Haystack: Per User Information Environments

David R. Karger*
Computer Science and Artificial Intelligence Laboratory
M. I. T.

January 29, 2006

1 Introduction

Every individual works with information in his or her own way. In particular, different users have different needs and preferences in regard to

- *which information objects* need to be stored, retrieved, and viewed;
- what *relationships and attributes* are worth storing and recording to help find information later;
- how those relationship and attributes should be *presented* when inspecting objects and navigating the information space;
- what *operations* should be made available to act on the presented information;
- and how information should be gathered into *coherent workspaces* in order to complete a given task.

Currently, developers make such decisions and hard-code them into applications: choosing a particular class of objects that will be managed by the application, deciding on what schemata those objects obey, developing particular displays of those information objects, and gathering them together with relevant operations into a particular workspace. The Haystack project takes a different approach. We posit that no developer can predict all the ways a user will want to record, annotate, and manipulate information, and that as a result the applications' hard-coded information designs interfere with users' ability to make the most effective use of their information.

Our Haystack system aims to give the end user significant control over all four of the facets mentioned above. Haystack stores (references to) arbitrary objects of interest to the user. It records arbitrary properties of the stored information, and relationships

*M.I.T. Computer Science and AI Laboratory, 32 Vassar St., Cambridge MA 02139. E-mail: `karger@mit.edu`.
URL: `http://theory.lcs.mit.edu/~karger`.

to other arbitrary objects. Its user interface flexes to present whatever properties and relationships are stored, in a meaningful fashion.

To give users flexibility in what they store and retrieve, Haystack coins a *uniform resource identifier (URI)* naming anything of interest to the user—a digital document, a physical document, a person, a task, a command or menu operation, a view of some information, or an idea. Once named, the object can be annotated, related to other objects, viewed, and retrieved.

To support information management and retrieval, a Haystack user can record arbitrary (predefined or user-defined) *properties* to capture any attributes of or relationships between information that the user considers important. Properties of an object are often the information users are seeking when they visit the object. Conversely, they may help users find the objects they want: the properties serve as useful query arguments, as facets for metadata-based browsing [YSLH03], or as relational links to support the associative browsing typical of the World Wide Web.

Haystack's user interface is designed to flex with the information space: instead of using predefined, hard-coded layouts of information, Haystack interprets *view prescriptions* that describe how different types of information should be presented—for example, which properties matter, and how their values should be (recursively) presented. View prescriptions are themselves customizable data in the system, so they can be imported or modified by a user to handle new types of information, new properties of that information, or new ways of looking at old information. Incorporating a new relationship or even a new type of information does not require programmaticaly modifying the application or creating a new one; instead, an easy-to-author view prescription can be added to describe how to blend the new information seamlessly into existing information views.

Beyond letting users customize the information they work with, Haystack lets users customize their information management activities. By "snapshotting" partially completed dialogue boxes, a user can create specialized operations to act on their data in common ways. At a higher level, a variation of view prescription approach is used to define *workspaces* for a particular user task, describing which information objects are involved, how they should be laid out, and what operations should be available to invoke upon them. With Haystack's unified information model, any heterogeneous set of objects can be brought into a coherent visualization appropriate for a given task.

The need to flexibly incorporate new data types, presentations, and aggregations is not limited to individual users. As is demonstrated by this volume, researchers keep proposing beneficial new attributes, relationships and data types. Plaisant et al. propose to tag all information objects with a "role" attribute that can be used to determine under which circumstances a given information object is relevant, and which operations on it should be available. Fisher and Nardi propose that information management will be improved by recording and displaying linkages from information objects to the people relevant to those objects. Freeman advocates recording and presenting according to the access time of all of a user's information objects. The articles all make good cases, suggested that each is correct *some of the time*. The Haystack system demonstrates an infrastructure that would make it much simpler to incorporate such new ideas in a single system as they arise, and invoke each of them at the times that are appropriate, as opposed to crafting new and distinct applications (and convinving users to migrate to them) for each new idea.

1.1 Principles

Haystack's design is guided by a number of principles and concepts. Many of them seem obvious and almost wasteful to assert. But all of them might be debatable, so we attempt to justify them in our motivation section below.

Universality. Users should be able to record any information object they consider important or meaningful, and should be able to seek, find, and view it later.

The centrality of metadata and relationships. Much retrieval of objects is based on recalling specific attributes of the objects and relationships to other objects. Thus, the system must be able to record whatever attributes and relationships matter to the user, display them, and support their use in search and navigation.

One information space. There should be no a-priori segregation of a user's information by "type" or application. Rather, all information should exist logically in a single space. Users should be able to group and relate any information objects they choose.

Personalization. No developer can predict what kinds of information objects a user will want to store, or what attributes and relationships will be meaningful to them for retrieval. Thus, the system must let the end user define new information types and properties, and adapt to store, present, and search using them.

Semantic Identity. There should be only one representation of a particular information object in the data model (as opposed to having distinct representations stored by different applications). Any visible manifestation of that object should be "live", offering access to that object (as opposed to, say, simply acting as a dead text label for an object that must be located elsewhere).

Separate data from presentation. The development of multiple views of the same information object should be encouraged, so that the right view for a given usage can be chosen. It should be possible to use each such view to be used in whatever contexts are desired, instead of restricting each view to certain applications.

Reuse presentations. Many types (such as "email message") are instances of more generic types ("message") that have other incarnations (newsgroup posting, instant message, telephone call) and to which many attributes (sender, recipient, subject) and operations (reply) apply uniformly. We should design views to apply generically when possible, so that the user can ignore differences that are irrelevant to their information management needs.

This chapter explains the motivation for these principles and to describe the system we have built to apply them.

1.2 A Tour of Haystack

To begin exploring Haystack's design, we take a brief tour through an end-user's view of Haystack. In Figure 1, we see a screen shot of Haystack managing an individual's inbox. As is typical of an email application, Haystack shows the user's inbox in the primary browsing pane. The layout is tabular, with columns listing the sender, subject, and body among other things. Less usual is the fourth "Recommended categories" column, which the user added to the display by dragging the Recommended Categories view from the message on the lower right into the Inbox column header. As is usual, the collection includes a "preview" pane for viewing selected items which is currently collapsed.

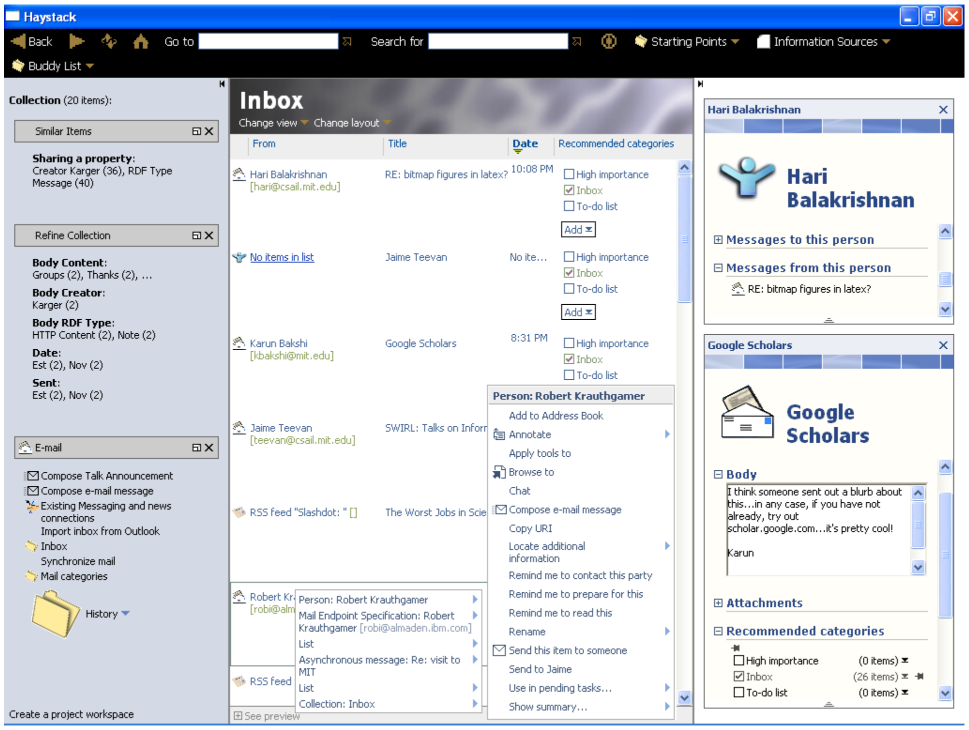
While the Haystack inbox looks much like a typical email collection, it contains much more. Some of the items in the inbox are not email messages. There are stories from RSS feeds, and even a person—perhaps placed there as a reminder that the user needs to meet with her. The RSS message has a sender and a date, but the person does not. This is characteristic of Haystack: rather than being inextricably bound to an "email reader application", the inbox is a collection like all other Haystack collections, distinguished only as the collection into which the user has specified that incoming email (and news) be placed. It is displayed using the same collection view as all other collections. Any items can be part of the Inbox collection, and will be properly displayed when the inbox is being viewed. This means that the Inbox can serve as a general purpose "Todo List". Bellotti et al. [BDHS03] that many users have forced email into this role but then had to cope with the constraint that only email could be in their todo list; Haystack does away with the constraint.

It is also worth noting that RSS was a "late arrival" in Haystack. The view showing the Inbox was created before RSS was developed. When we made the decision to include RSS stories as a new type of information to be handled by the system, we did not make any change to the user interface. Instead, we simply added a small *view prescription*—a small annotation explaining which attributes of an RSS object were worth seeing—and Haystack was immediately able to incorporate those stories as is shown in the figure. This is standard for Haystack: new types of information, and new attributes of information, do not force modifications to the visualization tool. Instead, lightweight annotations give Haystack sufficient information to seamlessly incorporate the new types and attributes among all the existing data.

On the right hand side of the screen is a clipboard-like "holding area" for arbitrary items; it currently contains an email message (about Google Scholars) and a person (Hari Balakrishnan). Various aspects of the message are shown, including the body, attachments (currently collapsed) and recommended categories. Displayed aspects of the person include messages to and from them; others such as address and phone number are scrolled out of view.

The bottom of the left panel shows that the "Email" task is currently active, and lists various relevant activities (composing a message) and items (the inbox) that the user might wish to invoke or visit while performing this task, as well as a history of items that the user previously accessed *while performing this task* (expanded in Figure 2). The tasks can be invoked, and items visited, by clicking on them.

Indeed, the user can click on any item on the screen in order to browse to a view of



I think someone sent out a blurb about this…in any case, if you have not already, try out scholar.google.com…it's pretty cool!

Karun

Figure 1: Haystack viewing a user's Inbox collection. A person and an email message are displayed to the right. The user right-clicks on the person "Robert Krauthgamer" to open a context menu of person-relevant operations.

that item—the individual messages, the various individuals named as senders or recipients, or any of the "recommended categories". Similarly, the user can right click on any visible item in order to invoke a context menu of operations that can be applied to that object. The user right-clicks on one of the people listed as a message sender and a menu (and sub-menu) opens up listing operations that might be invoked on that person, such as sending him an email message, initiating a chat, or entering him in the address book. The operations are not limited to those typical of an email application; rather, they are the ones applicable to the object being viewed. One of the operations, "Send to Jaime", was created by the user because he performs that operation frequently. He saved a partially completed "Send this to someone" operation; Haystack automatically recognized that this new operation was applicable and added it to the context menu.

Finally, the user can drag one item onto another in order to "connect" those two items in an item-specific way—for example, dragging an item onto a collection places the item into the collection, while dragging an item into a dialog box argument (see Figure 2) field binds that argument to the dragged item. These three actions—click to browse, right click for context menus, and drag and drop—are pervasive. They can be invoked at any time upon any visible object in a uniform fashion.

