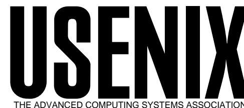
USENIX Association

Proceedings of the General Track: 2003 USENIX Annual Technical Conference

San Antonio, Texas, USA
June 9-14, 2003



© 2003 by The USENIX Association All Rights Reserved For more information about the USENIX Association:
Phone: 1 510 528 8649 FAX: 1 510 548 5738 Email: office@usenix.org WWW: http://www.usenix.org
Rights to individual papers remain with the author or the author's employer.
Permission is granted for noncommercial reproduction of the work for educational or research purposes.
This copyright notice must be included in the reproduced paper. USENIX acknowledges all trademarks herein.

A Logic File System

Yoann Padioleau and Olivier Ridoux
IRISA / University of Rennes 1
Campus universitaire de Beaulieu
35042 RENNES cedex, FRANCE
{padiolea,ridoux}@irisa.fr, http://www.irisa.fr/lande

Abstract

On the one hand, hierarchical organizations are rigid in the sense that there is only one path to each document. On the other hand, keyword-based search is flexible because many sets of keywords may lead to the same document, but it lacks a navigation mechanism. We present the new paradigm of a *logic file system*, which integrates navigation and classification, and the possibility of expressive queries. This paradigm associates logical descriptions to files, and logical deduction serves as a basis for navigation and querying; *paths are formulas*. A key notion is the *extension* of a logical formula: i.e., the set of all files whose description satisfies the formula. The root directory is determined by formula *true*, and sub-directories of a directory are determined by formulas whose extension strictly intersects the directory extension. This gives a logical ground for considering navigation as computing *relevant hints* to help refining a query. A prototype implementation demonstrates encouraging performances.

1 Introduction

Computer users can access a very large variety and number of digital documents. Managing so many documents is a difficult task. An important difficulty is to find quickly one desired document among many files; this is the information retrieval problem. Information retrieval techniques cannot be thought independently of organization methods; they form *paradigms*. The most well-known information retrieval/organization paradigms are the *hierarchical paradigm*, the *boolean paradigm*, and the *relational paradigm*. This introduction continues by presenting the advantages and disadvantages of these paradigms. The Logic File System is an original combination of these three paradigms.

1.1 Hierarchical organizations

Traditional file systems belong to the *hierarchical paradigm*, as well as Web portals such as *Yahoo!*, and many e-mail managers. Information is organized by first creating a hierarchy of *concepts*, which induces a tree-like structure, and then putting *objects* in this tree, which is the classification process. Information is searched by going up and down in this tree; this is the navigation process. In a file system, concepts are *directories*, and objects are *files*.

The advantages of the hierarchical paradigm are:
- the system proposes navigation hints to the user, e.g., directory names. If the user does not remember or does not know the exact classification of an object, then the computer will help him.
- the system proposes relevant hints only. By first proposing global concepts and then sub-concepts, the system helps in finding an object step-by-step.

The disadvantages are:
- since there is a single path to a file, one must know it exactly. This requires either luck or erudition. Gopal and Mander say [8] "Beyond a certain scale limit, people cannot remember locations for explicit path names, after so many years, it is still amusing to see even experienced UNIX system administrators spend time trying `/usr/lib` or was it `/usr/local/lib`, maybe `/opt/local/etc/lib` or `/opt/unsupported/lib`. There are of course many search tools available, but organizing large file systems is still too hard". Our objective is to take into account this *human factor*; users of a file system tend to look for files that they do not know where they are. So, users should not commit themselves to expressing definite paths.
- one can put an object in only one place, which means one cannot associate several independent (i.e., not ordered by the subconcept relation) concepts to one object.

However, one often wants to associate several independent concepts to an object: e.g., price and weight.
• another disadvantage is that the logic of the query language is limited to conjunctive formulas (e.g., a/b for traditional file systems), though disjunction is a clean formal way of expressing non-definite paths. Disjunction can be simulated using shell tricks, such as in

`ls -r */(keyword1|keyword2)/`,

but `cd */(keyword1|keyword2)/` means nothing. Negation is still more difficult to simulate. To express non-definite paths, a user would like to do just `cd !keyword1`, or `cd keyword1|keyword2`.

1.2 Boolean organizations

Web search engines such as Google belong to the *boolean paradigm*. Information (in the Web case, HTML files) is organized by describing information elements with keywords. One finds an information by specifying the keywords that its description must contain. It can be extended by boolean formulas, such as

$$keyword_1 \vee keyword_2 \wedge keyword_3.$$

The advantages of the boolean paradigm are:
• As opposed to the hierarchical paradigm, it is easy to describe an information element by a conjunction of concepts. It is flexible because of the many boolean formulas that are equivalent. E.g., queries $man \wedge ps$ and $ps \wedge man$ are equivalent, as opposed to a hierarchical file systems where `man/ps` and `ps/man` are not equivalent at all.
• The query language is expressive; it permits conjunction, disjunction and negation.

The disadvantages are:
• As opposed to the hierarchical paradigm, the system does not answer a query with relevant keywords; instead, it returns a list of all information elements whose description is partially matched by the query. So, the exact keywords that permit to select an information element must be guessed by the user; the system does not help.
• Since the system answers with a list of information elements the result of a query can be long, which can mean useless, because a user cannot/does not want to go through the entire list to find the desired file. That means that the user needs to find further keywords without help.

So, one would like that a boolean system proposes relevant keywords that will refine the search; one wants a navigation capability with a boolean query capability.

1.3 Relational organizations

Databases belong to the *relational paradigm*. Data is organized by first creating tables, according to a pre-established *schema*, and then by filling in those tables with items. One searches an object by using a complex language based on relational algebra.

The big advantage of databases is the expressiveness of their query language. E.g., one can associate valued attributes to items, and then make complex queries such as

`select ... where size > 45`.

The difficulty is that this expressiveness comes at a price. Databases are more complex systems than file systems, and are less easy to learn (a UNIX-shell course: 1 hour, an SQL course: 40 hours). This is because there are many concepts to learn, such as selection and projection. Moreover, as for boolean organizations, it does not propose navigation, which makes it difficult to find an information without knowing the schema of the database.

1.4 A new paradigm

All paradigms described in this introduction have their advantages and drawbacks. The main contribution of our work is to propose a new organization which makes it possible to combine navigation, querying, ease of use, and expressiveness, and which can be implemented reasonably efficiently. As opposed to previous works (see Section 6 for related works), the combination of querying and navigating is unrestricted. In particular, one can navigate in the result of any query. This organization is an instantiation of a theoretical approach [3, 4] that generalizes file paths to almost arbitrary logic formulas. Information systems based on this approach are called *Logic Information Systems* (*LIS* for short). Once an organization is chosen, the question remains of what will be its place in a system architecture. The proposed organization is implemented as a file system. This makes it usable by all sorts of applications, e.g., shells, editors, compilers, multimedia players. The file system will be called *Logic File System* (*LISFS* for short, and to avoid confusion with the Log-structured File System [16]).

The article is organized as follows. We first present the principles and usage of a logic file system in Section 2. Then we expose an implementation scheme, with its data structures and algorithms in Section 3. We present additional features and extensions to the basic scheme in Section 4. Section 5 describes an actual implementation

and the results of experiments. Section 6 presents related works. Finally, we present future search directions and conclusions in Sections 7 and 8 .

