A New Data Model: Persistent Attribute-Centric Objects*

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

June 16, 1999

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, and often they 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.

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 a new data model to information representation based on the use of persistent objects with dynamic attributes and search operations over them. This representation is

*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.

organization-neutral, thereby giving a flexible substrate for anyone to build multiple simultaneous organizations. In addition, it is uniform, allows easy sharing of objects, and can be simply extended to the Web. We present an initial prototype of this idea (AVS), and to show its potential, two system prototypes that have as their main goal to organize personal information: DomainView and KnownSpace.

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 files 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.

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, it should try to represent information as closely 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, 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.

The question is how to organize personal information and how to present it to the user in a meaningful way through a simple but effective user interface. Any system for managing information, implemented either on top of a file system or a database, is limited by the underlying data model of the storage mechanism. On the other hand, if we write down as the goals of the system what functionality the underlying data model should provide, we do not necessarily obtain a known data model.

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 that data through a user interface. In this paper we present a new approach for the first problem: a new model of the storage, representation, and organization of information based on what we call *persistent attribute-centric objects* (PACO). PACO is based on:

- storing data in objects having attributes and values in a uniform way, each having a dynamic structure.
- being able to organize objects in collections which are also dynamic and independent of the storage mechanism, through powerful search capabilities.

We present our vision of how to manage information relative to our data model in [1]. That paper also discusses a few user interfaces.

This paper is organized as follows. We first present some of the motivations behind this paper, which are the basic functions needed to build efficient user interfaces for managing information together with a list of the main limitations of today's file systems and databases. Next, we present the main concepts behind the new data model and how they can be used to manage information, clearly separating the storage, representation, and organization layers of the data being handled. We also compare our data model with previous work and discuss some of the main differences and consequences of it. Next, we show a specific generic prototype of this data model, AVS, the Attribute Value System [14], which maintains a collection of objects composed solely of attribute/value pairs, and which provides facilities for creating, altering, and locating these objects. This 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.

To show the potential of our model we summarize the functionality of two systems to manage personal information: DomainView [2] and KnownSpace [15]. Although they were developed without using the data model proposed, their essence is based on it. In fact, the main ideas of this data model were developed independently by the three authors and later were unified into a single view. We finish by presenting work in progress of the authors, and the consequences of our vision of organizing information with respect to programming, user interfaces, and the Web. Since we change basic assumptions, we have to start from basic principles and levels, which may seem naive for some readers or very radical for others.

2 Motivation: Organizing Information

Web search engine queries now often return millions of irrelevant pages. Those pages are not spatially arranged, collected 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 any other application, those pages then become lost on our desktops. They are not analyzed in any way, grouped 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, and, as we discuss in the next section, we must do that task ourselves by organizing information in a file system.

Hierarchical file systems are valid as an internal storage mechanism for operating systems, but the user need not be aware of them. Many users do not even understand this concept and typically use applications that each put all their files in one directory (a flat hierarchy). Not all information can be classified as a tree; there are many other ways to classify a group of objects, and many hierarchical ways to classify the same information.

Let us start over, without assuming anything about computer resources. What might be the right way to store and organize information? Which functionality should a data model provide to best help users manage their information? We give one possible answer to this question.

To help make our claims concrete we present them in the context of aiding the development of two prototypes that organize personal information. They are DomainView (DV) and KnownSpace (KS). DomainView [2] is 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 and by a user, who creates his or her own document-driven and knowledge-domain world. KnownSpace [15] is an adaptive, 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.

Both prototypes manage information using collections of objects, each object having attributes with values. 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. While interacting with the DV or KS interface, the user may want to do any of the following:

1. navigate through a space of objects,
2. search for various kinds of objects,
3. organize objects into groups and regroup objects already in groups,
4. markup various objects, either singly or in groups, and edit or delete markups, and
5. browse, create, and delete objects.

Note that these groupings are *not* mutually exclusive. A single object can be in several dozen, or more, different groupings of objects, simultaneously. The user may choose to view any of these groupings at any time.

