A Logic File System

Yoann Padioleau, Olivier Ridoux

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A Logic File System

Yoann Padioleau , Olivier Ridoux

N˚4656

Décembre 2002

THÈME 2

ISSN 0249-6399 ISRN INRIA/RR--4656--FR+ENG

RENNES

A Logic File System

Yoann Padioleau , Olivier Ridoux

Thème 2 — Génie logiciel
et calcul symbolique
Projet Lande

Rapport de recherche n ° 4656 — Décembre 2002 — 26 pages

Abstract: We present the new paradigm of *logic file systems*, its implementation, and first experimental results. It offers in an integrated way navigation and classification, the possibility of expressive queries, ease of use, and possible heterogeneity of data. This paradigm is object-centered. It associates logical descriptions to objects, and logical relations between descriptions serve as a basis for navigation and querying. We compare logic file systems with the hierarchical, boolean, and data-base paradigms. We present briefly the rôle of logic in logic file systems, and in more details the implementation issues of a particular logic file system that uses a simple logic.

Key-words: file system, information retrieval, applied logic

(Résumé : tsvp)

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Un système de fichiers logique

Résumé : Nous proposons le nouveau paradigme de *système de fichiers logique*, sa mise en œuvre, et les premiers résultats expérimentaux. Les systèmes de fichiers logiques offrent de façon intégrée des possibilités de navigation et classification, d'utilisation de requêtes expressives, d'usage simple, et de gestion de données hétérogènes. Ce paradigme est centré sur les objets. Il leur associe des descriptions en logique, et les relations logiques entre les descriptions servent de base pour naviguer et questionner dans le système de fichiers. Nous comparons les systèmes de fichiers logiques et les paradigmes hiérarchiques, booléens et bases de données. Nous présentons brièvement le rôle de la logique dans un système de fichiers logique, et en plus grand détail la mise en œuvre d'une instance de ce nouveau paradigme dans un système de fichiers qui utilise une logique simple.

Mots-clé : système de fichiers, recherche d'information, logique appliquée

Contents

1 Introduction

Nowadays, every computer user can access a very large variety and number of digital documents (e.g., music files, photos and videos for personal usage; programs, specifications, tests, for a software engineering usage). Managing so many documents implies some difficulty. Among them, an important one is to find quickly one desired document among many files; this is the information retrieval problem. Indeed, with thousands of files, one cannot use the naive method of going through all the files.

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. Logic file systems are an original combination of advantages of these three paradigms.

1.1 Hierarchical organizations

Traditional file systems belong to the *hierarchical paradigm*, as well as Web portals *à la Yahoo!*, and many e-mail managers.

Information is organized hierarchically by first creating a hierarchy of concepts, which induces a tree-like structure, and then putting information elements 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.

The principle of navigation/classification is concrete and intuitive since it merely generalizes how many things in the physical world are organized. For example, books are a hierarchical organization of parts/chapters/sections/subsections.

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. It would not be helpful of the system to propose all concepts at once, because the total number of concepts can be as big as the number of files. By first proposing global concepts (e.g., part names) and then sub-concepts (e.g., chapter names), the system helps in finding an object by incremental query. The classification principle is the basis of many fast search algorithms, such as binary search, and it is extensively used in real life.

The disadvantages are:

- one can put an information in only one place, which means one cannot associate several independent (i.e., not ordered by the subconcept relation) concepts to one information. However, one often wants to associate several concepts to an information, which means that one is forced to choose one concept to classify an information, and forget about the other concepts. The problem is that in the future, a user may want to access this information using a forgotten concept. So, the file is virtually lost.

  Moreover, as information elements may differ from each others only by a few aspects, not being able to precisely describe them (by mentioning all the concepts they belong to) means that many information elements will be put together in the same concept, which means that the user will be forced to explore all those information elements when he wants to select one.

  Hierarchical file systems provide three solutions to associate several concepts to an information: to use subdirectories, to use links, or to put information in file names (see below). However, this comes at a price; all the advantages of navigation are lost.

  Indeed, assume one wants to classify computer science documents by *year*, *author*, *genre*, and *type of file*, as in figure 1:

  - (1a) and (1b) deviate from the principles of concept/subconcept by ordering orthogonal concepts. This has several drawbacks. First, the number of directories, like *minsky* or *ps*, explodes, which leads to an over-complex creation and management. One can still write a script, but it requires special skills from a user, which we cannot assume.

    Second, some queries will be abnormally difficult to formulate. Assume one searches for a document and knows it talks about algorithms, *alg*, and it is a Postscript file, *ps*, but does not know any other facet of the document. In (1a), one can do first `cd alg/ps`, and then `ls -r` to get the list of all relevant documents but, yet again, if one wants to refine one's search by providing the author's name this structure will not help at all. It would require more coding. In (1b), one can only use the "try and see" technique; try first `cd minsky/ps/78/alg`, then `cd knuth/...`, etc, which is tedious, or use shell tricks as in `ls -r */ps/*/alg`. The disadvantages of those tricks are:

    * the result is no longer a directory. One cannot do `cd */ps/*/alg`, which means that this pseudo-directory cannot be navigated.
    * it requires to know that *ps* is in the second place in file paths, and that *alg* in the fourth place. It requires some discipline to maintain those rules which must be documented and followed by every user who wants to add new files[1]. In fact, in hierarchical file systems this

[1] Note that FHS (Filesystem Hierarchy Standard) is precisely this kind of document. However, it only proposes a standard usage for directory names like *usr*, *etc*, and *bin*, but says nothing about application level hierarchies.

