New Approaches to Information Management: Attribute-Centric Data Systems*

Ricardo A. Baeza-Yates
Dept. of Computer Science
University of Chile
Blanco Encalada 2120
Santiago 6511224, Chile
rbaeza@dcc.uchile.cl

Terry Jones
Distributed Cognition & HCI Lab
Cognitive Science Dept.
Univ. of California
San Diego, La Jolla
CA 92093-0515, USA
terry@cliffs.ucsd.edu

Gregory J. Rawlins
Dept. of Computer Science
Indiana University
Bloomington
IN 47405, USA
rawlins@cs.indiana.edu

Abstract

Trying to find information on the Web is like trying to find something at a jumble sale: it's fun, and you can make serendipitous discoveries, but for directed search it's better to go to a department store; there, someone has already done most of the arranging for you. Unfortunately, the Web's continuing explosion in size, its enormous diversity of topics, and its great volatility, make unaided human indexing impossible.

This problem is just a special case of the general problem of organizing information to create knowledge. A similar problem arises on the desktop when dealing with file systems, where users must search by name. When searching for a particular file, however, users often do not remember the file's name or location. File names are artifacts of current operating systems, but human understanding neither requires objects to be named, nor does it have problems with multiple objects sharing properties—names, for instance. That more general approach is not developed in current file systems or user interfaces.

The limitations mentioned above result because today's computer systems do not analyze the files they are asked to store. Instead they note only simple attributes like creation date and file type and leave the bulk of organizing, naming, annotating, and finding those files to their users. This may have made sense in the 1970s when computers were slow and expensive, but it makes no sense today.

We argue for an approach to information representation based on the use of attributes and search. This representation is organization-neutral, thereby giving a flexible substrate for anyone to build multiple simultaneous organizations. We argue the approach from three perspectives: Attribute Value System (AVS), a networked storage system where objects are composed solely of attribute-value pairs; DomainView (DV), a desktop metaphor where objects do not have explicit names and retrieval is done by content; and KnownSpace (KS), a personalized desktop data manager.

*The work of the first author was supported by Chile Fondecyt Grant #1990627, and the second author by DARPA Contract #N66001-94C-6039 and by a grant from Intel.

1 Introduction

Storing and organizing information is the kernel of any computer application. The widespread use and exponential growth of the World Wide Web, as well as other data sources, has had a crucial impact on the problem of managing personal information. Currently, the abstraction of file and folders is ubiquitous as the main way to organize and store documents. This abstraction, however, arose when computer resources were expensive, which is not true nowadays. In fact, we usually associate information with documents.

A collection of named files of information located in a hierarchical file system (HFS) is perhaps the most ubiquitous feature of modern operating systems. Virtually everyone who sits in front of a computer stores information in files and places these files within a HFS. As well as explicitly storing information in files, we store information in the hierarchical structure itself, in file names, and rely on our brains to maintain information about what we create. Our applications also operate within this framework, often encoding information in specific formats within files.

This working environment is so widespread that it is almost possible to forget that computational systems were once without the luxury of convenient information containers (files) and a structure in which to place them. It has become difficult to imagine an operating system without the familiar backdrop of files and folders. However, although there are occasions when it makes good sense to store information in a hierarchy of named files, being obliged to do so is a burden when the information being stored does not fit this paradigm, or has little or no relevance to the rest of the hierarchy structure.

Users accept HFS's because they are not hard to understand, they resemble physical archiving, and because there is no alternative. However, HFS's have several disadvantages. First, they try to simultaneously solve different problems: storing, representing, and organizing information. Second, they rely on the user's memory because every file and folder has to be named (and consistently). Third, the only semantic information about a file is given by the name path to it, which could have been named by another person or without enforcing a specific naming strategy. Fourth, naming is not easily scalable (typically there are limitations on how long names can be and what symbols can be used). Fifth, many files could conceptually belong to more than one folder, and although feasible in some systems using symbolic links (or their equivalent), it is not done for simplicity. In summary, we often have problems finding a specific file because we do not remember where it is and what it is named.

Current graphical user interfaces are based on windows, which, in most cases, either run an application or present a view of a HFS [6]. This means that the different problems of storing, representing, and relating information have the same solution at both the operating system and the user interface levels. This is purely for historical reasons, as the user interface designers had no choice but to use a HFS. This is reinforced by user interfaces showing hierarchical views of objects (we later detail this problem). On top of that, this solution is also being used on the Web, where HFS's are extended with domain names. Solutions to this problem include using a standard database or to add attributes to files, but they do not really address the intrinsic representation issue.

The semantics of a document/file depends on its use. It could depend on information about its creation and definition, a specific context, associations, structure, and its content. Some proposals, like semantic file systems, try to cover all these views using several approaches [10]. A subtle assumption is that a document has a content and metadata (attributes with values—which are data related to the content). This asymmetry is based on our physical abstraction of a document, but it is not necessary for the storage mechanism. A document could be just a set of attributes and values, one of them being the content. For different applications, different attributes will be more important than others, but intrinsically there is no reason to have data and metadata.