To support these actions, the interface needs to be able to pose the following queries to objects of the underlying data model:

- do you have attribute X? do your attributes satisfy property X?
- are you an attribute of any object? which objects are you an attribute of?
- what collections do you belong to? do you belong to collection X?
- what are your readable/modifiable attributes?
- are you locked at the moment? is your attribute X locked at the moment?
- can I read/write your attribute X? can I add/delete your attribute X?

and the following requests:

- give me read/append/modify access to all the objects with property X
- tell me the value of your attribute X
- change the value of your attribute X to Y
- add attribute X to yourself

- delete your attribute X
- tell me when your attribute X changes
- tell me when your attributes no longer satisfy property X
- attach the following (new or old) object to yourself
- delete object X from yourself
- give me a list of all the objects attached to you whose attributes satisfy property X
- delete all objects attached to you whose attributes satisfy property X

Note that we don't really need all these requests—some can be expressed in terms of others.

The only way we have found to support all of these abilities is to use the flexible data model outlined in this paper. Any object may have any other object (of whatever type) as an attribute. Further, those attributes may in turn have yet more attributes of their own, and so on recursively. This underlying data model gives both DV and KS the ability to do arbitrarily sophisticated things in their interfaces (each of which may be replaced by another at runtime). However, presently this comes at great cost, since to maintain maximum flexibility, our current implementation is quite slow. This is the single biggest issue to be worked on next and by formalizing the model, we make one more step on that direction. In the next section we discuss why current data models does not allow easy implementation of the capabilities above.

3 What is Wrong with File Systems and Databases

File systems and databases have the same final goal: to store and organize information. However, in both cases, they trade flexibility of data organization for access speed and reduced space used. Although more recent database models (object oriented, multimedia) are more generic, they still have limitations. For example, typically, object structure is static, and cannot change at run-time. Other limitations are the lack of a standard query language (for example, SQL in relational databases) and the power of the query language itself. File systems, on the other hand, are even more rigid and we will concentrate on this issue, as our proposal aims to replace both file systems and databases, which in fact should be the same thing. In fact, the final goal should be to have the capabilities of file systems as well as all database models, without the current problems arising when we use different data types or if we try to integrate two different database models. In particular, we would like to have at the same time all the good things of relational databases and of full-text databases.

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—mainly in file names—and rely on our memory 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 easy to forget that computational systems were once without the luxury of convenient information containers (files) and a structure in which to place them (directories). It has become hard 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 hierarchic structure.

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, of course, its content. Some proposals, like semantic file systems, try to cover all these views using several approaches [12]. A subtle assumption is that a document has content as well as metadata (that is, 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, only one of them being the content. For different applications, different attributes will be more important than others, but intrinsically there is no reason to separate 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) [3]. 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 too 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.

Users accept HFS's because they are not hard to understand, they resemble physical archiving, and—mainly—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 named 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 as well). 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 too awkward to be done routinely by most users. In summary, we often have problems finding a specific file because we do not remember where it is and what it is named.

Specific problems with the HFS usually include the following:

- Files can only live in one HFS. Although UNIX symbolic links and roughly equivalent mechanisms in other operating systems can be used to point at objects in another file system, files actually live in a single file system. These mechanisms are really just stopgap attempts to fix a fundamental problem. They create problems of their own since they essentially maintain a copy of a name of a file that is elsewhere. Problems arise when the destination file disappears or moves, and attempts to deal with these issues are expensive. Not dealing with these issues (as in some types of UNIX), leaves dangling symbolic links, another problem.

- A HFS is the only organizational structure in which a file can appear, and every file must be in a HFS. It is not possible to natively maintain a set of files in (for example) a priority queue, or a linked list or other organizational structure. Of course this illusion can be created with external programs and interfaces, but the underlying storage is still a HFS. A file cannot be on disk but not in a HFS. Thus there is no native provision for the organization of files, other than in a hierarchical tree.

