Using Properties for Uniform Interaction in the Presto Document System

Paul Dourish, W. Keith Edwards, Anthony LaMarca and Michael Salisbury

Computer Science Laboratory
Xerox Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto CA 94304 USA
{dourish,kedwards,lamarca,salisbury}@parc.xerox.com

ABSTRACT

Most document or information management systems rely on hierarchies to organise documents (e.g. files, email messages or web bookmarks). However, the rigid structures of hierarchical schemes do not mesh well with the more fluid nature of everyday document practices. This paper describes Presto, a prototype system that allows users to organise their documents entirely in terms of the properties those documents hold for users. Properties provide a uniform mechanism for managing, coding, searching, retrieving and interacting with documents. We concentrate in particular on the challenges that property-based approaches present and the architecture we have developed to tackle them.

Keywords

Document management, document properties, document interfaces, interaction models.

INTRODUCTION

Much of the time we spend interacting with computer systems is devoted to working with documents: text documents, spreadsheet documents, presentation documents, Web documents, email documents, and so forth. In turn, most of the electronic documents that we work with are stored in hierarchies. Filesystems, for example, present a hierarchical model of directories and subdirectories, while email systems present a hierarchical model of folders and subfolders. The hierarchy is the principal structure that users are offered for information management.

However, although hierarchies are a computationally convenient means to organise large amounts of data, they are often too rigid for everyday document management practices. The problems with hierarchies are borne out by studies of electronic file management. For example, Barreau and Nardi [2] looked at how people managed files on personal computers, and found that people displayed a considerable preference for spatial organisation and informal grouping, rather than the structure offered by hierarchical filing; the hierarchies they observed tended to be very broad and shallow, with little or no cross-linking. In many ways, this echoes the experiences reported by Marshall and Rogers [12] exploring the use of structured hypertext systems, who also found that the rigid structures they offered were poorly suited to the more fluid and informal ways in which people organised information.

Our work explores opportunities for moving away from this rigid approach. This paper describes ongoing research we are conducting into a new approach to document management that aims to provide users with new ways of organising, structuring, managing and interacting with document collections, while none the less retaining the level of integration with file-based platforms that we all need to work in our everyday computational environments.

Placeless Documents

The work reported here has been carried out under the auspices of a project called "Placeless Documents". The starting point for the Placeless Documents project is that conventional approaches put the emphasis on the wrong place, by forcing people to think not in terms of what a document is, but rather where it is stored.

Filesystems, for example, exhibit this focus on document locations rather than on the documents themselves. The path "T:\dourish\papers\presto\uist99\draft.doc" names a place in the filesystem and only incidentally the file that might be stored there. One way to see that this names a place and not a file is that there may not be such a file at all, but the system clearly knows where to look to determine whether or not there is.

For us, the particularly interesting thing about filenames such as these is the way we use them to encode information that is relevant to us about the documents they locate. Each of the elements of the path indicates something about the document – that it belongs to the user "dourish", that it's a paper, that it's about the Presto project, that it's for UIST'99, that it's a draft, and that it's a Microsoft Word file (the ".doc" extension). In fact, even the fact that it's stored on the "T:" drive is significant; in the PARC environment, the "T:" drive is a network file system that is regularly backed up; so if the document is stored there, rather than on the un-archived "C:" drive, then the author must feel that it is important enough to warrant backing it up.

In other words, documents and document collections have a variety of properties, relevant to document users. In the

Placeless Documents project, it is not locations but properties that play the central role in our designs.

We envision a system in which users can manipulate and exploit document properties directly. We make a strong separation between document *content* (the bits that make up the document itself), document *properties* (elements that users associate with the document), and document *storage* (an external repository where the content resides).

This approach offers a number of benefits:

1. Document properties are directly associated with documents, rather than with document storage locations. This means that documents will retain properties even when moved from one place to another, and that property assignment can have a fine granularity.
2. A document can have any number of properties associated with it, and these properties are unordered. Documents can be organised without the limitations of hierarchies ("should I create a 'Presto' sub-folder in my 'papers' folder, or a 'papers' sub-folder in my 'Presto' folder?")
3. Properties can be specific to individual document consumers. When you interact with a document in terms of its properties, those could be properties relevant for *you* rather than for the original author of the document.

Our vision is of a system that allows users to organise their documents according to the properties of those documents. Document properties are "things you already know about your documents"; they directly relate documents to their use. Properties can manage document interactions not simply in the filesystem, but in any document application.

This paper discusses recent research we have been conducting into these issues. We introduce and outline our approach, and describe Presto, an early embodiment of these ideas that we have been exploring. In particular, our focus in this paper is on the challenges that property-based interaction presents, and the approaches we have taken to meet them. We will begin by describing the basic elements of our approach, the design of Presto. We will then consider the interactions Presto supports with other system components, and with users, before discussing some early uses of Presto supporting other projects at PARC, and the lessons we have learned.

DOCUMENT PROPERTIES

If our goal is to provide a document system organised around document properties, then it seems reasonable to ask, what *are* document properties, and where do they come from?

In Presto, properties are name/value pairs. For instance, "author=kedwards" is a property whose name is "author" and value is "kedwards." Any number of document features could be expressed through name/value pairs. Values can be arbitrary objects. Of course, existing systems already provide some notion of document properties which might operate in this way, such as the document length or author, the last modified time or the storage format. In our work, we want to generalise this restricted notion of properties, and allow users to express arbitrary features of the document that are uniquely relevant to them.