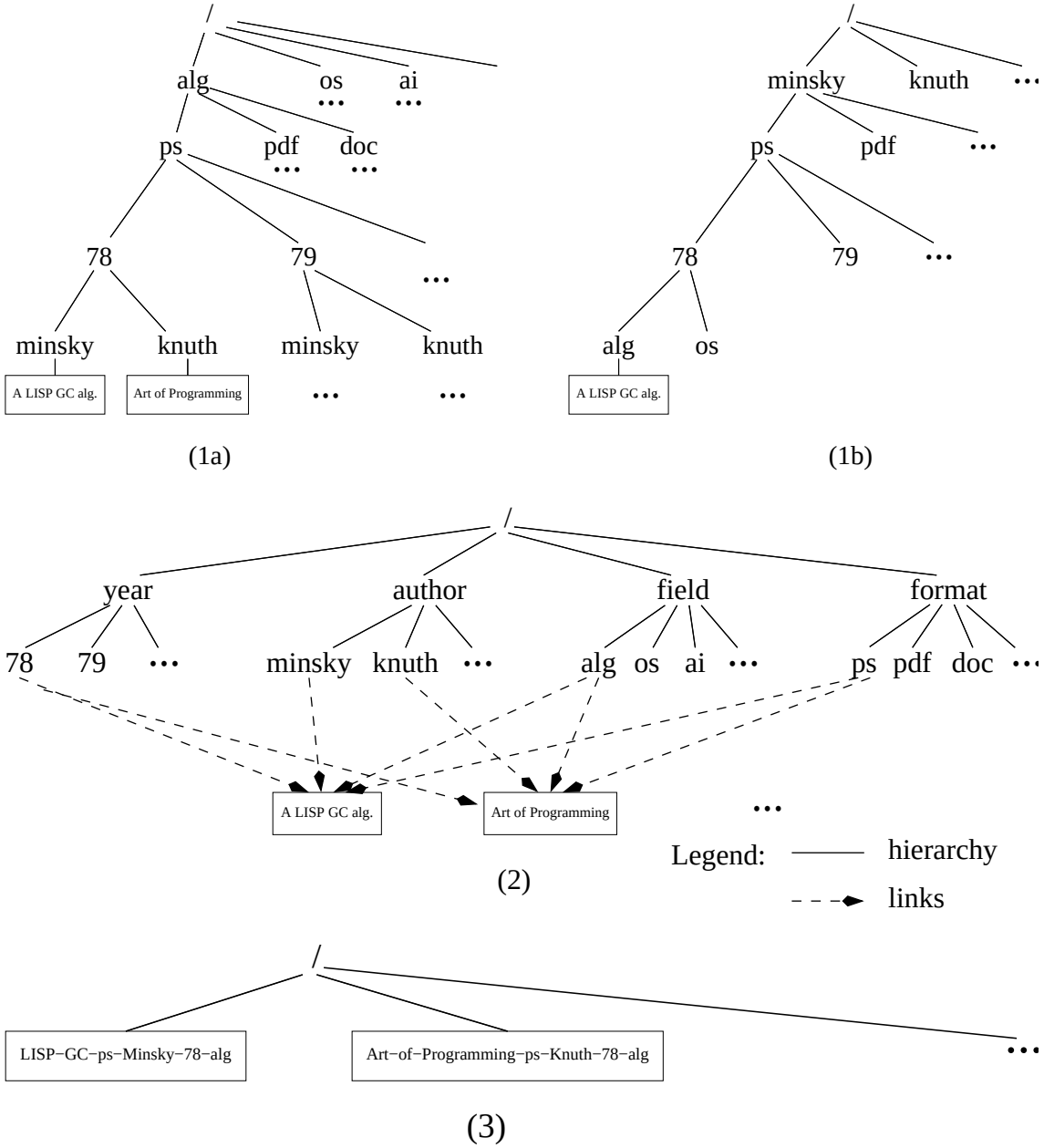

Figure 1: simulating a conjunction of concepts

is rarely followed. For example, *bin* can be found in first position (*/bin*), second position (*usr/bin*), etc. Gopal and Mander says [GM99] "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". In this example, users try to associate orthogonal concepts to a file: *usr/notusr*, *local/notlocal*, *supported/notsupported*. To do the trick we used previously, it would require to reorganize file hierarchies, and to put a file in *notlocal/notusr/lib/supported* which is tedious. It is simpler that */lib* means implicitly *notlocal/notuser/lib/supported*.

  - the link solution (2) looks practicable, but it makes it tedious to create files, and make links manually. More importantly, one cannot even do a conjunctive query such as `cd alg/ps`. It would require to first do `ls alg`, then `ls ps`, and then to calculate the intersection of those 2 directories. With this solution, one loses again the power of navigation, because the system does not propose further concepts to refine the query.
  - finally the name solution (3) consists in encoding the properties of the file in its name. In fact, it just amounts to emulate a boolean organization in a hierarchical system. One can use *find* and *grep* to select the file but as we will see in Section 1.2, but the navigation capability is lost.

  We can conclude that in practice hierarchical organizations fail to allow the user to classify an information in several independent concepts.

- another disadvantage is that the logic of the query language is limited to conjunctive formulas (e.g., a/b for traditional file systems). Disjunction can be simulated using shell tricks (such as in

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

  but negation is not so easy to simulate. 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 lists of keywords. One finds an information by specifying a list of keywords that its description must contain. It can be extended by boolean formulas, such as $keyword1 \vee keyword2 \wedge keyword3$.

The principle of boolean queries is concrete and intuitive, and generally well-accepted.

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 very flexible because of the many boolean formulas that are equivalent. E.g., queries $alg \wedge ps$ and $ps \wedge alg$ are equivalent, as opposed to a hierarchical file systems where *alg/ps* and *ps/alg* 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 new 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.

Under a hierarchical organization, the principle of ordering concepts and sub-concepts (e.g., directories and subdirectories) allows one to have in the result of a query (e.g., *ls*) a limited number of information elements and of sub-concepts.

So, one would like that the system proposes, from the context, some relevant keywords that will refine the search; that is 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, with names of columns, and then by filling in those tables with items. One searches an object by using a complex language based on relational algebra operations.

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: 1h, an SQL course: 40h). 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 structure of the database, its *schema*.

1.4 A new paradigm

As we have seen, 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 and querying, ease of use and expressiveness, and heterogeneity of description, and which can be implemented reasonably efficiently.

This organization is an instantiation of a theoretical approach [FR00, FR01] that generalizes file paths to almost arbitrary logic formulas. Information systems based on this approach are called *Logic Information Systems* (*LIS* for short). In this article, we have restricted the logic by allowing to create only files with a conjunctive description, where atoms can be ordered manually to form a taxonomy. Despite the restriction to conjunctive descriptions, the user can use disjunctions and negations in search queries. The restriction makes the system simpler, but we plan to allow disjunction and negation in descriptions in future works. However, this restricted instantiation seems rich enough for a first prototype.

Once an organization is chosen, the question remains of what will be its place in a system architecture. The proposed organization will be implemented as a file system. This makes it usable by all sorts of applications, e.g., shells, editors, compilers, multimedia players. Moreover, the file system interface via a shell (*cd* and *ls*) is very easy to use, and it supports intrinsically the heterogeneity of data. The file system will be called *Logic File System* (*LFS* for short).

The plan of this article is 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. In all these sections, we describe logic file systems in terms of shell commands like `cd`, `ls`, and `touch`, because they are more user-oriented. Only when entering into deeper details, do we consider actual file system operations, like `lookup` and `readdir`. Nevertheless, it is a new file system that we present. The shell is only an interface to the file system, and we can use any existing shell. Section 6 presents related works, since it happens that though our answer is new, the original question "How to get the advantages of several information system paradigms in a single paradigm?" is an old one. Finally, we present future search directions in Section 7 and conclude in Section 8.

2 Principles of a logic file system

We will first describe a file system based on the boolean query paradigm, *à la* Google, and then we will extend it to add the navigation capability. The boolean file system plus the navigation capability forms the