- Files cannot be annotated without being disturbed. File formats and file and directory permissions often make it impossible to annotate existing files. Permissions will typically prevent a users from modifying another user's files, and specific file formats often make it impossible to read or alter an existing file using anything but a single application (which may not be available or even known to a user wanting to annotate a file).

- A single organization of files into directories must be chosen. The problem with choosing the directory structure ahead of time is that very often one's first choice of organization will not be the best. Once made, the directory structure in use can be awkward to change, Reorganizing large collections of files and directories into an alternate organization is not a well-supported activity and is fraught with dangers. Once done, there is no support for reverting to the former organization.

- It can be difficult to find files, either by name or content, or (especially) both. UNIX, for example, stores file information in three locations: i-nodes (permissions, size, dates, etc.), directory files (file name), and the files themselves (content). This organization makes it awkward to, for example, simultaneously search for files by name and content as different tools to read these different sources of information must be invoked and their results somehow combined. While this at least can be done, it is very awkward.

- The hierarchical structure can only contain files. A hierarchical organization structure can be appropriate for many tasks, but a HFS only makes this structure available for the organization of files. It is not possible to maintain a hierarchy of desktops or databases, unless information representing these complex objects is somehow encoded into and decoded from files. The HFS does not separate the logical organization of files into a hierarchy from the objects (files) that it organizes into this structure. With a separation between these two, hierarchies of other objects could be supported.

- The use and organization of an HFS relies heavily on our brains. People typically maintain a tremendous amount of information about a HFS in their heads. For example, we know what kinds of names we tend to use for files and that email correspondence can be found in a top level directory called "mail". Users are often forced to deal with file name extensions, whose meaning the are expected to remember. Users are expected to remember the structure of the hierarchies they create, as well as aspects of the underlying system hierarchy, and, often, equally idiosyncratic hierarchies created by other users.

- File names must be chosen and remembered. A big part of working with a HFS requires users to constantly be choosing and remembering file names. Conventions arise and must also be remembered.

- File locations must be chosen and remembered. As well as remembering names and naming conventions, users must remember positional information about the placement within the tree of files and directories.

- Further symptoms of the problems with the hierarchical structure found in today's operating systems are the very presence of symbolic links, magic numbers used to indicate file content type, so-called "invisible" files that follow a naming convention that applications can be separately written to optionally ignore, and the very frequent encoding of information about file content in file names. All of these mechanisms were afterthoughts designed to get around problems inherent in the original underlying organizational design.

Note that some of these problems are also valid for databases.

Adding to the restrictions inherent in the HFS, today's applications and user interfaces provide almost nothing to help the user deal with these problems. There is typically little or no support for deducing potential names or locations of files, for retrieval based on content rather than name, for categorizing existing files into more useful organizational structures, for navigation other than the one-dimensional up and then down walk through a hierarchy, for helping to re-organize hierarchical structures or to support multiple simultaneous organizations, or for anything but the most primitive notions of history of user behavior.

In short, being forced to operate on files with coarse ownership, often holding data in fixed formats, all organized within a single hierarchical tree presents a wide range of problems. Making matters worse, applications and user interface software typically offer virtually nothing to help alleviate these problems.

4 A New Data Model

We can distinguish three different layers in a data model: data storage (physical), data representation (logical), and data relationships (organizational). In most data models used today, such as a data structure implemented in a programming language, a HFS, or a relational database, those three layers are closely bound together for different reasons (efficiency, consistency, etc.). We believe that this binding restricts the way programmers, designers, and users can manipulate data; and it forces early design decisions that affect whole systems, and even, as we've seen, user interfaces.

Our data model explicitly separates these three layers. First, we do not impose any condition on how objects should be stored as that depends on technology. We only want our objects to be *persistent*. 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 it makes that transparent to the programmer and to the user.