2 Principles of a logic file system

We first describe a file system based on the boolean query paradigm and then we extend it to add the navigation capability. In terms of Gopal and Mander example (see Section 1.1), the boolean file system allows to do `ls lib` to get every library files. With navigation, the answer is not formed of all library files, but of all keywords that can be used to make the query more precise: in their example, `usr`, `opt`, ... Then, the user can do `ls lib/usr`, and the answer will be all files that are fully identified by keywords `lib` and `usr` (in fact, the files of UNIX directory `/usr/lib`), plus keyword `local`, because it helps identifying files that are in `usr/lib/local` and not in `usr/lib`. Keyword `opt` is not listed because it is not relevant to `lib/usr`. In this framework, '`/`' commutes, so that `usr/lib` and `lib/usr` are equivalent.

The boolean file system plus the navigation capability forms the core of the logic file system. We present the semantics of shell commands in the resulting file system. Then we explain how this new paradigm affects the security model. We describe a file system in terms of shell commands, because they are more user-oriented. Only when entering into deeper details in Section 3.3, do we consider actual file system operations, like `lookup` and `readdir`.

2.1 A boolean file system

As we have chosen a standard file system interface we have to deal with files and directories. Boolean properties will play the role of directories. To handle boolean queries in a file system, we associate a conjunction of properties of interest (i.e., a directory) to every file. This is done by first registering property names with command `mkdir`, e.g., `mkdir man; mkdir latex;...` At this stage, this indicates that `man` and `latex` are subdirectories of `/`, but it also indicates that `man/latex` and `latex/man` are paths to potential subdirectories. Second, command `cd` sets the working directory (variable `PWD` in some shells) to a desired list of property names. Finally, commands like `touch` and `cat` will create

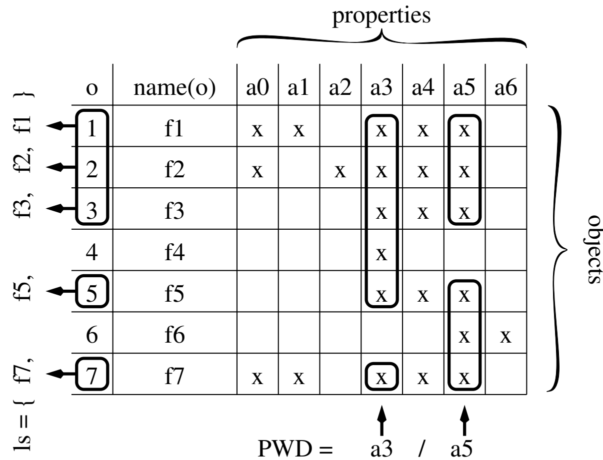


Figure 1: a boolean file system

files associated with the properties named in the `PWD`, e.g., `cd man/latex/french; touch myfile`. Note that `latex` needs not be a subconcept of `man`, nor `french` be a subconcept of `latex`, as in a hierarchical file system. The slash (`/`) must be read as a conjunction, hence it is commutative.

Boolean logic defines a language, i.e., its formulas, and an entailment relation that is usually written $\models$. $f_1 \models f_2$ means that if f_1 is true, then f_2 must be true too. For instance, $a \wedge b \models a$ holds in boolean logic. In the current prototype, formulas must be presented in conjunctive normal form: e.g., a, $\neg a$, $a_1 \vee a_2 \vee a_3$, or $(a_1 \vee a_2) \wedge a_3 \wedge (\neg a_4)$. These formulas are written `a`, `!a`, `a1|a2|a3`, and `(a1|a2)&(a3)&(!a4)` in the concrete syntax. The entailment relation is that of usual boolean logic.

Let $\mathcal{O}$ be the set of all the files in the file system, $d(o)$ be the description of a file o (in our case, $d(o)$ is a conjunction of properties), $name(o)$ be the name of file o, and $c(o)$ be the content of file o. The state of a boolean organization is well-represented by a $name \times property$ matrix (see Figure 1 for an illustration). The answer to `ls` in a `PWD` p is $\{name(o) \mid o \in \mathcal{O}, d(o) \models p\}$. This set is called the *extension* of formula p. Note the risk of name clashes in extensions; several o's with different descriptions, but same name, may clash in an extension. This risk will disappear in the final design of Section 2.2. The root directory, or `/`, is equivalent to formula $true$. So, doing `ls` at the root will list the names of all the files in the system, because anything entails $true$.

The `PWD` evolves incrementally. Given a property `x`, doing `cd x` in a `PWD` `p`, changes the `PWD` into $\texttt{p} \wedge \texttt{x}$. A logic path must always be composed with the `PWD`, tak-

ing into account relative and absolute paths, and special names like `".."`. E.g., doing `cd /y` sets `PWD` to `y`.

As files evolve, their descriptions evolve too. So, one needs a way to update the description of a file. In hierarchical file systems, the place where a file is located is the current description of this file, and when one wants to modify its description, one just changes its location using command `mv`. This works in the same way in a boolean file system. Concretely, if one executes `mv p1/f1 p2/f2`, the effect is to change the name of `f1` into `f2`, and to change its description, deleting attributes of `p1` and adding those of `p2`. E.g., assume file `f` is created with description `man`$\wedge$`latex`$\wedge$`french`, then command `cd /man/ps; mv f ../pdf` results in the new description `man` $\wedge$ `pdf` $\wedge$ `latex` for file `f`.

Executing `rmdir x` removes a property name (i.e., a virtual directory). It first checks that `x` is a simple atom (neither a disjunction, a conjunction, nor a negation), and that this property is empty, i.e., $\{o \mid o \in \mathcal{O}, d(o) \models \mathtt{x}\} = \emptyset$. Finally, `rm` proceeds as in a hierarchical file system.

At this stage, a boolean file system has all the advantages, but also the drawbacks, of a boolean organization. In the following section, we add navigation to the boolean file system. This defines the full *logic file system*.

2.2 Boolean file system, plus navigation

The answer to a boolean query can be very large (typically, `cd /; ls` would list the extension of the root, i.e., every file), so one would like to add a navigation capability to the boolean file system. So doing, the system will answer a query with *relevant* properties that refine the search. Those properties will be presented as sub-directories. By "relevant" we mean that if in a `PWD` p all files have keyword a_1, some have the keyword a_2, and none have the keyword a_3, then neither a_1 nor a_3 is relevant to distinguish between the files, but a_2 is relevant. Command `ls` lists sub-directories, but also files that cannot be distinguished any further by relevant properties. In a logic file system, the files located in a `PWD` p are the files whose description satisfies p, but does not satisfy any of its subdirectories. In the preceding example, if one of the files had just the properties mentioned in p and a_1, and nothing more (i.e., not a_2), then it would be listed.