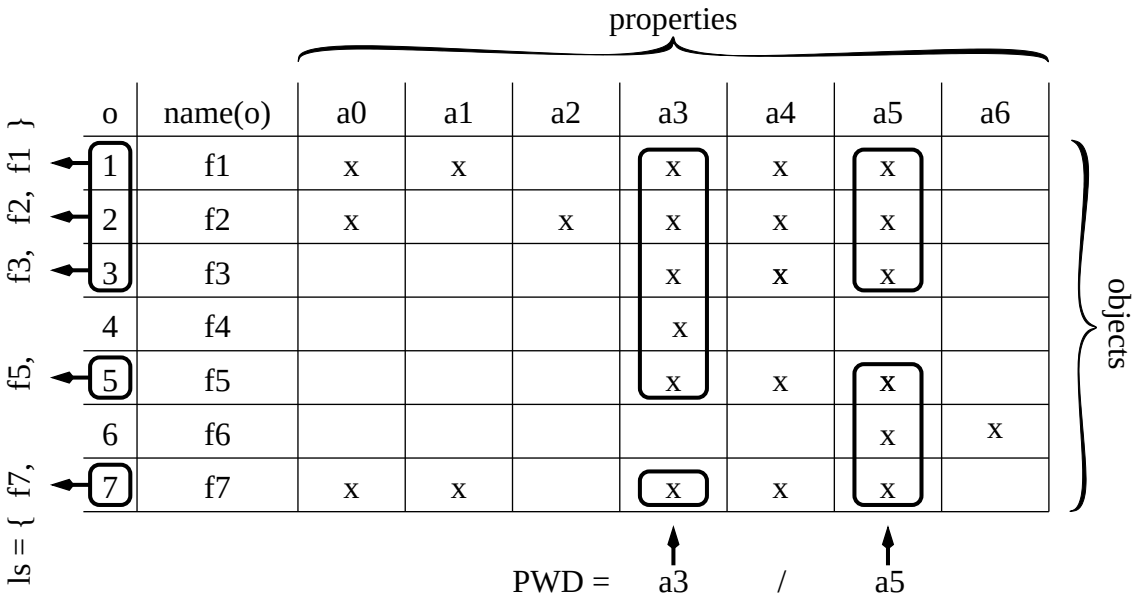

Figure 2: a simple boolean file system

core of the logic file system. We will present what is the semantics of the shell commands `cd`, `ls`, `mv`, `touch`, `mkdir`, `rmdir`, `rm`, `edit` in the resulting file system. Then we will explain how this new paradigm affects the security model.

2.1 A boolean file system

As we have chosen a file system interface we are faced with 2 kinds of objects: files and directories. Boolean properties will play the rôle 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 minsky; mkdir alg;` .... Second, command `cd` sets the `PWD` to a desired list of property names (i.e., it goes virtually to a directory, or goes to a virtual directory). Finally, commands like `touch` and `cat` will create files associated with the properties named in the `PWD`, e.g., `cd minsky/ps/alg; touch myfile`. Note that `ps` needs not be a subconcept of `minsky`, nor `alg` be a subconcept of `ps`, as in a hierarchical file system. The slash (`/`) must be read as a (commutative) conjunction.

In the boolean file system, command `cd` plays the rôle of the dialog box where one enters a Google query, as in `cd alg/ps`. Command `ls` plays the rôle of button "submit", and its answer is the name of all the files whose description *satisfies* the query. So, it remains to formalize what it is for a description to satisfy a query.

A logic is defined by a language, i.e., its formulas, and an entailment relation that is usually written $\models$. $f1 \models f2$ means that if $f1$ is true, then $f2$ must be true too

For instance, $a \wedge b \models a$ holds in propositional logic. In the current prototype, the logic is a restricted form of the logic of proposition. Formulas are either simple atoms (e.g., a), negation of atoms (e.g., $\neg a$), disjunctions of atoms (e.g., $a1 \vee a2 \vee a3$), or conjunctions of formulas (e.g., $(a1 \vee a2) \wedge a3 \wedge (\neg a4)$). The entailment relation is that of usual propositional logic. These formulas are written `a`, `!a`, `a1|a2|a3`, and `(a1|a2)&(a3)&(!a4)` in the concrete syntax.

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 answer to `ls` in a `PWD` p is $\{name(o) \mid o \in \mathcal{O}, d(o) \models p\}$ (see figure 2 for an illustration). This set is called the *extension* of formula p.

The root directory, or /, is equivalent to the true formula. So, doing `ls` at the root will list the names of all the files in the system, because anything entails true.

The `edit` *filename* command works as in other file systems, i.e., it modifies the contents $c(o)$ of a file o such that $name(o) = filename$.

The `PWD` evolves incrementally. Given a property x, doing `cd` x in a `PWD` $p1$, changes the `PWD` into $p1 \wedge x$. If x is `"..."`, one just goes back in the history, as with hierarchical file systems, and if x is $/y$, the new `PWD` is simply y. In summary, a logic path must always be composed with the `PWD` taking into account relative and absolute paths, and special names like `"..."`.

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` p_1/f_1 p_2/f_2 at the root of the logic file system (i.e., `PWD`= /), the effect is to change the name of f_1 into f_2, and to change its description, deleting p_1 and adding p_2. If the same command is executed at an arbitrary `PWD`, the p_1 and p_2 must be composed with this `PWD`, which results in p'_1 and p'_2. Then, command `mv` checks that p'_1 and p'_2 are conjunctions of atoms (as is also done for command `touch`), calculates the common part of p'_1 and p'_2, then calculates the difference between p'_1 and this common part, and suppresses it from the description, and finally calculates the difference between p'_2 and the common part, and adds it to the description. E.g., command `cd /alg/ps; mv f ../pdf` results in the new description `/alg/pdf` for file `f`.

Executing `rmdir` x removes a property name (i.e., a virtual directory). It first checks that (1) x is a simple atom (neither a disjunction, a conjunction, nor a negation), (2) this property is empty, i.e., $\{o \mid o \in \mathcal{O}, d(o) \models x\} = \emptyset$. Finally, `rm` proceeds as in a hierarchical file system.

In hierarchical file systems, we can have many files with exactly the same names, 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, we are sure that a path always exists where only one of these 2 files is listed, and so where there is no ambiguity.

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 a navigation capability to the boolean file system. This leads to defining the full *logic file system*.

2.2 Boolean file system, plus navigation

As said in the first section, the result of a query can be very large (typically, `cd /; ls`), 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 subdirectories. In a hierarchical file system, one can have thousands of files under `/local`, but those files are classified in subdirectories (e.g., `bin`, `doc`) which helps in reducing the size of an answer to `ls` in `/local`.

So, instead of `ls` listing the whole extension of a formula, it will return the remaining *relevant* property names needed to refine the query further, which we call *increments*.

By "relevant" we mean that if in a `PWD` p, there are 100 files, and that those 100 files have all keyword $a1$, some have the keyword $a2$, and none have the keyword $a3$, then neither $a1$ nor $a3$ is relevant to distinguish between the 100 files, but $a2$ is relevant.

Command `ls` lists increments (i.e., sub-directories), but also names of files that cannot be distinguished any further by increments. In the logic file system paradigm, the files located in a `PWD` p are the files whose description satisfies p, but does not satisfy any of its subdirectory. In the preceding example, if one of the 100 files has just the properties mentioned in p and $a1$, and no more (i.e., not $a2$), then its name 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 the answer of `ls` in a `PWD` p is divided in two parts, the increments Sub, and the files $Files$, such that