A “browsing advisor” in the left pane suggests various “similar items” to those in the collection—such as items created by Karger, or of type message—and ways to “refine the collection” being viewed—for example, limiting to emails whose body contains certain words, or that were sent at a certain time.

2 Motivation: Personalizable Information Management

Before embarking on a detailed discussion of the design of the Haystack system, we attempt to motivate the design by elaborating the problem we are trying to solve. We observe that current applications straitjacket users into managing information in ways that may not be natural for them, and argue that good information management tools must give users more control over what kinds of information they store and how they visualize and manage it.

2.1 Impersonal Applications

The Haystack project is motivated by the observation that different people have distinct and often idiosyncratic ways of managing their information. For example, I use a traditional subject classification for his books (with many exceptions for books that are in heavy use). My wife arranges her books chronologically by birth date of the author. A friend groups her books by color. Each of these three organizations works well for its owner, reflecting what he or she remembers about books they are seeking, but would fail for many other people.

This variability in physical organizations is squeezed into conformity in the management of digital information.[1] Instead of our bookshelves, current applications are one-size-fits-all: someone else has decided on the right “organizing principles,” and we are stuck using them whether we like them or not. The application’s choices may be “optimal” with respect to some hypothetical average user, but always seem to miss certain attributes a given individual would find useful. In a typical email program, we can sort only by the application’s predefined fields such as subject, sender, date, and perhaps importance, but not by by more idiosyncratic features such as sender’s age, “where I was when I read this”, number of recipients, importance of sender, or “needs to be handled by”.

Sometimes, this is because the desired information is recorded in the application but no appropriate interface is offered to make use of it. For example, although all modern email clients track the threading/reply-to structure of email messages, Microsoft’s Outlook Express 6 does not permit a user to display or sort according to the number of messages in a given thread. So if what I remember about an email message is that it generated much discussion, there is no easy way for me to exploit that information to find the message. Mozilla Thunderbird does have this capability, because an application developer decided it was worth incorporating. And although the data is present

[1]A direct observation of this phenomenon can be found in Nicolson Baker’s famous essay “Discards” [Bak94], in which he laments how the transfer of paper card catalogs to electronic form lost fascinating information that had been penciled onto them by patrons.

to answer the question, neither tool lets me display or sort messages according to how long it has been since I have heard from the sender.

In other cases, the problem starts even earlier, when information a user cares about cannot even be *recorded* in an application. Email programs do not contain a "due by" field for email messages. Although mp3 music files come with ID3 tags for recording various sorts of metadata, there is no field or representation for a user to record the dance steps associated with a given piece of music. And while photographs come with metadata fields, none of them is designed to hold a pointer to the email message that was used to send me (and tell me about) the photo. Neither Microsoft's nor Thunderbird's address-book tools allow me to include a photograph of a given contact. As an academic, I might even want address book entries for my colleagues to contain descriptions of, and links to, some of their papers I have read. But there is no "publications" field in typical address books, because the application developers did not think it worth including.

In many cases, one could try to resort to using one of the generic, "note" or "comment" fields available as a last resort in many applications, but this abandons all opportunity for the application to use those fields in a meaningful way. Although I could write down the filename of a photo of a given contact, I couldn't see that photo when looking at their address book entry—instead I would have to manually seek out and open the named file. And even if I recorded my colleagues' publications in an address book "custom" field, I wouldn't be able to use the address book to select those colleagues who have published in a given conference. If I record a "due by" date as text in the comment field of an email message, I would likely not get the by-date result I desire when I sort by that field, since such fields are generally sorted lexicographically. To fix this problem I would have to invent and remember a specific "backwards" representation of dates (year-month-day, with careful attention to always using two digits for date and month).

2.2 Unification

A common reason that an application does not record a given type of information is that the information is considered a "different application's job"—a developer could argue that recording publications seems a job for a bibliography tool, not an address book, or insist that answering an email by a certain date should be handled by your to-do list or calendar instead of your email program. After all this passing the buck, the information a user needs is stored in *some* application, but it is not possible to gather it all into one place and use it together, or even to navigate easily from one piece in one application to a different piece in another application. If my calendar shows that I have to deal with a certain email, I have to go find the message in my email program before I can deal with it. Such data fragmentation can also lead users to record duplicate information in multiple applications, which can then lead to inconsistencies when the user changes one but not all copies of that information. If a follow up message arrives eliminating the need for a response, I might forget to delete it from my to-do list (especially as it would involve more work to get from the email message to the corresponding to-do item). Our recent article on data unification [KJ06] discusses this issue at great length.

Difficulties multiply with more applications. Many people make use of an email-reading client, a music management tool, a photo album, a calendar, and an address book. The email client and address book may be somewhat linked, but the other applications manage their own data independently. Now consider the plight of an entertainment reporter following the music industry. She exchanges email with musicians, schedules interviews with them, attends scheduled concerts where they play certain songs, and writes reviews and interviews. It seems likely that such a user would want to

- Associate email about a certain interview with the interview article she is writing;
- Link musicians to messages from them (as is demonstrated in the Person object in the clipboard of Figure 1), concerts they played, songs they performed, and photographs they are in;
- "Caption" performance photographs of musicians with the song being performed in the photo;
- Place songs or albums in a calendar according to release date;

and so on. At present, while each item of interest is managed by *some* application, none is aware of the other item types or applications. The applications' internal models are not expressive enough to refer to the other item types (except through English-language "comments") and their user interfaces do not display the linkages that interest the user, or bring together the related objects into a single coherent representation. The system does not know that the artist recorded in the ID3 tags of a song is the same one mentioned in the address book. The best the reporter can hope for is to open all the relevant applications for this data simultaneously, at which point the information she actually cares about is lost in a clutter of other information less relevant to their particular task. The reporter must struggle to take applications targeted to certain tasks "repurpose" them for for a new task. She would likely prefer an application targeted specifically at her task.

In a study of users' desktop environments, Ravasio et al. [RSK04] observed that users are themselves aware of this issue: "the systematic separation of files, emails and bookmarks-was determined by three users to be inconvenient for their work. From their points of view, all their pieces of data formed one single body of information and the existing separation only complicated procedures like data backup, switching from old to new computers, and even searching for a specific piece of information. They also noted that this separation led to unwanted redundancy in the various storage locations."

This fragmentation of related information across applications blocks an important information seeking strategy. In a recent study [TAAK04], we highlighted the important role played by *orienteering* in the quest for information. Rather than giving a precise description of their information need, and expecting the computer to "teleport" them directly to the relevant object, users often *orienteer*, starting somewhere "near" the object in their information space, then navigating to the object through a series of local steps. When related data is spread across multiple applications that are not "linked", it becomes impossible for users to follow natural orienteering paths.

2.3 New Data Types

Beyond adding information to existing types, users may also discover a need for brand new types of information. Sometime this may be an enrichment of an existing application. For example, calendar programs often let one record the location of an event. But this record is merely a text string. A user might want a richer representation that incorporates directions to the location, a map of it, a list of nearby hotels, or a link to the site's web page if it exists. At other times, a user may create a new type from scratch. Our music reporter may realize that she wants to record and retrieve information about concerts—where they happened, who else attended, who played, how good they were, how many people were arrested, and so on. Where should she place this information?

Existing applications are even worse at incorporating new types of information than they are at extending existing types. They offer no widgets for displaying the new type, no operations for acting on it, and no slots to attach it to other information objects in the application.

Faced with this situation, users often turn to a spreadsheet, creating a tabular record in which each row is an "item" and each column corresponds to some attribute of the new information. This "poor man's database" does give some ability to work with the information: users can sort on a column to find all information with a given value of some attribute. But the presentation is necessarily generic, without any of the task- and data-specific presentations, menus, and operations that applications offer to ease work in specific domains.

2.4 Personalization

Application developers could easily solve each of the specific problems offered as examples in this section. An application developer could add a place for a contact's photo to the address book, or write an address book function that queried the email database and listed "messages from this person," or make live links from a photograph to an email message. One might even build a specialized data management application for music reporters, complete with a new "concert" data-type.

But there are far more application users than application developers, and each will have different needs for their application's information and presentation. Even if application developers could somehow keep up with all these individual wishes, the resulting applications would be cluttered with data and presentations that most people do *not* care about. Kaptelinin and Boardman in this volume argue that efforts to serve all users within a single application bloats the applications to the point that they are no longer useful for their original intended task. And the set of desired attributes is surely a moving target: no matter how many kinds of information the developers fit into their application, there will be a user who decides he wants one more.

One, and perhaps the only, way to surmount this problem is to give end-users themselves the ability to customize the data managed by their applications and the way it is presented. Such customization is already offered to some extent: most mail programs, music managers, and file/directory explorers give users the ability to choose which columns (attributes) are displayed in the tabular view of their displayed collections. But this customization is limited mainly to tabular/aggregate views; the user has less control over the presentation of single information objects. When it comes to adding and then working with *new* data types and attributes, much less support is offered—often, the ubiquitous and generic textual "comment" field is the only place to hold a user's new information type.

2.5 Lessons from the Web

The World Wide Web would appear to address many of the problems we have outlined with today's applications. On a web page, users can record any information they want about any type of object. To add a new information attribute, users just type it into the existing web pages—no applications need to change. They can link to other web pages, regardless of the type of object that other web page describes—there are no "partitions by application". The web is thus ideally suited for orienteering: a search engine will generally take one to the right neighborhood, from which a series of helpful links can be followed to the desired information.

Should users then abandon applications and move all their data to the web? Hypothetically, a user could create a separate web page for each email message, each directory, each music file, each calendar appointment, each individual in their address book, and so on. Editing these pages, the user could indicate arbitrary relationships between their information objects. Feeding these web pages to a tool like Google would give users powerful search capabilities; combining them with the orienteering opportunities offered by the user-created links would surely enhance users' ability to locate information.

Of course, such an approach could never work: it requires far too much effort on the part of the user. It is not clear that the payoff in better retrieval is worth the investment of organizational effort.[2] And the investment will be huge, given the current state of web content creation tools. Far more people consume web content than create it; they treat it as a read-only resource. And "read-only" should be taken literally—users generally inspect web pages by eye in order to extract their information, rather than feeding them to any sophisticated automated tools. Conversely, when people work with their own information, they manipulate and modify it in various ways. Such manipulation and modification needs to be easy, so each application comes with its own specialized interfaces and operations for manipulating the data it manages. We need to let users manage their information without forcing them to become web site developers.

2.6 Typing and Structure

Applications offer *structured* data models involving carefully defined types and relationships. These models make it easier for users to manipulate, navigate and display information in natural ways than on the web. Many of the information objects people work with, attributes of them, and relationships between them have meaningful types. People are a type of object that tends to have attributes like name, address, and phone

[2]Even we are optimistic about the payoff, a quick perusal of colleagues' offices and desks suggests that many of us are too shortsighted to invest information organization effort now that will pay off in better retrieval later.

number. Mail messages are a type of object that typically has attributes like a sender and a receiver, both of which are often instances of the person type. And while the sender and receiver may both be people, these two attributes represent distinct roles with respect to the message that should not be confused.

On the web, it is typically left to the reader to induce the types of objects and relationships. The type of an object being viewed may be implied by the title of the object, or by its exhibited attributes, or its placement in context. The types of the attributes are often indicated in English in the text preceding or anchoring links—thus, for example, the Internet Movie Database page for a given movie has headers like “Directed by”, “Writing credits”, and “Genre” introducing links to objects that exhibit those relationships to the given movie.