More formally, let $\mathcal{F}$ be the set of all property names, let $ext(f) = \{o \in \mathcal{O} \mid d(o) \models f\}$ (the extension of f), then

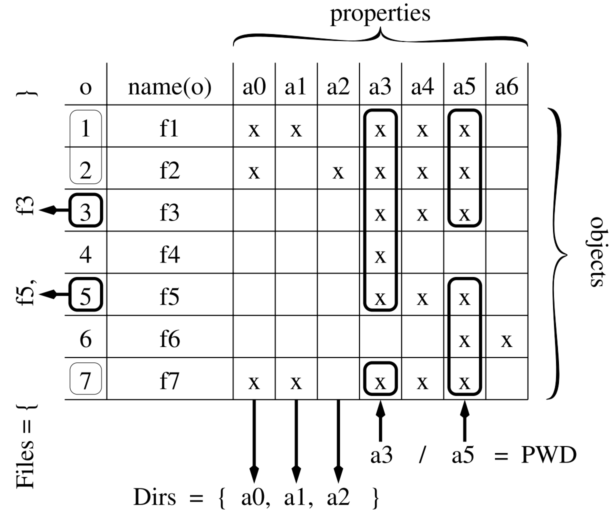


Figure 2: querying and navigating

the answer to `ls` in a `PWD` p is divided in two parts, the sub-directories *Dirs*, and the files *Files*, such that

$$Dirs = \{f \in \mathcal{F} \mid \emptyset \subset ext(f \wedge p) \subset ext(p)\}$$

and

$$Files = \{o \in \mathcal{O} \mid d(o) \models p, \neg\exists f \in Dirs \mid d(o) \models f \wedge p\}$$

(see Figure 2 for an illustration). Sub-directories in *Dirs* are also called *increments*.

In hierarchical file systems, several files may have exactly the same name, provided that they are not located in the same directory; otherwise, the user would not be able to disambiguate which file he wants to manipulate. In the boolean file system, the same problem arises, so one must ensure that if two files have the same name, they must not have exactly the same description. If this condition is kept true, a path always exists where only one of these two files is listed, and so where there is no ambiguity. Navigation always finds this path. For instance, users Alice and Bob may both have a file called `foo`; one file `foo` will have attribute `alice`, and the other one will have `bob`. Alice in her homedir `/home/alice` will see her own `foo`, and Bob in his homedir `/home/bob` will see his own `foo`. A file `foo` may also have attributes `alice` and `bob`; in this case, it is visible by both users.

2.3 Navigation in a taxonomy

With this scheme, the number of file names listed by a command `ls` is reduced, but there may be too many increments anyway. For instance, at the root directory, the system would list nearly all the property names. The principle of navigation is obviously to propose concepts

that refine the search, but also to propose the most general such concepts. E.g., assuming computer science documents, one would like to classify the keywords so that the system first proposes the main fields of computer science, e.g., *algorithms*, *databases*, *operating-systems*, and then the subfields, e.g., *UNIX*, *Linux*, *Windows*.

So, one needs some means for stating that an atomic property is more general than others. To this aim, we use the `mkdir` command to create a hierarchy of concepts: a *taxonomy*. So, to say that a property a_1 is more specific than a_0, one simply executes `mkdir` a_0`;` `cd` a_0`;` `mkdir` a_1, as for files. If an object x has property a_1, then it must also have property a_0. In logical terms, this means that $d(x) \models a_1$ must entail $d(x) \models a_0$. So, the extension of a_0 must contain the extension of a_1. This is achieved by considering the taxonomic relations as axioms, e.g., $a_1 \models a_0$. Note that the taxonomic relation is a *directed acyclic graph* (a *DAG*). E.g., doing

```
cd /OperatingSystem/TradeMark
mkdir Unix
```

makes `Unix` a subconcept of `OperatingSystem` (axiom `Unix` $\models$ `OperatingSystem`), and of `TradeMark` (axiom `Unix` $\models$ `TradeMark`; UNIX is a registered trademark of The Open Group). Now, if a user does `ls /OperatingSystem` all files with property `Unix` will also satisfy this query.

Command `ls` is refined accordingly, by proposing the most general increments. So, the answer to `ls` in a `PWD` p becomes

$Dirs = max_{\models}(\{f \in \mathcal{F} \mid \emptyset \subset ext(f \wedge p) \subset ext(p)\})$,

where

$max_{\models}(\mathcal{F}) = \{f \in \mathcal{F} \mid \neg\exists f' \in \mathcal{F}, f' \neq f, f \models f'\}$,

and *Files* does not change (see Figure 3).

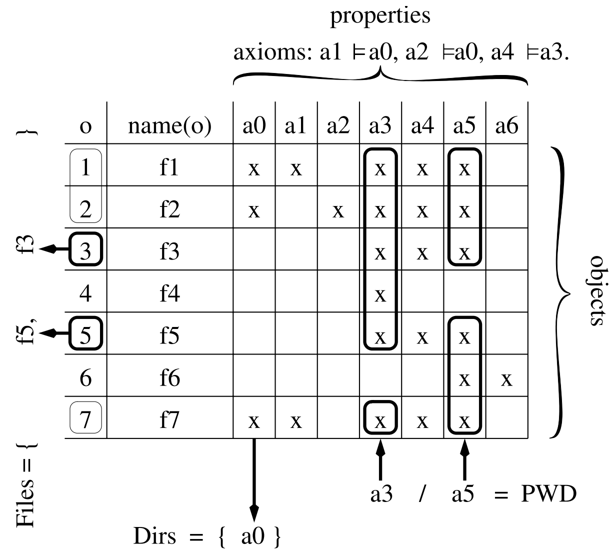


Figure 3: querying and navigating in a taxonomy

2.4 A security model

One of our goals in designing LISFS was to be as much compatible as possible with existing file systems. So, there is no reason to change the way file permissions are handled beyond what is implied by the new navigation paradigm. This is quite easy to do for files, because their nature does not change with LISFS, but we had to design a new security model for directories, because their nature changes a lot with LISFS.

Trying to mimic the way hierarchical file systems handle directory permissions introduces the following question: what are the permission/ownership of a conjunction of property names such as `man/ps`? Under hierarchical file systems, it is the permission/ownership of the last directory on the path (`ps` in the example) that gives the permission/ownership of the whole path. This is sensible since creating a file in `ps` does not modify the content of `man`. However, in the logic file system, the order of property names in a path is immaterial. So, the permissions of `man/ps` should be a conjunction of the permissions of `man` and `ps` to make it certain that, if the owner of `man` has so decided, nobody can create a new object in `man` or `man/ps`. This way of handling directory permissions makes commands that create objects (e.g., `touch` or `mkdir`) safe.

LISFS permits that a user "sees" a file, without specifying all its keywords; it suffices to specify enough keywords so that the file is designated unambiguously. This is like seeing any distant subdirectory in a hierarchical file system. This has strange side effects on security; one can see a file one does not own, from a directory where one has the write permission. This means that, under the traditional security semantics, one could either delete this file, even without the write permission on all the properties describing this file, or add a property to the description of a file (with the `mv` command). So, we refine the security model by allowing only the owner of a file to delete or change the property list of a file. The same kind of difficulty arises in hierarchical file systems with the directory `/tmp`. Every user can create files in `/tmp`, which implies the write permission for all on `/tmp`. However, one does not want that a user deletes the files of another user. The "sticky-bit" solves this difficulty. In a directory having this bit, the system does not look only at the permissions of the directory,

but also at the `uid` of the requesting process in order to check whether the owner of the file is the same user as the owner of the current process.

