The Naming System Venture

Hans Reiser

November 21, 2007

Abstract

For too long the file system has been semantically impoverished in comparison with database and keyword systems. It is time to change! The current lack of features makes it much easier to use the latest set theoretic models rather than older models of relational algebra or hypertext. The current file system syntax fits nicely into the newer model.

The utility of an operating system is more proportional to the number of connections possible between its components than it is to the number of those components. Namespace fragmentation is the most important determinant of that number of possible connections between OS components. Unix at its beginning increased the strategy. Let's take the file system namespace, and one-by-one eliminate the reasons why the filesystem is inadequate for what other namespaces are used for, one missing feature at a time. Only once we have done so will the hobbles be removed from OS architects, or even OS conspiracies.

Yet before doing that, we need a core architecture for the semantics to ensure we end up with a coherent whole. This paper suggests a set theoretic model for those semantics. The relational models would at times unacceptably add structure to information, the keyword models would at times delete structure, and purely hierarchical models would create information mazes. Reworking their primitives is required to synthesize the best attributes of these models in a way that allows one the flexibility to tailor the level of structure to the need of the moment.

The set theoretic model I propose has a syntax that is Linux, MacOS, and DOS file system syntax upwardly compatible, as well as CORBA naming layer upwardly compatible.

This is a planning document for the next major version of ReiserFS, that is, a description of vaporware. It is useful to ReiserFS users and contributors who want to know where we are going, and why we are building all sorts of strange optimizations into the storage layer (and especially those who are willing to help shape the vision in the course of discussions on the reiserfs-list@namesys.com mailing list...). Currently the storage layer for ReiserFS is working and useful as an everyday FS with conventional semantics. That storage layer is available as a GPL'd Linux kernel patch at http://namesys.com.

Introduction

Many OS researchers have built hierarchical namespaces that innovate in their effect on the integration of the operating system (e.g. Plan 9 and their file system [PPT+93]). Relational and keyword researchers rightfully scorn hierarchical namespaces as 20 years behind the state of the art [Dat86], but pay little attention to integration of the operating system as a design objective in their own work, or as a possible influence on data model design. I won't go into that here. Limiting associations to single key words is an unnecessary restriction.

A Naming System Should Reflect Rather than Mold Structure

The importance of not *deleting* the structure of information is obvious; few would advocate using the keyword model to unify naming. What can be more difficult to see is the harm from *adding* structure to information; some do recommend the relational model for unifying naming (e.g. OS/400).

By decomposing a primitive of a model into smaller primitives one can end up with a more general model, one with greater flexibility of application. This is the very normal practice of mathematicians, who in their work constantly examine mathematical models with an eye to finding a more fundamental set of primitives, in hopes that a new formulation of the model will allow the new primitives to function more independently, and thereby increase the generality and expressive power of the model. Here I break the relational primitive (that a tuple is an unordered set of ordered pairs) into separate ordered and unordered set primitives.

Relational systems force you to use unordered sets of ordered pairs when sometimes what you want is a simple unordered set. Why should a naming system match rather than mold the structure of information? For systems of low complexity, the reasons are deeply philosophical, which means uncompelling. And for multiterabyte distributed systems. . . ?

Reiser's Rule of Thumb #2: The most important characteristic of a very complex system is the user's inability to learn its structure as a whole.

We must avoid adding structure, or guarantee that the user will be informed of all structure relevant to his partial information. Avoiding adding structure is both more feasible and less burdensome to the user. Hierarchical, relational, semantic, and hypersemantic systems all force structure on information, structure inherent in the system rather than the information represented. If a system adds structure, and the user is trying to exploit partial knowledge (such as a name embodies), then it inevitably requires the user to learn what was added before he can employ his partial knowledge. With complex systems, the amount added is beyond the capacity of users to learn, and information is lost.

Example:

"My name is Kali, your friendly `whitepaper.html` technical support specialist for *REGRES*. Our system puts the *Library of Congress* online! How many I help you?"

George doesn't know Santa Claus' name: "I'm trying to find the reindeer chimneys Christmas man, and I can't get your system to do it."

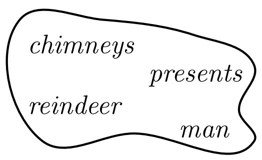

Figure 1: Graphical representation of a typical simple unordered set that is difficult for relational systems

> Kali says: "OK, now let's define a query: `is-a` equals *man*, that's easy. But *reindeer*? Is reindeer a property of this man?"
>
> "Uh no. I wish I could remember the dude's name. I read this story about him a long time ago, and all I can remember is that he had something to do with reindeer and chimneys. The story is on-line, somewhere."
>
> "'Reindeer chimneys presents man', that's the sort of speech pattern I'd expect from a three-year-old," Kali corrects him. "Let's see if we can structure this properly. Is *reindeer* an `instance-of` of this man? A `member-of` of this man? It couldn't be a generalization of this man. Hmmm. . ."
>
> "No! It's not that complicated. They just have something to do with him."
>
> "Pavlov would probably say you associate reindeer with this man, the way the unstructured mind of an animal thinks. But here in technical support we try to help our customers become more sophisticated. Is *reindeer* a property of this man?"
>
> "No. Try `propulsion-provider-for`."
>
> "Do you think that was the schema the person who put the information in our system used?"
>
> "No. Shoot. I can think of a dozen different columns it could be under. But what are the chances that the ones I think of are going to be the same as the ones the dude who put the information in used?"
>
> Kali feels satisfaction. "Guess it can't be done, not if you can't structure your *REGRES* query properly. I'll put you down in my log as a closed ticket. 190 seconds to resolution, not bad."
>
> "A keyword system could handle *reindeer chimneys christmas man*," George grumbles as he stares in despair at his display. Unfortunately, the *Library of Congress* is only one of *REGRES*' many reference aids. George could spend his life at it, and he'd never learn its schema.
>
> "But a keyword system would delete even necessary structure inherent to the information. It couldn't handle our other needs!" Kali says before she hangs up.