The drawback of such visual or English-language cues is that they can *only* be understood by a human. While this may be fine for basic orienteering, it prevents our applications from exploiting the implicit types to improve the presentation and retrieval of information. Depending on context, a user may not want to see *all* information about an object. To support email browsing, for example, mail tools generally offer a condensed view of (a list of) messages showing only sender, date, and subject. Such information is painful to extract from an arbitrarily human-formatted web page, but easily accessible from a structured data model. In this sense, web pages are much like the flexible comment fields available in many applications—able to record everything, but not in a way that can be used effectively by the supporting applications.

Storing information in structured form also enhances search capabilities. Individual applications typically exploit structure this way—for example, a music-playing application will let users arrange their music by composer, album, or performer. More generally, Hearst et al [YSLH03] have argued that faceted metadata browsing, in which a user can choose to organize the corpus according the value of chosen attributes, is an important information retrieval tool.

While applications have always imposed such structure on their data, the web is trending in that direction as well. To support navigation and search, web sites like the IMDB store their information in structured form and then render it into human-readable form through the use of templates. Structured storage means that sites can offer “site search” tools that exploit the available metadata in richer ways than pure full text search. For example, Epicurious.com lets users search and browse on various classes of recipe ingredient or cooking method—a kind of faceted metadata search [YSLH03].

But this move by web sites to be more like data-specialized applications means that they run into some of the same problems as those applications. Users can find that the attributes *they* consider important are not exposed for navigation, or are not available at all. When the data users are in spans multiple web sites, no one site is able to offer them an aggregated view of the information, or support navigation that moves between the sites.

What we need, then, is an approach that fuses the information-flexibility of the web with the rich interactions supported by applications’ structured data models.

2.7 User Interface Consequences

We have laid out our motivation for letting end users make their own choices about their information models—what information objects, attributes, and linkages matter to them. Accepting such a position imposes interesting challenges on the user interface.

Perhaps the simplest task of a UI is to display information. Over traditional application data models this is relatively straightforward. The developer considers, for each data type and display context, which features of that object need to be presented to the user. By considering the expected type of each feature, the developer determines some meaningful way to display each feature (a string, a number, a color, an icon) and some effective aggregation of all the individual feature presentations.

But in our personalizable data model, much less can be assumed about the data that will be displayed. The user may start recording or connecting to novel information types, for which no presentation mechanism was developed. Even for expected information types, the user may have developed new relations, or violated the schemas of old relations, so that the developers' assumptions of what needs to be displayed become invalid.

It follows that a user interface for our flexible data model will need to be equally flexible, adapting its presentations to the actual, rather than planned, content of its data model.

The web browser may seem like a promising start in this direction—it makes no assumptions about the structure of the information it is presenting, but simply renders formatted HTML. But where does that HTML come from? One possibility is that it is produced by hand—but above, we have argued that it is implausible to record all user information in HTML. Another possibility is to produce the HTML through the application of some templating engine to the underlying data model, as is done in many data oriented web sites. But this just pushes the problem down a level: current templates require the same strong assumptions about the structure of the data that limit applications.

Equally problematic is the "read-only" focus of web browsers. We need a user interface that also lets users *manipulate* their information with the ease they expect of typical applications.

2.8 Summary

In this section, we have outlined our motivation for a semistructured data model that can flex to the needs of any individual user, and for a user interface that can flex to fit the data model, incorporating new information attributes, new linkages between information, and new types of information. In the remainder of this paper, we describe our attempts to meet these goals in the Haystack system. After addressing the core data modeling and UI issues, we discuss some of the opportunities such a system offers.

3 Semantic Networks—The Haystack Data Model

Above, we have discussed the importance of letting users work with arbitrary information objects, and letting them record and use arbitrary new properties of those objects. Before we can think about an interface to support these activities, we need to develop a data model flexible enough to hold the information.

An effective generic representation supporting flexible information is a *semantic network:* a graph in which the nodes denote the information objects to be managed and the edges are labeled with property names to represent the relations we would like to record. An edge can only directly represent a *binary* relation, not one between more than two entities. However, the majority of relations we have encountered are binary, and higher arity relationships can generally be represented by reifying the relationship (creating a new information object to represent a particular relationship tuple, and using binary connections from the tuple to the entities that participate in the relationship), so this binary restriction has not been a burden.

In addition to what we think of as relations, semantic network edges can also represent what we think of as "attributes," "properties," or "features" of an object, by creating a link, labeled with the attribute name, between the object and the value of the given attribute. This highlights the fact that from a formal perspective these concepts are equivalent to relations. While the user may maintain an intuitive differentiation (e.g., that properties are intrinsic to an object while relations connect the object to other distinct objects), we will avoid drawing this distinction in the data model, and instead carry it into the user interface that aims to present the data in a way that matches the user's intuition.

3.1 RDF

While the original version of Haystack [AKS99] implemented its own semantic network representation, we have since adopted the *Resource Description Framework (RDF)* propounded as a standard by the World Wide Web Consortium [MM03]. RDF meets our representational goals. It uses uniform resource identifiers (URIs) to refer to arbitrary information objects—these are much like URLs, but need not refer to information stored on a particular web server (and certainly need not resolve over HTTP). In RDF, information objects are referred to as *resources.* Relationships are referred to as *properties*. And specific assertions that a given property holds between two resources are referred to as *statements*. The two resources linked by the statement are referred to as the *subject* and *object* while the chosen property is called the *predicate*. Properties are also named by URIs, which allows us to make statements about the property—such as a human-readable name for it, or the assertion that each resource should have only one value for that property. Statements too can be reified and given URIs, to allow one to record for example who asserted a given statement.

RDF also supports a type system with inheritance. A `Type` property is reserved to specify that a given resource is of a given type. Some resources, of type `Class`, represent types; these are the typical objects of a `Type` statement. There is a (most generic) class called `Object`; all resources are instances of this class. Properties are asserted to be of type `Property`.

RDF lets users define a collection of types and properties appropriate to a given usage. These properties can all be defined in a single (RDF) file; if that RDF file is given a URL, then individual classes and properties in it can be referred to using a label syntax (`http://url/#label`). The root URL is referred to as a *namespace* for the defined classes and properties. For example the *Dublin Core* defines types such as `dc:document` and `dc:person` and properties such as `dc:author` and `dc:title` (here `dc:` is shorthand for the Dublin Core namespace, while each suffix labels a specific class or property in the namespace).

Building atop RDF, the *RDF Schema language (RDFS)* and *Web Ontology Language (OWL)* [MvH03] can be used to define *schemata* for the classes and properties. RDFS and OWL are collections of properties and classes (defined in the RDFS and OWL namespaces) that can be used to assert typical schematic rules. For example, RDFS and OWL can be used to assert that the subject of a `dc:author` statement must be a `dc:document` and the object a `dc:entity`, or that a `dc:document` has at most one `dc:date`. We do not enforce schemata in Haystack; nonetheless, such schemata can be used to establish appropriate views of the information, or to guide (but not force) users in filling in values.

3.2 Why RDF?

One might question our choice of RDF as opposed to either XML or a traditional table-per-property relational database representation. In many ways, this question is unimportant. All three representations have equal expressive power. It is true that unlike traditional databases, RDF *can* be used without any schemata. However, RDFS and OWL can be used to impose schemata on an RDF model if we so choose. RDF has a standard representation in XML (RDF-XML) and can also be stored in a traditional database (with one table of triples, or with one binary table per named property). Conversely, XML or database tuples can be represented in RDF. Of course, the choice of representation might have tremendous consequences for the *performance* of the system as it answers a variety of queries. However, the end user will likely neither know nor care which representation lies under the covers of the user interface.

Nonetheless, a few features of RDF led us to select it. The lack of (enforced) schemata is an appealing feature we will discuss below. The use of URIs (uniform resource identifiers) for all information objects provides a uniform location-independent naming scheme. Also appealing is the fact that RDF places all information objects on a level playing field: each is named by a URI and can become the subject or object of arbitrary assertions. This contrasts (positively) with XML's hierarchical representation of information objects, in which the root object is "special" and related objects are nested deep inside a particular root object. RDF is more in keeping with our belief that the information designer cannot predict which information objects will be of greatest interest to a given user. Shades of this same argument appear in Codd's seminal paper [Cod70], where he argues that a hierarchical representation of information that is not fundamentally hierarchical introduces an undesirable *data dependence* that can trip up database users. A similar argument can be made regarding a relational database. Defining a database table with many columns suggests that those fields should be considered in aggregate, but different users may be interested only in some of those fields.

We could offer to project onto a subset of columns, but RDF surrenders from the start to the idea that each individual column may be interesting in its own right and deserve its own table, avoiding the whole question of how to project.

Yet another driver in our adoption of RDF is its structural similarity to the World Wide Web. The power of the web comes from its links, which let users navigate from page to related page. Similarly, the semantic net highlights the linkage of objects rather than highlighting the relations as a whole. This is important for two reasons. First, it captures a notion of "locality." When a user is working with a particular information object, it is quite common for them to want to visit "adjacent," related information objects in the semantic network. Second, linkage is an appropriate emphasis given the important role orienteering plays in individuals' information seeking behavior [TAAK04]. Rather than carefully formulating a query that precisely defines the a desired information target, users often prefer to start from a familiar location, or a vague search, and "home in" on the desired information through a series of associative steps. In RDF, the statements connecting subject and object form natural associative links along which a user can orienteer from subject to object. The database perspective might be more appropriate if a user wished to formulate a complex query, reflecting operations such as "join" and "project" that can be expressed concisely in a database language such as SQL. However, we do not consider typical users capable of working with such database query languages, so exposing them will be of limited value.

The various attributes displayed for each item in Figure 1 are often just other information objects related by some predicates to the displayed object. Haystack's user interface lets the user click on any of those information objects in order to browse to them, providing support for orienteering. As will become clear when we discuss the user interface, RDF's single notion of "predicate" is exposed to the end user in a number of different ways—sometimes as a relationship to another object, and other times as an attribute of the current object. "Properties" or "attributes" of a given object and "relationships" between pairs of objects are all represented by predicates in the data model.

3.3 A Semistructured Model

Beyond named relationships, structured data models often have *schemata* expressing knowledge of how different information objects and types will be related. For example, we might declare that the composer of a symphony will invariably be a person, or that any individual can be married to at most one other individual at a given time. Such schematic declarations are very useful. They can protect the user from errors in recording information, catching for example when a user swaps the composer and the title while entering information about a new symphony. They can assist in the presentation of information, letting us deduce that only one line will be needed to present the spouse in an address book entry.

But these protections are at the same time restrictions imposed by communal sense that might go against the desires of an individual. Consider someone with an interest in computer music: their attempt to record a particular computer program as the composer of a symphony will be blocked if the above schemata are enforced. Similarly, a researcher of polygamous societies might find themselves unable to view critical information in their records about people.

Thus, while schemata may be of great *advisory* value, we argue against *enforcing* them. There must always be a way for the user either to modify the schema or violate it if, given fair warning, they conclude that is the best way to record information they care about. This perspective is a natural extension of the idea of letting the user record whatever information she considers important. If we are faced with the choice of violating a schema or refusing to let a user record information she cares about, we choose the former. While developers may consider it unlikely for the sender of an email message to be an animal, and schematize the sender as a person, a user may decide otherwise. While documents typically have authors, a user might not care to record them. Semantic nets depend less on schemata than databases do: each named link can exist or not independent of any global schema.

A representation like this, in which it is possible to represent database-type structure but the structure is not enforced, is known as a *semistructured* data model. While we have argued that a semistructured model is essential to support a user's recording of information, it poses some problems when it comes time to actually present or manipulate that information. But these are problems at the *user interface* level, which we will address there, instead of trying to solve them by restricting the *data model*. As we shall see there, schemata can play an important role in semistructured information management; the difference is that the schemata become *optional* and *advisory* instead of being enforced. Thus, semistructured information is best seen as "schema optional" rather than "schema free."