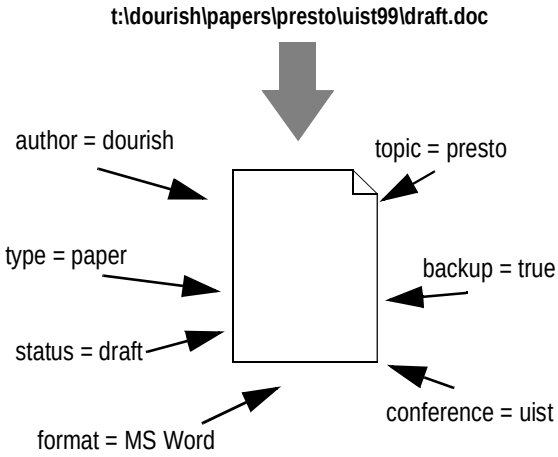

FIGURE 1: In our systems, documents are organised according to their properties, rather than according to their locations.

Although properties are designed to be meaningful to users, they can also be exploited by system components. For instance, a property that indicates that a document is important might signal to a backup service that this document is a candidate for off-line storage. Similarly, a property that indicates that a document is currently being worked on might signal to a replication service that an up-to-date copy should be kept on my laptop so I can work on it when I take a trip. In other words, properties can act as a point of coordination between the user level and the system level. The primary level of interaction, however, is the user level.

We can take this approach to user/system interaction one stage further. Since our systems may exploit document properties as configuration mechanisms, they should also be able to use document properties as a way of recording their own processing on documents. So, any application that wants to store data associated with a document can store it as a property attached to that document.[1] In this way, information which would otherwise be locked inside specific applications can be reflected in the same space as all other relevant document information. In turn, this makes all document information viewable, browsable and searchable through a uniform property-based mechanism and a single property space.

Properties for Document Consumers

We have already made the observation that document properties are oriented towards the document consumer, rather than the document creator or author. However, most documents have more users than creators, and clearly those users may have different relationships to the document. This implies that different users may want to attach different properties to a document.

1. Although we will typically use text string values in our examples, property values in Presto can be arbitrary data objects, so applications can store structured data, links to other documents, or even runnable code in document properties.

Our document model supports the idea of "document references," which appear as normal documents in our system, but actually refer to documents stored elsewhere, and under the control of some other user. Each document reference encapsulates its own set of properties, as shown in figure 2. In this example, Paul has created a document (called the "base document"), to which are attached a set of properties. Two other users, Mike and Anthony, have document references, which appear to them like documents, but which are actually pointers to Paul's base document. Mike and Anthony have their own properties associated with the document; but these are private to them, and do not interfere with each other. Mike may have marked the document as "interesting", while Anthony has marked it as "prior work" (for a project he is working on); since Anthony has not marked it as "interesting", it should not appear to display this property, or be returned from queries for interesting documents. However, both Mike and Anthony will see the properties directly attached to the base document (the "base properties"); these are properties relevant to everyone, such as the size of the document. Our approach allows users to query each other's spaces, so that they can essentially ask "show me the documents that Mike thinks are interesting"—a form of collaborative filtering [9].

A PROPERTY-BASED DOCUMENT INFRASTRUCTURE

Our prototype, Presto, is an early embodiment of these ideas. We had two principal goals in developing Presto. The first goal was to develop a platform for everyday document management organised entirely in terms of properties. By developing this platform, we wanted to explore, first, whether the approach was practical; second, what the interactional consequences might be of a property-based approach; and third, how such a model could be integrated with existing document applications. These issues will occupy us for most of this paper. The second goal was to use this platform to work with potential customers to elicit requirements for a larger design effort around a richer property model. In the later parts of the paper, we will discuss some of these efforts and the lessons learned from them.

Properties and Documents

Although Presto's basic entities are documents, it does not store document content. Instead, Presto makes use of external document repositories such as file systems or the World Wide Web. Presto itself stores just the document properties.

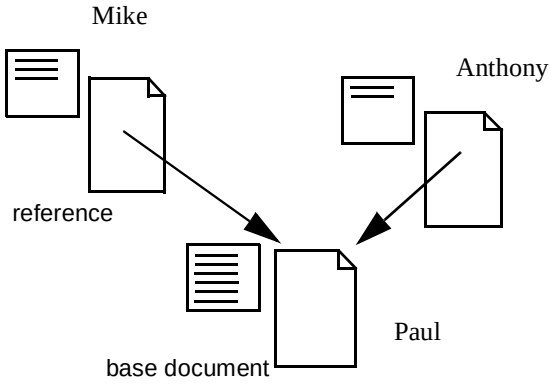

FIGURE 2: Documents can contain references to other documents, allowing each user a separate set of properties for the same base document.

This separation of document properties and document storage provides a number of advantages. First, it means that properties constitute a uniform document management paradigm across disparate substrates. I can store, search and manipulate web documents and file system documents with precisely the same mechanisms (and, indeed, in everyday interaction, I may not know or care which are which). Second, it integrates those various substrates, allowing documents to be managed collectively even though they may "live" in separate repositories. Third, it provides us an opportunity to separate the notion of "document" from the notion of "file". A document need not correspond exactly to a single file; it may integrate the content of multiple files, refer to only a segment of a file, or not correspond to any real stored content at all.

Since users may have different meanings for properties, their values are untyped. Coordination is achieved simply through a hierarchical property naming scheme.

Queries

All documents respond to a common interface for attaching properties, reading their value, and removing them. The document space can be queried according to property values. Since document queries are the primary means by which users interact with document spaces, we require that query performance be good. On a small database (342 documents, 4911 properties), the query "Mail.From=dourish" takes 30ms to return 8 documents, while "Type=text/html or Type=text/java and read within 1 month" takes 140ms to return 32 documents. On a larger database (2558 documents and 27921 properties) the same queries take 90ms (8 documents) and 620ms (300 documents) respectively.[2] This is fast enough to achieve our primary goal, a feeling of directly manipulating the document space, rather than the usual "submit a query, await the results" model.