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

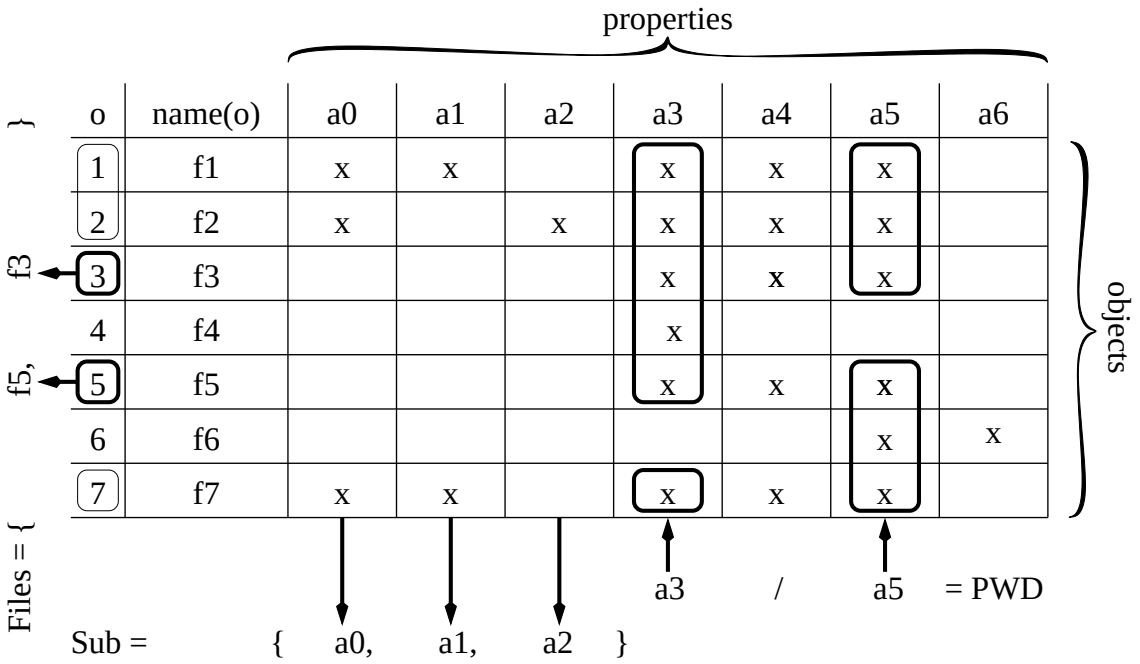

Figure 3: querying and navigating

and

$Files = \{o \in \mathcal{O} \mid d(o) \models p, \neg\exists f \in Sub \mid d(o) \models f \wedge p\}$
(see figure 3 for an illustration).

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 propose 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 we manage computer science documents, we 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 some atomic properties are more general than others. To this aim, we use the `mkdir` command to create a hierarchy of concepts, as in hierarchical file system, but since navigation has changed, this does not re-introduce the disadvantages of hierarchical file systems, namely rigidity of navigation. Indeed, one needs not create subdirectories to express $a \wedge b$; it suffices to do `mkdir /alg;mkdir /ps` and then `cd alg/ps` or `cd ps/alg`. We are now able to create a hierarchy of concepts, and create files that belong to many concepts.

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. Then, a property name has a list of all its parent properties (i.e., more general properties) in its description. I.e., doing `cd /`a`/`b`/`c`; mkdir` d makes d a subconcept of a, b, and c. Note that the subconcept relation is a *directed acyclic graph*, a *DAG*.

As a_1 is more specific than a_0, it means that if an object x has property a_1, then he must also have property a_0. Indeed a document about Unix is also a document about operating system, which is again also a document about computer science. This means that every document which is in the extension of the property $Unix$, must be also in the extensions of the properties *operating systems* and *computer science*. To achieve this transitivity, we wanted that if $d(o) \models Unix$ then $d(o) \models OperatingSystems$, by adding an axiom $Unix \models OperatingSystems$ we ensure this.

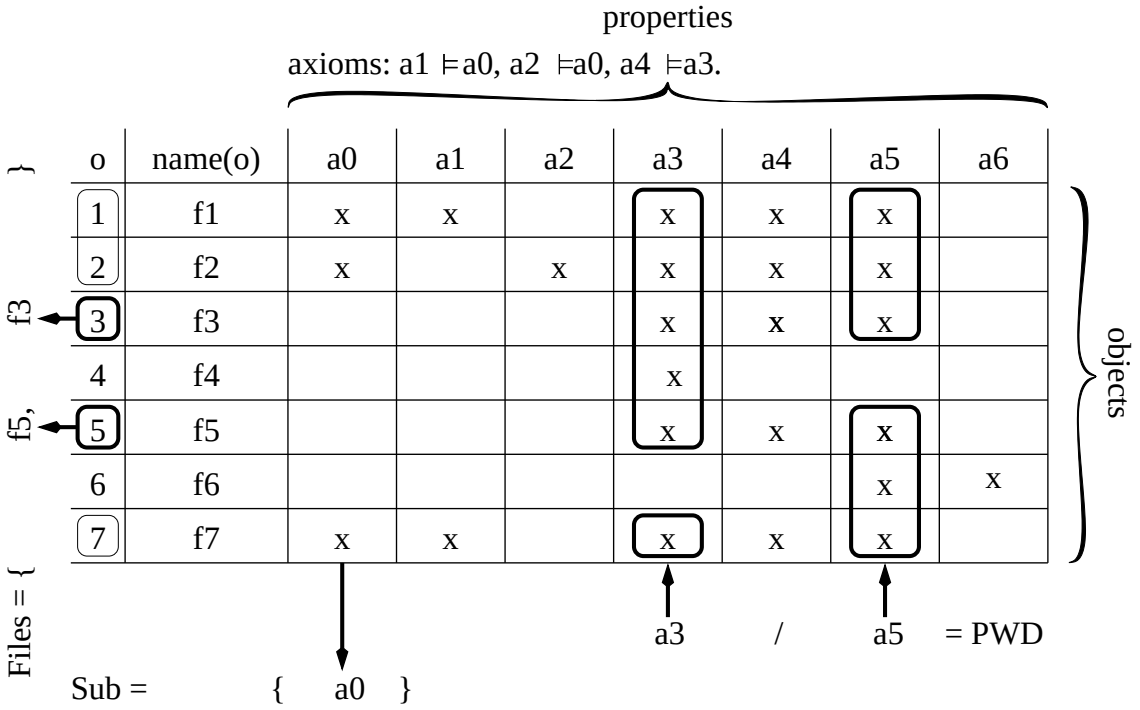

Figure 4: querying and navigating in a taxonomy

More formally, `mkdir` x in a `PWD` p (1) checks that p is a conjunction of atoms, and (2) adds the axiom $x \models p$. Command `ls` changes accordingly, by proposing the most general increments. So, `ls` in a `PWD` p becomes

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

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 4 for an illustration).

2.4 A security model

One of our goals in designing LFS 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. When a user wants to modify the content of a file, then the file system checks the permission/ownership of the file, and the permission/ownership of the current process, and decides whether the operation is permitted. This is quite easy for files, because their nature does not change with LFS, but we had to design a new security model for directories, because their nature changes a lot with LFS.

2.4.1 Security semantic of *touch* and *mkdir*

Trying to mimic the way hierarchical file systems handle directory rights introduces the following question: what are the permission/ownership of a conjunction of property names such as *alg/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 *alg*. However, in the logic file system, the order of property names in a path is immaterial. More subtly, adding a file in *alg/ps* affects the result of future *ls* operations in *alg*. For instance, command `cd /alg; ls` prints *ps* as a subdirectory, since *ps* is now an increment. So, the permissions of *alg/ps* should be a conjunction of the permissions of *alg* and *ps* to make it certain that if the owner of *alg* has so decided nobody can create a new file in *alg* or *alg/ps*.