In addition to the searcher's difficulties, having to manufacture structure by specifying the column for *reindeer* also adds unnecessary cognitive load to the author's indexing tasks.

A Few of the Other Approaches to This Problem

There is lurking at the heart of my approach a subtle difference between my analysis for naming, and the analysis of at least some others. I started my research by systematically categorizing the different structures embodied by names, placing them into equivalency classes, and then picking one syntax out of each class of functionally equivalent naming structures, on the assumption that each of the equivalency classes has value. For example, I considered that languages sometimes convey structure by word endings (tags), and sometimes by word order, but while the syntax differs, the word order and word ending techniques are equivalent in their power to convery structure.

In my analysis of the effect of word ordering, I decided that either the ordering mattered, or it did not, and that was the basis for two different naming primitives.

Others have instead studied the inherent structure of data, and then from that derived ways of naming.

The hypersemantic system [SS77][PT88] represents an attempt to pick a manageably few columns which cover all possible needs. Generalization, aggregation, classification, and membership correspond to the `is-a`, `has-property`, `is-an-instance-of` and `is-a-member-of` columns, respectively. The minor problem is that these columns don't cover all possibilities. They don't cover *reindeer*, *presents*, or *chimneys* for George's query. The major problem is that they don't correspond as close as is possible to the most common style of human thought, simple unordered association, and require cognitive effort to transform.

The first response of relational database researchers to this is usually to ask: "Why not modify an existing relational database to contain an 'associated' column, put everything in that column, and it would be functionally equivalent to what you want." This is like saying that you can do everything Pascal can do using TEX macros. (They are both Turing complete.) We don't design languages to be Turing complete, we design them to be useful. I have seen a colleague do in six lines of SQL (nonstandard SQL) a simple three-keyword unorderd set that I do in three words plus a pair of delimiters, and that traditional keyword systems also handle easily. Doing simple unordered sets well is crucial for highly heterogeneous namespaces, and the market success of keyword systems in Internet searching is evidence of that. If you look at the structure of names in human languages, they are not all tuple structured, and to make them tuple structured might be to distort them.

I have merely discussed the burden of naming columns. Most relational systems also require the user to specify the relation name. If column naming is a burden, naming both the column and the relation is no less a burden.

Many systems invest effort into allowing you to take the key that you know, and figure out all the relation names and columns that you might choose to pair with it. This is a good idea, but not as good as not imposing extraneous structure to begin with.

[Sal86] can be read for devastating critiques of the document clustering system, but there is a worthwhile idea lurking within that system. Perhaps it is worthwhile to keep track of a small number of documents which are "close" to a given document. The document creator could be informed upon auto-indexing the document what other documents appear to be close to it, and asked to consider associating it with them. This is not within our current plan of work, but I don't reject it conceptually.

In summary, modularity within the naming system is improved by recognizing unordered grouping and ordering as two different functions that deserve separate primitives rather than being combined into a tuple primitive. The tuple is an unordered set of ordered pairs. There are other useful combinations of unordered grouping and ordering than that embodied by the relation, and the success of keyword systems suggests that a plain unordered set without any ordering at all is the most fundamental and common of them.

Names as Random Subsets of the Information In an Object

A system may still be effective when its assumptions are known to be false.

You may regard the above as an overstatement of the notion that we are neural nets, and sometimes our abstract systems deal with assumptions that are not true or false, but are somewhat true. After we are finished stating them in English they lose the delicate weighting posessed by the reality of the situation. Someteimes we find it easier to model without that weighting. Classical economics and its assumption of perfect competition is the best-known example of an effective system based on assumptions known to be substantially false. Introductory economics classes usually spend several weeks of class time arguing the merits of building models on somewhat false assumptions. This paper will now use such a somewhat false model to convey a feel for what mandatory pairing of name components causes problems.

Assume the user's information from which he tries to construct a description will be some completely random subset of the information about the object. (Some of that information will be structural, and the structural fragments selected will be just as random as the rest.) Assume a user has 15 random clues of information selected from 300 pieces of information the system knows about some object. Assume the *REGRES* naming system requires that data be supplied in threesomes (perhaps column name, key name, relation name), and cannot use one member of a threesome without the other members of the threesome. Assume the *ANARCHY* naming system lacks this restriction, but does so at the cost that it can only use those 10 of the 15 information fragments which do not embody structure. Assume the statistical distribution of the 15 pieces of information the user has to construct a name with are fully independent and equally likely (this is both substantially wrong, and unfair to *REGRES*, but ...) Assume each clue has a selectivity of 100 (it divides the number of objects returned by 100).

Then *ANARCHY* has a selectivity of $100^{10} = 10^{20}$ = good.

REGRES has a selectivity of:

$$100^{(C_{\text{other two}} \times 15)} = 100^{\left(\frac{9}{300} \times \frac{8}{300} \times 15\right)} = 1.05 = \text{very bad}$$

where $C_{\text{other two}}$ is the chance that the other two members of an object's threesome are possessed by the user.

While it is not true that the clues are fully independent, it is true that to the extent that they are not fully dependent, *ANARCHY* will gain in selectivity compared to *REGRES*. Attempting to quantify for any database the extent of the dependence would be a nightmare, and so this model assumes a substantial falsity, through which it is hoped the reader can see a greater truth. For databases of the lower heterogeneity and complexity that the relational model was designed for, the independence within a threesome can be small, and the ability to also employ the 5 of 15 fragments which are structural is often more important than the difficulty of guessing any structure added.

There is an implicit assumption here that you are looking for information that others have structured, and this argument in favor of *ANARCHY* becomes much less strong without this assumption. I feel obligated to stress once again that I do not advocate low structure over high structure, but I do advocate having the flexibility to match the amount of structure to the needs of the moment. Only with such flexibility can one hope to use all of the 15 fragments that happen to be possessed.

The Syntax In More Detail

What's needed is a naming system intended to reflect just the structure inherent in the information, whatever that structure might be, rather than restructuring the information to fit the naming system.

Orthogonal or Unoriginal Primitives and Features

There are many primitives that the ultimate naming system would include but which I will not discuss here: macros, OR, weight for subnames and AND–OR connectors [SFW88], rules, constraints, indirection, links, and others. I have tried to select only those aspects in which my approach differs from the standard approach.