Properties and Collections

Properties apply to documents, but we also need a way of grouping documents and acting on them collectively. Presto provides *collections* for this purpose. Unlike queries, which are transient searches over the document space, collections are persistent entities that group documents. At the same time, collections are also a type of document, and so all core document operations apply to them. They can have properties attached, and can themselves be added to collections.

Collections in Presto play roughly the same role that folders and directories do in other systems. There are some significant differences, though. The two primary ones are that, first, documents can appear in multiple collections (or none at all), and, second, that collection membership can be organised dynamically according to document properties.

Since Presto document content might be stored in various possible repositories, such as filesystems or the Web, this means that collections in fact integrate content from different

2. These times are the mean of 15 trials on a 200MHz Pentium Pro.

stores. A single collection can mix documents from different locations. So, the logical organisation that collections offer is not limited by the fact that document content may actually live in different places; I can keep all my project documents together wherever they are stored.

The goal of the collection design was to allow users to organise their document space in terms of properties. Since the set of properties defined on documents and the values of those properties are subject to change continually, it follows that the membership of collections is also dynamic. Dynamic collections are very powerful, but they present difficulties in interactive systems. When a collection's members are being continually recalculated, it is hard to interact with the collection; documents appear and disappear under your feet. When developing Presto, we needed a richer collection design to support stable interaction.

Fluid Collections

One solution to these problems would be to allow manipulations of the collection members to indirectly manipulate the query. While this approach may be suitable for some applications, it also has a number of problems. First, it is hard to give a coherent account to end-users of *how* their document manipulations will result in transformations of the query.[3] Second, the query itself becomes more complicated as exceptions and additions are made to it. We want to be able to maintain the coherence of the query. Further manipulations express features of the collection, not of the query.

As an alternative, we adopted a design we call "fluid collections." Collection membership is defined by three components: the *query*, the *inclusion list* and the *exclusion list*. The query component is a query in the Presto query language that defines a set of documents according to their properties; all documents matching the query are members of the collection. For example, I might have a collection titled "Recently Used Documents" with the query component "used within 5 minutes – collection" (all documents read or written within the last five minutes, excluding collections). All documents matching the query at any given moment will be members of the collection. Similarly, I can define a collection to be my electronic mail "inbox" by setting its query component to be "mail.status=unread"; as I read messages, they automatically disappear from the collection since the query no longer applies.

The two other components of fluid collections serve to modify the query component. The inclusion list contains a set of documents that should be included in the collection *even if they do not* match the query. In turn, the exclusion list contains a set of documents that should be excluded from the collection *even if they do* match the query. The result is that fluid collections allow static modifications to the dynamic query, preserving stability for user interactions. When I look at a collection whose members are given by a dynamic query, I can still manipulate the collection—drag new documents into it (adding them to the inclusion list) and dragging others out (adding them to the exclusion list)—and even though the query is still "live", my changes will be preserved.

Each of the three query components can be null. In the case where the inclusion and exclusion lists are null, the collection behaves exactly like a normal query. In the case where the query component is null, the static lists serve to define the membership of the collection, and so it behaves just like a folder in that it contains only and exactly what the user has put there[4]. So, fluid collections can simulate the behaviour of purely static and purely dynamic collections as well as a middle ground which is dynamic but stable.

Properties and Services

Since properties are the primary mechanism for all document interactions in Presto, it follows that the power of the system depends on the range of properties attached to documents. Our model is that "there's no such thing as too many properties"; any property that a user might need to organise or retrieve their documents should already be there if we can put it there. However, although we allow users to attach arbitrary properties to documents, representing whatever characteristics of documents they deem relevant, the system would be cumbersome if the system stored only properties that had been explicitly added. In Presto, we address this by allowing system components to add properties too. These components can automatically derive document features to be recorded in property space. We call these *services*.

A service is an application which runs over the document space and adds new properties to documents. Services typically deal with particular document types and exploit local knowledge of the structure of documents of that type. For instance, an email service operates over documents containing email messages; it can read the contents of the document, parse the mail headers and annotate the document with properties describing who the message is from, what is the subject, and so on. Similarly, an HTML service processes HTML files and annotates the document with properties indicating such features as the title, included images and linked documents, while a Java service parses Java source files and uses document properties to describe the packages imported and the methods defined.

In general, then, what services do is to exploit the semantics of particular document types in order to reflect document *content* in the document *properties*. Services are the only components of the system that need to be aware of the details of document format and structure; by making aspects of content available as properties, they enable a uniform mechanism for querying and organising the document space for users. Since everything is encoded as properties, users can exploit properties as a single mechanism for all their interactions, and can search over all aspects of documents at once. So, for example, looking for a document that arrived by email, I can issue a query that looks not only at the email properties (such as header information), but also at aspects of its structure (e.g. heading text) and content (e.g. summary or

3. One could imagine using graphical query mechanisms (e.g. [10]), but they lack the directness that we were trying to achieve.

4. In practice, this means just the inclusion list, since removing a document from a collection will first remove it from the inclusion list if it is present there, before trying to add it to the exclusion list.

keywords). This sort of mixed-application query approach is not available using other mechanisms.

APPLICATION INTERFACES

What we have described so far is simply an infrastructure on which property-based document applications can be built. Our goals for Presto imply a number of requirements for application interfaces. First, we want to develop new applications that can exploit the novel features of property-based document management. Second, we wanted to be able to reuse existing component software wherever possible. Finally, we also need to provide compatibility with "legacy" applications such as Microsoft Word, without modification. Presto offers three levels of interface for these different needs.