This way of handling directory permissions makes commands that create files (e.g., `touch` or `mkdir`) safe.

2.4.2 Security semantic of `rm`, `rmdir`, `mv`, `chown`, and `chmod`

Under LFS, the `ls` algorithm permits that a user "sees" a file, without specifying all its keywords. 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), which will pollute future `ls` calls by the owner of this file.

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 under Unix 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. Unix proposes the "sticky-bit" to solve 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, the security check for the `rm`, `rmdir`, `chmod`, and `chown` operations is: check that the owner of the current process is the owner of the file or property name. The security check for `mv` is: check 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.

As changing the permissions of a complex directory makes no sense, such as in `chmod a+x key1|key2`, we first check that when the operation involves a directory, the last element in the path to the directory is a single keyword. E.g., doing `chmod a+rw "."` in a directory $a/b/c$ will modify only the permissions of c.

2.4.3 Security semantic of `ls` and `cd`

The new `ls` algorithm has also strange side effects on the `r` permission of a property. Under hierarchical file systems, disabling `r` means that others will not even be able to see the name of the file in this directory. In the logic file system, a file can be seen from many directories, and as the `ls` algorithm can "jump over" keywords, a user will be able to see a file, even if the file have properties without the `r` permission. In commands `ls` and `cd`, we could first check that one has effectively the right to see the name of those files by looking at the permissions of all the properties of the description of the file. As this makes those algorithms more complex, we decided not to solve this problem. In fact, we can see the read permission on a property as the right to see what are the file having this property. This means that one cannot so easily hide files. We will see in Section 4.3 a way to reintroduce this facility.

The same kind of difficulty comes with the execution permission of a directory. If a directory does not allow `x`, then no subdirectory will be reachable. Since under LFS, we can go from anywhere in any directory, one can "jump over" a parent directory which does not have the `x` permission. So, when requesting to go in a directory, LFS could check that all the parents have the `x` permission, or that there exists at least a path with the `x` permission. We prefer to not do any check to simplify the design.

3 Algorithms and data structures

We will first describe the `LS` algorithm that computes the answer to `ls` as it is the most original LFS operation. Moreover, it 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 LFS operations.

3.1 The `LS` algorithm

The "`LS` algorithm" is the most original operation of LFS, and the most costly. In a hierarchical file system, it just consists in reading the contents of a file on the disk, whereas in LFS it involves a real computation, with the increment and file calculus (see definition of *Sub* and *Files* in Section 2.2). There are so many possible combinations of properties that it is not feasible to create statically all the corresponding directories.

E.g., with a and b we have directories a/b, b/a, $a|b$, $a/!b$, ... Moreover, even if we could create them all, each time a user adds or deletes a file, the contents of all the directories must be updated as their *Sub* and *Files* part may have changed. So we prefer to create directories on-demand.

In the prototype LFS, the logic is a restricted form of the logic of proposition. The formula representing the `PWD` can be either a single atom (e.g., a), the negation of an atom (e.g., $\neg a$), a disjunction of atoms (e.g., $a1 \vee a2 \vee a3$), or a conjunction of formulas (e.g., $(a1 \vee a2) \wedge a3 \wedge (\neg a4)$).

We present first a naive algorithm for identifying "hot spots" in the computation. Then we propose a better algorithm that avoids the hot spots.

3.1.1 The naive algorithm

The naive algorithm follows closely the specification.

We can represent each property name in $\mathcal{F}$ by an *internal keyword identifier* f_i, and represent each object in $\mathcal{O}$ by an *internal object identifier* o_i.

To represent the description $d(o)$ of an object, a table `obj->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).

A `PWD` formula is represented by a list of an union value, which is either an internal keyword identifier, either the tag `Or` associated with a list of internal keyword identifiers (the operands of the disjunction), either the tag `Not` associated with an internal keyword identifier. The list will play the rôle of the conjunction.

The axioms of the taxonomy are represented as a DAG of internal keyword identifiers (that could be implemented as a table `keyword->children` or 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*. Note that this is not a return to a hierarchical file system, since this structure is not exposed to the user; it only plays an internal rôle.

The pseudo-code for function *extension* that takes a formula f as a parameter and returns a list of internal object identifiers is as follows:

```
extention(f) =
 filter oi from O
  where
    description = obj->keywords[oi]
    forall x in the list f
      if x is a single keyword      then
         check_downwards(x,description)
      if x is a OR with elements xs then
        check that at least for one y of xs:
          check_downwards(y,description)
      if x is a NOT y  then
        check not check_downwards(y, description)

check_downwards(x, xs) =
  if x is in xs    then true
  if x have no more child concepts then false
  else
   check that at least
     for one y in keyword->children[x]:
       check_downwards(y, xs)
```

Function `check_downwards` traverses recursively the taxonomy DAG. Indeed, if an object x has a description b and if $b \models a$ (i.e., b is a descendant of a in the taxonomy DAG), then x is also in the extension of the formula a.

Thus, the `LS` algorithm is:

```
LS(f) =
  ois  = extenstion(f)
  # possible increments
  fis  =
    filter fi from F where
      card = cardinality(inter(extension(fi),ois))
      0 < card < cardinality(ois)
  # maximal increments
  maxfis = filter fi from fis where
     not check_upwards(fi,minus(fis,fi))
  sub = maxfis
  files = minus(ois, union_extension(sub))

check_upwards(x,xs) =
  if x is in  xs then true
  if x is the top node then false
  else
   check that at least
     for one y in keyword->parents[x]:
       check_upwards(y,xs)
```

where functions `minus`, `union`, `inter` and `cardinality` are the usual set theory functions, and where `union_extension` consists in calling `extension` on each elements in `sub` and unioning their result.

3.1.2 Hot spots and an improved algorithm

The naive algorithm is too costly. Indeed, each call to `extension` (1) goes through all objects (see line `filter oi from O`), and (2) can involve a recursive exploration of all the taxonomy DAG by function `check_downwards`. So, the worst response time for function `extension` is at least proportional to $\|\mathcal{F}\|.\|\mathcal{O}\|$. Finally, (3) function `LS` goes through all the internal keyword identifiers (see line `filter fi from F`), which in turn involves again going through all objects for the extension calculus of each fi, and like in (2) it involves another recursive traversal of the taxonomy DAG by `check_upwards`.

Going through all objects As we have seen, starting from every object to check whether it satisfies the `PWD` formula is costly. The main feature of our implementation is to do the opposite, that is to start from the `PWD` formula and compute the objects. Instead of using a table `obj->keywords`, we will use the inverted table `keyword->objects`. This uses standard indexing techniques.

So, the extension of $a \wedge b$ is computed by intersecting the extension of a and the extension of b. Similarly, the extension of $a \vee b$ is computed by unioning the extension of a and the extension of b. Finally, the extension of $a \wedge \neg b$ is computed by subtracting the extension of b from the extension of a. The extension of an atom property a still involves a recursive process to merge `keyword->objects[a]` with the extensions of the sub-properties of a.

The recursive process The second feature of our implementation is to avoid the recursive extension computation by "in-lining" the extension of a child sub-property in the extension of its parent property. So doing, 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 recursively (using table `keyword->parents`) the `keyword->objects` entry of x and of all its ancestors in the taxonomy DAG. This operation is costly, but we think that the main operation to optimize is `ls`. So, we prefer to transfer the hard-work to command `touch`. Moreover, `touch` does not require an immediate update, so that its response time can be good anyway if it delays the computation.