3.4 Importing Data

Though RDF is appealing, most data is presently not in that form. Haystack generates RDF data by applying a collection of *extractors* to traditionally formatted data. At present we can incorporate directory hierarchies, documents in various formats, music and ID3 tags, email (through an IMAP or POP3 interface), Bibtex files, LDAP data, photographs, RSS feeds, and instant messages. Each is incorporated by an appropriate parser that is triggered when information of the given type is absorbed into the system.

Another outstanding source of semistructured data is the web itself. Many web sites use templating engines to produce HTML representations of information stored in back-end databases. We have studied machine-learning techniques to automatically extract such information from the web pages back into structured form in RDF [Hog04, HK04]. In our approach, the user "demonstrates" the extraction process on a single item by highlighting it and labeling its parts; the system then attempts to induce the (tree-shaped) structure of HTML tags and data elements that represent the object on the page. If successful, it can notice that structure on future pages and automatically perform the same extraction. Of course, Haystack does not care about where its RDF comes from, so other extraction methods [MMK99] can easily be incorporated.

4 Viewing Information

Given the representational power of the data model, the next question is how it should be presented to users so that they can effectively view and manipulate the stored information. Simply modifying traditional applications to run atop the unified data model would offer some limited benefit—for example, by reducing the amount of information duplicated over multiple applications, and therefore the amount of inconsistency among those duplicates. But it would leave users as constrained as before by the developers' sense of what and how information should be presented in what contexts. Instead, we must make it simple for the user interface to flex according to the users' preferences and the data it is called upon to display. We achieve this goal through a recursive rendering architecture in which essentially each object is asked to render itself and recursively makes the same request of other objects to which it is related [HKQ02, QK03].

4.1 Views

Most elementary information management applications present a hierarchical display of information on the screen. To display a particular object in a certain region of the screen, they subdivide that object's region into (typically rectangular) subregions, and use those subregions to display various attributes of the given object and to display other objects to which the object is related. Thus, a typical email application will present an email message by creating a region showing the sender, another region showing the subject, another region showing the body, and so on. The message might itself be in a subregion as part of a larger display of, say, a collection of messages, using distinct columns to present each message's (relationship to a) sender, subject, and date. A calendar view displays in each day a list of appointments, and an address book has a standard format for displaying an individual by listing properties such as name, address, phone number, and notes in some nicely formatted layout. The address itself may be a complex object with different sub-properties such as street, city, and country that need to be laid out.

When applications are targeted at specific domains, they can assume a great deal about what is being displayed in their subregions. The sender of an email address will be a person; he will have a name and address that can be shown in the sender region of the display. An address-book entry will describe a person who has an address. In Haystack we do not wish to make such assumptions: our Inbox above contains RSS stories, which perhaps do not have the same sort of sender as an email message. But we can still apply the recursive display principle. We can construct a view of any object X by (i) deciding which properties of X and relationships to other objects need to be shown, (ii) requesting recursive rendering of views of the objects required by X , and (iii) laying out those recursively rendered views in a way that indicates X's relation to them. As a concrete example, when rendering a mail message we might consider it important to render the sender; we do so by asking recursively for a view of the sender and then laying out that view of the sender somewhere in the view of the mail message. The recursive call, in rendering the sender, may recursively ask for a rendering of the sender's address for incorporation in the view of the sender.

The key benefit of this recursive approach is that the root view only needs to know about the root object it is responsible for displaying, and not about any of the related objects that end up inside that display. Incorporating RSS feeds into the inbox did not require a wholesale rewrite of a mail application; it simply required the definition of a view for individual RSS messages. Once that view was defined, it was invoked at need by the collection view showing the inbox.

4.2 View Prescriptions

Formally, views are defined by *view prescriptions* that are themselves data in the model. A view prescription is a collection of RDF statements describing how a display region should be divided up and which constants (e.g. labels) and related objects should be shown in each subdivision. It also declares that certain standard graphical widgets such as scrollbars and text boxes should be wrapped around or embedded in the display.

When a view prescription is invoked, it will require some *context* in order to render properly. Most obviously, we need to know how much space the rendered object should occupy. It is often useful to pass other state, such as current colors and font sizes, down from the parent view in order to get a consistent presentation. This is done by dynamic scoping—the view has access to an environment of variables set by the ancestral view prescriptions in the recursive rendering process. It can examine those variables, as well as modify them for its children.

The key task of Haystack's interface layer is to decide which view prescription should be used to render an information object. At present, we take a very simplistic approach: we select based on the *Type* of object being displayed and the *size* of the area in which it will be shown. Each view prescription specifies (with more RDF statements) the types and sizes for which it is appropriate; when a rendering request is delegated, Haystack uses an RDF query to determine an appropriate prescription to apply. Type and size are the most obvious attributes affecting the choice of prescription; an open research question of great interest is to expand the vocabulary for discussing which views are appropriate in which contexts.

When matching against type, Haystack uses a type-hierarchy on information objects and selects a view appropriate to the most specific possible type. The type hierarchy lets us define relatively general-purpose views, increasing the consistency of the user interface and reducing the number of distinct prescriptions needed. For example, RSS postings, email messages, and instant messages are all taken to be subtypes of a general "message" type for which we can expect a sender, subject, and body [QBK03]. Thus, a single view prescription applies to all three types. To ensure that all information objects can be displayed in some way, Haystack includes "last resort" views that are always applicable. For example, the "small" last-resort view simply displays the title or, if unavailable, the URI of the information object, while the "large" view displays a tabular list of all the object's properties and values (rendered recursively).

One might argue that our view architecture is remarkably impoverished, offering only rectangular hierarchical decompositions and delegation based on object type and size. While agreeing that this is in impoverished architecture, we assert that it captures much of the presentational power of current (equally impoverished) information management application displays, and hold up Figure 1, which can pass as a typical mail client, as evidence. While matching the presentational capabilities of existing applications, our delegation architecture enables the easy incorporation of new data types and cross-domain linkage of information.

One key improvement relative to existing applications is that views can be invoked anywhere. The right panel of Figure 1 holds a "clipboard" of sorts, into which any information object can be dragged for display. Thus information about the individual "Hari Balakrishnan" can be inspected without launching an entire address book application; similarly, the email about "Google Scholars" can remain in view even if we choose to navigate away from our inbox and stop "doing email". This idea of getting at data without the enclosing application connects with the Wincuts technique propounded by Tan et al. [TMC04].

Our view architecture also makes it straightforward to offer multiple views of the same information object, allowing the user to choose an appropriate view based on their task. The center pane of Figure 1 offers a "change view" drop down menu. From this menu, the user can select any view annotated as appropriate for the object being displayed.

It is also important to recognize that at the base of the view recursion, presentation of complex data objects can be delegated to special purpose widgets. Haystack's view prescriptions would be inadequate for describing the presentation of a scatter plot and the interactive manipulations a user might want to invoke while viewing it, but a prescription can certainly specify that some "scatter plot widget" is the proper view to invoke when a scatter plot needs to be displayed. This approach could even allow the embedding of entire applications within Haystack, so long as they can be told what data object to focus on.

4.3 Lenses

While it may suffice to display a list of attributes of a given object, the attributes often group naturally to characterize certain "aspects" of the information being presented. Such a grouping in Haystack is effecting by defining a *lens*. Lenses add another layer of indirection to the presentation of information. Like views, lenses are described in the data model as being appropriate to a certain type of object. The person and mail message in the right pane of Figure 1 are being displayed using a *lens view*. The lens view is applicable to *all* object types. It simply identifies *all* the applicable lenses for the given type, and displays each of them. Each lens has a title describing the aspect it is showing, such as "Messages from this person."

Unlike recursively rendered views, these lenses are "reified" in that the user can visually address each one, choosing to expand or collapse it (with the small plus/minus sign adjacent to the lens name). The choice is stateful: the user's choice of which lenses to show is remembered each time the lens view is applied for that type of information object. This provides a certain level of view customization. Furthermore, many of our lenses are simple "property set lenses"—they are described simply by a list of which properties of the object they will show, and those properties are shown in a list. Users can easily modify these lenses by adding properties to or removing them from the list. Thus, if a user chooses to define a brand new property in their data model, it is straightforward for them to adapt the user interface to present that property to them.

Lenses can also be *context sensitive*. For example, some lenses might only be present when a given task is being performed. The "recommended categories" lens shown for the Google Scholars email message is present only when the user is performing the "organizing information" task. A "help" lens could aggregate useful help information about any object, but should be visible only when the user is actually seeking help.

Users can further customize their views of information by manipulating lenses. For example, the fourth "recommended categories" column in the view of the inbox was created by dragging the "recommended categories" lens from the Google Scholars view onto the header of the Inbox collection. This would be a useful action if the user wanted to quickly skim and organize their email based on the headers, without inspecting the details of each. This tabular collection view lays out one item in each row, and applies a lens in each column to determine what information to show in that column about the object in a given row. Any lens can be placed in a column of this collection view, allowing the user to construct a kind of "information spreadsheet" showing whichever aspects the user cares to observe about the objects in the collection.

4.4 Collections

Collections are one of the most common data types people work with. Nearly every application offers tools for managing collections of its primitive elements: directory/folder managers for files, bookmark managers for web browsers, mail folders for email, and so on. Generally, these collections are limited to the type of object the given application "owns." Under Haystack's unified data model, it becomes possible to aggregate arbitrary collections of information germane to a given task. Perhaps the closest analogue in existing desktop systems is the file manager. Directories are able to hold files of arbitrary types, meaning that the user can group files by task instead of by file type. The limitation, of course, is that such management can be applied only at the file level. Thus, items whose representation is wrapped up inside an application's data file, such as individual contacts in an address book or individual mail message in a mail folder, cannot be organized into heterogeneous collections. Haystack, by providing a uniform naming scheme for all objects of interest, extends the benefits of heterogeneous collections to arbitrary objects. We have already noted how, in Figure 1, non-email objects such as RSS stories and people can be placed seamlessly into the inbox.

The availability of multiple views is particularly important for collections, which are perhaps the central non-primitive data type in Haystack. Since collections are used for so many different purposes, many views exist for them. Figure 1 shows the standard row-layout for a collection, but also available are a calendar view (in which each item of the collection is displayed according to its date—this view is applied to the inbox in Figure 2), a graph view (in which objects are shown as small tiles, and arrows linking the tiles are used to indicate specific chosen relationships between them), and the "last-resort" view showing all properties of the collection and their values. Each view may be appropriate at a different time. The standard view is effective for traditional email reading. The graphical view can be used to examine the threading structure of a lengthy conversation. And the calendar view could be applied by the user to rearrange email according to its due date instead of its arrival time.

Yet another collection view is the menu. When a collection is playing the role of a menu, a left click drops down a "menu view" of the collection, which allows quick selection of a member of the collection. Implementing menus this way gives users the power to customize their interfaces: by adding to and removing from the collection of operations in a menu, users modify the menu. Users can similarly customize the pinned-in-place *task menus* in the left pane (such as the Email task menu displayed in Figure 1) in order to make new task-specific operations and items available.

Traditionally, drop down menus are used to present collections of *operations*. While Haystack certainly does present operations in menus (see below), any object can be in the collections presented this way. Thus, the notion of lightweight access and putting away of a collection is separated from the issue of access to operations. For example, as shown in Figure 2, the results of a search in the search box at the top of the system are presented as a drop-down menu (but can also be navigated to for closer inspection and manipulation).