Unifying the namespace does not require unifying automatic name generation, and those who read the [BM85] vs. [Sal86] controversy likely understand my concluding that whatever the benefits might be of unifying automatic name generation, it is not feasible now, and won't be feasible for a long time to come. The names one can assign an object are kept completely orthogonal from the contents of the object in the implementation of this naming layer. It is up to the owner of the object to name it, and it is up to him to use whatever combination of autonaming programs and manual naming best achieves his purpose. He may name it on object creation, and he may continually adjust its various names throughout its lifetime. See the section defining the "⟨`Key_Object`⟩ primitive" for a discussion of why names should be thought of this way.

Technically, object creation only requires the object be given a ⟨`Storage_Key`⟩. In practice most users will in the same act that creates the object, also associate the object with at least one name that will spare them from directly specifying the ⟨`Storage_Key`⟩ in hex the next time they make a reference to it. For applications implementing external name spaces, they can interact with the storage layer by referencing just the ⟨`Storage_Key`⟩.

Namesys will provide a manual naming interface, and the API autonaming programs need to plug into it. Companies such as Ecila will provide autonamers for various purposes.

Ecila is implementing a program which scans remote stores, creates links to them in the unified name space, but leaves the data on the remote stores. Other programs may also be implemented to perform this general function. To be more specific, the Ecila search engine scans the web for documents in French, and uses the filesystem as an indexing engine. However, they are writing their engine to be a general purpose engine, they have sold support and the addition of extensions to it to other search engine companies, and it is open source. For now we are simply functioning as part of their engine, and the interface is by web browser: at some point we may be able to add their functionality to the namespace. While the implementation of Microsoft's attempt to blur the distinction between the filesystem name space and the web namespace is one more of appearance than substance, it is surely the right thing to do for Linux as well in the long run. We should simply make our integration one with substance and utility, rather than integrating mostly the look and feel.

When the store is external to the primary store for the namespace, then stale names can be an issue with no clean resolution. That said, unification at just the naming layer is, in a real rather than ideal world, often quite useful, and so we have Internet search engines.

GUI based naming is beyond the scope of this paper, except to mention that it is common for GUI namespaces to be designed such that they are not well integrated with the other namespaces of the OS. They are often thought to necessarily be less powerful, but proper integration would make this untrue, as they would then be additional syntaxes, not substitutes. These additional syntaxes should possess closure within the general name space, and thereby be capable of finding employment as components of compound names like all the other types of names. The compound names should be able to contain both GUI and non-GUI based name components. Integration would make them simply the aspect of naming that applies to what is present in the visual cache of the screen, and to how to manage and display that cache most effectively.

Vicinity Set Intersection Definition (Also Called Grouping)

Suppose you have a set X of objects. Suppose some of these objects are associated with each other. You can draw them as connected in a graph.

Let the *vicinity* of an object A be the set of objects associated with A.

Let there be a set of query objects Q. Then the *set vicinity intersection* of Q is the set of objects which are a member of all vicinities of the objects in Q.

When thinking of this as a data model, it seems natural to use the term *vicinity set intersection*. When thinking of this syntactically, it seems natural to use the term *grouping*, because it implies that the subnames are grouped together without the order of the subnames being significant. There is exactly one data model primitive (set vicinity intersection) posessing exactly one syntax (grouping), and I rarely intend to distinguish data model primitive from syntax primitive (I can be criticized for this), and yet I use both terms for it, forgive me.

Synthesizing Ordering and Grouping

I am going to describe a toy naming system that allows focusing on how best to combine, grouping and ordering into one naming system. This synthesis will contain the core features of the hierarchical, keyword, and relational systems as functional subsets. It consists of a few simple primitives, allowed to build on each other. It sets the discussion framework from which our project will over many years evolve a real naming system out of its current storage layer implementation.

Resolving the second component of an ordering is dependent on resolving the first — unlike set theory. In set theory one can derive ordered set from unordered set, but because resolving the name of the second component depends on the first component one cannot do so in this naming system. For this reason it can well be argued that this naming system is not truly set theory based.

Now that I have mentioned this difference I will start to call them grouping and ordering, rather than unordered and ordered set. These two primitives take other names as sub-names, and allow the user to construct compound names. Either the order of the subnames is significant (ordering), or it isn't (grouping), and thus we have the two different primitives.

Because I have myself found that BNFs are easier to read if preceded by examples, I will first list progressively more complex examples using the naming system, and then formally define it. The examples, and the simplified syntax, use `/` rather than `:` or `\`, but this is of no moment.

Examples

Ordering and grouping are not just better; file system upward compatibility makes them cheaper for unifying naming in OSes based on hierarchical file systems than a relational naming system would be. This approach is fully upwardly compatible with the old file system. Users should be able to retain their old habits for as long as they wish, engage in a slow comfortable migration, and incorporate the new features into their habits as they feel the desire. Elderly programs should be untroubled in their operation. Many worthwhile projects fail because they emphasize how much they wish to change rather than asking of the user the minimal collection of changes necessary to achieve the added functionality.

```
/etc/passwd
```

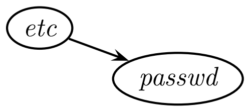

Figure 2: Graphical representation of `/etc/passwd`

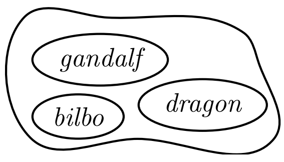

Figure 3: Graphical representation of `[dragon gandalf bilbo]`

```
[dragon gandalf bilbo]
```

Mr. B. Bizy looking for a dimly-remembered story (The Hobbit by Tolkien) to print out and take with him for rereading during the annual company meeting.

```
case-insensitive/[computer privacy laws]
```

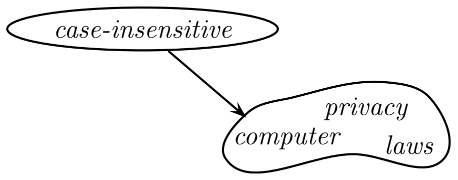

Figure 4: Graphical representation of query

When one subname contains no information except relative to another subname, and the order of the subnames is essential to the meaning of the name, then using ordering is appropriate. This most commonly occurs when syntax barriers are crossed. This is when a single compound name makes a transition from interpreting a subname according to the rules of one syntax to interpreting it according to the rules of another syntax. Ordering is essential at the boundary between the name of the new syntax as expressed in the current syntax, and the name to be interpreted according to that new syntax. Some researchers use the term context rather than syntax. The pairing of a program or function name, and the arguments it is passed, is inherently ordered. While that is usually the concern of the shell, when we use a variety of ordering functions to sort ⟨`Key_Object`⟩s of different types it affects the object store.