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 to table `keyword->objects`.

Still, the extension of an atomic property can be large, and set operations on it can be costly. Indeed, to get the entry in `keyword->objects` may require to read many sectors on the disk, and then to compute the extension of a `PWD` may involve intersecting and unioning large sets of objects. Anyway, experiments show that this a viable design (see Section 5).

The naive extension algorithm causes $\|\mathcal{O}\|$ accesses to table `object->keywords`, which is very big even if we can pack some entries together on the same disk sector. The refined solution causes $\|$`PWD`$\|$ accesses to table `keyword->objects`, which is by far smaller. Moreover, extensions can be compressed by using an interval representation, and specialized set operations can be used. This is especially useful for the most general properties which contain in their extensions all or nearly all the objects.

Finally, experiments show that the overhead involved in "in-lining" extensions in parent properties is negligible in disk space (see Section 5).

Going through all keywords A fast extension calculus does not solve problem (3); function `LS` still goes through all the property names. As for problem (1), instead of starting from all the formulas and then sorting their extensions, the final feature of our implementation is to start from the sorted formulas directly.

The final 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 contain the `PWD` extension, and has a non empty intersection with it, are the maximal increments to list in Sub.

This avoids 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 "in-lined" extensions of its children. Similarly, extensions of sub-properties of a property that strictly intersect with the `PWD` are not considered either, because those sub-properties are not maximal increments, and would be filtered out by `check_upwards`.

The final algorithm The final algorithm using tables `keyword->objects` and `keyword->children` is as follows:

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

LS(f) =
  ois  = extension(f)
  fis  =
    collect keywords fi from the graph concept
```

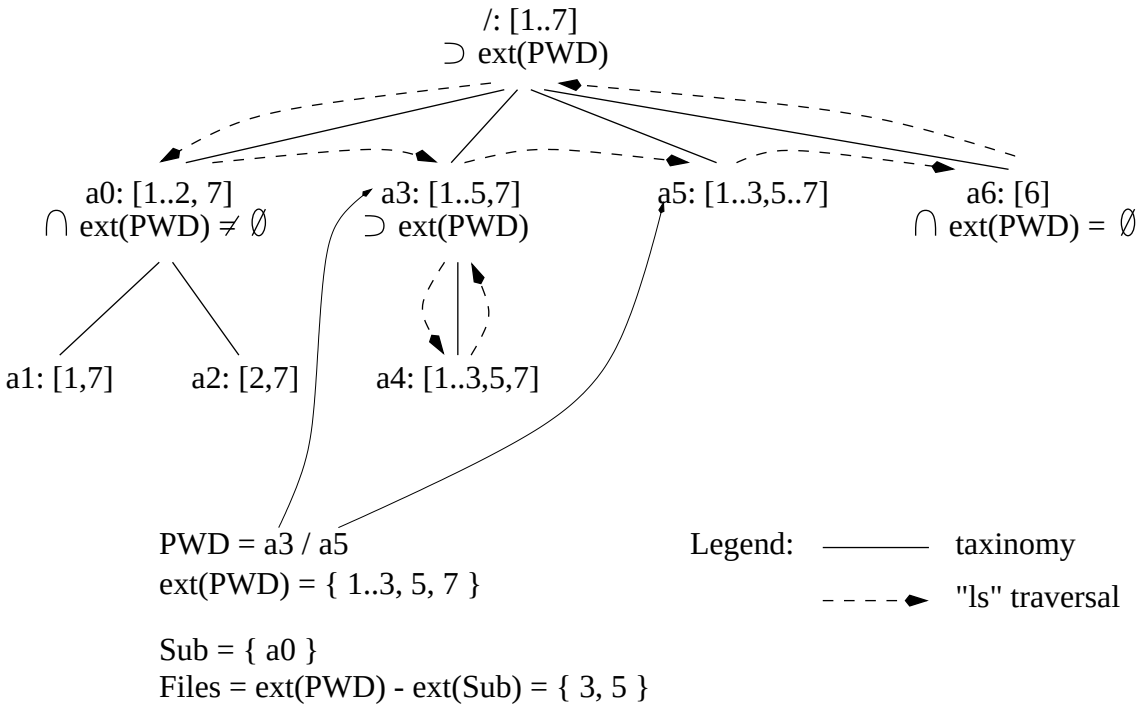

Figure 5: the final algorithm