Supporting Custom Applications

There are two interfaces that we use to develop custom Presto applications.

First, the Presto object model, structured in terms of document objects, properties, queries and collections, is offered to Java programmers as a set of classes they can use in their own Java programs. This is the primary mechanism for building new applications that exploit Presto's novel features.[5] Our browsers, for example, are built to this interface, and the applications we will describe later are custom Java systems built on top of the Presto code base in this way.

Second, we support the Java IOStreams interface which is used by Java Beans components. This allows us to incorporate Beans to view and modify document content. For instance, our current browsers include third-party Bean components for processing image, text and HTML files which were directly incorporated using this interface. These components, then, can be seamlessly integrated, but only within the context of new applications. This makes it easy to incorporate new data types, but does not address the legacy application problem.

Supporting Legacy Applications

Since people already have a significant investment in their documents and document tools, we need to be able to support these in Presto. However, legacy applications like Word do not understand how Presto manages document content and properties, and extending it to do so is not feasible, especially when we consider the wide range of applications that might have to be modified in this way. Instead, we take a different approach.

The common feature of these legacy applications is that they expect to run on a filesystem. Our solution is, essentially, to give them one. Presto exports access to its documents using the standard Networked File System (NFS) protocol. A Presto data store can be mounted as a new networked disk from the perspective of the client machine; filesystem reads and writes will act on the Presto documents indirectly. Word need never know that it's talking to Presto.

5. Presto is implemented 100% in Java. It is based on Java 1.1, and uses JDBC for database interactions, and Swing for its user interfaces. We run it on both Windows and Solaris platforms.

Our NFS protocol is implemented as a Presto application. It uses a Java RPC implementation and offers the NFS RPC protocol, which it then translates into requests to the Presto object model. However, this "translation" is not entirely straightforward, since the semantics offered by Presto are not the same as those of a conventional filesystem. In actual use, applications depend on a number of invariant properties of filesystems that do not hold in Presto, and so our implementation needs to work around these. We will discuss four here.

1. In a conventional filesystem, there is a top-level directory. Clearly, this is not true in Presto. The NFS server maintains a "virtual" top level directory, which is a query for documents with the property "Root=true".
2. In a conventional filesystem, all files are reachable along some path from the top level. Since Presto documents need not appear in any collection at all, this is not true for us either. To solve this, the NFS server recognises a special syntax in directory names as indicating a query. So, for example, the path "/#type=html" names a directory containing the results of a query for all documents with property "type=html". Since each document can be retrieved by a query using its unique document ID, this means that any document can be named in the filespace. (This is similar to Semantic File Systems [8]).
3. In a conventional filesystem, the combination of directory and filename uniquely identifies a file. In Presto, however, documents have an intrinsic identity regardless of where they appear and what properties they hold. This becomes relevant when an application updates a file by renaming the old document and then writing a new one with the same name in its place. In most filesystems, this replaces the old file; in Presto, it creates a new one. The NFS server recognises this sequence of action and restores the original document (and hence, the original properties) with the content of the new one.
4. In a conventional filesystem, two files in the same directory will always appear together. This means that an application can safely store related information alongside (i.e. in the same directory as) an original file and expect to find it in the same directory later. However, in Presto, a document can appear in multiple collections, and so the related information will not always appear in the same place. If a document appears in the "important" collection, for instance, that does not mean that the spelling dictionary is also "important". Our NFS server detects attempts to create related files of this sort, and notes the relationship as a property on the secondary document. When the application is run in future, the server searches for all related documents and temporarily reinstantiates them.

So, since the goal of the NFS server is to support the use of legacy filesystem-based applications, it is not enough simply to support what filesystems do; we must also support the way that filesystems are used. To this end, techniques like these can preserve the invariants upon which traditional applications depend, and so integrate them into a Presto-based document management system.

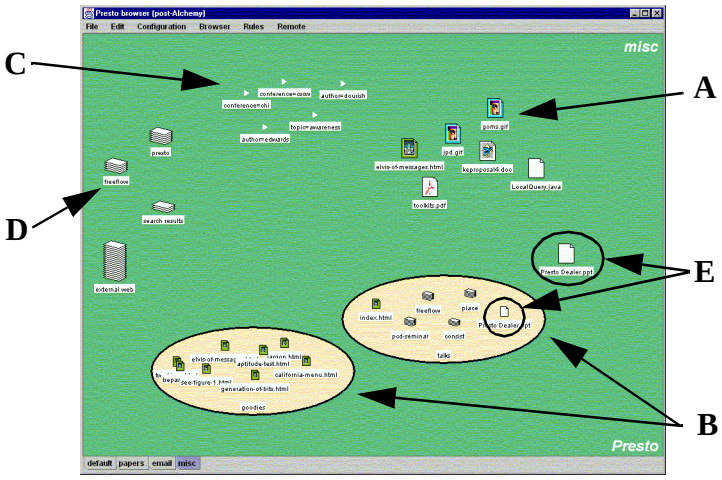

FIGURE 3: A snapshot of Vista, a Presto interface. Vista offers multiple Rooms-like workspaces onto the same document database, and displays documents (A), collections (B) and properties (C). Closed collections are shown as piles (D), giving cues as to their current size. Note that the nature of the system implies that a document may actually appear in multiple places at once (E).

INTERACTION CHALLENGES

A property-based document infrastructure such as that embodied by Presto presents interesting challenges for interaction design. On the one hand, we must exploit the familiar models of document interaction which are presented by traditional interface designs; and on the other, we want to be able to capitalise on the novel features provided by a property-based infrastructure. In the interfaces we have developed, we have encountered a number of these challenges. In this section, we will discuss some of these with particular reference to Vista, a desktop-like workspace browser for Presto.