In this example the ordering serves as a syntax barrier. Case-insensitive is the unabbreviated name of a directory that ignores the distinction between upper and lower case. For Linux compatibility this naming layer is case sensitive by default, even though I agree with those who think that it would be better were it not.

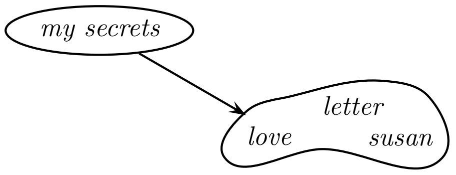

Figure 5: Graphical representation of searching for love letter to Susan

```
[my secrets]/[love letter susan]
```

Devhuman (that's the account name he chose) is the company's senior programmer. Six years ago he wrote a love letter to Susan, which he put in his read protected secrets directory. (He never found the nerve to send it to her.) He's looking for it so he can rewrite it, and then consider sending it. Security is a particular kind of syntax barrier (you have to squint a bit before you can see it that way). Here the ordering serves as a security barrier. (He certainly wouldn't want anyone to know that an object owned by him with attributes love letter susan existed.)

```
[subject/[illegal strike] to/elves from/santa document-type/RFC822
ultimatum]
```

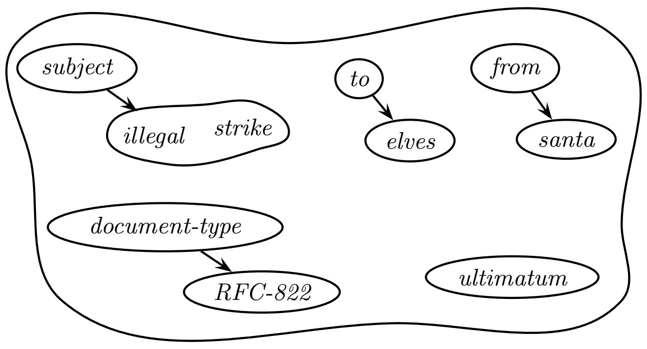

Figure 6: Graphical representation of search for Santa's ultimatum

Devhuman knows his object store cold. He is looking for something he saw once before, he knows that it was auto-named by a particular namer he knows well (perhaps one whose functionality is similar to the classifier in [Mea]), and he knows just what categorizations that namer uses when naming email. Still, he doesn't quite remember whether the word 'ultimatum' was part of the subject line, the body, or even was just elvish manual supplementation of the automatic naming. Rather than craft a query carefully specifying what he does and does not know about the possible categorizations of ultimatum, he lazily groups it. If Devhuman's object store is implemented using this naming system with good style, someone less knowledgeable about the object store would also be able to say:

```
[santa illegal strike ultimatum elves]
```

and perhaps get some false hits as well as the desired email (instead of finding mail from santa perhaps finding the elvish response). Notice that if you delete *illegal* and *ultimatum* to get

```
[subject/strike to/elves from/santa document-type/RFC8221]
```

the query is structurally equivalent to a relational query. Many authors (e.g. semantic database designers) have written papers with good examples of standard column names which might be worth teaching to users. So long as they are an option made available to the user rather than a requirement demanded of the user, the increased selectivity they provide can be helpful.

```
[_is-a-shellscript bill]
```

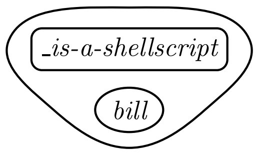

Figure 7: Graphical representation of `[_is-a-shellscript bill]`

This name finds all shellscripts associated with bill. Names preceded by `_` are pruners. Pruners are analogous to the predicate evaluators of relational database theory. If you have read papers distinguishing between recognition and retrieval, pruners are a recognition primitive. They are passed a list of objects, and return a subset of that list which matches some criteria. They are a mechanism appropriate for when a nonlinear search method that can deliver the desired functionality is either impossible, or not supported by existing indexes. There are many names for which we cannot do better than linear time search algorithms (perhaps simply as a result of incomplete indexing) that are useful. `_is-a-shellscript` checks each member of its list to see if it is an executable object containing solely ASCII. The user can use it just like any other ⟨`Key_Object`⟩ within an association, it will prune the results of the grouping. Since set intersections are commutative its order within the grouping has no meaning, and optimizers; are free to rearrange it.

The Formal Definitions

```
⟨Object Name⟩ ::= ⟨Grouping⟩
                | ⟨Ordering⟩
                | ⟨Key_Object⟩
                | ⟨Storage_Key⟩
                | ⟨Orthogonal and Unoriginal Primitives⟩
                ;
```

See the section listing orthogonal and unoriginal primitives for a discussion of what primitives I left out of the definitions of this grammar that are necessary to a real-world working system.

The name resolver will have a method for converting all of the primitives into ⟨`Storage_Keys`⟩, and when processing the compound names it first converts the subnames into ⟨`Storage_Keys`⟩, though the object may have null contents, and serve purely to embody structure. This allows the use of anything which anyone can invent a way of allowing the user to find a ⟨`Object Name`⟩ for, and then invent a method for the resolver to convert the ⟨`Object Name`⟩ into a ⟨`Storage_Key`⟩, as a component of a grouping or ordering. In a word, closure. Extensible closure.