```
      using a depth first search starting
      from the top node downward
      using keyword->children until:
    card = cardinality (inter(extension(fi),ois))
    0 < card < cardinality(ois)
sub = fis
files = minus(ois, union_extension(sub))
```

See figure 5 for an illustration.

3.2 Data structures

We describe more precisely the actual data structures of LFS. We have implemented LFS as an instance of Linux *VFS* (*virtual file system* [Kle86]).

Traditionally, file systems divide the disk in 2 sections: (1) the meta-data section, containing a bitmap of free inodes numbers, a list of free storage blocks, indexing structures, and superblock information, and (2) the data section, containing the contents of the files. With LFS, the meta-data information will be bigger, and of varying size, so the meta-data will be spread on the disk, as for the contents of the files.

File contents are stored on the disk as usual. More generally, all file operations use standard file system techniques.

3.2.1 Main tables

As said previously, the internal data structures use internal object and keyword identifiers, instead of plain names. This is more space and CPU efficient. In 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. The extension table

uses interval compression to represent the set of objects. This is more space efficient, and it improves in turn the speed while reading the entry.

Moreover, in order to preserve the in-lining 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.

In order to return plain names in `ls` answers, we introduce two new 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`.

3.2.2 Unix tables

In the Unix context, a typical user manipulates files with names, whereas the operating system uses an internal identifier: the inode number.

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 the file, such as its mode, permissions, data block addresses, whether or not it is a directory, etc. Under these file systems the size of this table is fixed, because there is a statically bounded number of files and directories. With LFS, they are too many to be created statically, then we prefer to create them on-demand, and allocate new inodes only when needed. So, LFS directory inodes correspond to kind of *virtual directories*.

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, as described in section 3.1.1. Each union value contains either the internal property identifier of a property, or the tag `Or` associated with a list of internal property identifiers (the operands of the disjunction), or the tag `Not` associated with an internal property identifier.

Moreover, answers to `ls` are cached. Directory inodes contain block addresses in the data section that contain 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 LFS, inodes contain a local timestamp indicating the time the current result was computed. LFS 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 filled in the `pwd` field of the `PWD`, and their local timestamp is set to 0 (to force the next call to `ls` to recompute increments). When increments are recomputed, LFS will fill in storage blocks with the result computed by the `LS` algorithm, and will adjust accordingly the block addresses in the inode.

For atomic properties, a table `keyword->property` stores the rights, owner and group of every atomic property. This table is used when setting the rights of the directories inode in `inode_table`.

Finally, we must take care of deleted atomic properties. Indeed, a deleted atomic property may have been mentioned in several queries, and so its corresponding internal keyword identifier can be present in the `pwd` field of some directories. Future `ls` executions may look for an internal keyword identifier that, either is not used anymore, or if the internal property identifier have been reallocated, corresponds to the wrong atomic property. We solve this problem by maintaining in a table `keyword->inodes` all the inodes that use in their `pwd` field a given keyword, and by adding a tag `valid` in the inode structure. Then, deleting an atomic property p consists in going through the list `keywords->inodes`$[p]$, and mark as invalid all those inodes, so that future operations on those inodes will return an error code.

3.3 Concrete operations

In this section, we switch from shell operations to file system operations to be able to enter in more details. For space reason, we enter in the details of only a few LFS 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.

We also abstract the description of the operations, to be able to focus on the difference between LFS and hierarchical file systems. So, we do not talk about memory and block layout, concurrency and synchronization, as we think our paradigm have few impacts on those questions, and traditional techniques can be used.

For fault-tolerance, we use journaling for implementing *transaction*. Every time several tables must be updated in an atomic way, a transaction is started, that logs the updates. In case of failure, the next reboot will redo (or undo) the log (see [RO92] for more information about 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 have coherent contents.

File system operations can be divided into three groups: global operations (aka. superblock operations in Unix), directory operations (aka. inode operations in Unix), 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.

Function `put_super` (`umount`), `write_super` and `stat_fs` (`df`) are similar to their counterparts in hierarchical file systems.

3.3.2 Inode operations

- `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 3.1.2) is applied to the `pwd` field of the parameter inode. The result is a pair `Sub` 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 `Sub` a fresh inode number `ino` is allocated. Then, each entry `inode_table[ino]` in the `pwd` field is filled in with the conjunction, `PWD'`, of `fi` and the previous `PWD`, and has its local timestamp 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.

- `lookup` can be called via the command `cd keyword`. It takes as a parameter an inode `i` (the current directory) and a string `s`, which is the plain name of a file or of a property, and it returns the corresponding inode, or an error condition. Operation `readdir` is called to look if the string is in the list of pair, 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, either 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 entry `inode_table[ino]` is filled in with in the `pwd` field of the conjunction of `fi` and of the `PWD` of the current directory, and the freshly created 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 have the form of a negation `! x`, then the operation is similar, but putting this time in the `pwd` field the tag `Not`.

- `create` can be called via command `touch`. It takes as a parameter an inode `i` (the current directory), a string `s` and returns the inode corresponding to the object just created. The current directory must correspond to a conjunction of atomic properties (as we have restricted the description of object) `fis`. 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 in the table `filename->objects[s]`, with the same description, 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.

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` and the `keyword->property` tables, associating the appropriate mode, owner, group, ...to the corresponding inode, using the security semantic described in section 2.4.1.

3.3.3 File operations

As said previously, 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 LFS. In reality this is only a framework, in which more features can be introduced. We present in this section features that have actually been introduced in the prototype LFS.

4.1 Valued attributes

LFS allows one to create directories with arbitrary names, except names containing special symbols (`|`, `&` or `!`). This 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, ...).

Hierarchical operating systems provide a command (called *find* in Unix), which allows to make elaborate queries on a hierarchy as a whole. For instance, `find -size +45` looks for files having a size greater than 45 bytes. With databases also, one can form a request such as "*select file where size* >= 45". As our query language provides disjunction, we could simulate such a query by enumerating all the directories satisfying the condition, e.g., `cd (size:45|size:46|...)`. It would be very inconvenient, and we prefer to extend the query language to support more operations on valued attributes such as `cd size:>45`.

Concretely, the `lookup` operation parses the name (i.e., a string) passed as a parameter. If it matches the pattern `attr:>val` where `attr` is an arbitrary string, and `val` an integer, the keyword `attr` is looked for, all its increments are gathered in a list, which is then filtered to keep only those increments that match the pattern `attr:x` where `x>val`. Finally, a new inode is allocated as usual, and the `pwd` field is set to `Or (k_1,k_2, ...)` where the `k_i`'s are the selected increments.

Our prototype only supports integer operations (`<`, `>`, `<=`, and `>=`) and operations on regular expressions (e.g., `cd auteur:=~ m.?in.+y.*`). This can be easily extended.

This extension only affects the parsing and interpretation of names in operation *lookup*. It is essentially free.

4.2 Transducers

Being able to describe a file with valued attributes, still requires to fill in the values. E.g., consider a *size* valued attribute; it changes almost every time a file is updated. One cannot envisage to maintain it manually.

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 [GJSJ91]).