A particularly noteworthy collection view is the "check-box view" being exhibited in the bottom right of the display. This forms a somewhat inverted view of collections, in that it shows which of the collections from a given *category set* the Google Scholars email is in. Checking and unchecking a box will add or remove the item from the given collection. Of course, the collection itself is live—items can be placed in the collection by dragging them onto the collection name, and the collection can be browsed to by a left click. But in a past study [QBHK03], we demonstrated that presenting the collections to users as checkable "categories" made a big difference in the way they were used. Many email users are reluctant to categorize email away into folders, fearing that any email so categorized will be lost and forgotten from their inboxes. Many mail tools allow a user to *copy* and email message into a folder and leave a copy behind in the inbox, but apparently users find this too heavyweight an activity. In particular, once two copies are made, the user may have trouble keeping them in sync—an annotation on one copy will not appear on the other. Checkboxes, on the other hand, feel like a way of annotating the message, rather than a putting away, and therefore encourage multiple categorization. In our study, users given the option to categorize with checkboxes made use of it, and found that it improved their ability to retrieve the information later. In the underlying data model, of course, the checkboxes represent collections like all others that can be browsed to (indeed, the inbox itself is one of the checkable categories).

4.5 Creating New Views

We continue to explore ways to let users customize their information presentation. We have created a "view builder" tool that lets users design new views for given information types [Bak04]. The users use menus and dragging to specify a particular layout of screen real estate, and specify which properties of the viewed object should be displayed in each region and what kind of view should be used to display them. The representation of view prescriptions as data, rather than as code that is invoked with arbitrary effects, makes this kind of view definition feasible—it involves simple manipulation of the view data. This work is still in its early stages; while the system has the view-construction *power* we want, we continue to seek the most intuitive interfaces exposing that power to users. The current scheme requires users to explicitly refer to properties, types, and views, which may be beyond the capabilities of many users. Ultimately, we aim for users to edit the views in place, manipulating the presentation of the information by dragging appropriate view elements from place to place. Such design “by example” is likely to be within the capabilities of more users.

Even with ideal tools, many users will likely be too lazy to design new views. However, the description of views as data means that, like other data, views can be sought out from elsewhere and incorporated into the system. We imagine various power users placing view prescriptions in RDF on web sites where other users can find and incorporate them, much the way individuals currently define “skins” for applications such as MP3 players.

Longer term, we hope to explore application of machine learning to let Haystack create and modify views automatically. By observing the way a user examines and manipulates their information, the system may be able to hypothesize which attributes are actually important to a user in a given context, and construct views showing only those attributes.

At a higher level, the same view construction framework can be used to design entire *workspaces*—collections of information objects laid out and presented in a specific way, to support the performance of a particular task. We discuss this issue in Section 6.2.

5 Manipulation

Besides viewing information, users need to be able to manipulate it. Most of Haystack’s views offer on-the-spot editing of the information they present, as a way to change specific statements about an object. More generally Haystack offers a general framework for defining *operations* that can be applied to modify information objects in arbitrary ways. Most operations are invoked by *context menus* that can be accessed by right clicking on objects. Particularly common operations are supported by a natural drag and drop metaphor.

5.1 Operations

The basic manipulation primitive in Haystack is the *operation*. Operations are arbitrary functions that have been reified and exposed to the user. Each function takes some number of arguments. When the operation is invoked, the system goes about collecting its arguments. If the operation takes only one argument and the operation is invoked in a context menu, the argument is presumed to be the object being clicked. If more than one argument is needed, a dialog box is opened in the right pane to collect the other arguments. Unlike in many traditional applications, this dialog box is *modeless*. It does not force the user to finish filling it out before turning to other tasks. In particular, the user can use all of Haystack’s navigation tools to seek and find the arguments he wishes to give to the operation (by dragging and dropping them onto the dialog box) before invoking it.

Operations are objects that can be manipulated like any other objects in Haystack. In particular, users can drag operations into (menu) collections in order to make them accessible wherever the user wishes.

5.2 Invoking Operations

Context menus provide a standard way to access all the operations germane to a given object. Statements in the data model declare which operations are applicable to which types of objects; a right click leads to a database query that creates the collection of operations (and other items) that apply to the clicked object.

Drag and drop provides a way for a user to associate two information objects by dragging one onto the other. Dragging onto a collection has the obvious semantics of placing the object in the collection. Dragging onto a particular property displayed in a lens has the effect of setting the dragged object as a value for that property with respect to the object the lens is showing. Dragging into a dialog box argument assigns the dragged item as an argument to the operation being invoked. More generally, a view can specify the operation that should be invoked when a specific type of object is dragged into the view.

5.3 Customization

Like other data, operations can be customized by the user. In particular the user can fill in some of the invoked operation's arguments, then "curry" the result, saving it as a new, more specialized operation [QHKM03]. For example, a user may take the standard "email an object" operation, fill in their boss' email address as the destination, and then curry it into a "mail this to my boss" operation. Since the curried operation takes only one argument (the object to send), it can be invoked in a right-click context menu with no need for any dialog box. Once created, the new operation can be given a name, and then dragged into various collections of commands (menus) so that it can be accessed when needed.

We are working to offer users more powerful operation customizations. In addition to currying operations, we would like to let users define new operations by composing existing ones—passing the result of one operation as an argument to the next. We are also exploring techniques like those we use to extract information from web pages (Section 3.4) that let a user encapsulate web operations (accessed through web forms) as Haystack operations, which can then by accessed (and customized) through Haystack's interface. without visiting the web site.

Like views, operations offer an opportunity for arbitrary, fine-grained extensions of Haystack. Operations are defined in RDF, so can be created and offered up by power users for download by any individuals who find them useful. Some operations may simply be carefully curried operations; others may include newly crafted database queries, or even arbitrary code.

5.4 Example

Figure 2 shows what happens after a user invokes the "send this item" operation on a particular object. A dialog box in the right pane gathers the necessary arguments, including the object to send (already filled in) and the person to whom it should be sent. To fill in that person, we show how the user might drop down the email-specific history in the left pane, listing items recently used in while handling email. Since the desired recipient is not present, the user can perform a search in the search box in the top navigation bar. The (single) result matching this search appears in a drop-down menu. From there it can be dragged and dropped onto the dialog box in order to indicate that it is the intended recipient. If the user has cause to believe that he will need to send this particular item to other individuals, he can drop a context menu from the dialog box (shown) and select "save this as an operation" to create a new operation for which the item to send is prespecified, and only the intended recipient needs to be filled in. The resulting operation, which takes only a single argument (the intended recipient), will become available in the context menu that drops down when right clicking any person. A complementary operation, in which the recipient is pre-specified but the item to send is not, shows up as "Send to Jaime" in the context menu of Figure 1.

6 Tasks

Another concept we consider it crucial to model in Haystack is *tasks*. Without attempting a formal definition, we recognize that many people spend time engaged in what are commonly called tasks: dealing with their email, planning their day, writing a paper, surfing the web, shopping for an item, and so on. For each of these tasks, there is information the user will likely need to work with (the inbox for email, the calendar for day planning, the paper being written, and so on) and a collection of operations the user is likely to invoke while doing the task (sending a reply to an email message, scheduling an appointment, or spell-checking a document). Nowadays, it seems that people are often doing more than one task at a time; however, at most a few are likely to be simultaneously in mind.

6.1 The Task Window

In Haystack, we are exploring two approaches to supporting tasks. The first is the task pane shown on the left of the figures. The task pane can display a collection of objects and operations useful for a given task. For example, in Figure 1 we see an "E-mail" task window containing objects (such as the inbox) and operations (such as composing a message) relevant to the email task. The user can navigate to task-relevant objects, or invoke task-relevant operations, by clicking on them in the task window. The task window is simply presenting a collection, which can be manipulated like any collection. In particular, if the user decides that other objects or operations are frequently useful for the task, she can drag those items into the task-collection so that they will be accessible in the future. Of course, a user can also create brand new tasks, and populate them with relevant items.

Also visible in the task windows is a *task-specific* history collection, containing the items accessed in the recent past *while the user was performing this task*. Unlike a generic history such as might be found in a web browser, the task specific history does not become cluttered with irrelevant items visited while the user is performing other unrelated tasks. If there are items that a user accesses often while doing the given task, those items will tend to be found in the history. Thus, even if the user does not go to the trouble of customizing the task window to include items he needs for the task, the history provides a kind of automated customization accomplishing the same thing.

The task window is much lighter-weight than the typical application, but at the same time is significantly more detailed than an "minimized application. We believe that this middle ground can be very effective for multitasking users. Instead of cycling through expansion and collapse of various full-screen windows as they try to work with information from multiple applications, users can keep a little bit of state from each of their tasks in view.

The task windows can be seen as similar to the dockable "toolbars" currently available in many applications. However, their modeling as standard collections means users are free to incorporate any objects or operations they find useful.

Task windows become active in two ways. First, users may explicitly begin doing the task. For example, the user can select "E-mail" from the starting points menu (top right) in order to invoke the task. Alternatively, a user can type "E-mail" into the search box, and select the "E-mail" task from the result set. This is analogous to launching an application: the user explicitly states that she wishes to begin doing the task. A second option is for the system to guess at what tasks the user is doing. For example, in Figure 2, a grayed out "Addresses and Contacts" task is visible in the left pane. This is a sign that the system believes the user may be performing this task. If the user clicks on the grayed out header, the task window will expand to show the items relevant to that task. At present, such guesses are hard-wired into the system—certain objects and types are explicitly associated (by an appropriate RDF statement) with certain tasks. For example, the inbox is explicitly associated with E-mail, so any time the user navigates to the inbox, the E-mail tasks is offered in the left pane. In the longer term, we see this problem of discovering which tasks a user is currently performing as a fruitful target for machine learning research.

6.2 Workspaces

Our second approach to tasks is at a larger scale. A *workspace* is a (presumably rather large) region filled with (relatively detailed) views of various information objects that can be used to tackle a given task. For example, a traditional email application may present a region holding a collection of current messages, a region holding a particular current message, a region holding an address book, and so on. A user working on a paper about a particular research project may wish to gather and lay out the relevant research data, useful citations and documents, spell checking functionality, and mail-sending operations to their coauthors (cf. Section 5 on customizing operations). Continuing our main argument, that developers cannot predict what workspaces end users will want, we would like to give end users the ability to create their own workspaces, deciding what pieces of information should be presented (and in what way) to let them carry out a given task.

Creating a workspace is much like creating a new view. While a view may be intended to apply to many pieces of information, a workspace is typically created once, for a single task. While a view typically presents information associated with the object being viewed, workspaces instead present information associated with the task to be performed—in a sense, the workspace can be seen as a view of the task.

Given their similarity, we can apply tools similar to those for the construction of views to the construction of workspaces. To construct a workspace, the user needs to choose a collection of items to be shown in the workspace, choose a view for each of those items, and determine how those views should be laid out in the workspace. Choice of items (creating a collection) and selection of (predefined) views is already available as a standard part of Haystack. We have designed a prototype tool for managing the layout of the items so as to create workspace [Bak04].

Figure 3 shows a paper-writing workspace constructed with our drag-and-drop tools by assembling views that were also constructed with our drag and drop tools mentioned in Section 4.5.

7 Search

Beyond reading and writing information, search is perhaps the key activity in information management. Haystack offers a number of search tools in the system. We aim to make search both pervasive and lightweight—rather than dropping what they are doing and initiating a search, we want users to think of search as a no-overhead activity that is performed as part of regular navigation activity.

As we argued above, *orienteering* is a natural search mode. Should a plausible starting point be visible, we expect users to "hyperlink" their way from object to object, homing in on the one they are seeking. By placing user-definable task-specific collections of information in the left panel, we aim to maximize the chances that the user will find a good jumping-off point for their search.