Compound names are interpreted by first interpreting the subnames that they are constructed from. At each stage of subname interpretation an ⟨`Object Name`⟩ is converted into a ⟨`Storage_Key`⟩ for the object that it is resolved to. The modules that implement the grouping and ordering primitives do not interpret the subnames, they merely pass them to the naming system which returns the ⟨`Storage_Key`⟩s they resolve to.

It was a long discussion which led to the use of storage keys rather than objectids. A storage key differs from an objectid in that it gives the storage layer directions as to where to try to locate the object in the logical tree ordering of the storage layer. If the logical location changes, then in the worst case we leave a link behind, and get an extra disk access like we get with an inode. (Inode numbers are functionally objectids) In the better case, the repacker eventually comes along, and changes all references by key to the new location, at least for all objects that have not given their key to external naming systems the repacker cannot repack. A ⟨`Storage_Key`⟩ is assigned by the system at object creation, and serves the purpose of allowing the system to concisely name the object, and provide hints to the storage layer about which objects should be packed near each other. The user does not directly interact with the ⟨`Storage_Key`⟩ any more often than C programmers hardcode pointers in hex. The packing locality of keys may be redefined.

The Primitives

⟨`Key_Object`⟩

A description of the contents of an object using the syntax of the current directory. For objects used to embody keywords this may be the keyword in its entirety. If it contains spaces, etc. it must be enclosed in quotes. Note that making it easy for third parties to add plug-in directory types is part of Namesys's current contract with Ecila. Ecila wants space efficient directories suitable for use in implementing a term dictionary and its postings files for their Internet search engine.

Example: `[reindeer chimneys presents man]`

In this, *presents*, *reindeer*, *chimneys*, and *man* are the contents of objects associated with the Santa Claus story. Each of them is searched for by contents, and then when found they are converted into their ⟨`Storage_Key`⟩s, and then the grouping algorithm is fed their three ⟨`Storage_Key`⟩s. The grouping module then looks in the object headers of the three objects, gets the three sets of objects the ⟨`Key_Object`⟩s group to, and performs a set intersection.

Besides greater closure, another advantage of storing ⟨`Key_Object`⟩s as objects is that non-ASCII ⟨`Key_Object`⟩s and ordering functions can be implemented as a layer on top of the ASCII naming system, allowing the user to interact with the naming system by pressing hyperbuttons, drawing pictures, making sounds, and supplying other non-ASCII ⟨`Key_Object`⟩s that the higher layers convert into ⟨`Storage_Key`⟩s.

There are endless content description techniques, if the directory owner supplies an ordering function for the ⟨`Key_Object`⟩s in a directory, one can generate a search index for the directory using an directory plug-in which is fully orthogonal to the ordering function, though perhaps slower in some cases than one that is tailored for the ordering function. Users will find it easier to write ordering functions than index creation objects, and will not always need the speed of specialized indexes. We will need one ordering function for ASCII text, another for numbers, another for sounds, perhaps someday one even for pictures of faces (perhaps to be used by a law enforcement agency constructing an electronic mug book, or a white pages implementation), etc. No system designer can provide all the different and sometimes esoteric ordering functions which users will want to employ. What we can do is create a library of code, from which users can construct their own ordering function and their own directory plug-ins, and this is the approach we are taking on behalf of Ecila.

For an Internet search engine one wants what is called a postings file, which is like a directory in that there is no need to support a byte offset, and one frequently wants to efficiently perform insertions into it.

Grouping

```
      ⟨Grouping⟩ ::= [ ⟨Unordered List⟩ ]
                     ;
⟨Unordered List⟩ ::= ⟨Unordered List⟩
                   | ⟨Object Name⟩
                   | ⟨Pruner⟩
                     ;
        ⟨Pruner⟩ ::= _⟨Object Name⟩
                     ;
```

A ⟨`Grouping`⟩ is a list of ⟨`Object Name`⟩s and ⟨`Pruner`⟩s whose order has no meaning. Every object has a list of objects it groups to (associates with in neural network idiom) in its object header. A grouping is interpreted by performing a set intersection of those lists for every object named in the grouping. In the sense of the data model, the interpretation of a grouping is interpreted by performing what is in the sense of the data model a set vicinity intersection.

Grouping is not transitive: $[A] \Rightarrow B$ and $[B] \Rightarrow C$ does not imply $[A] \Rightarrow C$ though it does imply that $[[A]] \Rightarrow C$

A ⟨`Pruner`⟩ is an ⟨`Object Name`⟩ which has been preceded with an underscore (`_`) to indicate that the object described should be passed a list of objects named by the rest of the grouping, executed, and it will return a subset of the list it was passed. Whether a member of the set is in the returned subset must be fully independent of what the other members were of the set, or else the results become indeterminate after application of a query optimizer, as with an optimizer in use there is no guarantee provided of the order of application of the pruners.

Ordering

```
          ⟨Ordering⟩ ::= ⟨Object Name⟩/⟨Object Name⟩
                       | ⟨Object Name⟩/⟨Custom Programmed Syntax⟩
                         ;
⟨Custom Programmed Syntax⟩ ::= varies; provides extensibility hook
                         ;
```

An ordering is a pairing of names, with the order representing information. The first component of the ordering determines the module to which the second component is passed as an argument. In contrast, a grouping first converts all subnames to ⟨`Storage_Key`⟩s by looking through the same current directory for all of them in parallel, and then does its set intersection with the subdescriptions already resolved.

Example: In resolving the query `[my secrets]/[love letter susan]` the system would look for the objects with contents *my* and *secrets*, find both of them and do a set intersection of all of objects those two objects both group to (are associated with). This will allow it to find the `[my secrets]` directory, inside of which it will look for the three objects *love*, *letter*, and *susan.* It will then extract from their object headers the sets of objects those three words ('love', 'letter', and 'susan') group to, and do a set intersection which will find the desired letter. The desired letter is not necessarily inside the `[my secrets]` directory, though in this case it probably is.

A *directory* is an object named by the first component of an ordering, to which the second component is passed, and which returns a set of ⟨`Storage_Key`⟩s. One can in principle use different implementations of the same directory object without impacting the semantics and only affecting performance, as is often done in databases.