To summarize,
• `touch` and `mkdir` security semantics rely on the fact that the permissions of a path are the conjunction of the permissions of each path elements.
• `rm`, `rmdir`, `chmod`, and `chown` operations check that the owner of the current process is the owner of the file or property name;
• `mv` checks that the owner of the current process is the owner of the file, and check that one has the write permission on the properties one deletes, and the write permission on the properties one adds.

3 Algorithms and data structures

We will first describe the `LS` algorithm that computes the answer to `ls` as it is the most original LISFS operation. Indeed, it computes the sets *Files* and *Dirs* for getting local files and sub-directories (see definition of *Dirs* and *Files* in Section 2.2), whereas the usual operation performed by `ls` is merely to read a directory file. Moreover, this operation dictates the choice for the data structures that we will present just after. Then we will give an overview of the concrete implementation of a few representatives LISFS operations.

3.1 The `LS` algorithm

Each property name in $\mathcal{F}$ is represented by an *internal keyword identifier* f_i, and each object in $\mathcal{O}$ is represented by an *internal object identifier* o_i. To represent the description $d(o)$ of an object, a table `object->keywords` indexed by internal object identifiers contains lists of internal keyword identifiers (indeed, in this prototype, the description of an object is a conjunction of properties). This corresponds to rows of the *name* × *property* matrix. A formula is represented by a list of a union value, which is either an internal keyword identifier, or the tag `Or` associated with a list of internal keyword identifiers (the operands of the disjunction), or the tag `Not` associated with an internal keyword identifier. The list will play the role of the conjunction. The axioms of the taxonomy are represented as a DAG of internal keyword identifiers implemented as a table `keyword->children` and a table `keyword->parents`. We call it the *taxonomy DAG*. Adding an axiom $x \models a \wedge b \wedge c$ means attaching x as a child of the internal keyword identifiers a, b and c. In this data structure, the top node represents the most general property, i.e., $true$ because anything implies $true$.

Extensions are not computed with table `object->keywords`; instead, they are pre-computed in the inverted table `keyword->objects`. This uses standard indexing techniques, and this corresponds to the columns of the *name* × *property* matrix. The extension of $a \wedge b$ is computed by intersecting a and b extensions, i.e., $ext(a \wedge b) = ext(a) \cap ext(b)$. Similarly, $ext(a \vee b) = ext(a) \cup ext(b)$, and $ext(a \wedge \neg b) = ext(a) \setminus ext(b)$. The underlying assumption for using an inverted table is that the number of attributes per path and file description is small wrt. the number of files. This is confirmed by experiments, and by analysis. Indeed, most of the attributes are computed automatically by functions that do not depend on the number of files (see *transducers* in Section 4).

In fact, the table `keyword->objects` does not contain for every property only the objects described by this property, but also the objects that are described by some sub-property of this property. This amounts to precompute the extensions of all the atomic properties. Then, adding a file with property x requires to update (using table `keyword->parents`) the `keyword->objects` entry of x and of all its ancestors in the taxonomy DAG. This is costly, but we think that the main operation to optimize is consultation rather than creation. So, we prefer to transfer the hard-work to creation commands. Moreover, creation does not require an immediate update; its response time can be good anyway if it leaves the computation to a background task.

So, table `keyword->objects` returns extensions at a constant cost. The `PWD` formula is usually small, because it is often the trace of a navigation by a human being. So, to compute `extension(PWD)` costs only a few accesses ($\|PWD\|$) to table `keyword->objects`. Moreover, extensions can be compressed by using an interval representation (where lists of consecutive elements are represented by their *min* and *max*), and specialized set operations can be used. This is especially useful for the most general properties which contain in their extensions nearly all the objects.

The algorithm first locates the internal keyword identifiers that are present in `PWD`, and computes the extension of `PWD`. Then it searches the taxonomy tree downwards from the root node while the extensions of traversed nodes contain the `PWD` extension. The highest internal keyword identifiers whose extension does not con-

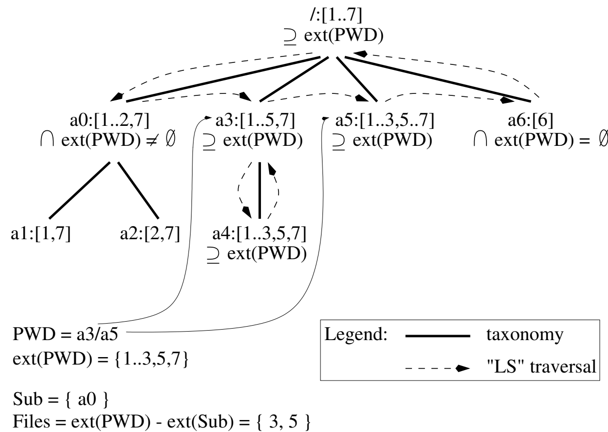


Figure 4: the final algorithm

tain the PWD extension, and has a non empty intersection with it, are the maximal increments to list in *Dirs* (see pseudo-code below). This avoids having to go through all the formulas. The extensions of all sub-properties of a property that does not intersect with the PWD are not considered, because if keyword->objects[x] does not intersect with PWD then no descendant of x will intersect since keyword->objects[x] contains the "inlined" extensions of its children. Similarly, extensions of sub-properties of a property that strictly intersects with the PWD are not considered either, because those sub-properties will also intersect with the PWD but are not maximal increments.

```
extension(f) =
  let set =
    forall x in the list f
      compute intersection of
        either x is a single keyword
          then keyword->objects[x]
        either x is a OR with elements xs
          then
            forall y in xs
              compute the union
                of keyword->objects[y]
        either x is a Not y
          then do nothing
  in ext := set
     forall x in the list f
       if x is a Not y then
         ext := minus(ext,
                  keyword->objects[y])
       otherwise do nothing
    return ext

LS(f) =
  let ois  = extension(f)
  let fis  =
    collect keywords fi
      from the taxonomy DAG
      using a depth first search
      starting from the top node
      using keyword->children
      until
        card = cardinality(
                   inter(extension(fi),
                         ois))
        0 < card < cardinality(ois)
  in Dirs = fis
     Files = minus(ois,
                   union_extension(dirs))
```

Figure 4 illustrates this algorithm. The context is the same as in Figure 3. Every node of the taxonomy is represented by its name and extension (e.g., $/$:$[1..7]$ for the root node). The thick lines represent the taxonomy. The two thin arrows point to the atomic formulas of the current PWD. The dashed arrows represent the actual traversal done by the algorithm for answering command ls. For each traversed node that has subnodes, the algorithm compares the extension of the node with the extension of the PWD; the result of this comparison is written under the node description (e.g., $\supseteq ext(\texttt{PWD})$ for the root node). Only when the result is $\supseteq ext(\texttt{PWD})$ does the algorithm enter the subnodes. Sub-directories are those that do not contain PWD but have a non-empty intersection with it (e.g., a_0). a_6 does not count as a sub-directory because it does not intersect PWD. Files in the extension of PWD but not in a sub-directories are returned (i.e., $\{3, 5\}$).

The complexity of computing *Files* and *Dirs* is essentially the complexity of the product of a $name \times property$ matrix by a vector of properties (the PWD), plus the product of the transposed matrix by a vector of names (the *Files*). The first product computes the *Files* as of the boolean file system (see Section 2.1), and the second product computes the *Dirs*. Assume there are P properties and N names. Both products cost $P \times N$ operations. However, both the matrix and the vector of properties are sparse. Indeed, the number of attributes of a file seldom depends on the number of files. So, one can assume it is constant. This is confirmed by experiments; e.g., the coding of a set of man pages as a logic file system costs about 45 attributes per file whatever the number of files. Let us call p the number of properties per file, $p \ll P$. Moreover, the vector of properties (the PWD) has usually less non-zero elements than p. Let us call it π, $\pi < p$. Then the first product costs only $\pi \times N$. It results in a vector of names with n non-zero elements, $n \ll N$. The second product costs $P \times n$. Finally, since the scalar operations are boolean operations, they have good absorbing properties (e.g., $x \wedge \mathit{false} = \mathit{false}$ whatever is x). This allows

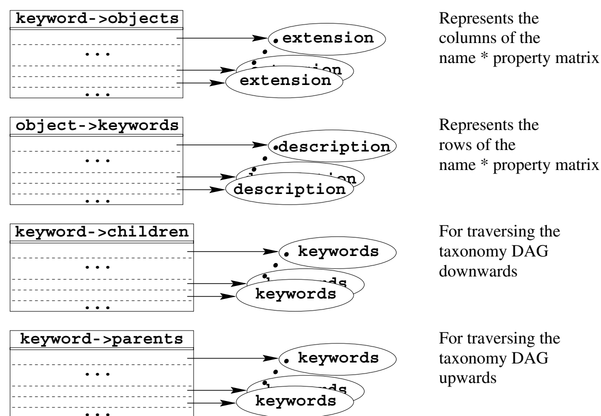


Figure 5: the main tables

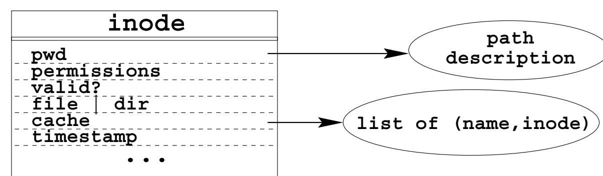


Figure 6: the inode

to short-circuit the computation, without computing every internal products. The essence of algorithm LS is to perform these matrix-vector products, taking advantage of sparseness and taxonomy axioms, and short-circuiting products as soon as possible. In summary, assuming the number of properties per name does not depend on the number of names, the worst-case complexity of algorithm LS is in $\mathcal{O}(N)$. It is much less in practice because of the caching of answers and the short-circuits in boolean operations. Note that the total number of files, N, must be considered with respect to the current PWD. Assume a machine with a million files; this number only affects operations at the root of the file system. As soon as one moves to subdirectories with smaller extensions, N decreases. Section 5 presents the measured cost of LISFS operations on a prototype.

3.2 Data structures

The internal data structures use internal object and keyword identifiers, instead of plain names. This is more space and CPU efficient. In a traditional file system, these internal identifiers are the inode number of a file (or of a directory). In our specification, o's are such internal object identifiers. The meta-data consists mainly in tables `keyword->children` (which represents the taxonomy DAG) and `keyword->objects` (which represents the extension table) used by the LS algorithm. Moreover, in order to preserve the consistency of extensions, command `touch` updates the entries in `keyword->objects` for all the ancestors of the internal keyword identifiers in the current PWD. This is done recursively using a table `keyword->parents`. Similarly, command `rm file` updates the extensions of all internal keyword identifiers used in the description of `file`. This uses a table `object->keywords` which was called $d(o)$ in our specification. All these tables are presented in Figure 5.

In order to return plain names in `ls` answers, we introduce two tables to get the name of a file or property given its internal identifier: `keyword->keywordname` and `object->filename`. As we allow to do `cd` x even if x is not a subdirectory of the current directory, we need to know to what internal identifier x corresponds to, so we add a table `keywordname->keyword`. In order to check, when adding a file, that there is no other file with the same name and same description, we also introduce a table `filename->objects`.

In the UNIX context, a typical user manipulates files with names, whereas the operating system uses an internal identifier: the inode number. The inode structure is presented in Figure 6. Traditional file systems store on disk an `inode_table` (indexed by the inode number) where each entry (aka. the inode) contains the control information to manage a file or a directory, such as its mode, permissions, data block addresses, etc.

Algorithm LS starts from the PWD formula, which it knows only as an inode. So, we add a `pwd` field to directory inodes, which contains a list of a union type. Each union value contains either the internal property identifier of a property, or the tag `Or` with a list of internal property identifiers (the operands of the disjunction), or the tag `Not` with an internal property identifier.

Moreover, answers to `ls` are cached. Directory inodes contain the addresses of blocks that store the result computed by `ls`, which mimics the contents of a directory under a hierarchical file system. As this result must be recomputed every time someone modifies the contents of a LISFS, inodes contain a local timestamp indicating the time the current result was computed. LISFS maintains a global timestamp that is incremented every time someone adds or deletes a file or an atomic property. Operation `ls` compares the global timestamp with the local one, to decide if the increments must be recomputed.

All this means that every time new increments are computed, fresh inodes are allocated, and their local time-

stamp is set to 0 (to force the next call to `ls` to recompute increments). When increments are recomputed, LISFS will fill in storage blocks with the result computed by the `LS` algorithm, and will adjust accordingly the block addresses in the inode.

3.3 Concrete operations

We switch now from shell operations to file system operations to enter in more details. We use Linux *VFS* terminology (*Virtual File System* [10]). To save space, we enter in the details of only a few LISFS operations. Note that these details are often direct consequences of the data structures, so that details of other operations can be inferred for a large part.

For fault-tolerance, we use journaling for implementing *transactions*. Every time several tables must be updated in an atomic way, a transaction is started, which logs the updates. In case of failure, the next reboot will redo (or undo) the log (see [16] on journaling and transaction). For example, renaming a keyword needs to update table `keyword->keywordname` and its inverse table `keywordname->keyword`. This must be done in a transaction to ensure that the two tables are coherent.

File system operations can be divided into three groups: global operations (aka. superblock operations), directory operations (aka. inode operations), and file operations. Global operations deal with the management of the file system: (un)mounting, and statistics. Directory operations deal with navigation/querying, and creation/deletion of files/properties. Finally, the file operations deal with the file contents. Note that in hierarchical file system, operation `readdir` is considered as a file operation, though we consider it as an inode operation. This is because in these systems to `readdir` is actually to read the content of a *directory* file, though in our case it implies a real computation. Directory operations are original wrt. hierarchical file systems, whereas file and global operations are mostly similar to their counterparts in hierarchical file systems.

3.3.1 Global operations

Operation `read_super` is called via the user program `mount`. It locates the different tables on the disk, it stores this information in the superblock structure for further use, and it fills in the first entry in `inode_table` that corresponds to the root inode. The `pwd` field of this entry is set to a list of one element containing a single internal identifier of the top node of the taxonomy DAG which corresponds to property $true$. The local timestamp is set to 0, and the global timestamp is set to 1.

Operations `put_super` (shell command `umount`), `write_super` and `stat_fs` (shell command `df`) are like their counterparts in hierarchical file systems.