The second layer of our data model, the representation layer, is crucial, for it is that which presents: *objects with dynamic attributes and values.* In normal programming, objects have a fixed number of attributes and dynamic values. In most OO programming languages, objects with more attributes can be created through inheritance or delegation. In our case we go one step further: we can add and delete attributes to an existing object at run-time, as in prototyping languages like Self [7]. This means that we can also add and delete attributes to a collection of objects. Formally, an object in our model is:

- A set of attribute instances, which may be empty.
- Each attribute instance is a value and an object, either or both of which may be empty.
- A value can be any predefined atomic data type or it can be a reference to an object.

Hence, our objects have identity as in OO programming, attributes have names, and values are typed. An attribute may contain an object because we want to have attributes of attributes, and so on. As attributes are dynamic, ownership and permissions should be on attribute instances and not on objects. The same is true for concurrency, and locking is also done on attribute instances.

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 at any time. For example, the same data can simultaneously appear in a search tree and a hash table simply by manipulating attributes. 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 dynamically create a set of objects satisfying any given attribute-value query. Consequently, the representation layer gives power and flexibility to the relationship layer.

Formally, we want support for the following type of queries and operations on objects:

- Search for objects having a given attribute or with an attribute value satisfying a certain property. The property will depend on the value type and could be a range if it is a number, or a regular expression if it is a string, and so on. We do not impose any restriction on what this property can be.
- Same as before, but with any level of recursion, such that we can query on sub-attributes and so on.
- Operations on sets of objects: union, intersection, and subtraction.
- The addition or removal of attributes (or sets of them) to sets of objects.

Although a PACO system might use a back-end relational database as a storage tool, it is not itself a database. The data model we advocate prescribes only object structure (attributes and values). It is independent of object content and says nothing about information organization. A relational database does just the opposite. The design of a set of database tables is heavily dependent on a priori knowledge of content. Given this knowledge of content, a database implements a single organization of the data. The prior conditions and the result are quite different. Our data model 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. This is notwithstanding database research on materialized views, since, for the most part, they are merely subsets of the database, and not truly new extensions of the data. However, recent work on adapting traditional relational databases to the exigencies of the web is increasing along the lines of *evolvable materialized views* [16].

An important part of our data model is that objects have no special piece of content to which attributes are attached. Although this has been done in the past (see the discussion of related work), it seems to be a move that has been ignored by more recent work. We believe this is an important step because:

- It allows objects that have no content (but other useful attributes). Such objects might, for example, hold attributes that serve to organize other objects.
- Objects may have many pieces of content. For example, different translations of the same document, all equally important, or various versions of a document or program.
- Attributes may exist before the content does. For example, notes by various people regarding a meeting may collect as attributes of an object before the minutes of the meeting (the main content) is available and added to the object (as the value of yet another attribute).
- Attributes may exist after the content goes away. Attributes may summarize a large object such as an image or book. It should be possible to remove the original content and not have all attributes suddenly vanish too.
- The focus of an object may change. The content which provokes the creation of a new object may, over time, become less important than some other attribute in the same object. By not treating the original material in a special fashion, there is no problem with changing focus.
- It increases uniformity of storage and access. Special content does not have to be stored or accessed differently from other information in the object. This means that awkward situations requiring the use and combination of multiple tools to access the same conceptual object do not arise.
- It is in accord with our general aims of providing more freedom to people who end up using our system.

While it is true that some of these problems can be avoided in a system that adds attributes to a special content object by simply leaving that object empty, that approach is less attractive. Under that approach, everyone using the system must deal with the decision to always have a special content object around which attributes aggregate. Instead, with no special content, systems that require all objects to have such content can simply add it as an attribute and treat it specially. Systems that do not require this do not inherit the obligation to deal with this irrelevant (to them) content object.

5 Comparison with Related Work

Using attributes as a mechanism of avoiding the difficulties of files or simply as a flexible storage layer is an approach that has been taken in several earlier projects. We can classify the majority of related work by the characteristics described in the following sections.

5.1 Object Persistence

Persistent objects are not new. They are the core of object oriented databases, which tipically are implemented over the standard file system. Another related topic is persistent programming languages, where everything persist from one run to the next, storing the state of a program in secondary memory. The main difference of our proposal to standard object oriented databases is that objects have dynamic attributes. That changes completely how they should be stored and also how the database can be queried. This difference also applies to persistent programming languages. In addition, our proposal replaces both, a database and a file system layer. It is a diiferent paradigm to access information.