7.1 Text Search

At times, of course, no such starting point is clearly visible. A simple scheme to fall back on at that point is text search. Information objects are often associated with memorable text, such as a title, a body, or an annotation. Haystack's upper navigation bar includes a search box into which an arbitrary textual query can be entered. The results of this search are a collection. The collection is presented in the drop-down-menu view of a collection, which optimizes for rapid selection of an item in the common case where the search is successful. However, the collection of results can also be "navigated to" to provide the starting point for a more complex search.

In Haystack, text is associated with many items—not just traditional data, but other objects such as operations. Thus, a user can search to find (and then invoke in place) an operation by describing it—essentially dynamically specifying a menu of commands relating to the given description. It also becomes natural to use search at a fine grain to locate small items, for example to locate a particular value to fill in as the argument to a dialog box.

Unlike text search of traditional corpora, where the text associated with a given item is clear (the text in the file plus its metadata), the question of what text to associate with a given RDF resource is complex. It is natural to associate with a resource any text directly connected to it by a statement, but one might also imagine associating text located at greater distance along a chain of statements.

7.2 Fuzzy Browsing

Much research has been done in the database community on search. Some [BMP01] have even looked for ways to offer ranked or approximate matching, avoiding the off-putting "all or nothing" effect of boolean database queries. However, as we argued above, attention needs to be given to orienteering, which manifests in search as an iterative process of query specification, inspection of the results, and refinement of the query. Orienteering along statements is natural to get from one resource to another, but when a user starts by issuing a query, they are faced with a *collection* of better or worse matches from which they need to orienteer. Yee et al [YSLH03] have explored *faceted metadata browsing* as a way to let users orienteer through a collection of data by choosing to restrict on certain attributes of the information.

In Haystack, we are exploring ways to bring orienteering tools from the text-search domain to the database domain [Sin03, SK04]. We propose to think of a resource's attributes and values (predicates and objects) as features of that resource that can be used for search and similarity estimation, much as the words in a document are used in text search. Put another way, we can think of associating to each item a "virtual document" containing "words" such as "author:yc1yb87Karger" and "Send-Date:012937" (note that URIs are kept in the terms in order to differentiate values that are lexicographically identical but semantically distinct). We can apply all the well-studied techniques of fuzzy text search to those virtual documents.

For example, given any item, we can define "similar items" to be those which share many of the same attribute values. These may well be worth displaying when we are looking at an item, as they will likely assist orienteering by the user. Text search research suggests various *term weighting* approaches to decide which attributes are "important" in deciding similarity—for example, extremely common attributes should likely be ignored. When it comes to the common search process of issuing queries, browsing the results, and modifying the query, the text search community has also developed various *query refinement* and *relevance feedback* techniques that can be used to suggest next steps. It is just such suggestions that are presented in the left pane of Figure 1.

7.3 Database Search

We also offer a general purpose "find" interface that lets people design a database query against the RDF model. At present it is limited to expressing constraints that specific predicates must take on certain values. We have invested relatively little effort in this interface, because we see the need to express a query in this way as a sign of failure of the more lightweight navigation tools. Instead of a generic query interface, we expect that specific useful queries will likely be packaged up by developers as operations (discussed above) that use domain-specific dialogs to capture the information necessary to formulate the query.

8 Discussion

Having presented the Haystack system, we now turn to a discussion of some of our design choices and of some of the open questions that we continue to examine.

8.1 Why a Semantic Network?

Our discussion of Haystack may lead one to ask why we use a database or structured model at all. The user sees almost no sign of the underlying database: tuples are never shown, and database querying is deprecated. One might think, given our focus on link traversal, we would be better off simply storing user information as HTML in some kind of "personal web."

On the contrary, we argue that a semistructured data model is absolutely critical to the design of a personalizable information management system. Much of the data users work with clearly *is* structured, relying heavily on properties and relationships to other items. Unlike the web, in which each link must be manually labeled with a textual description of its role, a structured model gives a concise machine-readable way to indicate that role played by a certain class of links. Our view-rendering architecture can make use of that structure to render information objects in a variety of informative ways. And the representation of links in machine-readable form means that, even if complex database queries are beyond the capabilities of end users to construct, power users can package up complex database queries (as operations) and information presentations (as views and lenses) that can then be incorporated by typical users to increase the capabilities of their system. Even more generally, the structure available in the model makes it possible to write various autonomous agents that can import and manipulate data on behalf of the end user.

8.2 Semantic Networks are Universal

We also argue that a semantic network, and RDF in particular, offers a natural "universal data model" that should be widely adopted in the development of applications. The semantic network is rich enough to represent a tremendous portion of the information that users need to work with. At the same time, it is simple enough to be incorporated into any application design with very little effort. All that is needed is a mechanism for naming individual objects, and a representation for specific relationships connecting those named objects.

With such a representation, applications can easily make use of data created by other applications, even if they understand nothing else about those applications' semantics. An object name is enough to let an application create a live link to the object in another application. A relation connecting two objects can be exploited by an application without much understanding of the meaning of that link (as is the case on the web). Invariably, applications will hold some information that is too “complicated” to expose into the semantic network—the individual pixels of an image, or the fuzzy classification scheme of some complex data filer—but these can easily hide inside the individual information objects named in the network, and be handled by applications that understand the internals of those objects. Meanwhile, search tools can let users query and browse on the metadata represented by the semantic network without understanding the semantics of those relationships or the information objects to which they relate. Much like text or files, relations are universal enough to be worth giving a standard representation, so that cross-application tools (like clipboards and desktop search engines) can help to reduce the problems of data fragmentation across applications.

It is also worth noting that much of what each application does is straightforward manipulations of relations and attributes. Nearly every application offers some sort of “collection” framework, with the same drag and drop interactions for moving items among collections. Many offer “annotations”—customized fields that can be filled in with arbitrary text—again using similar interfaces. Offering these capabilities application by application is a waste of developer effort, and also means that they cannot be used across applications. Given the essential simplicity of the intended data model manipulations, there is good reason to expose it at a system-wide level, much as the manipulation of files (and ASCII text) is exposed in existing systems.

8.3 The Role of Schemata

While we rely heavily on a structured representation, the same is not obviously true of schemata. We allow the user to to relate arbitrary objects in schema-violating fashion—the author of a document can be a piece of furniture, and the delivery date a person. And we allow users to craft arbitrary new relations to connect objects, without providing any schematic descriptions.

8.3.1 On not using schemata

On the whole, we believe this schema-light approach is necessary in an personal information management system. Given schemata, we must choose whether to enforce them or not. As with developers designing applications, we will invariably find users wanting to record information that will violate our schemata. At that point, we must choose whether to enforce our schemata and forbid users from recording information they consider important, or we must choose to violate our schemas. Although the latter choice makes it challenging for us to architect our system, the former defeats the fundamental goal: to let users record information they need. Mangrove [HED$^+$03] takes a similar tack, arguing that in practice schema will need to be crafted to fit existing data, rather than the reverse.

Of course, one might argue that the user does not know best. Perhaps enforcement can be couched as an educational experience that teaches users how they ought to be structuring their information. We suspect, however, that users are too set in their ways for such an approach to work. Even if an interface can steer users to *record* information the "right" way, we expect users returning to seek that information will look for it the "wrong" way that they original envisioned, and be unable to find it because it was recorded "right." We need to record information the way we expect users to seek it, even if we expect them to seek it incorrectly.

8.3.2 On using schemata

Although we do not envision enforcing schemas, they nonetheless pervade Haystack. For the sake of consistency, we do attempt to steer users towards reasonable information schemata. We expect that the "preexisting conditions" established by the large number of schemata initially distributed with Haystack will lead to users having similar-in-the-large knowledge representations, so that standard views, queries, and operations work with them.

Schemata play a particularly important role in the design of views. In particular, we make heavy use of `Type` assertions to decide on appropriate views and operations; a user with a highly-nonstandard type system will also need a highly nonstandard interface to work with it. The choice of which attributes to display in the view of an object of a given type is also schematic—it expresses an expectation that those attributes will typically be available, and that other attributes will not (or will not be important). Users, when they modify views, are in a sense modifying the schemas associated with the viewed types. A key difference, however, is that the schematic constraints suggested by views are "soft." While a view implies that certain attributes are expected, the lack of one simply results in no information being displayed. We can see this in Figure 1: while the inbox display suggests the need for a sender and date associated with each object, a person can be included in the collection, with the only consequence being some blank fields. Equally important is the fact that multiple views mean that, in a sense, different schemata can be imposed on the same object at different times, depending on the task the user is undertaking.

Although we do not enforce schema, our user interface's manipulation primitives often make very strong suggestions. Schematic annotations about whether a given property is single-valued or multi-valued affect the behavior of drag and drop: dropping on a single-valued field may *replace* the value of the property while dropping on a multi-valued field may incorporate an *additional* value for the property. Again, these suggestions are not rigidly enforced: with sufficient effort, a user can add a second value to a schematically-single-valued attribute. At that point, views which assume single-valuedness may end up displaying only one of the two assigned values nondeterministically. Of course, there is always the opportunity for the user to modify the view to repair this flaw. And database queries, that address the data without the constraints imposed by views, can make full use of the multiple values.

Underlying our use of schemas is the general research question of how to make use of database schemata that are "usually true." We have already discussed ways that usually-true schemata can assist the design of information views. At the programmer level, schemata let the developer write clearer code, as they can avoid complex case analyses for dealing with data. As a simple example, knowing that a given property is always present means one can skip the code needed to deal with its absence. An intriguing question is to what extent usually-true schemata can be used to maintain clear code. At present, Haystack operations are filled with various blocks of code dealing with schema exceptions—for example, an operation that sorts on dates needs to explicitly check whether each date is actually of type date. In other cases, operations fail silently when they encounter unexpected exceptions (arguably this is reasonable behavior, effectively refusing to apply the operation to schema-violating data). One might hope instead to write code in which all schema violations are caught implicitly and branched of to some kind of exception handling block. But this begs the question of describing that exception handling code, and in particular giving clean descriptions of the ways the schema can be violated and the desired defaults to apply when they are.

8.4 Haystack Limitations

Our use of Haystack has highlighted assorted limitations and flaws in the design. A significant one is "UI ambiguity". Given that every object on the screen is alive, it is sometimes difficult for the user interface to guess which object a user is addressing with a given click. Any point on the screen is generally contained in several nestings of hierarchically displayed objects, and when the user clicks it is unclear which level of the nesting they are addressing. For context menus, we resolve this problem by giving the author access to menus for *all* the objects nested at the click point—as can be seen in Figure 1, the context menu offers access to operations on the email sender, on the email message of which that sender is a part, and on the inbox of which the email message is a part. When the user drags and drops an object, we make the heuristic decision to address the "most specific" (lowest in the hierarchy) objects at the click and drop points. This is often correct, but sometimes leads to difficulties. For example, in order to drop an item into a display of a collection, one must carefully seek out a portion of the collection display that is *not* owned by any recursively rendered member of the collection. Much research remains to be done on the best way to disambiguate UI actions.

The power we give users over the data model can also be damaging. Haystack does not offer users much protection to users as they perform operations that could destroy their data. Beyond the users' own data, since the entire interface is described as data, users can corrupt their interfaces in ways that make them impossible to use. For example, users can dissociate views from the data-types they present, and suddenly find themselves unable to view information.

The proper solution to this problem is to develop effective access control (particularly write-control) methods on the data. We have not addresses this critical issue, and pose it as an open problem below.

9 Other Applications

In this section, we speculate on some other roles for the architecture we have created: to let users consume the semistructured data being produced by the Semantic Web effort [BLHL01], and to let individual users contribute to that effort by sharing or publishing some of their own semistructured information.

9.1 The Semantic Web

Whether or not one accepts the need for a semantic network on each user's desktop, semantic networks seem destined to play a critical role in information dissemination as the so called *Semantic Web* [BLHL01] evolves. The web is an extremely rich source of information, but its HTML documents present that information in "human readable" form—i.e., one in which the semantics of the documents are decoded by human beings based on their understanding of human language. Such documents cannot be easily digested by automated agents attempting to extract and exploit information on behalf of users. Thus, momentum is building behind an effort to present information on the web in RDF and XML, forms more amenable to automated use.

One might think that the richer semantics offered by the Semantic Web versus the traditional web could also increase human users' ability retrieve information from it. But at present the opposite is true, because no good interfaces exist for the Semantic Web. On the Semantic Web, data and services are exposed in a semantics-rich machine-readable fashion, but user interfaces for examining that data, when they exist at all, are usually created from centralized assemblies of data and services. For example, with a semantic portal (e.g., SEAL [SMS^{+}01] or Semantic Search [GMM03], database administrators aggregate semantically-classified information together on a centralized server for dissemination to Web users. This helps users access the semantic web through a traditional web browser.

But a web portal interface has the same drawbacks as traditional applications. It seems unlikely that one designer can create an interface that is "just right" for all the distinct individuals who will use it. Also, the design of any one portal has in mind a fixed ontology; arbitrary information arriving from other parts of the Semantic Web ("other applications") cannot be automatically incorporated into views generated by the portal. If some schema is augmented, no portal will be able to present information from the augmented schema until the portal developer modifies his or her display system. Thus, portals take us back to the balkanized information structures we tried to remove with a semantic network model.

On the other hand, if the user's client software could perform this data aggregation and user interface construction on a per-user basis, then we could restore a user's ability to freely navigate over information and services on the Semantic Web. Our view architecture offers just such an opportunity to integrate data at the client end [QK04]. Separate pieces of information about a single resource that used to require navigation through several different Web sites can be merged together onto one screen, and this merging can occur without specialized portal sites or coordination between Web sites/databases. Furthermore, services applicable to some piece of information need not be packaged into the Web page containing that information, nor must information be copied and pasted across Web sites to access services; semantic matching of resources to services (operations) that can consume them can be done by the client and exposed in the form of menus. By crafting and distributing views and operations, users can create and publish new ways of looking at existing information without modifying the original information source.

9.2 Collaboration and Content Creation

Our discussion so far has focused on one user's interaction with their own information (and then the Semantic Web). But we believe that our system can enhance the recording of knowledge by individuals for communal use, as well as the search for and use of that knowledge by broader communities.

One of the tremendous benefits of the World Wide Web is that it dramatically lowered the bar for individuals wishing to share their own knowledge with a broader community. It became possible for any individual, without sophisticated tool support, to record information that could then be located and accessed by others. If the same were done on the Semantic Web, then information recorded by users can be much richer, making it more useful to other individuals (and automated agents) than plain HTML.

Unfortunately, the state of the art tools for authoring Semantic Web information are graph editors that directly expose the information objects as nodes and properties as arcs connecting those nodes [EFS+99, Pie]. Such tools require a far more sophisticated user than do the simple HTML editors that let naive users publish their knowledge to the World Wide Web.

Haystack makes it easy for users to author structured information, which is already represented in the Semantic Web's native RDF format. This lowers the bar for a user who decides to expose some of their "internal use" information to the world at large. Traditionally, someone who read a document and annotated it for their own use would have to do substantial work to convert those annotations (and possibly the document) to HTML to be pushed on the web. With a semantic network representation, the document and annotations are already in the right form for publication onto the Semantic Web, and the user only needs to decide who should have access to them.

Of course, the access-control problem is a difficult one, made harder by the fine granularity of the data model. We need a simple interface letting user's specify which properties and relationships on which objects should be visible to which people.

On the opposite side, when information is being gathered from numerous sources, an individual must start making trust decisions. Again, interfaces must be developed to let users specify which Semantic Web assertions they wish to incorporate as "truth" in their own semantic networks.

Another significant issue that must be tackled when users collaborate is the problem of divergent schemata. If each use is allowed to modify their information representation at will, then it is unlikely that these representations will align when data is exchanged. We hope that this problem can be ameliorated by sharing view prescriptions and operations along with data.

A piece of related work that we should mention here is the REVERE system, and in particular the MANGROVE project [HED+03]. REVERE shares many of Haystack's goals and methods. Like Haystack, REVERE aims to colonize a useful point somewhere between structured and unstructured information. Haystack focuses on helping each individual manage their own information better. For REVERE, in contrast, collaboration is a primary goal. Thus, issues of schema alignment that can pushed to the future for Haystack become primary drivers for the design of REVERE.

10 Related Work

Much recent work has highlighted the problems created by application-centric data management, and proposed ways to stretch or coordinate applications to address the problem. Bellotti at al. [BDHS03] observed that email applications were being used for task management, and showed how to augment an email application's "views" to support this additional task. Ravasio et al. [RSK04] have given evidence of the problems users run into when trying to perform tasks whose data spans multiple applications. In this volume, Kaptelinin and Boardman argue that one must instead take a "workspace centric approach" that brings together the data needed for a task, instead of the data managed by one single application.

There have been several efforts in the past to center information management on the idea of relations. The Presto project [DE00] proposed to do away with static directories as the key organizing framework documents, and instead base location on queries against metadata that was recorded for each file. Lifestreams, discussed in this volume, focused on one piece of metadata above all: the time of last use. These two systems continued to focus on the file as the basic unit of information, however, and emphasized queries rather than linking, display, and browsing.

The ObjectLens system [LM88] is a clear forerunner of many of the idea we explore in Haystack. ObjectLens emphasized the idea of arbitrary information objects drawn from a type hierarchy, with attributes and links to other objects. OVAL [MLF95] was a tool for rapidly creating applications out of "objects, views, agents, and links" similar to the workspace design concept in Haystack.

The WinCuts tool [TMC04] demonstrates an alternative approach to freeing data from applications: it cuts out small windows into applications so that individual pieces of data from those applications can be viewed (near data from other applications) without the clutter of the rest of the application. Because wincuts operates at the pixel level, it is extremely generic—it can snag information from near any application. But this is also its weakness. Since only the pixels of different applications are unified, and not the data, WinCuts creates no additional semantic linkage between the data in multiple applications. Dragging data from one WinCut to another works only if the two underlying applications are already set up to share data.

Several chapters in this volume propose interesting new relations that are worth recording between information objects, or interesting new visualizations of existing or new relationships. Plaisant et al. propose to tag all information objects with a "role" attribute, and to use that attribute to decide whether given objects or operations should be shown by considering in which of their several roles a user is acting at a given moment. Fisher and Nardi propose that information management will be improved by recording and displaying linkages from information objects to the people relevant to those objects. Freeman advocates recording and presenting according to the access time of all of a user's information objects.

Under the current approach to application development, each of those tools must be developed from scratch, and extensive work invested in attaching to and remote controlling existing applications for working with the given data objects. This kind of integration work must be repeated for each new approach. And the work would multiply even further if someone were ambitious enough to try to build an application that incorporated *all* the different approaches. This seems wasteful, given that in each case the core idea is simply to track some additional relations and to create some views that exploit those relations. An infrastructure such as Haystack would make it much simpler to incorporate such new relations and views in a single system as they arise, and invoke each of them at the times that are appropriate, as opposed to crafting new and distinct applications (and convincing users to migrate to them) for each new idea.

11 Conclusion

The Haystack framework demonstrates some of the benefits of managing user information uniformly in a semistructured data model. Its separation of data and presentation lets us knock down the barriers to information manipulation imposed by the current application model. Moving to the new model, however, imposes requirements on the database and user interface layers that current research has not addressed.

References

[AKS99] Eytan Adar, David R. Karger, and Lynn Andrea Stein. Haystack: Per-user information environments. In *Proceedings of the 8th International Conference on Information and Knowledge Management*, pages 413–422, 1999.

[Bak94] Nicholson Baker. Discards. *The New Yorker*, pages 68,81–83, April 1994.

[Bak04] Karun Bakshi. Tools for end-user creation and customization of interfaces for information management tasks. Master's thesis, Massachusetts Institute of Technology, June 2004.

[BDHS03] V. Bellotti, N. Ducheneaut, M. Howard, and I. Smith. Taking email to task: The design and evaluation of a task management centered email tool. In *Proceedings of the ACM CHI Conference on Human Factors in Computing Systems*, volume 5, 2003.

[BLHL01] Tim Berners-Lee, James Hendler, and Ora Lasilla. The semantic web. *Scientific American*, May 2001.

[BMP01] P. Bosc, A. Motro, and G. Pasi. Report on the fourth international conference on flexible query answering systems. *SIGMOD Record*, 30(1), 2001.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *CACM*, 13(6):377–387, 1970.

[DE00] P. Dourish and et al. Edwards, W.K. Extending document management systems with user-specific active properties. *ACM Transactions on Information Systems*, 18(2):140–170, April 2000.

[EFS$^+$99] H. Eriksson, R. Fergerson, Y. Shahar, , and M. Musen. Automatic generation of ontology editors. In *Proceedings of the 12th Banff Knowledge Acquisition Workshop*, 1999.

[GMM03] R. Guha, R. McCool, and E. Miller. Semantic search. In *Proceedings of the World Wide Web Conference*, 2003.

[HED$^+$03] Alon Halevy, Oren Etzioni, AnHai Doan, Zachary Ives, Jayant Madhavan, Luke McDowell, and Igor Tatarinov. Crossing the structure chasm. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR)*, January 2003.

[HK04] Andrew Hogue and David Karger. Wrapper induction for end-user semantic content development. In *Interaction and Design for the Semantic Web Workshop at the* 13^{th} *annual World Wide Web Conference*, New York, NY, 2004.

[HKQ02] David Huynh, David Karger, and Dennis Quan. Haystack: A platform for creating, organizing, and visualizing information using RDF. In *Semantic Web Workshop at WWW2002*, Hawaii, May 2002.

[Hog04] Andrew Hogue. Tree pattern inference and matching for wrapper induction on the world wide web. Master's thesis, M.I.T., May 2004.

[KJ06] David R. Karger and William Jones. Data unification in personal information management. *Communications of the ACM*, January 2006.

[LM88] Kum-Yew Lai and Thomas W. Malone. Objectlens: a spreadsheet for cooperative work. *ACM Transactions on Office Information Systems*, 6(4), 1988.

[MLF95] Thomas W. Malone, Kum-Yew Lai, and Christopher Fry. Experiments with oval: A radically tailorable tool for cooperative work. *ACM Transactions on Information Systems*, 13(2):177–205, April 1995.

[MM03] Frank Manola and Eric Miller. Rdf primer. http://www.w3.org/TR/rdf-primer/, 2003.

[MMK99] I. Muslea, S. Minton, and C. Knoblock. A hierarchical approach to wrapper induction. In *Proceedings of the Third International Conference on Autonomous Agents (Agents '99)*, 1999.

[MvH03] Deborah L. McGuinness and Frank van Harmelen. Owl web ontology language overview. http://www.w3.org/TR/owl-features/, 2003.

[Pie] Emmanuel Pietriga. Isaviz. http://www.w3.org/2001/11/IsaViz/.

[QBHK03] Dennis Quan, Karun Bakshi, David Huynh, and David R. Karger. User interfaces for supporting multiple categorization. In *INTERACT:* 9^{th} *IFIP International Conference on Human Computer Interaction*, Zurich, September 2003. International Federation for Information Processing.

[QBK03] Dennis Quan, Karun Bakshi, and David R. Karger. A unified abstraction for messaging on the semantic web. In *Proceedings of the* 12^{th} *International World Wide Web Conference*, page 231, 2003.