3.3.2 Inode operations

Operation `readdir` is called via the user program `ls`. It takes as a parameter an inode, and returns a list of pairs containing the plain name of a file or property and the corresponding inode number. If the global timestamp is equal to the local timestamp of the inode, then the list of pairs is read at the block addresses in the inode. Otherwise, an actual computation must be started: algorithm `LS` (described in Section 3.1) is applied to the `pwd` field of the parameter inode. The result is a pair `Dirs` and `Files` which contain respectively a list of internal keyword identifiers and a list of internal object identifiers. Then, a new buffer is allocated, and for each `oi` in `Files` the pair `(object->filename[oi], oi)` is stored in the buffer. For each `fi` in `Dirs`, a fresh inode number `ino` is allocated, and the pair `(keyword->keywordname,ino)` is also stored in the buffer. Then, for each `ino`, the `pwd` field of `inode_table[ino]` is filled in with the conjunction of `fi` and the current `PWD`, and its local timestamp is set to 0. Finally, previous blocks used by the contents of this directory are freed, new blocks are allocated and filled with the contents of the current buffer, the local timestamp is set to the value of the global timestamp, and the list of pairs contained in the buffer is returned.

Operation `lookup` can be called via the command `cd keyword`. It takes as a parameter an inode (the current directory) and a string `s`, which is the plain name of a file or of a formula, and it returns the corresponding inode, or an error condition. Operation `readdir` is called to look if the string is in the list of pairs, in which case the corresponding inode is returned. Otherwise, the string must correspond either to a property not listed as an increment of the current directory, or to a complex formula. If the string has the form of a single name, then `keywordname->keyword[s]` gives the corresponding internal keyword identifier `fi`. If this entry is empty, an error code is returned, otherwise a fresh inode `ino` is allocated, and the `pwd` field of `inode_table[ino]` is filled in with the conjunction of `fi` and of the `PWD` of the current directory, and inode `ino` is returned. If

the string has the form of a disjunction `x|y|z|...`, the atomic property names are translated into their internal keyword identifier `xi`, `yi`, `zi`, ... If there is a property name without a corresponding internal keyword identifier, then an error code is returned. Otherwise, a fresh inode is allocated as above, putting this time in the `pwd` field a conjunction of the `PWD` of the current directory with the disjunction (with the `Or` tag) of the internal keyword identifier `xi`, `yi`, `zi`, ... If the string has the form of a negation `!x`, then the operation is similar, but putting this time in the `pwd` field the tag `Not`.

Operation `create` can be called via command `touch`. It takes as a parameter an inode (the current directory), a string `s`, and returns the inode corresponding to the object just created. The current directory must correspond to a conjunction, `fis`, of atomic properties (as we have restricted the description of object). This is checked first. Then a similar check is done for the string `s`; it must not contain connectives `|`, `&` or `!`. Finally, another object `y` with the same name and description must not exist. This is checked by looking for an object `y` in table `filename->objects[s]`, and comparing `fis` with the corresponding entry in `object->keywords[y]`. If one of those checks fails, then an error code is returned, otherwise a transaction is started for updating several tables in an atomic way. Then, a fresh internal object identifier `o` is allocated. It is added in `filename->objects[s]`, `object->filename[o]` is set to `s`, `object->keywords[o]` is set to `fis`, and for each keyword `f` in `fis`, `o` is added to `keyword->objects[f]` and recursively in all the ancestor nodes of `f` using table `keyword->parents`. Then, the transaction is ended, and `o` is returned.

Operation `mkdir` behaves the same way as `create`, but this time for atomic properties. It allocates a fresh keyword internal identifier, `fi`. Then, it adds new entries in tables `keywordname->keyword` and `keyword->keywordname`, and it sets to empty `keyword->children[fi]` and `keyword->objects[fi]`, and for each atomic property, `pi`, of the `PWD`, it adds `fi` in `keyword->children[pi]` and `pi` in `keyword->parents[fi]`.

Operation `unlink` undoes what has been done in `create`, checking that the current user is the owner of the file to be deleted.

Operation `rmdir` behaves the same way as `unlink`, but for atomic properties.

Finally, operations `notify_change` and `read_inode` manage the `inode_table`, associating the appropriate permissions to an inode, using the security semantic described in section 2.4.

3.3.3 File operations

Operations `lseek`, `read`, `write`, `open`, `release`, `truncate` are standard. For example, operation `read` takes as a parameter an inode, and a buffer to be filled in. It first gets the inode number `oi`, looks in `inode_table[oi]` to get the block addresses of the contents of the file and fills in appropriately the buffer passed in parameter.

4 Extensions

Section 2 exposed the core of LISFS. In reality this is only a framework, in which more features can be introduced. We present in this section a part of the features that have actually been introduced in the prototype LISFS. A more complete account, particularly on ACL-like security and variants of navigation, is given in a research report [14].

LISFS includes the possibility of valued attributes. For instance, one can create a directory `author:minsky`, or `size:45`. Since related properties can be grouped in a taxonomy DAG, one can gather all properties of the form `size:x` as sub-concepts of the property `size`. This increases the readability of `ls`, which will propose first the coarse categories (`size`, `author`, ...) and then the finer sub-categories, that is the valued attributes (`size:1`, `size:2`, ...). LISFS supports operations on integer attributes, e.g., `cd size:>45` and string attributes, e.g., `cd auteur:=~ m.?in.+y.*`. We simulate such a query by constructing a disjunction of all the existing properties that satisfy the condition, e.g., `cd (size:46|size:72|...)`. This is done in the `lookup` operation.

As many properties can be automatically inferred from the file contents, we designed *transducers* which are functions that extract attributes and values from file contents (as in the Semantic File System [7]). If attributes are new, they are created on-the-fly. In our prototype, transducers are defined for all the system attributes of a file, e.g., its size and last modification time, but also for its extension, e.g., `ext:c` is extracted from

`foo.c`. For the sake of experimentation, we have also defined a transducer for MP3 music files, which extracts the `genre`, `author` and `year` from the meta-data encoded in the file. This permits requests like `cd genre:Disco/year:1980`. We could also easily define other transducers to support more file types, but we prefer to offer the user a simple interface to define his own transducers.

Conceptually, the description of a file is split in two parts: the *extrinsic* part, made of properties assigned by the user, and the *intrinsic* part, made of properties inferred by transducers. As the content changes, the intrinsic part changes too. This is done in operations `release` and `notify_change`. Intrinsic attributes are not updated by the `read` or `write` operations. Indeed, calling the transducers is costly, and we prefer to update the intrinsic attributes only when the user closes a file, that is when doing `release`. To update the intrinsic part, each transducer is called in turn, with the contents, name and system attributes of the file as arguments.

	Andrew	MP3	Man
NbFiles	860	633	11502
FSize	10 Mb	1772 Mb	246 Mb
TSize	2 Mb	3.1 Mb	43.3 Mb
AvNbAttr	26/23	36/20	21/24
NbAttr	1686	3730	43442
AvFSize	11.6 Kb	2799 Kb	21.4 Kb
SpOverH%	20 %	0.17 %	17.6 %
SpOverH/F	2.3 Kb	4.9 Kb	3.7 Kb
SpOverH/A	1.2 Kb	0.84 Kb	1 Kb
SpOverH/FA	47 bytes	87 bytes	84 bytes

NbFiles = Number of files, FSize = Total size of files, TSize = Total size of LISFS tables, AvNbAttr = Average number of file attributes (intrinsic/extrinsic), NbAttr = Total number of attributes, AvFSize = Average file size, SpOverH% = Space overhead (per cent), SpOverH/F = Average space overhead per file, SpOverH/A = Average space overhead per attribute, SpOverH/FA = Average space overhead per attribute of file.