5.2 Adding Attributes to Existing Objects

The addition of attributes to special objects: Placeless Documents (documents) [9]; Tapestry (mail messages, news articles) [13]; Semantic File Systems (files) [12, 10]; SHORE (files) [6]; the BeOS File System (files) [10]; the Synopses File System (files) [4].

Our data model does not treat as special any pre-existing object, such as a document or file or other content. This approach was taken earlier in work on Entities [18] and Self [7], but is a departure from more recent work in which attributes were added to special objects, as mentioned above. A discussion on this can be found in the section on our data model.

Our objects could be implemented using associative arrays as in AWK or perl, or string hashing tables in Java. However, in these contexts, implementing the search capabilities of our model would be extremely inefficient. On the other hand, object persistence is related to persistent programming languages and object oriented databases.

Our work is closely related in spirit to the Placeless Documents project at Xerox PARC [8, 9]. That project has a wide range of aims for building document management systems based on attributes and search.

5.3 Emphasis on Automated Processes

Some related work on attributes emphasizes automated processes which are used to gather information and incorporate it into the system as attributes - either from outside sources or by looking at local objects (mainly files): Semantic File Systems [12], Harvest [5].

As well as allowing for the automated addition of attributes to objects, especially in KnownSpace, our work also explicitly emphasizes the importance of ad hoc addition of attributes to objects, particularly by normal users interacting with information through graphical interfaces. In this it has similarities with the Presto's Vista browser [8], with a distinct focus on enabling the end user to interact in a flexible fashion with information by means of attributes.

5.4 Applications Versus Platforms

Very often, the move to use attributes is made as a means to an end: building a single application. It is clear that information processing via attributes is a flexible and powerful approach, but it has not resulted in the separation into an attribute-based information system that may be used to build multiple applications. Exceptions to this rule are Entities [18] and the BeOS file system [10].

We believe it is time that the power and flexibility of the attribute-based approach was separated from any given application and taken down to a more fundamental level in computational systems. In this way, the advantages of the approach will be available to any application that cares to build on this foundation. This seems to have been an aim in the BeOS file system (BFS) [10], the NT file system, and others. The AVS system described in this paper will eventually aim for a low-level implementation, such as that taken with the BFS, though the aims of the BFS implementers seem to have been less broad: directed mainly towards allowing attributes for a small number of applications (e.g., a mail reader) using a small number of attributes. An important difference is our move away from monolithic file objects with coarse ownership to which attributes are attached. Instead, we use PACO's, persistent objects composed solely of attributes and values.

5.5 Tuples

Our work has similarities with Linda [11] and its derivatives (Java Spaces from SUN and T Spaces [19]). Both present a flexible persistent forum for communications amongst applications, and the objects involved have a uniform representation that is not based on any pre-existing content. The focus of these tuple systems is centered on communication, probably reflecting the initial focus of Linda. The AVS system described in this paper also provides a persistent forum for this kind of inter-application communication, though that is not its main focus. We are more concerned with flexible persistent information representation and organization.

6 An Instance of Our Model: Attribute Value Systems

AVS is a prototype implementation of the above data model. Its explicit aim is to provide a convenient and flexible storage substrate, and hence computational environment. Many of the problems that AVS seeks to address stem from the difficulties inherent in a HFS discussed earlier. 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. AVS is designed to make these operations simple.

6.1 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.

The remainder of this section presents a broad outline of a system which provides an environment in which all these objectives are satisfied in a natural fashion.

6.1.1 Attribute Value Objects

In 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 always 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 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.