[QHKM03] Dennis Quan, David Huynh, David Karger, and Robert Miller. User interface continuations. In *Proceedings of UIST (User Interface Systems and Technolgies*, 2003.

[QK03] Dennis Quan and David R. Karger. Haystack: A platform for authoring end-user semantic web applications. In *Proceedings of the International Semantic Web Conference*, 2003.

[QK04] Dennis Quan and David R. Karger. How to make a semantic web browser. In *Proceedings of the* 13^{th} *International World Wide Web Conference*, 2004.

[RSK04] Pamela Ravasio, Sissel Guttormsen Sch&#228;r, and Helmut Krueger. In pursuit of desktop evolution: User problems and practices with modern desktop systems. *ACM Trans. Comput.-Hum. Interact.*, 11(2):156–180, 2004.

[Sin03] Vineet Sinha. Dynamically expoloiting available metadata for browsing and information retrieval. Master's thesis, M.I.T., September 2003.

[SK04] Vineet Sinha and David R. Karger. Magnet: Supporting navigation in semistructured data environments. Submitted, 2004.

[SMS$^+$01] N. Stojanovic, A. Maedche, S. Staab, R. Studer, and Y. Sure. SEAL: a framework for developing semantic portals. In *Proceedings of the international conference on Knowledge capture*, October 2001.

[TAAK04] Jaime Teevan, Christine Alvarado, Mark Ackerman, and David R. Karger. The perfect search engine is not enough: A study of orienteering behavior in directed search. In *Proceedings of the ACM CHI Conference on Human Factors in Computing Systems*, 2004. To appear.

[TMC04] D. S. Tan, B. Meyers, and Mary Czerwinski. WinCuts: Manipulating arbitrary window regions for more effective use of screen space. In *Extended Abstracts of Proceedings of ACM Human Factors in Computing Systems CHI 2004*, pages 1525–1528, 2004.

[YSLH03] Ping Yee, Kirsten Swearingen, Kevin Li, and Marti Hearst. Faceted metadata for image search and browsing. In *Proceedings of ACM CHI Conference on Human Factors in Computing*, 2003.

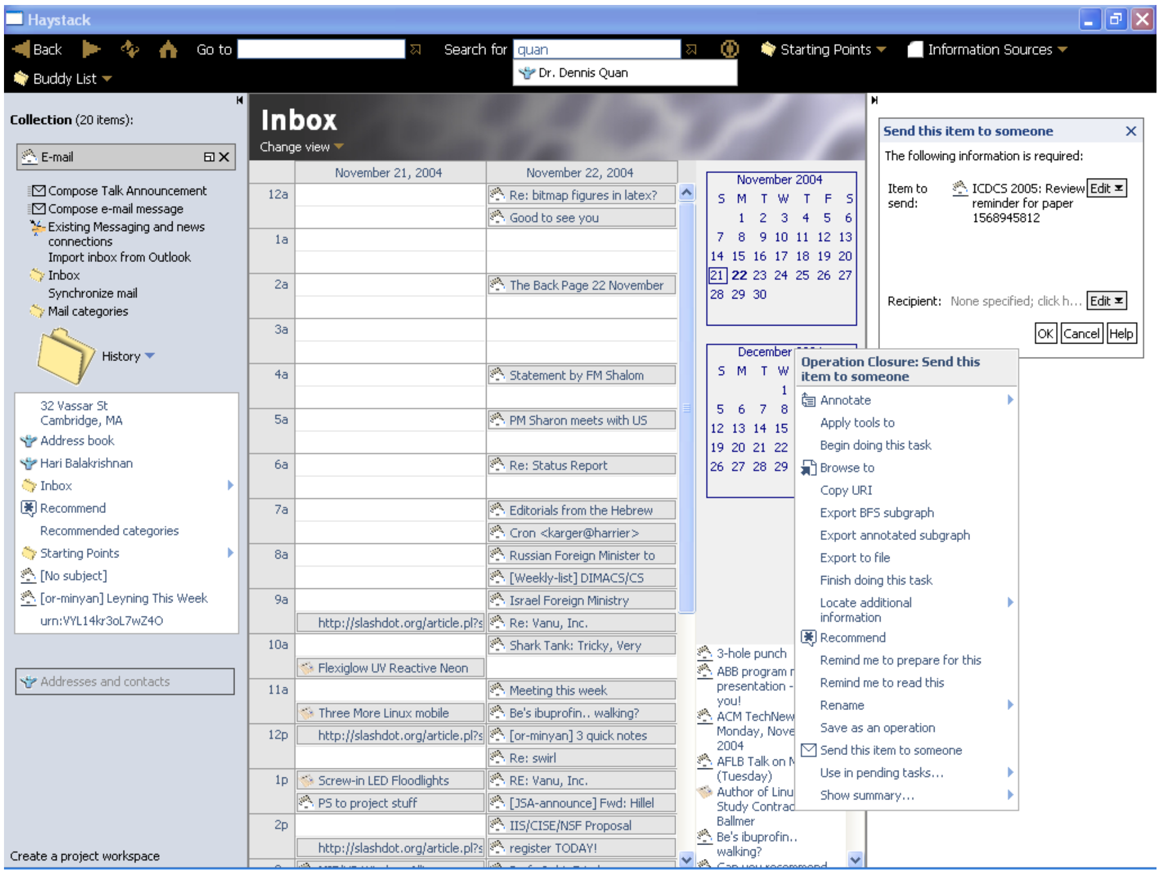
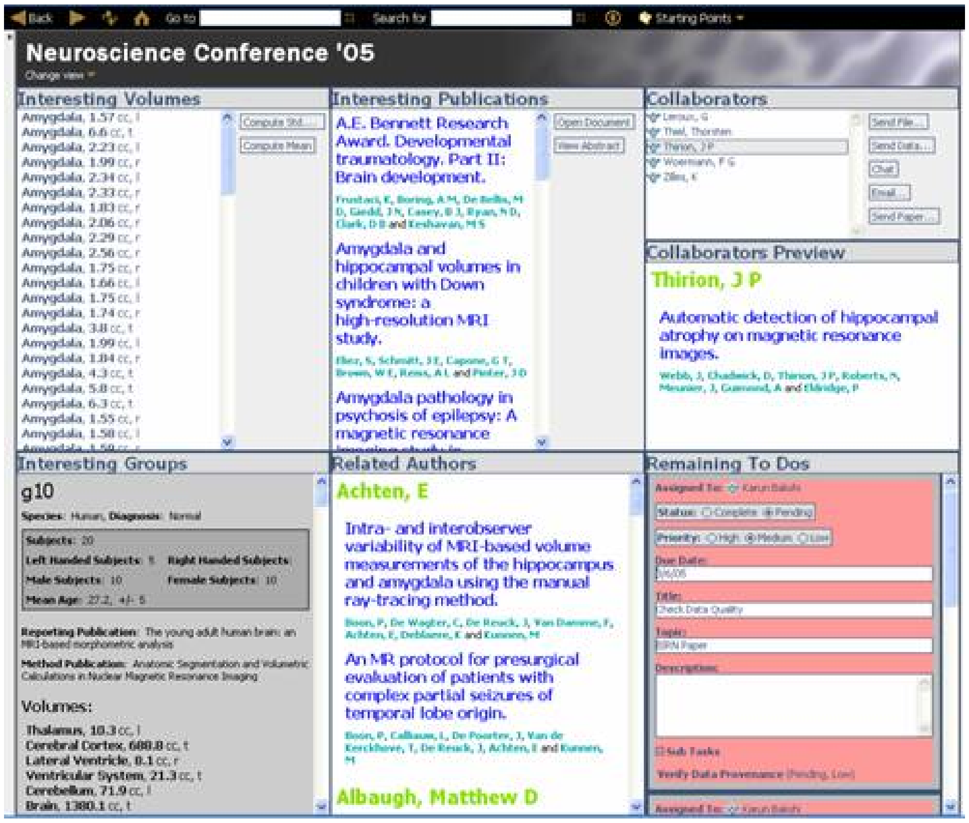


Figure 2: Invoking “send this item to someone” in Haystack. The Inbox collection is displayed in the calendar view. We show three distinct open menus—the task-specific history, the result of a search for “Quan,” and the context menu for an operation—though in actual use only one would remain open at a time.



Neuroscience Conference '05

Interesting Volumes

Amygdala, 1.57 cc, l
Amygdala, 6.6 cc, t
Amygdala, 2.23 cc, l
Amygdala, 1.99 cc, r
Amygdala, 2.34 cc, l
Amygdala, 2.33 cc, r
Amygdala, 1.83 cc, r
Amygdala, 2.06 cc, r
Amygdala, 2.29 cc, r
Amygdala, 2.56 cc, r
Amygdala, 1.75 cc, r
Amygdala, 1.66 cc, l
Amygdala, 1.75 cc, l
Amygdala, 1.74 cc, r
Amygdala, 3.8 cc, t
Amygdala, 1.99 cc, l
Amygdala, 1.84 cc, r
Amygdala, 4.3 cc, t
Amygdala, 5.8 cc, t
Amygdala, 6.3 cc, t
Amygdala, 1.55 cc, r
Amygdala, 1.58 cc, l

Interesting Publications

A.E. Bennett Research Award. Developmental traumatology. Part II: Brain development.

Frustaci, K, Boring, A M, De Bellis, M D, Giedd, J N, Casey, B J, Ryan, N D, Clark, D B and Keshavan, M S

Amygdala and hippocampal volumes in children with Down syndrome: a high-resolution MRI study.

Eliez, S, Schmitt, J E, Capone, G T, Brown, W E, Reiss, A L and Pinter, J D

Amygdala pathology in psychosis of epilepsy: A magnetic resonance

Collaborators

Leroux, G
Thiel, Thorsten
Thirion, J P
Woermann, F G
Zilles, K

Collaborators Preview

Thirion, J P

Automatic detection of hippocampal atrophy on magnetic resonance images.

Webb, J, Chadwick, D, Thirion, J P, Roberts, N, Meunier, J, Guimond, A and Eldridge, P

Interesting Groups

g10

Species: Human, **Diagnosis:** Normal

Subjects: 20
Left Handed Subjects: 5 **Right Handed Subjects:**
Male Subjects: 10 **Female Subjects:** 10
Mean Age: 27.2, +/- 5

Reporting Publication: The young adult human brain: an MRI-based morphometric analysis

Method Publication: Anatomic Segmentation and Volumetric Calculations in Nuclear Magnetic Resonance Imaging

Volumes:

Thalamus, 10.3 cc, l
Cerebral Cortex, 680.8 cc, t
Lateral Ventricle, 8.1 cc, r
Ventricular System, 21.3 cc, t
Cerebellum, 71.9 cc, l
Brain, 1380.1 cc, t

Related Authors

Achten, E

Intra- and interobserver variability of MRI-based volume measurements of the hippocampus and amygdala using the manual ray-tracing method.

Boon, P, De Wagter, C, De Reuck, J, Van Damme, F, Achten, E, Deblaere, K and Kunnen, M

An MR protocol for presurgical evaluation of patients with complex partial seizures of temporal lobe origin.

Boon, P, Calliauw, L, De Poorter, J, Van de Kerckhove, T, De Reuck, J, Achten, E and Kunnen, M

Albaugh, Matthew D

Remaining To Dos

Assigned To: Karun Bakshi
Status: Complete [✓ Pending]
Priority: High [✓ Medium] Low
Due Date: 3/6/05
Title: Check Data Quality
Topic: BIRN Paper
Description:
☐ **Sub Tasks**
Verify Data Provenance (Pending, Low)
Assigned To: Karun Bakshi

Figure 3: A workspace constructed by drag and drop. This workspace is specialized for writing a particular research paper, presenting research data, coauthors, and relevant references.
The publication view was created with the similar view-construction tool.