There are flavors of directories:

- *Custom programmed directories*, aka filters, are any executable program that will return a ⟨`Storage_Key`⟩ when executed and fed the second component as an argument. They provide extensibility. (They are the ordered counterpart of pruners.) Another term for them is filter directories. Custom programmed directories whose name interpretation modules aren't unique to them will contain just the name of the module (filter), plus some directory dependent parameters to be passed to the module. It should be considered merely a syntax barrier directory, and not a fully custom programmed directory, if those parameters include a reference to a search tree that the module operates on, and if that search tree adheres to the default index structure. The connotations conveyed by the term 'filter' of there being an original which is distorted are not always appropriate, but in honesty this is not an issue about which we deeply care.

- *Syntax barrier directories* allow you to describe the contents of the object they contain with a syntax different from their parents. Except for being sorted by a different ordering function, the indexes of syntax barrier directories are standard in their structure, and use a standard index traversal module. The index traversal module is ordering function independent.

There must be an ordering function for every ⟨`Key_Object`⟩ employed within a given syntax barrier directory. By contrast, ⟨`Custom Programmed Syntax`⟩ could be anything which the syntax module somehow finds an object with, possibly even creating the object in order to be able to find it.

- To cross a *security barrier directory* the user must use an ordered pair of names with the security barrier as the first member of the pair, and he must satisfy the security module of the secured directory. A security barrier directory may be both a security and a syntax barrier directory, or the security barrier directory may share the syntax module of its parents.
- *Fully standard directories* are those built using the default directory module, and adding structure is their only semantic effect.

There is an aspect of customization which is beyond the scope of this paper, in which one customizes the items employed by the storage layer to implement files and directories. That is, the storage of the files and directories are implemented by composing them of items, and these items have different types. We are now creating the code for packing and balancing arbitrary types of items using item handlers and object oriented balancing code, so as to make it easier to extend our filesystem.

Ordering can be implemented more efficiently than grouping

The set intersections performed in evaluating the grouping primitive are normally much more expensive computationally than performing the classical filesystem lookup. Imposing excess structure on one's data does not just at times reduce the cost of human thinking ⌣̈, it can be used to reduce the cost of automated computation as well.

When the cost to a user of learning structure is less important than the burden on the machine, use of highly ordered names is often called for.

The Motivation for Different Syntactic Treatment of Ordering and Grouping, and Some of the Deeper Issues Revealed by the Difference

An important difference between grouping and ordering affects syntax. It allows us to represent an ordering with a single symbol (`/`) placed between the pair, but requires two symbols (`[` and `]`) for each grouping. Imagine using `<` and `>` as a two-symbol delimiter style alternative notation for ordering:

```
<<father-of mother-of>sister-of> = <father-of<mother-of sister-of>>
= <father-of mother-of sister-of> = father-of/mother-of/sister-of
```

All of the expressions above are equivalent in referring to the paternal great aunt of the person who is the current context. The ones using nested pairs of symbols to enclose pairs of subnames imply a false structure that requires the user to think to realize the first two expressions are equivalent. The fourth is the notation this naming system employs.

Grouping is different: Fast Acting Freddy is looking through the All-LA Shopping Database for a single store with black reebok sneakers, a green leather jacket, and a red beret so that he can dress an actor for a part before the director notices he forgot all about him.

`[[black reebok sneakers] [green leather jacket] [red beret]]`
is not equivalent to `[black reebok sneakers green leather jacket red beret]`
which equals `[red sneakers black reebok jacket green beret]`

Ordering is not algebraically commutative (`father-of/mother-of` is not equivalent to `mother-of/father-of`). Groupings, however, are algebraically commutative (`[large red]` = `[red large]`).

Style

As a general principle, a more restricted system can avoid requiring the user to repeatedly specify the restrictions, and if the user has no need to escape the restrictions then the restricted system may be superior. This is why "4GLs", which supply the structure for the user's query, are useful for some applications. They are typically implemented as layers on top of unrestricting systems such as this one.

This paper has addressed issues surrounding finding information, particularly when the user's clues are faint. When supporting other user goals, such as exploring information, adding structure through substantial use of ordering can be helpful. [Marchionini][McAleese].

When the user goal is finding, one should assume that of all the fragments of information about an object, the user has some random subset of them. The goal is to allow the user to use that random subset in a name, whatever that subset might be. Some of that subset will be structural fragments. While requiring the user to supply a structure fragment is as foolish as requiring him to supply any other arbitrary fragment, allowing him to is laudable.

In the best of all worlds the object store would incorporate all valid possible structurings of ⟨`Key_Objects`⟩. The difficulty in implementing that is obvious. [Metzler and Haas] discuss ways of extracting structure from English text documents, and why one would want to be able to use that structure in retrievals. Unfortunately, there is an important difference between representing the structure of an English language sentence in a way that conveys its meaning, and representing it in a way that allows it to be found by someone who knows only a fragment of its semantic content. I doubt the wisdom of trying to advocate the use of more than essential structure in searching.

You can allow users to avoid false structure; you cannot force them to. It is important to teach those creating the structure that if they group a personnel file with *sex/female* they should also group it with *female.*

Type checking can impose structure usefully. Its implementation can enhance or reduce closure, depending on whether it is done right.

When To Decompound Groupings

There are dangers in excessive compounding of compound groupings analogous to those of excessive ordering.

Let's examine two examples of compound groupings, both of which are valid both semantically and syntactically. One of them can be "decompounded" with moderate information loss, and the other loses all meaning if decompounded.

Example: Finding a loquacious Celtic textbook salesman who told you in excruciating detail about how he was an ordinance researcher until one day he went to a Grateful Dead concert.

```
[[Celtic textbook salesman] [ordinance researcher]]
```
vs.
```
[celtic textbook salesman ordinance researcher]
```

These two phrasings of the same query are not equivalent, but they are "close".