Table 1: Results of Disk Space Benchmark

5 Experimental results

The current prototype of LISFS is implemented as a user level file system, using PerlFS. We use Berkeley DB [13] to manage the different meta-data (implemented as Btrees). The transactional module of Berkeley DB provides the necessary tools for preventing the possible corruption of meta-data by a crash. Finally, we use the underlying file system (EXT2) to manage the contents of files and tables.

Since there is no similar file system to compare with, a part of our experimentations aims at evaluating the overhead of LISFS with respect to a similar technology file system that implements the standard semantics. Since the LISFS prototype is built upon EXT2 and PerlFS, we evaluate the overhead with respect to these file systems. Another part of our experimentations aims at evaluating the performance of LISFS for tasks it has been designed for, like information retrieval. In this case, it is compared with a user-level application that performs the same task, like command `find`. We ran several experiments to determine the overhead of using LISFS, both in speed and disk space. The platform for all experiments was a Linux box running kernel 2.4, with a 2Ghz Pentium 4, 750Mb RAM, and a 40 Gb IDE disk.

The first experiment evaluates the disk space overhead used by the meta-data of LISFS (see Table 1). The experiment is run for a set of 633 MP3 music files, a set of 860 program files obtained by ten copies of the Andrew file system benchmark [9], and a set of 11502 man pages. The Andrew files are described by intrinsic attributes valued by the names of functions declared in them (as produced by command CTAGS). The MP3 files are described by intrinsic attributes valued by MP3-specific meta-data such as `genre` and `artist`. The man pages are described by the words of their description line. All have extrinsic attributes for ACL-like security and for representing their position in a user-defined hierarchy, plus other intrinsic system attributes for size, last modification time, etc. SpOverH/F measures the average cost of file descriptions (i.e., rows of the $name \times property$ matrix). SpOverH/A measures the average cost of each attribute (i.e., extensions, or columns of the matrix). SpOverH/FA measures the average cost of each individual attribute of each file (i.e. each non-empty position in the matrix).

In the second experiment, we ran the modified Andrew benchmark, first on the native file system (EXT2), then on a hierarchical file system implemented via PerlFS, then on LISFS where transducers were turned off then on (see Table 2). The Andrew benchmark has 5 phases: *Mkdir* constructs a directory hierarchy, *Copy* copies files, *Scan* recursively traverses the whole hierarchy, *Read* reads every byte of every file, and finally *Make* compiles the files. Note row *Read* where LISFS without transducer is faster than PerlFS because PerlFS goes into empty directories that LISFS avoids because they are not relevant. We also ran our own benchmarks that consists in creating

	Ext2	PerlFS	LISFS (transducer off)	LISFS (transducer on)
Mkdir	0.217s	0.986s	1.823s	3.703s
Copy	1.359s	5.943s	13.212s	46.296s
Scan	2.506s	5.141s	5.348s	6.638s
Read	3.548s	11.510s	11.119s	12.333s
Make	16.896s	28.384s	36.182s	46.260s
Total	24.526s	51.964s	67.684s	115.230s
MP3	2min28s	4min30s	5min	5min30s
Man	22min	29min	44min	85min

Table 2: Results of CPU Benchmark

the 633 music files, and creating the 11502 man pages. In both cases, creating also means indexing through the use of adhoc transducers.

We now compare the speed of search-like activities using UNIX `find` and `apropos`, and LISFS `lookup`. With the Andrew files `find`'ing function *GXfind* takes 2.899 seconds, but `look`'ing it up takes 0.081 seconds. Similarly, with the MP3 files `find`'ing *disco* music takes 3.292 seconds, but `look`'ing it up is immediate. Doing `ls change` with the man pages takes 1.370 seconds, and returns 110 increments. Doing `apropos change` takes 0.145 seconds, and returns 288 items. In other words, the increments reveal the organization of the items. Doing `ls change/directory` with the man pages takes 0.026 seconds, and returns 4 entries. Doing `apropos change | grep directory` takes 0.030 seconds, and returns the same 4 entries. We have also tested the speed of `ls` in directories of various sizes in the music files context. In directory `artist`, `ls` computes 155 increments in 0.405 seconds. In directory `size`, it computes 629 increments in 1.058 seconds, and finally it computes 28 increments in directory `genre` in 0.220 seconds. Note that in any case, only the first `ls` in a given directory takes time, as for further `ls` LISFS uses its cache, and answers immediately.

Figure 7 plots the creation times of the 633 MP3 files and 11502 man pages. It shows an almost constant creation time until 10000 files, and then a deterioration. We believe that a better representation of extensions will push this point to a greater value.

In summary, the space overhead is tolerable, and it could still be decreased by using better marshalling techniques. Operations `lookup` and `readdir` do not show a great performance penalty, especially considering the work they perform. Moreover, they compare positively with search tools like `find`. On the opposite, operation `create` suffers from a large performance penalty, because the extensions of the taxonomy DAG must be checked and updated. This is visible in the *Copy* row of Table 2. It can be cured by performing most of the `create` operation lazily, in the background during idle time. Indeed, the result returned by `create` does not depend on the updating of internal tables. We need also study alternative representations of the taxonomy DAG.

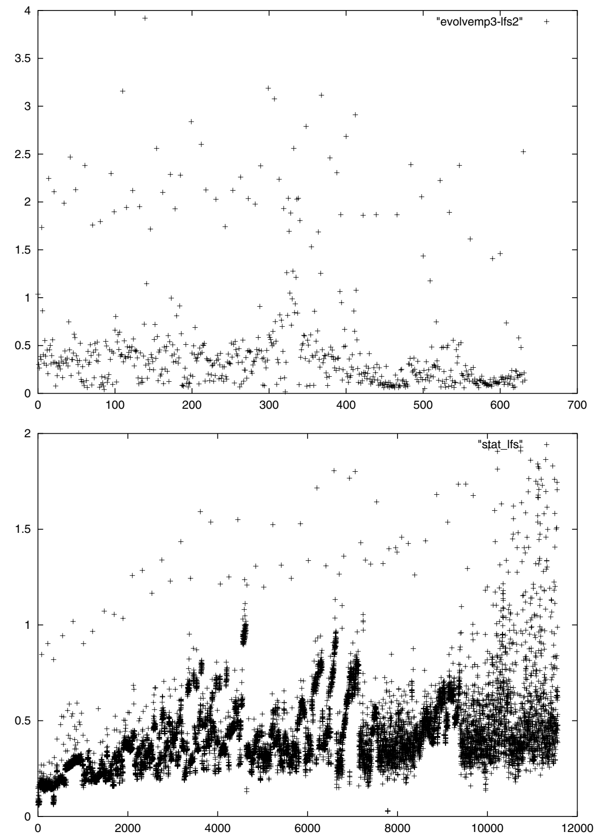


Figure 7: creation times (sec)

6 Related work

The Semantic File System (SFS [7]) was the first file system to combine querying and navigation. It remained compatible with the file system interface by using *virtual directories*. Some attributes were extracted from the contents of the files by *transducers*. This allowed users to express queries such as `cd ext:c/size:15`. However, users could not assign their own attributes to a file (i.e., all these attributes were *intrinsic*, see Section 4). More importantly, querying and navigation were two separated operation modes; one could not navigate

in a virtual directory that resulted from a query. The Hierarchy and Contents proposal [8] solved this problem in a way that leads to inconsistencies. Directories that represent answers to queries are no longer virtual; they are real directories in which one can navigate, and even write. However, one can write something which is inconsistent with the query that created the directory. Another file system that combines querying and navigation is the Be file system (BeFS [6]). Following the observation that hierarchical file systems fail to describe a file by a conjunction of independent concepts, BeFS allows the user to manually assign attributes to a file. But this extension is not compatible with a standard UNIX interface (as opposed to SFS and LISFS where one can use classical shells). So, one can either use a shell or browser and navigate, or use the new interface and do querying, but not both. Finally, the Nebula file system [1] allows a user to assign multiple attributes to a file and formulate query at the shell level. One can also create kinds of directories called *views*, which are just names assigned to queries, as for databases. Views can be organized in a hierarchy, where subviews refine parent's views with another query (restricting the set of objects). This allows to cache frequently used queries. One can both navigate following the subviews links, and query the file system, but as for SFS, one cannot navigate in the result of a query.

The principal contribution of our work is a seamless combination of querying and navigation, under the file system interface. The key features are to combine intrinsic and extrinsic descriptions, and to permit navigation in query results by computing relevant subdirectories.

This idea of combining querying and navigation via increments is not new. In the literature, increments are also called *co-occurrence lists*, *term suggestions*, *relevant informations*, *significant keywords*, ... In [11], a corpus of keywords is extracted from man pages, and via formal concept analysis [5], a lattice of keywords is computed to permit a user to find man pages by keywords, and getting as a result other keywords considered relevant (as increments in Section 2.2). Queries are limited to conjunctions of keywords, and keywords cannot be ordered which mean we get the disadvantages mentioned in Section 2.3. In [15], a similar approach is applied to bibliographic information retrieval. The querying mechanism of these applications is completely encompassed by LISFS; the answers of LISFS *are* the answers of formal concept analysis. In fact, the domain of information retrieval is aware of the need for integrating querying and navigation (e.g., see [2, 12]). However, the proposals in this domain remain at the application level, and are very often combined with visual interface issues.

All those systems are limited to conjunctive queries, and are more like front-ends over another information system for allowing to combine querying and navigating, which means that they do not handle updating in their interface. Our contribution is to offer all these services, querying, navigating, and updating, at the system level, so that many kinds of application can be built on it.

7 Future directions

There is much room for performance improvement in the prototype LISFS. E.g., operations `create` and `readdir` are too expensive. Tricks such as grouping of commands (as in X Window), amortization, lazy structures or use of idle time will certainly improve the performance of `create`. Finally, in place of a global timestamp, locating more precisely what is dirty could lead to less cache miss. Another future work is to make a "complete" logic file system, allowing arbitrary formulas in object descriptions as well as in queries. This requires to incorporate an automatic theorem prover for representing the $\models$ relation in LISFS. The goal here is to permit an open-ended range of file description styles. Even if some logics are undecidable (e.g. predicate logic) or unpracticable (e.g., propositional logic), there are many useful and practicable logics that could be used as a file description language: e.g., a logic of types for program components, or a logic of intervals for expressing dates. Another direction is to overcome the difference between directories and files. We would like to navigate in files in the same way as in directories. E.g., one would like to navigate inside a BibTeX file, or inside a program source file. Then, a user could do `cd !comment & security` to get all the parts of a source file that are not comments and that talk about security.

8 Conclusions

We have presented a new file system paradigm which allows to freely combine high-level file description and querying using logic formulas, navigation, and updating. This is called a *Logic File System*. The integration of querying and navigation goes beyond previous proposals; coherence is kept when writing in virtual directories, and navigation and querying can be freely intermingled. Such a file system gives at a system level services that are useful in many applications. A key technical aspect is to develop data structures and algorithms that permit

to implement a prototype LISFS with encouraging performances. Experiments show that though the prototype LISFS is slower than a more classical one, it passes usual benchmarks with reasonable performances: `create`-intensive benchmarks show a bad performance ratio for LISFS, but `ls`-intensive benchmarks show almost no penalty. Consider also that what operation `create` actually does is on-line indexing. Note also that the current implementation of LISFS is very *soft*, a user-level program based on PerlFS, and it could be improved by using more effective techniques. We believe that all this confirms the validity of LISFS.

9 Acknowledgments

We thank the anonymous referees and our sheperd, Prof. Darrell Long, for their thoughtful remarks. We also wish to acknowledge the collaboration with Sébastien Ferré for elaborating the theory of Logic Information System.

10 Availability

A prototype LISFS and more information on this project can be down-loaded at the following URL:

`http://www.irisa.fr/LIS`

References

[1] C.M. Bowman, C. Dharap, M. Baruah, B. Camargo, and S. Potti. A File System for Information Management. In *ISMM Int. Conf. Intelligent Information Management Systems*, 1994.

[2] P. Bruno, V. Ehrenberg, and L.E. Holmquist. Starzoom - an interactive visual interface to a semantic database. In *ACM Intelligent User Interfaces (IUI) '99*. ACM Press, 1999.

[3] S. Ferré and O. Ridoux. A file system based on concept analysis. In Y. Sagiv, editor, *Int. Conf. Rules and Objects in Databases*, LNCS 1861, pages 1033–1047. Springer, 2000.

[4] S. Ferré and O. Ridoux. Searching for objects and properties with logical concept analysis. In *Int. Conf. Conceptual Structures*, LNCS 2120. Springer, 2001.

[5] B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer, 1999.

[6] D. Giampaolo. *Practical File System Design with the Be File System*. Morgan Kaufmann Publishers, 1999.

[7] D.K. Gifford, P. Jouvelot, M.A. Sheldon, and J.W. O'Toole Jr. Semantic file systems. In *13th ACM Symp. Operating Systems Principles*, pages 16–25. ACM SIGOPS, 1991.

[8] B. Gopal and U. Manber. Integrating content-based access mechanisms with hierarchical file systems. In *3rd ACM Symp. Operating Systems Design and Implementation*, pages 265–278, 1999.

[9] H.J. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, N. Sidebotham, and M. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, 1988.

[10] S.R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *USENIX Summer*, pages 238–247, 1986.

[11] C. Lindig. Concept-based component retrieval. In *IJCAI95 Workshop on Formal Approaches to the Reuse of Plans, Proofs, and Programs*, 1995.

[12] R. Miller, O. Tsatalos, and J. Williams. Integrating hierarchical navigation and querying: A user customizable solution, 1995.

[13] M.A. Olson, K. Bostic, and M. Seltzer. Berkeley DB. In *FREENIX Track: 1999 USENIX Annual Technical Conference*, pages 183–192, 1999.

[14] Y. Padioleau and O. Ridoux. A logic file system. Research Report RR-4656, INRIA, 2002.

[15] G.S. Pedersen. A browser for bibliographic information retrieval based on an application of lattice theory. In *ACM-SIGIR'93*, pages 270–279, 1993.

[16] M. Rosenblum and J.K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.