One Document, Many Icons

One interesting way in which a property-based document system differs from a filesystem is that a document may appear in a number of different collections at once. It follows that in an interface to the document space, any given document may appear multiple times on the same screen. For instance, suppose my window contains two collections, each of which has a dynamic query. The first dynamically contains all documents related to the Presto project, while the other dynamically contains all documents containing purchase orders. A document detailing the purchase of a server for Presto will necessarily appear in both of these collections (and quite possibly others too). In a Presto browser, this situation arises naturally, and so must be accommodated by the interface.

In Vista, we allow multiple documents to appear in a workspace, but avoid the situation where two documents appear in the same *context*—that is, a document cannot appear more than once in any given collection, or more than once on the desktop. If the user attempts to move a document into a context where it already appears, then the "second" appearance will merge with the first when the user releases the mouse.

Controlling Dynamic Queries

Earlier, we discussed the way that document collections embody queries over the document space. Queries are typically constructed in a textual query language, but we wanted to provide users with a means for direct manipulation of queries, and the sense of immediacy that engenders. One solution to this is to use property objects not simply as entities which can be assigned to documents, but also as terms from which queries can be constructed.

One implementation works as follows. When a user creates a new collection object, it is initially completely empty. At this point, documents can be dragged onto it; they will be added to the inclusion list and appear in the collection, and in this way the collection behaves like a traditional desktop folder. However, if a property term is dragged onto the collection, then it becomes a query term for that folder. So, if I drag the property "project=presto" onto an open collection, then the term "project=presto" is set as the query term for that collection, and documents matching the query will appear inside the collection. This process can be repeated, with any number of query terms added to or removed from collection query; the collection shows the current query terms around its edge (figure 4).

As new documents match the query, they are immediately displayed in all matching collection windows. The inclusion and exclusion lists described earlier provide a way to perform stable manipulations to live query sets. Rather than complicate the interface by exposing the three-part structure

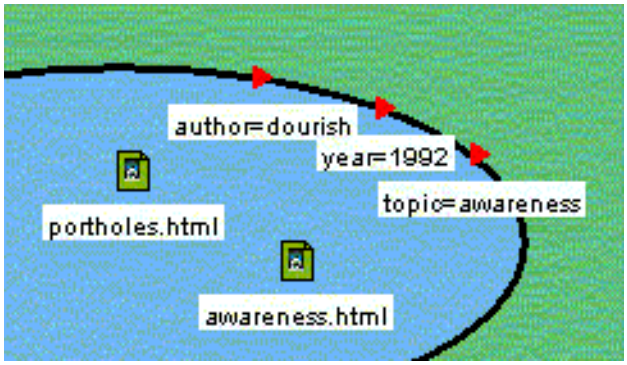

FIGURE 4: Properties can also be used as query terms on collections.

of collections, we support manipulation of the inclusion and exclusion lists through direct interaction with the collection itself. Dragging a document out of a live collection adds it to the exclusion list, while dragging a document into the collection adds it to the exclusion list. To make this manageable, these operations are also reversible; if a document has been dragged out of the collection, the dragging it back in will do the right thing (in this case, remove it from the exclusion list again). The user gets the sense of these queries as stable, manipulable collections even though they are realised by dynamic queries subject to continual re-evaluation.

This style of direct interaction with live queries supports our goal of providing users with the feeling of a directly manipulable document space. This is in contrast with traditional "query/response" systems, and more like the UMD work on "Dynamic Queries" [1] although it's not tailored to specific domains. We are considering other techniques for direct exploration of the document space, such as Magic Lenses [3].

The Vista browser, then, embodies some specific techniques that we incorporate in order to create a fluid interactive style for property-based document management. However, we can also use the Presto infrastructure to develop special-purpose applications that incorporate more detailed understandings of particular application domains. With this in mind, we have been working with various colleagues at PARC to develop other applications that explore the use of Presto as an application infrastructure.

APPLICATIONS

We have been working with other PARC research groups to build applications on top of Presto. Our goals are to evaluate our system and approach, and to derive requirements for further design efforts.

One interesting feature of Presto as an application infrastructure is that it can essentially serve two roles. First, Presto is a *document management system*, providing access to document repositories and document content stored there. Second, Presto is a simple and lightweight *object system*, providing a coherent model for storing and retrieving object attribute information. A number of Presto applications, including some internal parts of the Presto infrastructure itself, make use of the Presto property mechanism in this way, creating "content-less" documents whose significance resides entirely in the properties defined on them.

We will discuss two examples here. The first makes use of Presto as an object store supporting the implementation of novel interaction techniques. The second explores the ways in which users can control and share structures describing the space of possible property values.

Magic Office

The Magic Office project at PARC is working on tools that integrate the work we do in both the physical world and our computer-mediated virtual worlds. One of its first manifestations is a computer-augmented whiteboard called Flatland [13]. Flatland provides an environment in which the whiteboard is both an input and an output device. The system detects users' strokes on the whiteboard and can generate new output overlaying or replacing user input. Flatland aims to support the range of work people do on office whiteboards everyday; this is in contrast to systems like Tivoli [14] which are directed towards domain-specific applications such as meeting support.

This focus on the "everyday content" of the workplace has a number of implications for the underpinning infrastructure. First, there is a need for persistence. Flatland is envisioned as a constantly-available tool in the workplace, which can remember everything that has taken place on it since it was installed. Users of the system need to be able to return to past states of the whiteboard, search for specific pieces of prior content based on the context of their use, and so on. Since this requires a very fast interface to users for searching based on time and context, the infrastructure on which is built must make these operations not only possible but fast.

Second, Flatland needs to be able to store whiteboard information in an extremely light-weight way. Most of the scribblings on a whiteboard have value precisely because they are easy to create and easy to retrieve, so requiring users to drive a "Save as..." dialog would be totally inappropriate.

Third, Flatland needs a way of storing arbitrary bits of application state along with the actual whiteboard contents. In the Flatland architecture, the interpretation of strokes is divorced from the representation of those strokes on the board. Parcels of application functionality (called "Behaviours") can be dynamically associated with regions of content, lending their own interpretation and semantics to regions on the board. These behaviors need a convenient and quick way to associate application-specific data—which may be meaningful only to them—with arbitrary content items. Flatland meets these goals by using Presto as its storage layer.

Flatland creates a separate Presto document for each region of the board (called a segment). Each document contains a set of tokens representing the complete history of the segment associated with it. Histories can be played forward or backwards to "reset" the segment to any state in its history. Such a representation—storing the explicit history of an on-screen entity in a document's contents—could just as easily be done in an existing filesystem. The advantage of using Presto in such a setting comes about from the ease of integrating meta-information about the history into the document. Using Presto, for instance, we can store "markers" that index into interesting points in the history as properties on the document. Multiple markers can coexist at the same time, and can be updated independently (and quickly—simply storing the markers in the content, say at the first of the document, would necessitate rewriting the entire file when the markers changed).

The property system also enhances our ability to search through document content. Whiteboard segments containing text are passed through an off-line handwriting recogniser. This recogniser attempts to identify text in the segment and then annotates the content document with a property containing recognised words. The resulting keywords property can then be searched quickly without the need to retrieve or parse the actual stroke-based contents of the segment.

In the Flatland implementation, properties are used to build interconnections between segments, to provide indices into the representation of history stored in the content, and to permit extremely quick search over elements that may be computationally expensive to derive from the original content. They are also used to maintain information about the context of use of segments–where the segment was on the board, who was involved in its creation, and so on, allowing all of these contextual elements to be searched directly.

One of the most basic advantages of using Presto is the ability to create extremely light-weight documents, without the need to file or even name them. Presto supports the ability to create unnamed documents. (In Presto, the "name" of a document is simply an arbitrary, optional property.) These documents need not exist in any collections, but can be retrieved based on their properties. So any new stroke on the board can potentially result in the creation of a new, unnamed and unfiled, document. The metaphor used by some on the Flatland team is that of a "soup" of documents, strongly interconnected via properties.

Finally, the dynamic behavior mechanisms used by Flatland are very amenable to implementation atop Presto. One of the design challenges of Flatland was to build a system in which arbitrary bits of application code could be dynamically associated with a segment, grouped, removed, and reapplied. These code bits need a flexible way to store their state, associate arbitrary state with arbitrary strokes, and rebuild their state if they are reapplied to a segment. Presto's hierarchical property namespace allows application behaviors to "own" a chunk of the property namespace, and use it for their own needs without worry of collision with other behaviors. Since Presto allows arbitrary Java objects to be stored as the values of properties, the infrastructure essentially is treated as a lightweight object model for Flatland; each segment which represents on-screen content has associated with it arbitrary application data in named slots on the object.

Document Category Management

In another project, we are working with a group of ethnographers studying the document management practices of a project group in a California state organisation [6, 17]. This organisation has an extensive set of procedures for managing the files for any given project, which can extend to many shelves of documents packed into folders. For consistency across projects, they employ a "Universal Filing System" (UFS) to categorise project files. In practice, the use of the UFS is more complex than it might appear. Since the UFS is updated infrequently, and is a very general device, the people working with the files frequently find that they need to file documents in categories not supported by the UFS, or that they need a greater degree of specialisation than the UFS can provide.

We have been investigating the development of technology to support this organisation's document practices. We are particularly concerned with how the system can exploit the filing categories fluidly, supporting the perspectives of different users with different local category structures, and allowing users to share their customisations of the UFS.

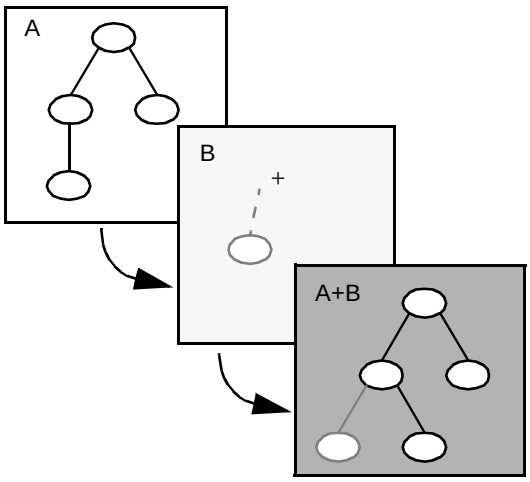

FIGURE 5: Layered definitions of the structure of a property value space. The hierarchical structure that results is composed of a sequence of layered modifications.

There are two challenges in using Presto to support these document practices. The first results from the fact that, in designing Presto, we deliberately eschewed any attempts to control the possible values of properties. Presto leaves the semantics of properties entirely in the users hands, introducing no restrictions. The UFS, however, introduces a structure into the space of values for relevant document properties. We want to be able to exploit the structure of this space, especially in queries; for example, a query for documents whose source is a government agency should correctly match those documents whose source property is any value of type "government agency".

Second, since we want to allow specialisation and refinement of these value space hierarchies, the structures that we introduce need to be reflected back into the Presto object space. As local customisations of the UFS are introduced, we want to be able to provide the means to share these within the group. Since Presto manages only documents, this implies that the value space structure introduced above must be concretely realised in the document space so that it can be shared with others, and further that this encoding be able to capture the multiple levels of customisation to be able to support this collaborative, shared refinement of the basic mechanism. Presto already supports the idea that different users have different views of documents; an explicit value space representation gives us a way to relate those different views.

As a normal document system, Presto already offers the ability to store the organisation's project documents, and record their properties. The challenge is to use Presto's "flat" property model to reflect the structured properties values of the UFS. Reflecting the duality of the Presto system, as both document store and object system, we can take a two-pronged approach to the problem. First, we create an encoding of the category system as properties and document objects. Each category in the UFS is represented by a minimal (no content) document object in Presto, and properties on those documents define a pattern of relationships between them. So, using links encoded in properties, we can create a tree of values represented as Presto documents. Once we

have this structure, we can use it to manage the space of UFS properties; instead of setting a document property to be the string "Highways Department", we can set it to be the document representing the "Highways Department" value, and so link it into our encoding of the UFS. With this structure in place, we can cause queries over the document properties to exploit the hierarchy, since property values can now be located with a hierarchy of values (so that "Highways Department" is recognised as being a kind of "Government Agency", and so will be a valid match for searches for government agencies). This meets our first criterion.

Taking this one step further, we define a whole set of whole and partial category trees by this same property-encoding mechanism. Each "layer" is either a simple hierarchy or a set of modifications to a hierarchy (adding a new item, consolidating two items into one, and so on). Partial trees can be thought of refinement of a basic hierarchy; each new layer specifies some change to the layers below, so that the end result is a hierarchy that incorporates the information from all the others. This structure is illustrated in figure 5.

With this encoding, the type space can be manipulated not simply in terms of the basic UFS, but also in terms of local modifications that have been introduced to it. Since these modifications can be layered on top of each other, there can be multiple sets of refinements introduced by work groups, individuals, etc. Those refinements are explicitly structured as Presto documents themselves, so they can be collected together, stored and shared.

We are working on a document browser which allows users to work in terms of categories defined in this way. Our goal is to be able to work in terms of these "layered" typologies, so that users can perform local manipulations to the set of categories while still allowing the document to be viewed from the perspective of an earlier or simpler structure.

LESSONS

We developed Presto as an early embodiment of the "propertied documents" ideas, and as a basis for experimentation and application development prior to a larger and more ambitious design effort. Presto has served as an excellent vehicle for developing and validating our original intuitions, and while we have not yet conducted any formal user tests, we have none the less discovered patterns emerging from informal use and from application development. Some features have emerged as more important than we had originally imagined.

One of the primary experiences of property-based document interaction is *uniformity*. All document-related information appears in the same space (the property space), and can be viewed, browsed and searched using the same tools. This makes it easy to integrate information from many different sources, when documents play multiple roles, since a single search query can encompass different pieces of information. What's more, document property management applies uniformly across document storage layers. Documents may be stored on the Web, on a local filesystem, or elsewhere, but they can be managed, searched and organised uniformly. Hierarchies need not be replicated across different storage repositories, and the information is managed in an integrated way. This uniformity has proven invaluable.

For end users, one major feature of Presto interactions is the *lower cost of storage*. There is no need to find a unique name for a document, and to find a place to store it where it is likely to be found again. Instead, our applications can automatically tag data items with any relevant contextual information (such as the time of day, related documents, and so forth), any of which can be used to retrieve the document again.

Integration with existing document applications is crucial to the appeal of Presto. A key property of Presto is that it provides its facilities in concert with the existing tools for managing and interacting with documents. Our NFS implementation allows us to interoperate with applications that expect to operate over a traditional filesystem, but without losing the advantages of property-based interaction.

In applications, we have found that the *duality* that the Presto system offers is of great use to application developers. Presto can be used as both a document system and an object system simultaneously. In practice, this means that applications tend to use the same persistent store for both content objects and for data structures that manage them.

Relatedly, in developing our applications, we have found that a pattern quickly emerges in which Presto is used, essentially, as an *associative store*. This exploits the fact that, in Presto, property values can be arbitrary Java objects, including structured data and, for instance, references to other documents. So, rather than storing data in external files, programmers use Presto's "object model" to store their data structures by attaching objects to documents, and then rely on queries as a means to reconstitute them afterwards. The flexibility of extensible properties, and the absence of any fixed typing, means that data items can be organised into a variety of data structures fluidly and easily, and can be recombined and revised as work progresses.

Finally, as a consequence of this model, *query performance* turned out to be a critical issue. Since applications rely on search to build structures, high performance queries are critical. This also allows us to rely on queries as a fundamental part of the user interaction model. Presto's speedy response to document queries allowed us to provide an interactive experience based around the direct manipulation of dynamic document objects, rather than around a traditional "submit a query, await a response" model. Interaction is based on using queries to explore the space, rather than relying on structural properties and with only occasional use of heavyweight, expensive queries.

RELATED WORK

Lifestreams [7] uses a timeline as the major organisational resource for document spaces. Like our approach, Lifestreams is inspired by the problems of a standard single-inheritance file hierarchy, and instead seeks to use contextual information to guide document retrieval. Unlike our approach, however, Lifestreams replaces one superordinate aspect of the document (its location in the hierarchy) with another (its location in the timeline).

Semantic File Systems [8] introduce the notion of "virtual directories" that are implemented as dynamic queries on databases of document characteristics. With the NFS server implementation, Presto essentially provides the same sort of functionality which was provided by the Semantic File System. However, Gifford and his colleagues were largely concerned with direct integration into a filesystem so that they could extend the richness of command line programming interfaces, and so they introduced no interface features at all other than the file name/query language syntax. In contrast, our concern is with how such an attribute-based system can be used day-to-day, and with how our interfaces can be revised and augmented to deal with it; Presto acts as a filesystem simply in order to support legacy filesystem-based applications, rather than as an end in itself.