In our prototype, transducers are defined for all the system attributes of a file, e.g., its size, 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.

4.3 ACL-like security

We mentioned in section 2.4 the problem that no file can be really hidden. I.e., one can hide their contents but not their name. We extend the security model to support this feature.

Each request of a user will now have in addition to normal attributes, an attribute that represents the user identifier (*uid*). For instance, the formula corresponding to the directory */alg/ps*, for a user with a *uid* of 500 is $alg \wedge ps \wedge 500\_l$, where l stands for "read listing".

So doing, a user sees only files that have this attribute. If a user wants to make his file visible only by him, he just has to not put this file in the uid_l directory of another user.

One often needs to make a file visible by every user, but it would be error-prone to manually list all the uid_l directories. Again, we use the principle of the taxonomy. Security attributes are ordered: *other* > *group* > *user*, (see figure 6 for an illustration).

If a file has the property $other\_l$, that means that this file is in the extension of all the parents of $other\_l$ (cf. Section 2.3), i.e., this file is in the extension of all the uid_l directories, i.e., all users will be able to see it, which is what is intended.

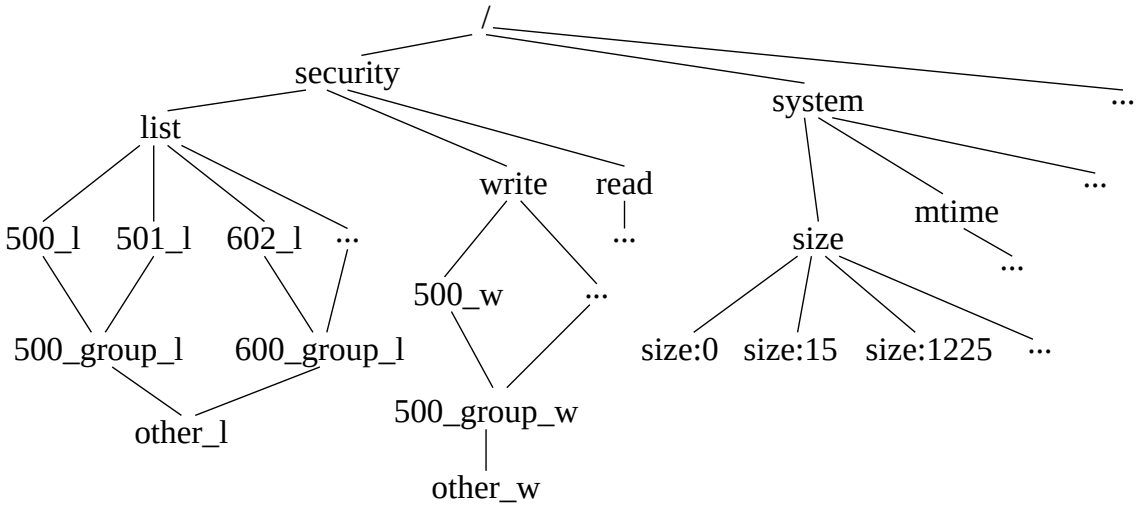

Figure 6: A typical hierarchy of concepts for security and system in LFS

This method has the advantage of providing a discretionary control on the visibility of a file. Indeed, one can assign individually which user can see a file, even if this does not form exactly a group.

This feature is implemented by just modifying the `readdir` operation, which must call the `LS` function with `LS(pwd & uid_l)` where `uid` is the *uid* of the current process.

We apply this method also to the readability/writeability of the contents of a file. We define 2 other hierarchies (uid_r, uid_w, ...), and we modify the `open` operation. When a user opens a file in `read`/`write` mode, `open` first check that the file is in the extension of `uid_r` and `uid_w` respectively. We do not apply this method to the x property, because there is no specific file system operation where one can intercept the execution of a file.

Note that we keep the original mechanism of Unix permissions, and just extend it. When a user writes in a file, the operating system first checks the Unix permission of the file, then calls the `open` operation of the file system in `write` mode. This is where the discretionary control is done.

In summary, this is an easy way to add ACL-like permissions while still using traditional commands. The user needs neither edit special permission files, nor use special commands.

4.4 `ls` operations

4.4.1 Relaxed increments

We use transducers to automatically add intrinsic valued attributes to a file, but this has a negative side-effect on the result of `ls`. Every file now has system attributes, and so the file organization is as showed in figure 6.

It means that the first `ls` from the root lists all the $size : x$, $modification : y$, ..., subdirectories. Indeed, it does not answer with the more general concepts, like *system*, *size*, ..., because those concepts do not refine the current query, since every file is in the extension of those concepts.

So, we decided to relax the condition for being an increment in order to exhibit really relevant increments, with a more general property. The `ls` increment semantics becomes

$Sub = max_{\models}(\{f \mid f \in F, \neg(pwd \models f), \emptyset < ext(f \wedge pwd) \leq ext(pwd)\})$.

The main novelty is that the rightmost inequality is no longer strict. With this scheme, the `ls` command at the root directory lists only the coarse categories. The $\neg(pwd \models f)$ constraint avoids navigating indefinitely in the same directory.

The two semantics are proposed to the user. The `inode_table` structure has a field that contains a tag indicating the chosen semantics. This tag is propagated to all the increments of the directory. It can be changed by the user doing either `cd .relaxed`, or `cd .strict`.

4.4.2 Views

LFS encourages to give very rich descriptions to individual files, according to several different points of view. This leads to long descriptions. E.g., in an experiment with BibTeX entries, every entry was described by an average of 70 attributes. The danger is that most of these attributes are simply noise when a file is considered from a single point of view. So, we like to better control this noise.

A new feature, *views*, allows one to constrain the set of increments that the algorithm may list. The user declares a view by adding the special symbol `^` (caret) to the end of a property name. This indicates that future `ls` operations may list only increments that are sub-concepts of the view. For instance, a user who searches a music file may do `cd music^`, to indicate that only sub-concepts of *music*, like *author* and *genre*, are of interest for him. Other concepts, like *size*, are simply ignored.

In summary, the `PWD` selects the kind of objects that are searched for, and views select the kind of increments that can be used in navigation.

The `inode_table` structure is again augmented with a field that indicates the current view. This tag is propagated to all the increments of the directory, and it can be changed in the `lookup` operation if the name string matches the pattern `.*^`. The `LS` algorithm is modified to start from the current view in place of starting from the top node.

4.4.3 Extensional answers

Under hierarchical file systems, one gets the list of all the files under a directory (i.e., its extension) by using command `ls -r`. This command recursively goes though all the sub-directories of a hierarchy. Under LFS, a recursive traversal is impracticable because too many paths lead to the same place. In particular, if a path of n names leads to some place, then the $n!$ permutations lead to the same place.

However, computing an extension is almost free in our system because they are represented in the extension table. To be able to get the extension of a query, we extend again the query language with a special keyword: `.ext`. So, a user doing `cd .ext; ls` will get the extension of the current directory.

We use the same techniques as before, adding a special tag in the inode indicating when we are in "extension mode".

5 Experimentation

5.1 Implementation

The current prototype is implemented as a user level file system, using PerlFS which is like Userfs [Fit96], but allows to prototype in Perl in place of C++. We use Berkeley DB [OBS99] to manage the different meta datas (implemented as Btrees). The transactionnal module of Berkeley DB provides the necessary tools for handling the possible corruption of meta-datas during a crash. Finally, we use the underlying file system (ext2) to manage the contents of the file, which means that we delegate the block handling managment.

5.2 Efficiency

We ran several experiments to determine the overhead of using LFS, both in speed and disk space.

The first experiment evaluates the disk space overhead used by the meta-datas of LFS. The results are shown in table 1. The experiment is run for a set of 616 music files in format MP3, and a set of 850 program files obtained by ten copies of the Andrew file system benchmark [KMN+88].

In the second experiment, we ran the Andrew benchmark, first on the native UNIX file system (ext2), then on LFS where transducers were turned on then off, and finally on a hierarchical file system implemented via PerlFS. The benchmark has 5 phases: (1) **Makedir** constructs a directory hierarchy, (2) **Copy** constructs files, (3) **Scan** recursivly traverses the whole hierarchy, (4) **Read** reads every byte of every file, and (5) **Make** launches a compilation of the files. We also ran our own benchmark that consists in copying 616 music files. The results are shown in table 2.

In the last experiment, we test the speed of `ls` in directories of various sizes in the music files context. In directory *artist*, `ls` computes 160 increments in 0.9 second. In directory *size*, it computes 616 increments in

	Andrew files	MP3 files
Number of files	788	634
Total size of files	9308 Ko	1771916 Ko
Total size of LFS tables	904 Ko	1808 Ko
Average number of attributes per file (intrinsic/extrinsic)	(15/10)	(38/15)
Total number of different attributes	1049	2893
Average size per file	12 Ko	2795 Ko
Space overhead per file	1.2 Ko	2.9 Ko
Space overhead	10 %	0.1 %
Space overhead per attribute	0.9 Ko	0.6 Ko

Table 1: Results of Disk Space Benchmark

File System	Makedir	Copy	Scan	Read	Make	Total	Copy MP3
Ext2	0s	1s	0s	1s	2s	5s	2min
LFS (transd. on)	0s	8s	1s	3s	6s	18s	14min
LFS (transd. off)	0s	3s	1s	3s	4s	11s	9min
PerlFS	0s	1s	0s	2s	2s	5s	4min

Table 2: Results of CPU Benchmark

3 seconds, and finally it computes 38 increments in directory *genre* in 0.3 second. Note that in any case, only the first `ls` in a given directory takes time, as for further `ls` LFS uses its cache, and answers immediatly.

Note also that the large number of increments does not contradict the overall aim of LFS. Properties like *artist*, *size*, and *genre*, are intrinsic properties that are shared by every file in the MP3 directory. They differ only in their values. Only user defined extrinsic properties could help in discriminating them. These directories have been chosen for this last experiment only because they stress command `ls`.

In summary, operations `lookup` and `readdir` do not show enormous performance penalty, especially considering the work they perform. 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" column of Table 2. We believe it can be cured by performing most of the `create` operation lazily. