examples of this type of system are Compaq's AltaVista Disovery tool [AV98] and Apple's Sherlock [AP98]. However, even these tools do not attempt to adapt to a user.

Information filtering attempts to deal with individual users' needs by performing online decisions of relevance of a particular piece of information to a user. Users then "train" the system by pointing out several interesting or relevant documents; the system then attempts to decide, based on this information, whether a given new piece of information will be interesting. Balabanovic et al. [BSY97] constructed a system that learns user preferences regarding web pages. The model is trained by presenting users with candidate web pages and asking them which are interesting. CMU's WebWatcher [JOA95] is another example of an agent that learns from user feedback. Similarily, Letizia [LIE95] provides surfing suggestions by pre-processing links further down the hypertext path and measuring them against user preferences.

These tools share several features of collaborative filtering that we aim to avoid in Haystack. They all work with a general model of "goodness" that is independent of any particular query—they try to get a broad idea of the user's likes and dislikes, without tackling specific information needs. Also, these tools require the user to explicitly rate candidate objects for quality to provide useful input to the system.

Haystack is an attempt to draw together some of the threads that have been explored separately in information retrieval and learning based information filtering. Integrating the information retrieval system into each individual's desktop instead of restricting it to large, communal corpora gives us the opportunity to personalize the information retrieval process, adapting it to individual attitudes about what is interesting and how it is described. But we focus on search, where the user's specific information need, and not just a general notion of goodness, drivies the search for results.

This blend of search and filtering has also been explored to some extent. Academic research on personal information tools stems from the early days of Hypertext research including Bush's Memex [BU45] and Engelbart's AUGMENT [ENG62]. The Lifestreams project from Yale [FRE95] is another example of personal information space management. Lifestreams, however is predominatly based on the storage of documents in a temporal context. Haystack allows for classification and retrieval on other document features. The Remembrance Agent [RHO96] indexes a user's e-mail and other local files. When the user works in the Emacs editor, the agent provides pure content based retrieval on related material.

3 A General Data Model

Our goals for haystack motivated the choice of a very general data model that can represent arbitrary pieces of data and metadata and the links between them. We choose to record metadata because it can provide useful information to a user, beyond the raw document object. This information, displayed to the user, may help them make a better judgement about the relevance of objects. We also allow the user to extend this metadata as they wish, for example to let them manually annotate objects with information they find useful. We cannot predict what information a given user may find useful, or in what ways a given user may choose to extend the model, so we want a model that can grow easily.

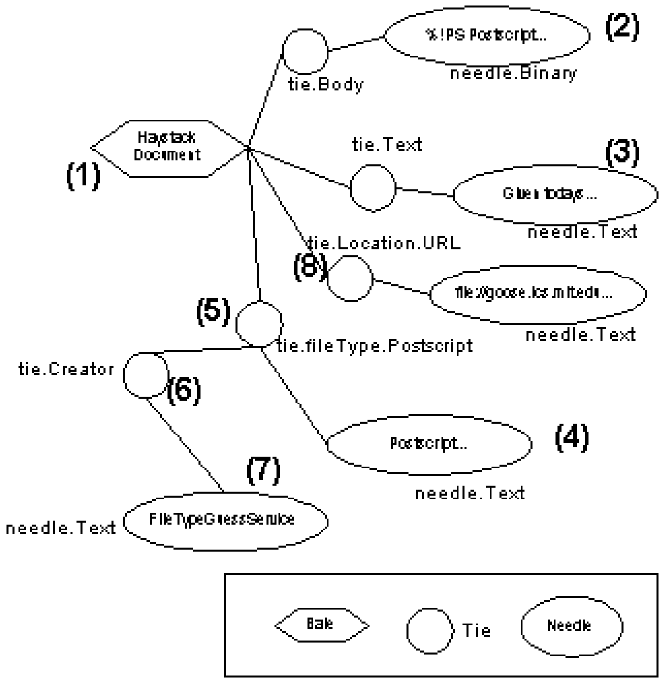

Figure 1: The Haystack Data Model

Besides being general enough to store any data types or relationships between them, our data model creates a natural linkage of “related information”—for example, a document can be linked to its author, and through its author to other documents written by the same person. These links provide support for “associative searching” by a user, which can home in on information from other, partially related information. They also provide indications of similarity that can be used in our indexing tools to modify retrieval performance.

The Haystack data model is represented by a graph structure exemplified by Figure 1. This graph is esentially a semantic network that lets us model associations. Nodes within the graph, which we call straws, represent units of information. Haystack's data types are further extended through an inheritance hierarchy. The base type is a straw, its subclasses are needles, bales and ties.

For example, the document's type (4) occupies one straw and the text of the document occupies another (3). These two particular pieces of information are examples of the simplest kind of node – containing a single primitive piece of information – in our system.[3] We call these primitive pieces of information *needles*, since it is presumed that they will be the most frequent desired results of searches. Specific needle types store integers, text, binary data, and other formats.

Other nodes do not contain any data themselves but rather serve as placeholders to collect a group of related straw objects. For example, a directory contains a number of files (or other directories). We call collection-type straws *bales*. A Haystack document (1) is a particularly important bale type representing a document archived within the Haystack. To the Haystack document we attach various other needles (as direct attributes of the document) and bales (as other collections relevant to the document in some way). Each bale has an associated set of straw objects representing its content. The relationships between a bale and its member straw are one particular type of the many edges in our data graph.

We call the edges of the graph *ties*. Examples of ties are nodes (5) and (6) in Figure 1. A tie allows us to represent arbitrary relationships between two other straw objects. This provides the basic facility for annotation and metadata. However, ties are themselves straws, which allows for recursive annotation of the metadata. It is therefore possible to represent more complex relationships. For example, the type tie (5) above was created by a specific client (7, a type guesser).

We can also return to our observation that the Haystack data model represents metadata about archived documents. To represent metadata, one usually requires the ability to express attribute/value pairs. Values are held within the needles described above. The associated attribute is represented by the *label* of the tie connecting the needle to another document. For example, we know that a given text needle (4) contains the type of the document because the label of the tie connecting it to the document is “type” (5). Labels actually form a type system, and Haystack allows arbitrary straw subclassing. For example, the general “location” is subtyped in the example document as “location.URL” (8).

Haystack data model nodes can also be used to interface Haystack to external “services.” An example is Kramer's work in the MIT Intelligent Room [KR97]. In this context agents that controlled external devices (a VCR, for example) were “indexed” in Haystack.

An important, if slightly mundane, feature of the data model is the ability to dynamically load and unload individual nodes from memory. Straw types correspond directly to Java objects. While in memory connections between straws are standard direct memory pointers. However, because of the potential complexity and size of Haystack data graphs it is not possible to hold a user's complete Haystack in memory. The implementation of straw allows for the connection between between nodes to be severed and rebuilt quickly by replacement of the memory pointers with unique straw identifiers. This allows us to store and load subgraphs to and from disk. The data model elements are managed by the persistent object service described previously.

4 Architecture

The Haystack system consists of a three-tiered architecture. At base are the data storage systems – databases and information retrieval engines – on whose behavior Haystack relies. Above this is the core Haystack system, which includes both a data model implementation and other operating-system-like services. Finally, Haystack provides a number of client level services that augment and use the data stored in a haystack. The complete Haystack architecture is depicted in Figure 2.

4.1 Database Layer

Haystack is not a project about information retrieval *per se*. We remain largely agnostic regarding the particular search tool(s) used. Instead, we are interested in augmenting the power of these tools by providing personalization of the information that they record and retrieve. As a result, Haystack delegates the actual tasks of storage and search to off-the-shelf information retrieval and database tools.

In order to integrate a particular information retrieval engine or database into Haystack, we wrap it

[3] Calling the document text "primitive" does not preclude analyzing it further. The document text is "primitive" only *qua* document text. As we describe below, a document text straw may also be related to other straw objects denoting other aspects or representations of the document.

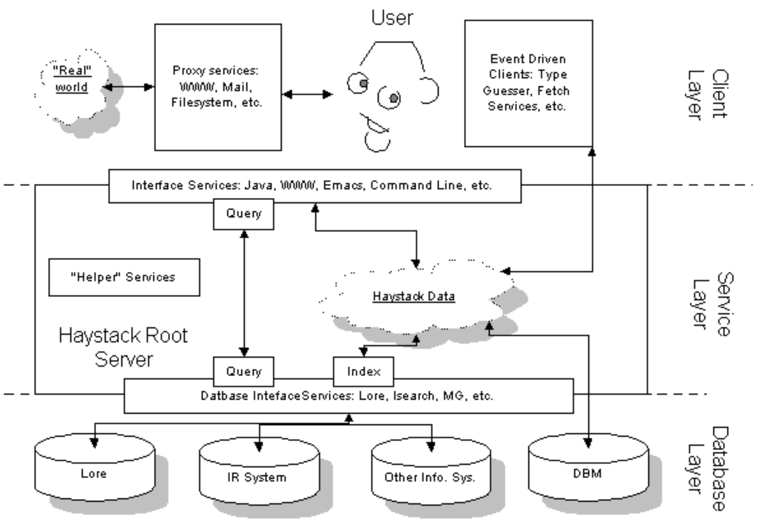

Figure 2: The Haystack Architecture

with a relatively simple adapter. This includes routines that turn Haystack data objects into a form suitable for storage in the off-the-shelf component as well as methods for retrieving objects so stored. We have successfully integrated a number of different traditional information retrieval systems into Haystack. In addition, we have worked with a semi-structured database [MCH97] and a persistent object store [AP98].

4.2 Haystack Root Server

The core of the Haystack system is the root server. This component provides a persistent, indexable, searchable, transaction-safe implementation of the uniform data model described in the previous section. In addition, the Haystack root server provides a variety of other services such as a name service, a thread pool, and event queueing. In essence, the Haystack root server is a small-scale operating system providing the necessary infrastructure for the client services which run above it.

The server layer contains Haystack components that serve utility purposes, and also acts as the "environment" within which the various data-driven clients operate. This tier consists of various services that all run within the context of one Haystack *root server* (basically, a Java Virtual Machine). The root server includes a name service by which other services bootstrap themselves. A persistent storage services, built around a DBM, allows for dynamic loading and unloading of Haystack data model structures. Other services, allow for system configuration, transaction support, and interfaces to the indexing and query functionality of the various systems that reside in the database layer.

Implementation details of the various utility services residing in the server layer as well as mechanisms by which external services communicate with the root server are elaborated on in [ADA98] and [ASD98].

4.3 Client Services

Client services augment and use the data stored in the haystack. All client services rely on the uniform data model provided by the root server. In addition, client services make us of other aspects of the infrastructure level including resource management, interprocess communication, and networking.

There are three major classes of client services. First, the user can interact with his haystack directly. These interactions are supported by a simple command-line interface or by a more complex custom graphical user interface. Haystack's direct user interfaces allow a human user to add new information to his haystack, to annotate existing information, to issue queries, and to browse the results of these queries as well as other information contained within his haystack. The uniformity of the data model means that any information within the haystack can be the target of annotation, query, or browsing activity.

The second class of Haystack clients is a set of proxy services that Haystack provides for other desktop applications. As before, the user also interacts with his electronic environment, including the World Wide Web, e-mail systems, and other information sources. Haystack slips invisible observers (proxies) between the user and these external information sources. Without disturbing the user, these proxy services allow his haystack to record what the user does and sees, remembering both relevant information and the (metadata) context within which it was encountered.

Finally, the client level of Haystack includes a number of automatic data-augmenting clients. These services act to modify the data within a haystack in a wide variety of ways. For example, a fetching client recovers the external document corresponding to a URL in a haystack. A textifier client produces a plain text version of (e.g.) a postscript document it finds in the haystack. A similar text finder service compares documents within the haystack, adding a link between two documents when it finds significant overlap. Even queries are handled by such clients: a query client polls the underlying information storage systems to address an information request placed into the haystack.

4.4 One Person, One Machine

We have chosen to implement haystack as a tool that runs on an individual's own machine, rather than something serving out of a centralized repository like the internet search engines. Although such systems are substantially more powerful than desktop computers, our approach lets each user have more cycles dedicated to them than they could get from any centralized tool. We can therefore apply more sophisticated search techniques without worrying about resource limitations. Location on the user's machine maximizes the amount of information that can be gathered from a user. It also gives at least the psychological illusion of more privacy, so that a user will be willing to commit more personal preference information to the system. Of course, this approach is only practical because an individual's corpus of interest is significantly smaller than the entire web.

4.5 Indexing

Although a full description of the indexing mechanisms for Haystack are beyond the scope of this paper, we provide a brief description. Text extracted from the data model is deposited in an information retrieval system. Haystack attempts to be neutral in respect to the type of information system in the bottom layer; we have built interfaces to two text retrieval engines, MG and Isearch, and are working on others. This approach will allow us to later integrate such nontraditional IR systems as Scatter/Gather [CKPT92]. Additionally, an initial attempt has been made to index the data model in a database [ADA98].

Indexing is done incrementally. The Haystack system monitors changes in the data model followed by a "calm," when no changes are made. When the data has reached this stable state Haystack executes a breadth first search through the graph structure starting at the "Document" anchor node. Each node generates textual information (if possible) and the text is collected and indexed as one unit in the information retrieval. This method allows us to associate pieces of information that may not have necessarily been obvious from just the text of the document. For example, it may not be obvious from reading a paper that Bob wrote it. However, because Bob was the creator of the file on the hard drive, and because Haystack has extracted this information and connected it to the document, Bob's authorship will be noted in the indexed text of the document. This approach is consistent with Spreading Activation Models (SAM) which describe memory functions in associative networks.

4.6 Search

The ultimate goal of Haystack is to provide high-quality retrieval. User queries abstractly consist of some information need (which may be implicit or hidden) and a number of hints that the user will leverage to cause the system to return something that truly satisfies that information need. For example, a user may be looking for a book on probability and may remember that the book was red. The fact that the book is red has nothing to do with the real information need, but if the user knows that this is a unique characteristic of the book (or can't remember other characteristics) this is how he would phrase his query. Users, in dealing with information they have seen or have created, will have a large and varied set of hints.

As indicated previously, Haystack searching makes use of various search engines. Although this is still work in progress, the Haystack data model is processed in various ways and converted into the "native" format of various search engines. A user is then able to use various modes of search including text, database, and hyperlinking.

5 Harvesting Data for Haystack

There are three distinct sources of data for Haystack:

Data driven clients that digest data already in Haystack to produce more data;

Observers that watch what the user is doing and place the resulting information in the Haystack;

Active human annotation carried out by the user to improve his data organization.

In the following sections, we describe these three mechanisms in detail.

5.1 Data driven clients

One primary source of new information in Haystack is the digestion of information that already exists in the repository. This processing is carried out by independent but cooperating data driven clients. In designing Haystack we realized that it would be impossible to forsee (let alone implement) all the clients users could ever use. Haystack therefore allows for the dynamic insertion of new clients in a scheme inspired by CORBA. Clients are implemented in Java and conform to a certain interface. Once loaded, clients register themselves with the name service held within the root server.

The data driven clients fall into several main categories:

- *Fetch clients* retrieve data from various other sources (from a URL, from an RMAIL file, etc.).
- *Type inference clients* decide the type of a document once it is retrieved (a latex file, a postscript document, an HTML page, etc.).
- *Extractor clients* attempt to extract textual information from the retrieved documents. For example, a postscript extractor knows how to convert postscript files into text.
- *Field finder clients* extract various pieces of metadata. For example, the *to*, *from*, and *subject* lines in an email message.

Haystack's data-driven clients are triggered by events occurring in the user's Haystack data. In addition to registering with the name service, clients register (with a special dispatcher service) interest in various changes to the data structure. These changes include the creation, deletion, and modification of straw objects. Interest is currently expressed in terms of a template which is submitted to a dispatcher service. The template corresponds to a one level tree representing essentially a regular expression.

For example, a service can register interest in the addition of an "author" node to a "document" node which already has a "date" node attached to it. Whenever a new node appears in the Haystack data model, the dispatcher quickly determines the other nodes locally effected by the change. The dispatcher then passes the regular expression templates over the effected subgraph. If there is a match for both event type and structure, the interested client is notified (and executes in its own thread). Because we limit template structure, and our dispatcher is tuned, it is possible for sub-graph matching to scale. Additionally, we internally manage a queue for events and a thread pool to prevent saturation of the system when a large number of new objects are added to Haystack.

5.1.1 Data Driven Clients in Action

An example of a number of data driven clients archiving a document in Haystack is illustrated in Figure 3. In (a) a user has added a new document to Haystack. This is done by submitting the URL through one of Haystack interfaces. The URL is anchored by some default behavior to a document node. A fetch service which has previously expressed interest in such a formation is notified and passed a reference to the effected subgraph. The service retrieves the bits for the document from the web, and in (b) adds a new node to the graph that contains these bits.

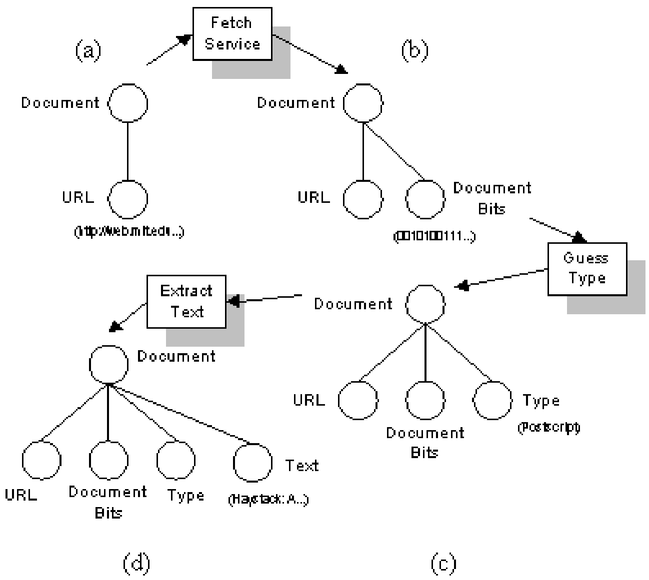

Figure 3: Data driven clients in action

Subsequently, a "type guesser" service is notified that there is a document with some attached bits. The type guess service in Haystack has been implemented with various heuristics and is able to recognize many file types. In (c) the type guess service labels the document as being of type postscript. Finally, a text extractor service (in this case one that knows how to convert postscript to text) is notified that a relevant structure has become available. In (d) this service has created a new node containing the text of the document. After some stable state has been reached (i.e. no more events are triggered) the text and graph structure are indexed for queries by the user.

5.2 Observers

As discussed before, we would like to maximize the amount of information we gather about individual Haystack users. Although ideally we would like to see users manually building up annotations of the data structure, the reality is that most will not. Haystack was designed with this reality in mind. By observing the user it is possible for Haystack to make annotations automatically so that organization and retrieval can be optimized over time.

Proxy services act as transparent intermediaries between the user and external information sources. As the user acquires new pieces of information the proxies submit the information and other observations about the user's state to Haystack's server layer. WWW and e-mail proxies are currently included in Haystack. Haystack also provides a crawler that traverses a user's file space at regular intervals looking for changes. To understand the type of information gathered from observation of the user we describe the web proxy in more detail.

A user configures his or her web browser to utilize the Haystack web proxy. The proxy acts as an intermediary between the browser and other servers. As a user browses the web, the documents he sees are pushed into Haystack. Just as we would automatically make annotations to the document if it was added explicitly by the user, we do the same in this observation mode. Additionally, we include information gathered from observations of the interaction of the user with the external web servers. For example, as the user clicks from page to page, Haystack draws edges from the archived representation of these pages. It is later possible, on retrieval, for the user to retrace his steps. Another example is our ability to annotate a document with a "visited" time. A user can then query for "all web pages that I looked at between 1:00 PM and 1:30 PM today." Although it is impossible for us to predict whether a user will actually make a query on this particular feature, we recall that certain users sort and search by color, size, and other metrics. By gathering as much information as possible we insure that an individual user is able to utilize whatever features that he finds useful for recall and retrieval.

5.2.1 Query Observers

IR systems are designed to answer questions in a predictable way. Generally, one IR system will interact with many users. Both in the interest of fairness and to make implementation easier, the system never adapts to individual users. For example, a query for the word "jaguar" will always return the same set of documents whether the user is a car buff, a computer gamer, or a zoologist. This is important for not confusing the many users of single search service nor biasing the system unfairly. However, because we are dealing with only a single user, in Haystack we have opted for a slightly different approach. As long as Haystack adapts in a consistent fashion with a user's expectations, we believe it is valuable to mold the data model to the user based on query interactions.

Just as Haystack observes the user's interaction with the "outside world," services within also observe the user's interaction with Haystack itself. Specifically, we are interested in representing the queries and paths that a user may take in browsing his Haystack collection. To achieve this we introduce the concept of query straws and query paths.

When queries are made into a Haystack interface two things happen. The first is that the query is routed to the appropriate information system and the results are presented to the user. The second is that a new node appears in the Haystack data graph representing that query. We call this node a query straw. This straw is actually of the bale type described above. Haystack attaches the nodes corresponding to the matched documents to the query straw. The query straw also contains the actual text of the issued query, the relative rank of the returned documents, and other possibly useful pieces of information about the query. Because the queries are part of the Haystack graph, users can also annotate, add to, and disconnect (irrelevant) documents from, the query straw. If a user does not get completely satisfactory answers from the information system, he can make adjustments to the result set. As the result set is archived within the data graph, subsequent queries of the same form will generate a pointer to the user-adjusted result set. By providing this facility in Haystack, we allow users to modify the system's idea of relevance to match their own. Haystack can also reinforce the weighting of documents frequently visited in response to a query. This approach takes inspiration from IR relevance feedback techniques [FBY92, Chapter 11], but differs in purpose and execution.

In a perfect world the user will get back data he is interested in instantly. In reality, however, it takes a user numerous iterations with the system to get at this information. Haystack observes this behavior and annotates the data to reflect the fact that various query straws are chained together. These chains are called *query paths*. The Haystack query interface provides a method to either start a new query or indicate that the user is continuing along the same path. We decide that a user has found what he is looking for when he asks to see the contents of a document. This is not a perfect solution, but it allows us to decide where a query path ends. Query paths give us two important benefits that help in subsequent retrievals. When searching for the same or similar documents, a user is presented with similar query objects. From the query objects a user can travel the same query paths he has in previous iterations. This allows users to reduce the number of steps necessary to find information they previously found relevant by having access to the full query path up-front.

Additionally, documents within Haystack are indexed when the structure surrounding them changes. When a document becomes the terminal point for a query path, the entire text of the query path (as it is local to the document node in the graph) is indexed with the text of the HaystackDocument cluster. The index of the document now contains added terms which change the relevance of the document with respect to those terms. For example, if a user makes a query for the word "cat" and receives no matches, he may change his query to "feline." If feline generates results, and those results are acceptable, Haystack notes this. In re-indexing the relevant documents the term "cat" is added to the document. Subsequent queries for "cat" will generate the matching documents on the first try. In this fashion, Haystack learns the vocabulary and weighting of terms for an individual user.

A detailed evaluation of the improvements to precision and recall using this technique has yet to be undertaken, but in the limited use among project members this approach appears to provide some utility.

5.3 Human Annotation

The third and probably best source of information for Haystack is active annotation by the user. Unfortunately, active participation by users is also hard to achieve. Carroll [CAR87] describes this difficulty as the "Production Paradox," where users ignore learning steps that require effort but which will produce better end results.

We have worked to make annotation as easy as possible, so as to lower the effort required to annotate. We are hopeful, as well, that users who are working with their own data will be more motivated to work on organizing it than those whose annotations in past social filtering systems have mainly provided "benefit" to other users of the system.

To ease the user's work in annotation, we provide multiple interfaces so that the user can choose whichever is most convenient at the time. Currently these interfaces include a custom (personal) web server, a Java based GUI, a command line tool, and an emacs interface. This variety of access mechanisms will perhaps encourage the integration of annotation into the user's workflow. However, significant user studies will be required to refine this approach.

6 Evaluation

Haystack is currently in a limited alpha release, but is approaching a stage in which it can be released to a public user-base. In anticipation of this we have built into Haystack an extensive logging facility that will allow us to monitor how users interact with the system. Once the public release occurs, our hope is that we will be able to construct a broader user-study to evaluate the utility of Haystack based on these logs.

Because of its personal nature, Haystack is hard to evaluate in large studies. The system requires seeding by a user and continued use for adaptation to occur. Additionally, relevance in the context of personal information is highly subjective. Standard methods for judging IR systems require a standardized corpora where experts have judged relevance of documents to various queries. These evaluation methods, which are largely based on precision/recall curves, are hard to apply to system such as Haystack.

7 Conclusions and Future Work

Haystack's goal is to draw together some of the threads that have been explored separately in information retrieval and information filtering. Integrating the information retrieval system into each individual's desktop instead of restricting it to large, communal corpora gives us the opportunity to personalize the information retrieval process, adapting it to individual attitudes about what is interesting and how it is described. Two different Haystacks might give completely different answers to two different individuals' queries, and be right because the two users mean different things. The fact that the system is explicitly designed to deal with queries creates an alternative to filtering systems' model of a "notification" service, instead allowing useful processing of a users specific information need at the time it becomes apparent.

The elements of the Haystack system described above have all been implemented, although the prototype requires scaling issues to be resolved. The work to date has focussed on the problems of data representation and gathering, but we can now turn towards many of the more interesting problems involved in exploiting the large amounts of data we have gathered. Among the tasks we hope to undertake:

- After building up a database of user queries, refinements, and reactions to the results, use machine learning tools to improve retrieval performance in future queries. Among the learning possibilities are learning which documents are "high quality" and preferred by the user (independent of the query) and learning what additional terms a user tends to associate with a given query term (personalizing and automating query expantion, so that the user needn't bother).

- Provide a better interface to "hybrid search" methods that allow users to mix full text, relational, and associative queries.

- Discover users' "interests" (based on what they own) and use the information to drive a recommender system that looks for interesting material on the web. Haystack gathers far more data about a user than typical recommendation systems, and we hope that this will let it make better recommendations.

If a person's bookshelf fails him, he still has an alternative to the library. The natural next step is to ask his colleagues in neighboring offices. Turning to a colleague offers several advantages over a trip to the library. Colleagues have their own personalized collections of information which they can search effectively. They share interests and vocabulary with the original questioner, and are thus likely to be able to understand that person's information needs and effectively communicate anything they might know that can help. A person can describe his problem in a language common to him and his colleague, and his colleague can then use her own knowledge of her collection to find what the original searcher wants. Finally, books in colleagues' personal collections are more "trustworthy" than random books selected from a library. Their presence in the colleague's collection indicates that someone we trust considers them valuable.

8 Acknowledgements

The authors would like to thank Mark Asdoorian, Aidan Low, and Ilya Lisansky for their work on Haystack. We would also like to thank Marti Hearst, Jan Pedersen, Lada Adamic, and Jeanette Figueroa for their valuable comments.

References

[ADA98] Eytan Adar. Hybrid-Search and Storage of Semi-structured Information. Master's Thesis, MIT, May 1998.

[AV98] AltaVista Discovery homepage `http://discovery.altavista.com`.

[AP98] Apple Computer's Sherlock `http://www.apple.com/sherlock/`.

[ASD98] Mark Asoorian. Data Manipulation Services in the Haystack IR System. Master's Thesis, MIT, May 1998.

[BSY97] Marko Balabanovic, Yoav Shoham, and Yeogirl Yun. An adaptive agent for automated web browsing. Technical Report CS-TN-97-52, Stanford University, 1997.

[BU45] Vannevar Bush. As We may Think. *Atlantic Monthly*, 176(1)641–649, January 1945.

[CAR87] John M. Carroll and Mary Beth Rosson. "Paradox of the Active User" in *Interfacing Thought: Cognitive Aspects of Human Computer Interaction*, ed. John M. Carroll., MIT Press, Cambridge MA, 1987, pp. 81–111.

[CKPT92] Douglass Cutting, David R. Karger, Jan Pedersen, and John W. Tukey. "Scatter/gather: A cluster-based approach to browsing large document collections." In *Proceedings of the 15^{th} Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 318–329, Copenhagen, Denmark, 1992.

[ENG62] Douglas C. Engelbart Augmenting Human Intellect: A Conceptual Framework. *Stanford Research Institute Technical Report*, Menlo Park, CA, October 1962.

[FBY92] William B. Frakes and Ricardo Baeza-Yates, editors. *Information Retrieval: Data Structures and Algorithms.* Prentice Hall, Englewood Cliffs, New Jersey, 1992.

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

[JOA95] Thorsten Joachmis, Tom Mitchell, Dayne Freitag, and Robert Armstrong. "WebWatcher: Machine Learning and Hypertext," *Proceedings of 15^{th} International Joint Conference on Artificial Intelligence*, 1997.

[KR97] Joshua Kramer. Agent Based Personalized Information Retrieval. Sc.M. Thesis, MIT, June 1997.

[LIE95] Henry Lieberman. "Letizia: An Agent That Assists Web Browsing," *Proceedings of the International Joint Conference on Artificial Intelligence*, Montreal, August 1995.

[LOW99] Aidan Low. A Folder-Based Graphical Interface for an Information Retrieval System. Master's Thesis, MIT, May 1999.

[MCH97] Jason McHugh, Serge Abiteboul, Roy Goldman, Dallan Quass, and Jennifer Widom. "Lore: A Database Management System for Semistructured Data." *SIGMOD Record*, 26(3):54–66, September 1997.

[RHO96] Bradley J. Rhodes and Thad Starner. "Rembeerance Agent: A continuosly running automated information retrieval system," *The Proceedings of the First International Conference* *on The Practical Application of Intelligent Systems and Multi Agent Technology*, April 1996, London.

[SAB94] Gerard Salton, James Allan, and Chris Buckley. Automatic Structuring and Retrieval of Large Text Files. *Communications of the ACM*, 37(2):97–108, February 1994.

[AP98] Sleepycat Software `http://www.sleepycat.com/`.