There are several studies about how users organize and retrieve documents in a desktop metaphor. Retrieval is usually based on location (that is, where the document is on the screen), content (by using a file searching tool), history (which file was being used before in an application—this is known as reminding), and in most cases by name, that is, where it is stored in the HFS (known as archiving) [4]. In all cases we are relying on the user's memory of past events, positions, and names. There is no real model for the user's actions.

Although searching by content is closer to a semantic search, it can return non-relevant documents because in a different context the same search keywords are valid (these are problems with polysemy and synonymy). This problem can be partially solved by adding additional information—for example, if we know that the document is not too old and is small. However, this kind of fuzzy information usually cannot be used, and even when it can, there is a lack of good systems that integrate search by content (text databases) and search by attribute values (relational databases). Part of the problem is due to the asymmetry mentioned before.

HFS's are valid as an internal storage mechanism for operating systems, but the user need not be aware of them. In fact, many users that do not understand this concept or use a few applications that put all their files in one directory (a flat hierarchy). Not all information can be classified as a hierarchical tree; there are many other ways to classify a group of objects, and many hierarchical ways to classify the same information. Suppose we were to start again, without assuming anything about computer resources. What might be the right way to store and organize information? Which functionality should be provided by the data model to manage information? We give one possible answer to this question.

In this paper we present a new approach to the storage, representation, and organization of information based on what we call attribute-centric data systems, and we show three prototypes that share, although in different levels of abstraction and implementation, the following ideas:

- They store data in objects having attributes and values in a uniform way, each having a dynamic structure.
- They can organize objects in collections which are also dynamic and independent of the storage mechanism.
- They are focused on searching and retrieving information without relying on human memory and easing the user interface (either for the end user or for a programmer).

This paper is organized as follows. We first present the main concepts behind the ideas above and how they can be used to manage information, clearly separating the storage, representation, and organization layers of the data being handled. We also present our goals for a user interface oriented toward information management. Then we show three different systems that share these concepts.

First, we present AVS, the Attribute Value System [12], a novel storage system which maintains a collection of objects composed solely of attribute/value pairs, and provides facilities for creating, altering, and locating these objects. This very simple substrate provides flexibility in the representation of information, emphasizes the role of search, generalizes hierarchical file systems, provides for the dynamic construction of arbitrary data structures and inter-object relationships, emphasizes the distinction between informational objects and structures that contain and organize them, facilitates multiple views of the same information, and provides a novel form of object ownership.

Next, we continue with DomainView [3], a desktop user interface that organizes information in domains chosen by the user using a uniform and simple interface. The interface is based on retrieval of documents by attribute values, including the content, in a flat and dynamic universe of domains. The desktop is tailored to/by a user, who creates his own document-driven and knowledge-domain world.

Last we present KnownSpace [14], a smart, visual, and autonomous information manager of all of a user's information, whether that information is data or programs, and whether it originates on the Web, via mail, news, ftp, an editor, or any other application. It is not a search engine, browser, desktop, or operating system, although it shares elements of all four programs. It is written in Java to make it portable across as many computers, operating systems, and browsers as possible.

We finish by comparing our ideas to related work, work in progress of the authors, and the consequences of our vision of organizing information, in particular regarding the Web. Notice that we really change basic assumptions, so we have to start from very basic principles and levels, which may seem naive for some readers or very radical for others.

2 Conceptual Framework

In this section we present the models behind our ideas: how to model data and how to model the user. The real limitations of any information system are not storage space or computing power. They are the narrow communication pipelines between the computer and the human user (typically through a screen), and between the user's eyes and brain. In that sense, any information system should not depend on the abilities of the user, should try to represent information as similarly as possible to the user's model of it, and any internal representation should be transparent to the user. However, the paradigms that are used today, as we mentioned in the introduction, do not satisfy any of these goals. Why? One main reason is that the historical development of software has forced the user to be aware of many unnecessary things. We can think of information as organized data which has to be comprehended and managed. So, there are two main problems to be solved: how to organize data, and how to communicate to a person through what is known as user interface. The two problems can be solved, as they have in the past, through a data and a user model. Next, we give our data model and our goals for a user model, as the later is still one of the major open problems in CS.

2.1 Data model

We can distinguish three different layers in a data model: data storage (physical), data representation (logical), and data relationships (organization). In most data models used today, such as a data structure implemented in a programming language, a HFS, or a database, those three layers are closely bound together for different reasons (efficiency, consistency, etc.) We believe that this binding restricts the way programmers, designers, or users can manipulate data; and forces design decisions that also affect user interfaces. We can give many different examples, some of them are already implicit in the introduction. One is how HFS's affect user interfaces: documents are stored in a tree that is traversed mostly linearly by opening folders. This should be surprising, as a tree is not a one dimensional structure and the user interface is communicating information through a two dimensional space (most interfaces do not take advantage of this fact).