6.1.2 Search

In 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 cause them to be found in later searches. A hierarchy manager will add attributes to an object to cause that object to appear at some location in the hierarchy. An application putting objects onto a graph gives the objects attributes whose values store the coordinates of the object. These will later be retrieved, via search, by the application that displays the graph. A desktop manager will add attributes that cause an object to appear on a desktop, at a given location. All these attributes may be completely independent of one another, or they may be shared by various applications. In each case, the object is as much a member of the hierarchy, graph, desktop, etc., as it is of any other structure.

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.

In AVS, all actions reduce to attribute editing and search. All applications and users wishing to operate on objects within the system must first find the objects they wish to act on (though references to previously found objects may also be used). Search does not discriminate amongst attribute names or values. There are no special attributes. Finding a file with content matching a regular expression and whose name has a certain suffix is difficult in an HFS, because the content and name are stored separately, and treated differently by the file system. In AVS, such distinctions (and thus problems) simply do not exist.

6.1.3 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.

6.1.4 Ubiquity of Attribute Editing Operations

There are a small number of basic operations in AVS. These are the addition and removal of attributes to/from objects, the altering of attribute names and values, and search. This extreme conceptual 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 simple operations.

6.1.5 An Interpreted Object Language

The AVS will provide an interpreted language allowing operations on objects and attributes. This language will be used by applications to specify operations to be executed upon specified events (attribute assignment or deletion, object found in a search, etc.). The language may also serve as an extension language for applications.

6.1.6 Permissions

The permissions model of AVS places restrictions on attribute names and their values. A user or application that creates an attribute controls whether other users and applications 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.

Attribute permissions exist independent of any instance of an attribute, and serve to implement a permissions policy for future instances of the attribute. In addition, attribute instances may carry permissions information relevant to the instance in the object that is associated with each attribute. In the absence of these instance-specific permissions, the permissions for an attribute instance are inherited from those of its attribute.

In order to prevent normal users from altering permissions on arbitrary attributes or attribute instances, the system initially creates various important attributes and gives itself the permission to add, delete, and alter these in objects. Various system properties that exist in normal PACO objects are protected in this way with attributes that are owned by the system. It is envisaged that tools such as compilers will have attributes (such as "executable") that only the compiler may attach to objects.

6.1.7 Attribute Managers

Applications in AVS will usually implement an Attribute Manager component. A standard task for an application will be to manage a set of attributes which it routinely attaches to and manages in objects that are or become relevant to the application. A hierarchy manager might add a `name` attribute whose value encodes the location of the object in the hierarchy. Although no other attributes are necessary to maintain a hierarchy, such a manager might also manage other attributes, such as dates and access history, or may split the name into a path attribute and a name attribute, etc. Other applications will manage sets of attributes in a similar fashion. For example, a desktop application might manage a set of attributes `desk-x`, `desk-y`, `desk-icon` which it attaches to objects that should appear on its desktop, and an annotation application might manage a set of `comment`, `offset`, `date` and `author` attributes.

6.2 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.

7 Applications of the Model

In this section we briefly present two prototypes of systems that use the proposed data model to manage personal information.

7.1 DomainView

DomainView [2] 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 now summarize how the proposed data model is used in DV.

An object is a document which has a dynamic number of attributes. Naming an object is optional (its 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 creating 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 with 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, chosen 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. In all cases, the thesaurus is dynamic and is 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 (moreover, it is forbidden as was the original intent of one of the Apple Macintosh designers [17]). Applications are associated with 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.

The user interface allows the following capabilities:

- Visualizing domains, that is, sets of objects. In particular their intersection.
- Visualizing a given domain or the result of a search, that is, a set of documents.
- Retrieving sets of documents by searching on document attributes (by ranges in numerical attributes or by full-text retrieval operations in strings).

Clearly, all these capabilities are easily implemented in our data model. Our user interface prototype has been implemented in Java, and has almost all the functionality already described.

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.

7.2 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 also be arbitrary (date added, phones numbers contained in, website pointed to by, and so on).