DLITE [5], a user interface for accessing digital library resources, explicitly reifies queries and search engines, providing users with direct access to dynamic collections. The goal of DLITE, however, is to provide a unified interface to a variety of search engines, rather than to create new models of searching and retrieval. So although DLITE queries are independent of particular search engines, they are not integrated with collections as a uniform organisational mechanism as they are in the Presto interfaces.

In terms of its simple object model, Presto is more akin to prototype-based systems like Self than it is to more traditional class-based languages like Smalltalk. Self, of course, offers a much richer programming model, but the design of Vista is also inspired by the directness and concreteness explored in the various Self user interfaces [4, 15].

CONCLUSIONS

We have been investigating a new approach to document management based on document properties. Properties encode information about a document and offer the potential to provide a uniform mechanism for document storage, management, retrieval and interaction. In addition, they allow us to organise the document management experience according to the needs of individual document consumers.

Presto is a platform for exploring this space. Our experience with applications has reinforced the value of this particular set of design choices. Presto's dual nature, as document system and associative object store, naturally supports not only a wide range of application which need to be able to provide fluid structures over a "document soup", but also those requiring the creation of more structured interaction.

Presto originated in work on the Placeless Documents project. Based on this experiment, we are currently exploiting the lessons we have learned in the development of the larger Placeless Documents prototype. Placeless Documents backs up the conceptual framework of propertied documents with a secure distributed infrastructure, and extends these ideas by incorporating live objects into the document space, opening up new areas for novel interactive document system development.

ACKNOWLEDGMENTS

The other members of the Placeless Documents project—John Lamping, Karin Petersen, Doug Terry and Jim Thornton—have contributed deeply to this work. We would also like to thank Tom Rodden for invaluable contributions.

REFERENCES

1. C. Ahlberg, C. Williamson and B. Schneiderman, "Dynamic Queries: An Implementation and Evaluation", *Proc. ACM Conf. Human Factors in Computing Systems CHI'92* (Monterey, CA), May 1992.
2. D. Barreau and B. Nardi, "Finding and Reminding: File Organization from the Desktop", SIGCHI Bulletin, 27(3), July 1995.
3. E. Bier, M. Stone, K. Pier, W. Buxton and T. DeRose, "Toolglass and Magic Lenses: The See-Through Interface", in *Proc. SIGGRAPH'93* (Anaheim, CA), August 1993.
4. B. Chang and D. Ungar, "Animation: From Cartoons to the User Interface", *Proc. ACM Symp. User Interface Software and Technology UIST'93*, October 1993.
5. S. Cousins, A. Paepcke, T. Winograd, E. Bier and K. Pier, "The Digital Library Integrated Task Environment", Technical Report SIDL-WP-1996-0049, Stanford Digital Libraries Project (Palo Alto, CA), 1996.
6. P. Dourish, J. Lamping and T. Rodden, "Building Bridges: Customisation and Mutual Intelligibility in Shared Category Management",to appear in Proc. GROUP'99, 1999.
7. E. Freeman and S. Fertig, "Lifestreams: Organizing your Electronic Life", *AAAI Fall Symposium: AI Applications in Knowledge Navigation and Retrieval* (Cambridge, MA), November 1995.
8. D. Gifford, P. Jouvelot, M. Sheldon and J. O'Toole, "Semantic File Systems", in *Proc. Thirteenth ACM Symposium on Operating Systems Principles* (Pacific Grove, CA), October 1991.
9. S. Jones, "Graphical Query Specification and Dynamic Result Previews for a Digital Library", *Proc. ACM Symp. User Interface Software and Technology UIST'98*, Nov. 1998.
10. D. Goldberg, D. Nichols, B. Oki and D. Terry, "Using Collaborative Filtering to Weave an Information Tapestry", *Communications of the ACM*, 35(12), December 1992.
11. R. Mander, G. Salomon and Y.-Y. Wong, "A 'Pile' Metaphor for Supporting Casual Organization of Information", *Proc. ACM Conf. Human Factors in Computing Systems CHI'92* (Monterey, CA), May 1992.
12. C. Marshall and R. Rogers, "Two Years before the Mist: Experiences with Aquanet", Proc. European Conf. on Hypertext ECHT'92, December 1992.
13. B Mynatt, T. Igarashi, K. Edwards, and A. LaMarca, "Flatland: New Dimensions for Office Whiteboards", *Proc. ACM Conf. Human Factors in Computing Systems CHI'99* (Pittsburgh, PA), May 1996.
14. E. Pedersen, K. McCall, T. Moran and F. Halasz, "Tivoli: An Electronic Whiteboard for Informal Workgroup Meetings", *Proc. InterCHI'93* (Amsterdam, The Netherlands), May 1993.
15. R. Smith, J. Maloney and D. Ungar, "The Self-4.0 User Interface: Manifesting a System-wide Vision of Concreteness, Uniformity and Flexibility", *Proc. ACM Conf. Object-Oriented Programming Systems, Languages and Applications OOPSLA'95* (Austin, TX), 1995.
16. Sun Microsystems, *Network File System Protocol Specification (RFC 1049)*, DDN Network Information Center, SRI International (Menlo Park, CA), March 1989.
17. R. Trigg, J. Blomberg and L. Suchman, "Moving Document Collections Online: The Evolution of a Shared Repository", *Proc. European Conf. Computer-Supported Cooperative Work ECSCW'99* (Copenhagen, Denmark), September 1999.