Our data model explicitly separates these three layers. We do not impose any condition on the storage as it depends on technology. In fact, AVS uses a relational database to store objects, DV uses a HFS to store documents, and KS uses Java objects to store entities. Moreover, our model allows objects to be distributed and that should be transparent to the programmer and to the user.

A crucial part of our data model is the representation layer: *persistent objects with dynamic attributes and values.* In normal programming, objects have a fixed number of attributes and dynamic values. In OO programming languages, objects with more attributes can be created through inheritance. In our case we go one step further: we can add and delete attributes to an existing object at run-time. This means that we can also add and delete attributes to a collection of objects. For more details in the date model see [2].

This representation choice has some fundamental consequences on the relationship layer. In particular, it does not restrict the relationships to compile time, or to tables in a database, etc. By adding attributes we can create any and many relationships between different groups of objects. For example, the same data can simultaneously appear in a search tree and a hash table by just manipulating attributes, very easily. This is difficult to do in a normal programming language. To stress this idea, we allow *search capabilities on attributes and their values*, such that we can create a set of objects satisfying a given attribute-value query. So, the representation layer gives power and flexibility to the relationship layer. AVS, DV, and KS all share this idea, although how they use and maintain collections of objects is different. In our systems, objects are also called entities, pages or documents, and collections are also called clusters or domains. In both, DV and KS the basic data relationship is the collection of objects, which is the cornerstone of information organization.

2.2 User Model

The only intersection between a user's understanding of the spatial metaphor of the desktop and anything the computer understands and (sometimes) pays attention to, is the user's current location within the HFS. That is almost the entire extent of the computer's understanding of space today. It doesn't even keep track of the locations users visit most often, or those that users visit when doing various things (for example when moving an object with the mouse across the screen to the waste basket but missing it by a few pixels: most user interfaces will overlay both icons instead of realizing what the user intended to do).

Further, directories and files, and the programs that operate on them, do not even know that they are in a tree (the HFS), let alone that they are on a desktop. All they know is their name—which is the same as their path from the root. Consequently, they aren't even aware of their siblings in the tree. So from a file's point of view, it's not even inside a tree—it's at the end of a linear list. Since all files are placed in a tree and all naming, placing, accessing, and categorizing is based on the tree, today's desktop only appears to be a two-dimensional space, but is in fact a one-dimensional space, as we mentioned before.

In addition, there is the implicit assumption that we should have one common interface for all users (at least at the desktop level.) However, there is no reason to do not have one different interface for each user. Hence, the interface could be personalized by/to each user. So, our goals for a good user interface are:

- A simple interface with powerful navigation and search capabilities to manage information without relying on the user's memory, which if possible learns and adapts to each user.
- Use the available communication medium, the screen, to its full capacity. Good use of a two-dimensional space may provide a real illusion of space such that the user feels immersed in it, providing a better navigation metaphor.

These are very generic goals, and both DV and KS try to satisfy them with some degree of success. In both cases, information is organized as collections of objects in the data model described earlier, and a global thesaurus is used to have semantic capabilities. DV uses a flat space for all information which is retrieved by queries on attributes (in particular, content). KS is much more ambitious and has several different interfaces with the same goal (different visualizations of the organization).

3 Attribute Value Systems

The AVS is an embodiment of a philosophy of information storage and retrieval whose explicit aim is to provide a convenient and flexible storage substrate, and hence computational environment. The system is an effort to make it simpler for users, programmers, and applications to work with information: to accumulate it, share it, add to it, delete it, and to organize, retrieve, and view it in many ways. The AVS is intended to replace or supplement the traditional HFS with a substrate that better reflects the types of operations on information that we hope to perform in many modern computational environments.

Many of the problems that AVS seeks to address stem from difficulties inherent in a HFS. A traditional HFS provides just one organizational structure for information, and modern operating systems insist that we use it as a basis for storage. The object storage (files) and organization (a hierarchy) are tightly bound. While it is clearly possible to use this base for many things, there are important common operations that are very awkward. The AVS is designed to make these operations simple.

3.1 Conceptual Model

AVS attempts to provide a flexible information management environment for programmers, users, groups of user, and applications running on their behalf. It does this by providing a very general form of object to hold information and a flexible attribute-based method of creating relationships between these objects. AVS is based on: 1) Objects composed solely of attribute/value pairs; 2) Editing of these attributes and values; and 3) Object relationships built via assigning attributes to objects and discovered via subsequent search. These mechanisms generalize the HFS and provide a natural basis for dealing with information.

There is no a priori set of attributes that objects contain. Objects, by virtue of their attributes, may exist in many organizational structures (e.g., on graphs, on spreadsheets, on a desktop, in hierarchies), simultaneously, and exist there for real (not just as a copy of the name of an object in a distant structure which then has to be used to fetch the actual object—supposing it still exists). The search system provides access to attribute names and values equally, allowing searches for objects based on their properties. Objects may be annotated by the addition of attributes, without disturbing the original content.