As with DomainView, a typical KnownSpace interface presents a universe of entities, each of which may represent a document—a webpage, an email message, and so on, but it may also present entities representing more abstract things that seemingly have nothing whatsoever to do with documents in the traditional sense–like a Person. A Person however, if a powerful organizing principle in daily life. A Person sends email, a Person has a phone number (and that phone number may be stored in an email form yet another Person). A Person has an address, and so on. All these pieces of information can be hung on one entity inside KnownSpace. These users can browse, not just traditional "documents", but also any arbitrary collection of pieces of information the interface chooses to support.

Further, interfaces in KnownSpace are not tied to the system in the sense that most other interfaces are tied to their systems. KnownSpace has many faces. It already has five interfaces, and more are on the way. Each user can (potentially) have a unique interface, since part of what KnownSpace supports is the ability to build new interfaces inside KnownSpace (this work is not yet complete). Consequently, users may even have one KnownSpace interface they use when they're at work, and at home use a completely different one (although perhaps carrying over the same set of icons for each entity).

Within a particular interface, users may browse, delete, create, edit, or markup any entity. However, KnownSpace itself further marks up those entities, and any entities it fetches autonomously for presentation to the user. KnownSpace uses all that markup (all those attributes) to cluster the entities in many different ways, each of which is available to any KnownSpace interface. It is up to the interface designer to choose which particular views of the data space it will emphasize to its users.

To make all of this flexibility possible KnownSpace depends heavily on the flexible object-attribute model we presented earlier.

8 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).

The new data model should 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. This data model 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.

Our model can also easily be 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, although for complex objects it would be better to use a link to them and maintain them in their native format. 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 will revolve around evaluating 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 those 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, although the later imposes many restrictions that are already mentioned. Finally, our model leaves open many problems that we have not being able to address (yet).

References

[1] Ricardo Baeza-Yates, Terry Jones, and Gregory Rawlins. New Approaches to Managing Information: Attribute-Centric Data Systems. Submitted, 1999.

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

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

[4] Mic Bowman and Ranjit John. The Synopsis File System: From Files to File Objects. Position paper for the Joint W3C/OMG Workshop on Distributed Objects and Mobile Code. June 1996.

[5] C. Mic Bowman, Peter B. Danzig, Darren R. Hardy, Udi Manber, and Michael F. Schwartz. The Harvest information discovery and access system. In *Proc. 2nd Int. WWW Conf.*, pages 763–771, October 1994.

[6] Carey, M., DeWitt, D., Naughton, J., Solomon, M., et al., Shoring Up Persistent Applications. Proc. of the 1994 ACM SIGMOD Conference, Minneapolis, MN, May 1994.

[7] Craig Chambers, David Ungar, Bay-Wei Chang, and Urs Hölzle. Parents are Shared Parts: Inheritance and Encapsulation in Self. In Lisp and Symbolic Computation 4(3), Kluwer Academic Publishers, June, 1991.

[8] Paul Dourish, W. Keith Edwards, Anthony LaMarca, John lamping, Karin Petersen, Michael Salisbury, Douglas B. Terry and james Thornton. Extending Document Management Systems with User-Specific Active Properties. Xerox PARC Working Paper, 1999.

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

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

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

[12] 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.

[13] David Goldberg, David Nichols, Brian M. Oki and Douglas Terry. Using Collaborative Filtering to Weave an Information Tapestry. Communications of the ACM, v. 35(12) pp. 61-70, December 1992.

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

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

[16] Elke A. Rundensteiner, Andreas Koeller, Xin Zhang, Amy J. Lee, and Anisoara Nica; Evolvable View Environment (EVE): Non-Equivalent View Maintenance under Schema Changes, SIGMOD'99, Software system demonstration, Philadelphia, USA, May 1999.

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

[18] Andrew John Wilkes. Workstation Design for Distributed Computing (see Chapter 4, Entities). Ph.D. Dissertation, University of Cambridge, 1984. Reproduced as Hewlett Packard Technical Report ACS-88-32, April 1988.

[19] P. Wyckoff, S. W. McLaughry, T. J. Lehman and D. A. Ford. T Spaces. IBM Systems Journal, v. 37, No. 3 - Java Technology, Aug 1998.