Our second example is the one in which Fast Acting Freddy tries to find a suspect by the objects he is associated with:

```
[[black reebok sneakers] [green leather jacket] [red beret]]
```
vs.
```
[black reebok sneakers green leather jacket red beret]
```

These two are not at all "close". The difference between the two examples of inequivalencies is that the subdescriptions within the second example describe objects whose existence within the object store independednt of the store described is worthwhile. The first does not, and it is more reasonable to try to design so that the "decompounded" version of the query is used. False hits will occur, but for large systems that's better than asking the user to learn structure.

A higher level user interface might choose to present only one level to the user at a time, and then once the user confirms that a subdescription has resolved properly it would let him incorporate it into a higher level description. There might be 6 models of `[black reebok sneakers]`, and Fast Acting Freddy should have the opportunity to click his mouse on the exact model, and have the interface substitute that object for his subdescription. Using such an interface an advanced user might simultaneously develop several subdescriptions, refine and resolve them, and then use the mouse to draw lines connecting them into a compound grouping. Closure makes it possible for that to work.

Examples of Creating Associations

- $\leftarrow$ creates an association between all of the objects on the left hand side and all of the objects on the right hand side.
- $A \setminus B$ is the set difference of A and B, and it resolves to the set of objects in A except for those that are in B.
- $A \cap B$ resolves to the set intersection of A and B, the object that are both in A and B.
- $[AB] = [A] \cap [B]$, by definition.

And so:

- animal $\leftarrow$ (lives, moves)
- mammal $\leftarrow$ ([animal], animal, ‘warm blooded’)
- cat $\leftarrow$ ([mammal], hypernym/mammal, mammal, meronym/fur, fur, meronym/whiskers, whiskers, hypernym/quadruped, quadruped, capability/purr, purr, capability/meow, meow)
- Basil $\leftarrow$ (owner/Nina, Nina, [siamese], siamese, clever, playful, brave/overly, brave, ‘toilet explorer’)
- bag $\leftarrow$ ([container], container, consists-of/‘highly flexible material’, ‘highly flexible material’)
- backpack $\leftarrow$ ([bag], shoulderstrap/quantity/2, shoulderstap, college-student, holonym/backpacker, meronym/shoulderstrap)
- mould $\leftarrow$ ([fungi] - green/not, furry, ‘grows on’/surfaces/moist, ‘killed by’/chlorine)
- fungi $\leftarrow$ ([plant], plant, leaves/no, flowers/no, green/not)
- bird $\leftarrow$ ([vertebrate], vertebrate, flies, feathers)
- penguin $\leftarrow$ ([bird] - flies, bird, hypernym/bird, swims, Linux, [Linux (mascot, symbol)])
- siamese $\leftarrow$ ([cat], cat, hair/short, short-hair)

  Notice how we don’t associate siamese with short despite associating it with hair/short, but we do associate Basil with Nina as well as with owner/Nina.
- small $\leftarrow_0$ little

The above means that small and little are synonyms, and are to be treated as 0 distance away from each other for vicinity calculation purposes. In other, traditional Unix, words, they are hardlinked together.

Creating a serious ontology is not our field or task, but worth doing. The reader is referred to WordNet (free), and Cyc by Doug Lenat (proprietary). While we will focus on implementing primitives that allow for creating better ontologies, we are happy to work with persons interested in contributing or porting an ontology.

Other Projects Seeking To Increase Closure In The OS

ATT’s Plan 9

[Plan 9] is being produced by the original authors of Unix at ATT research labs. It has influenced CORBA, and `/proc` is a direct steal from it to Linux.

Their major focus is on integration.

Their major trick for increasing integration is unifying the name space.

Name spaces integrated into the Plan 9 file system include the status, control, virtual memory, and environment variables of running processes. They have a hierarchical analog to what the relational culture calls constructing views, that the Plan 9 culture calls context binding.

Microsoft's Information At Your Fingertips

Plan 9 ignores integration of application program name spaces, concentrating on OS-oriented name spaces. Microsoft's "Information at Your Fingertips" name space integration effort appears to be taking the other approach, and focusing on integrating the name spaces of the various Microsoft applications via OLE and Structured Storage. The application group at Microsoft has long been better staffed and funded than the OS group, and FS developers have long preferred to simply ignore the needs of application builders generally. The primary semantic disadvantages of Microsoft's approach are primitives selected with insufficient care, a lack of closure, and the use of an object oriented rather than set oriented approach in both naming syntax and data model.

Realistically, one can say that folks within Microsoft have often made a statement favoring name space integration, and in various areas have successfully executed on it, but on the whole I rather suspect that the lack of someone in marketing making a business case for $X in revenue resulting from name space integration has crippled name space integration work at commercial OS producers generally, including MS.

Internet Explorer

Internet Explorer attempts to unify the filesystem and Internet namespaces. At the time of writing, the unity is so surface, with so little substance, that I would describe it as having the look and feel of integration without most of the substance. Perhaps this will change.

Microsoft's Well-Known Performance Difficulties

Despite having many of the leading names in the industry on their payroll, they have somehow managed to create a file system implementation with performance so terrible that it is for the Unix customer base a significant consideration contributing to hesitation in moving to NT. It may well have the worst performance of any of the major OS file systems. Their implementation of OLE's structured storage offers extremely poor performance, and their excuse that it is due to the incorporation of transaction concepts into their design is just a reminder that they did a poor job at that as well. That they managed to implement something intended to store small objects within a file, and implement it such that it still suffers from 512-byte granularity problems, problems that they try to somewhat overcome by encouraging the packing of several objects within "storages" at horrible kludge costs...

Storage Layers Above the FS: A Sure Symptom The FS Developer Has Failed

When filesystems aren't really designed for the needs of the storage layers above them, and none of them are, not Microsoft's, not anybody's, then layering results in enormous performance loss. The very existence of a storage layer above the filesystem means that the filesystem team at an OS vendor failed to listen to someone, and that someone was forced to go and implement something on their own.

You just have to listen to one of these meetings in which some poor application developer tries to suggest that more features in the FS would be nice, I heard one at a nameless OS vendor. The FS team responds to say disks are cheap, small object storage isn't really important, we haven't changed the disk layout in 10 years, and changing it isn't going to fly with the gods above us about whom we can do nothing. At these meetings you start to understand that most people who go into filesystem design are persons who didn't have the guts to pursue a more interesting field in CS. There is a sort of reverse increasing returns effect that governs FS research, in which the more code becomes fixed on the current APIs, the more persons in the field who react with fear to any thought of the field of FS semantics being other than a dead research topic, the less research gets done, and the fewer persons of imagination see a reason to enter the field. Every time one vendor gets a little forward in adding functionality, the other vendors go on a FUD campaign about it breaking standards and therefore being dangerous for mission critical usage. This is a field in which only performance research is allowed, and every other aspect is simply dead. Namesys seeks to raise the dead, and is willing to commit whatever unholy acts that requires.

There is no need for two implementations of the set primitive, one called directories, the other called a file with streams, each having a different interface. File systems should just implement directories right, give them some more optional features, and then there is no need at all for streams. If you combine allowing directory names to be overloaded to also be filenames when acted on as files, allowing stat data to be inherited, allowing file bodies to be inherited, and implement filters of various kinds, then in the event that the user happens to need the precise peculiar functionality embodied by streams, they can have it by just configuring their directory in a particular way. There was a lengthy Linux-kernel thread on this topic which I won't repeat in more detail here.

The tree architecture of the storage layer of this FS design will lend itself to a distributed caching system much more effectively than the Microsoft storage layer, in part due to its ability to cache not just hits and misses of files, but to cache semantic localities (ranges). For more on this topic see later in this paper.

Rufus

The Rufus system [Mea] indexes information while leaving it in its original location and format. While it does allow the user to create a unified name space, it does not choose to integrate that name space into the operating system. Even so, it is immensely useful in practice, and strongly hints at what the OS could gain if it had a more than hierarchical name space with a data model oriented towards what [Mea] calls "semi-structured information", such as you find in the RFC822 format for email. When you have 7000 pieces of mail, and linear searching the mail with a utility like grep takes 10 minutes, it is nice to be able to quickly keyword search via inverted indexes for the mail whose from: field contains billg and that has the words "exclusive" and "bundling" in the body of the message, as you hurriedly search for an old email just before an appearance at court.

Semantic File System

The Semantic File System comes closest to addressing the needs I have described. It is a Unix compatible file system with more than hierarchical naming (attribute based is the term they use). Its data model unfortunately has the important flaw of lacking closure (in it names of objects are not themselves objects). In my upcoming discussion of the unnecessary lack of closure in hypertext products, notice that the arguments apply to the Semantic File System (and so I won't duplicate them here).

OS/400

IBM's OS/400 employs a unified relational name space. The section of this paper entitled *A System Should Reflect Rather than Mold Structure* will cover its problems of forcing false structure. Inadequate closure due to mandatory type checking is another source of difficulties for it. While users moan about these two unnecessary design flaws, the essence of the opinions AS/400 partisans have expressed to me has been that the unification of its name space is a great advantage that OS/400 has over Unix. I claim these users were right, and later in this paper will propose doing something about it.

Conclusion

While I spent most of this paper on why adding structure to information can be harmful, particularly when it is intended to be found by others sifting through large amounts of other information, this was purely because it is a harder argument than why deleting structure is harmful. My goal was not to be better at unstructured applications than keyword systems, or better at structured applications than the hierarchical and relational systems — the goal is to be more flexible in allowing the user to choose how structured to be, while still being within a single name space.

I claimed that multiple fragmented name spaces cannot match the power and ease of name spaces integrated with closure: closure makes a naming system far more powerful by increasing its ability to compound complex descriptions out of simpler ones. The strong points of this naming system's design are various forms of generalizing abstractions already known to the literature, for greater closure.

Acknowledgements

David P. Anderson and Clifford Lynch helped enormously in rounding out my education, and improving my paper. Their generosity with their time was remarkable. David P. Anderson was simply a great professor, and it was a privilege to work with him. Brian Harvey informed me that it wasn't too obvious to mention that an object store should be unified. Cimmaron Taylor provided me with many valuable late-night discussions in the early stages of this paper. I would like to thank Bill Cody and Guy Lohman of the database group at the IBM Almaden Research Center for a wonderful learning experience.

Vladimir Saveliev kept this file system going when others fell by the wayside. He started as the most junior programmer on the team, and through sheer hard work and dedication to excellence outshone all the other more senior researchers. Of course after some time he could no longer be considered a junior programmer.

References

[BM85] David C. Blair and M. E. Marron. Evaluation of retrieval effectiveness for a full-text document-retrieval system. 28:289–299, 1985.

[CW84] Ronald Curtis and Larry Wittie. Global naming in distributed systems. *IEEE Software*, 1(3):76–80, July 1984.

[Dat86] C. J. Date. *An Introduction to Database Systems*. Addison-Wesley Pub. Co., Reading, Mass., fourth edition, 1986. Contains a well-written substantive textbook sneer at the problems of hierarchical naming systems, and a well-annotated bibliography.

[Mea] Eli Messinger and et al. Rufus: The information sponge.

[PPT$^+$93] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom. The use of name spaces in Plan 9. *Operating Systems Review*, 27(2):72–76, 1993. Plan 9 is an operating system intended to be the successor to Unix, and greater integration of its name spaces is its primary focus.

[PT88] Walter D. Potter and Robert P. Trueblood. Traditional, semantic, and hyper-semantic approaches to data modeling. *IEEE Computer*, pages 53–63, June 1988.

[Sal86] Gerard Salton. Another look at automatic text-retrieval systems. *Communications of the ACM*, 29(7):648–656, 1986.

[SFW88] Gerard Salton, Edward A. Fox, and Harry Wu. Extended boolean information retrieval. *Communications of the ACM*, 31(2):170–188, 1988.

[SS77] John Miles Smith and Diane C. P. Smith. Database abstractions: Aggregation and generalization. *ACM Trans. Database Syst.*, 2(2):105–133, 1977.

[VR79] C. J. Van Rijsbergen. *Information Retrieval, 2nd edition.* Dept. of Computer Science, University of Glasgow, 1979.