Although an AVS may use a back-end database as a storage tool, it is not a data base itself. The AVS prescribes object structure (attributes and values) independent of content, but not organization. A database does the opposite. An AVS contains no a priori view of how information should be organized, and permits many simultaneous organizations, but a database is an explicit implementation of a single view of the organization of given information.

3.1.1 Attribute Value Objects

In an AVS, objects are collections of named attributes and their corresponding values. There are no special attributes, and there is no separate entity to which the attributes are attached. An object corresponding to what we call a file will have an attribute (perhaps named `content`) whose value contains the content of the file. It will likely have other attributes to hold information such as `name`, `size`, `date`, etc.

Any user or application may attach attribute/value pairs to any object they are able to find. This allows the addition of information to the object without disturbing the existing content. Content in a proprietary format can be augmented by attributes containing comments, annotations, summaries, questions, additional content, etc. The addition of these attributes does not require tools (which may not be known, available, or usable) to edit the existing attribute values.

Attribute names exist within namespaces. Namespaces allow applications to use simple attribute names. AVS contains a namespace called `public` with common attributes that applications might want to attach to objects and share (e.g., `text`, `creation-date`).

There are a small number of basic operations in an AVS. These are the addition and removal of attributes to/from objects, the altering of attribute names and values, and search. This simplicity means that implementations can be small and easily written, and increases the potential for code re-use. Operations such as moving an object within a file system, changing the name of an object, adding comments to an object, causing the object to appear in an application, all reduce to these operations.

3.1.2 Search

In an AVS, search based on attribute names and values is fundamental and ubiquitous. Applications use search to locate objects and collections of objects. Objects are assigned attributes that may cause them to be found in later searches. Information about searches need not be discarded. Objects may contain 1) a search query (allowing a later identical or similar search); 2) a copy of search results, or 3) references to found objects. Search and its results are treated as important members of the information system. There is low level support to ensure that saving search information is a natural operation.

3.1.3 Objects are not Owned, Attributes Are

The objects in AVS are not owned. They are composed of owned attributes. Objects may frequently be composed of attribute/value pairs added by several users or applications. The original attributes may be later deleted, but the object continues to exist. The object remains even if all its attributes are deleted. Such an empty object might be used for later communication between applications, as a place to announce events, etc.

This arrangement reflects the aggregation of information into and around objects in the real world. Memories, opinions, comments, and various other attributes exist regarding objects, and the fact that some attributes may disappear (or never appear) does not alter the fact that there is still an ownerless conceptual ball of information concerning the original material.

3.1.4 Permissions

The permissions model for an AVS places restrictions on attribute names and their values. A user who creates an attribute controls whether other users can: 1) see the new attribute name (i.e., that the attribute exists, and/or that an object has an instance of the attribute), 2) create or delete instances of the new attribute, and 3) read or write the value of attribute instances.

This allows users to assign private attributes to objects. A user may build objects composed entirely of attributes that cannot be seen by other users or applications. This offers a form of privacy: In AVS if you cannot find an object by search, you cannot view or alter it. Objects may thus be inaccessible (no attributes visible), partly accessible (some attributes visible, alterable, etc.), or fully accessible (all attributes visible) according to the permissions on its individual attributes.

3.2 Data Relationships via Attributes and Search

In AVS, as relationships between objects arise they are made real by the addition of attributes linking the objects involved. These objects and their relationships are dynamic, and are later discovered via search rather than through formal predefined data structures or following pointers. There are no a priori assumptions about data structures. Multiple relationships between objects can exist simultaneously, allowing multiple views of the same objects, no one more important than any other.

An object with a collection of attributes and corresponding values is an instance of *some* data structure (recall that data structures in programming languages are composed of fields and values). However, unlike data structures as they are used in programming languages, in AVS no-one need anticipate the relationships that may exist amongst objects or the fields (attributes) that might comprise an object. Similarly, building relationships via the addition of attributes is quite different from building the same relationships through the design of object-oriented class hierarchies.

Predetermined data structures and class hierarchies are of limited use in a world full of unexpected relationships between objects, programmers, and users with widely differing brains, perspectives, and needs. By using attributes and search in place of formal data structures, classes and pointers, AVS eliminates the need to anticipate relationships or to support a fixed number of them. Similar comments apply to the restrictions imposed by traditional databases.

4 DomainView

DomainView is a desktop metaphor which tries to address the user interface problems mentioned in Section 2. Although the initial motivation for DomainView was to use retrieval by content in a smarter way and to present a simpler interface for the user, we later realized that a different conceptual framework was needed to organize information and store documents. We briefly describe the data model and the user interface.

4.1 Data Model

The storage unit is a document. In DomainView a document is a generic object which has a dynamic number of attributes. Naming an object is optional (name is just one of the possible attributes of a document). Attributes can be potentially added by the user or by applications. Some initial attributes are creation time and application, size, and content. User documents are organized in collections, each one defining an information or application domain. Domains have a flat organization. That is, there is no hierarchy associated to them. Nevertheless, domains can naturally nest or overlap. However, they are dynamic, so those set relationships are not fixed. Hence, a document may belong to more than one domain.

Domains can be predefined and/or created by the user and are dynamic. Each domain has associated a set of words which defines it, from a global thesaurus. Predefined domains could depend on specific user tasks or applications. As documents, domains may have a name, but this is optional. The thesaurus can be initially defined by a system manager, extracted automatically from a subset of documents, or created by the user itself. In all cases, the thesaurus is dynamic and are modified by user actions. Documents or a set of documents can be retrieved by using a set of words which will be searched on all attributes and/or specific values in specific document attributes such as date, size, etc. Queries are stored and their result can define a new domain. Notice that there is no notion of a HFS and a document can only be retrieved using the value (or a range of values) of one or more attributes.

Documents can only be used through the desktop interface. That means that the concept of an application opening a document does not exist (more over, it is forbidden as was the original intent of one of the MacIntosh designers [15]). Applications are associated to domains and executed by documents, and not the other way around. To create a new document, there is a generic new document with no attributes, which can call any application or the applications associated to the document domain, if any.

4.2 User Interface

Conceptually, the desktop has a fixed layout and simple interactions. In that sense, is much more ordered than usual desktop metaphors. We can distinguish three different areas:

- Domain visualization: a window shows a visual representation of domains, identifying nesting and overlapping and giving global information about each domain. Several visualization metaphors are available and the user chooses the easiest to understand for him/her. Clicking on a domain is like retrieving that set of documents, which will appear in another window by showing the value of a user chosen attribute (content, date, description, etc).
- Document visualization: A document selected from the document list mentioned above, is shown below that window by using another attribute selected by the user, the default being the content. How the value is shown depends on the attribute and might even have more than one way of seeing it. The user chooses from a menu the visualization that is best suited for his/her purposes.
- Retrieval interface: several operations are offered by using text boxes, buttons, and dials. The main operation is retrieval, where a set of words is specified and/or values (or a range of values) are given for some attributes. The result of the search appears in the document window already mentioned (this can be considered a dynamic domain obtained through a query). By double clicking in a document, a menu of applications opens, from where the user chooses one. This is done for creating new documents. If the document domain has applications associated to it, those applications will appear in the top positions of the menu and will be highlighted. We can also access past queries if we want to perform repetitive tasks.

We have implemented a prototype of this interface using Java, which has almost all the functionality already described. The desktop screen is shown in the Figure 1 which shows the three areas. We used the standard HFS to implement our storage organization, so each document is actually a file but the user is never aware of it.

Our desktop can be seen as a simple interface to a different operating system where there is no HFS, but a uniform universe of objects (in our case called documents). We think that this metaphor is closer to reality and does not rely so much on the user of the memory, although we expect normal users to name all domains. However, the number of domains will be in general small, so this organization is much more scalable than a HFS. On the other hand, a user can have just one domain (his/her universe) and retrieve everything by content. That should be the final goal.

5 KnownSpace

Web search engine queries now often return millions of irrelevant pages. Those pages are not spatially arranged, clustered by topic, or distinguished in any way other than by their titles, so we have no idea of the relevance of any page before reading it. The same is true for mail and news.

After we save webpages, mail messages, news articles, ftp pages, or any pages produced with an editor or

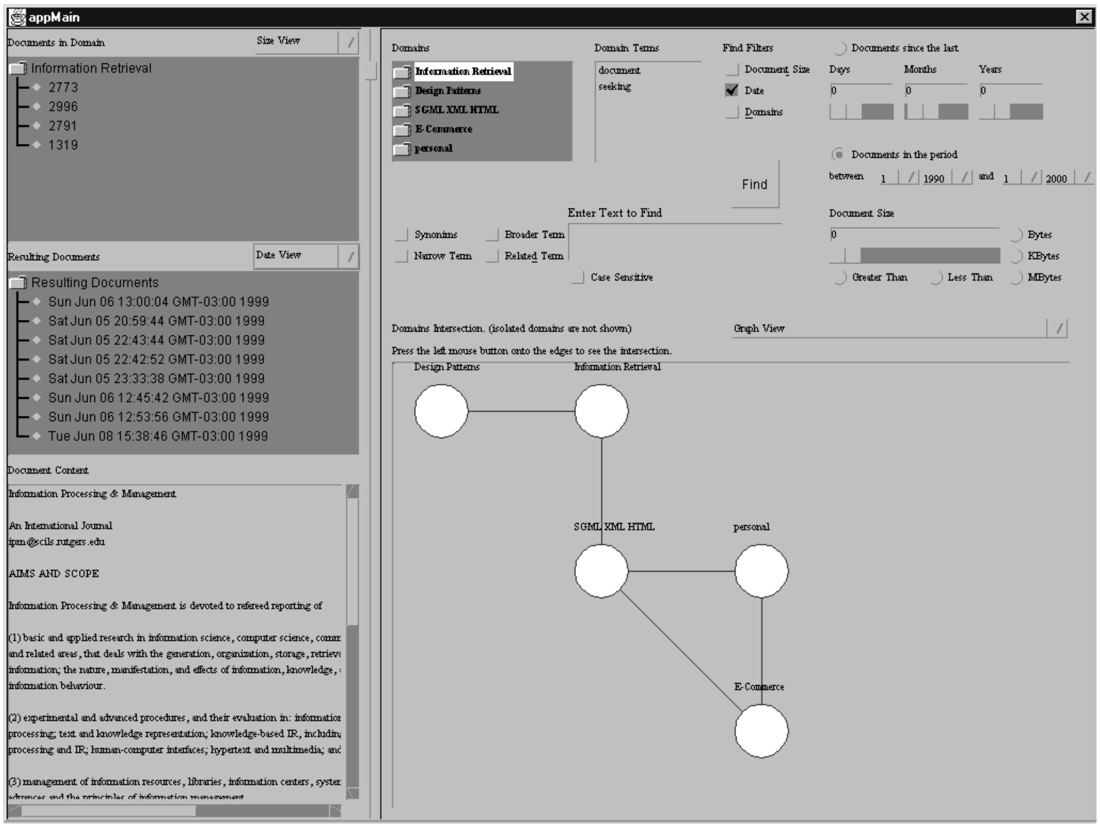

Figure 1. Current DV user interface.

any other application, those pages become lost on our desktops. They are not analyzed in any way, clustered according to our interests, or laid out spatially to show their similarities to other pages already there.

Even after we organize the pages on our desktops by hand there is no automation to help us reorganize them, search them, navigate through them, or find more pages like them. In short, our computers don't help us manage our own data.

Here we summarize the main features of KnownSpace, a smart data manager, as the whole system is too large to describe in the scope of this paper.

5.1 The Overall Design of KnownSpace

Data within KnownSpace is stored in Entities with Attributes. Entities may be anything (emails, webpages, desktop documents, or more abstract things like Persons, Organizations, and Websites). Attributes may be arbitrary (date added, phones numbers contained in, website pointed to by, and so on). KnownSpace also has an arbitrarily large number of Simpletons, which are agents all working on the Entities in parallel.

The Infopool holds the set of all Entities, and gives Simpletons access to appropriate Entities. When Entities enter the KnownSpace system they are put into the InfoPool. All Simpletons act by accessing Entities in the InfoPool, and either changing those Entities or learning from them. This is similar to a "BlackBoard" system in AI: lots of agents all working on a central store of data, each doing a little bit of computation and manipulation when they see a piece of data that they can work with.

This design maximizes modularity. A Simpleton programmer doesn't need to know what added the data that they will use, or what will use the data they add. This, for example, lets KnownSpace switch user-interfaces at any time, and it lets programmers add, remove or change Simpletons without causing errors. It lets programmers remain ignorant of how the rest of the system is working, and provides the minimum of limits on what programmers can do.

KnownSpace fetches data, analyzes data, shows it to the user, watches what they do, and learns.

The five modules that make up KnownSpace are:

- The Kernel, which contains the core of the KnownSpace system.
- The Collectors, which go out and get the information that goes in the KnownSpace system.
- The Adapters, which contain all the Simpletons that are constantly organizing and re-organizing the data within the system.
- The User Interface, which shows things to the user, accepts user input, and drops reports of what the users actions are back into the InfoPool for further analysis.
- The User Model, which watches for actions by the user and marks Entities to show how often they are used, when, and in combination with what other Entities.

These parts are themselves sub-divided into smaller parts (that is, Simpletons) that work independently. We now briefly summarize each module.

5.1.1 Kernel

The KnownSpace kernel contains the code for essential classes like Entity, Attribute, Simpleton, and InfoPool. The kernel keeps track of what types of Attributes currently exist in the KnownSpace system manages the Simpletons and other threads that are currently running. Further, the kernel manages Attribute locking and permissions (not every Simpleton has permission to change—or even read—every Attribute of every Entity).

The Kernel also contains various "tools", that is classes that Simpleton programmers might find useful. Eventually the Kernel will have responsibility for saving KnownSpace's state, but this is not yet implemented.

5.1.2 Collectors

The Collectors are responsible for getting data into KnownSpace. At the moment that means downloading from the web, from email, and from newsgroups. The Collectors should be pre-emptive: they should go out and get things if they are similar to things that the user has used before. The Collectors should also be going out and performing searches that the user has requested. However, it's not the Collectors job to decide what things should be fetched, nor is it their job to parse what they fetch; both jobs are handled separately.

5.1.3 User Interface

Currently KnownSpace has five user interfaces. A user could choose which interface they prefer, or swap which one they use depending on the task the user wishes to perform. Interfaces should take advantage of KnownSpace's highly adaptive data-set. Some interfaces are more focussed on the search capabilities that KnownSpace provides, others focus more on the way that KnownSpace organizes data spatially. This module has less Simpletons that the others, and more classes that are at a further remove from the rest of the KnownSpace system. Figure 2 shows one of the interfaces that have been designed for KnownSpace.

As well as the usual duties of a User Interface, the KnownSpace interfaces are responsible for watching the user's actions and dropping reports of what they have done (what they have accessed, when) into the InfoPool. This is a bottle-neck as far as the modularity of design is concerned, since other modules that want to use this information have to depend on the interface providing it. However, there seems to be no way to avoid this dependency.

5.1.4 User Model

The User Model takes the information about the user's actions that are placed into the InfoPool and uses that information to try and build a model of what the user's preferences are like. Currently this focus on when Entities are accessed, looking at frequency of access and at what things tend to be accessed together. Future versions will perform more sophisticated analysis. Information the User Model discovers from analysis is then dropped back into the InfoPool for use by other Simpletons, in particular the User Model determines what the Collectors should search for next.

5.1.5 Adapters

The Adapters look at the information in the InfoPool and try to find patterns within it. Typical Adapter Simpletons will clusters together any Entities in the InfoPool that have Attributes which contain the same words. The Adapters also analyze the Strings attached to Entities using a Semantic Space and try to map Entities onto locations in a multi-dimension space depending on how similar or different the Entities are.

The difference between Adapters and the User Model can be a bit fuzzy, but in general the User Model is a bunch of specialists who look just at what the user does, and the Adapters are looking at everything in the InfoPool. Together they build up a model both of the data and of the user using the data. The User Interface then uses this information to better serve the user.

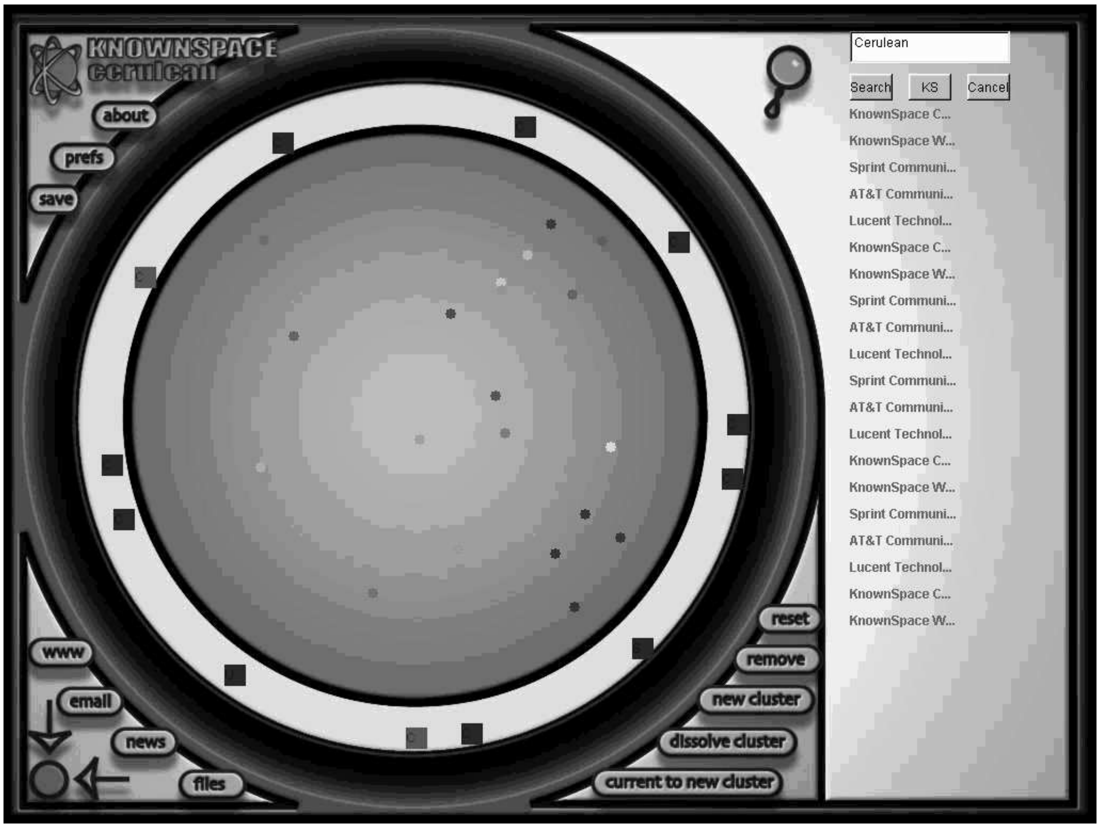

Figure 2. One of the KS user interfaces.

6 Concluding Remarks

Arguments for our data model include its underlying simplicity and flexibility, the removal of a priori assumptions about data structures and relationships between information objects, ease of use and implementation, its natural representation of real world information, the support for multiple views of the same information, and the fact that it generalizes the current structural imperative (the hierarchical file system).

AVS is closely related in spirit to the Placeless Documents project at Xerox PARC [7]. That project has a wide range of aims for building document management systems based on attributes and search. In practice, AVS resembles the BeOS file system [8], though the aims of the BFS implementers seem directed mainly towards allowing attributes for a small number of applications (e.g., a mail reader) using a small number of attributes. AVS also has much in common with Linda [9] and its derivatives (Java Spaces and T-spaces), in that it presents a flexible persistent forum for communications amongst applications. The focus of these tuple systems is centered on this communication, while AVS is more concerned with information representation and organization. These aims are partly found in work on semantic file systems [10]. In all these cases, the differences lie in one (or more) of: focus on a specific application (AVS is solely a platform); adding attributes to some special object (a document, a file); design emphasis (communications, HFS enhancements, document systems); and the emphasis on heavily automated addition of attributes.

The DV and KS user interfaces are focused in managing dynamic collections of richly attributed data that can be searched, although the real goal is to present more of a spatial interface. Although these interfaces uses windows, their use is minimal and oriented to effective use of the available screen. There are many other interfaces that partially cover our goals. One example is the Vista browser of the already mentioned Placeless documents Xerox project [7]. Also, there are many visualization metaphors to visualize collections of documents that are also related, such as BEAD [5], Vibe [13], TileBars [11], or Bookpile [1].

Our model can naturally extended to the Web. For example, all Web objects could be wrapped in XML, where we have attributes and values independent of the type of object (HTML, image, etc). To maintain the

XML philosophy, binaries would be converted to visible ASCII with no distinction between data and metadata. We think that this is a very uniform and portable object model for the Web. Consequently, searching the Web with agents would be much easier and any search engine index would be much more powerful because attributes give semantics to the data, which is also one of the goals of XML. All of our systems add richer layers of markup/interpretation to data which may (or may not) already be in some XML format.

Future work for AVS is to evaluate how programmers find the task of designing applications using the data model that we propose. Based on our experience with DV and KS, having AVS would have greatly simplified the development of these prototypes. Another important evaluation is how efficiently this model can be implemented, either from scratch or on top of a HFS or a relational data model.

Future work for DV and KS is to do a usability test, to determine if these new desktop metaphors are easier to understand and use by newcomers than existing user interfaces. As we propose a radically different interaction, users that already know current interfaces may dislike our proposal. Nevertheless, something similar happened with object oriented and structured programming. Changes are always hard.

References

[1] Ricardo Baeza-Yates. Visualizing large answers in text databases. In *Int. Workshop on Advanced User Interfaces*, pages 101–107, Gubbio, Italy, May 1996. ACM Press.

[2] Ricardo Baeza-Yates, Terry Jones, Greg Rawlins, A New Data Model: Persistent Attribute-Centred Objects, Technical Report DCC-99/4, Dept. of Computer Science, Univ. of Chile, June 1999.

[3] Ricardo Baeza-Yates and Claudio Mecoli. DomainView: A Desktop Metaphor based on User Defined Domains, Dept. of Computer Science, Univ. of Chile, 1999.

[4] Deborah Barreau, and Bonnie A. Nardi. Finding and Reminding: File Organization from the Desktop. SIGCHI Newsletter, Vol. 27 No. 3, July 1995.

[5] Matthew Chalmers and Paul Chitson. Bead: Exploration in information visualization. In *Proc. of the 15th Annual International ACM/SIGIR Conference*, pages 330–337, Copenhagen, Denmark, 1992.

[6] Andreas Dieberger. Navigation in Textual Virtual Environments using a City Metaphor, Ph.D. Thesis, Vienna University of Technology, Faculty of Technology and Sciences, November 1994.

[7] Paul Dourish, Keith Edwards, Anthony LaMarca, and Michael Salisbury. Presto: An Experimental Architecture for Fluid Interactive Document Spaces. Xerox PARC Working Paper, 1999.

[8] Dominic Giampaolo. Practical File System Design with the Be File System, Morgan Kaufmann, 1999.

[9] David Gelernter. Generative communication in Linda. ACM Transactions on Programming Languages and Systems, 2(1):80–112, January 1985.

[10] David K. Gifford, Pierre Jouvelot, Mark Sheldon, and James O'Toole. Semantic file systems. In 13th ACM Symposium on Principles of Programming Languages, October 1991.

[11] Marti A. Hearst. TileBars: Visualization of term distribution information in full text information access. In *Proc. of the ACM SIGCHI Conference on Human Factors in Computing Systems*, pages 59–66, Denver, CO, May 1995.

[12] Terry Jones. Attribute Value Systems: An Overview, Dept. of Cognitive Science, Univ. of California at San Diego, 1998.

[13] K. Olsen, R. Korfhage, K. Sochats, M. Spring, and J. Williams. Visualization of a document collection with implicit and explicit links: The Vibe system. *Scandinavian Journal of Information Systems*, 5, 1993.

[14] Gregory J. Rawlins. KnownSpace, `http://www.knownspace.org//`, 1999.

[15] Bruce Tognazzini. Tog on Software Design, Addison Wesley, 1996.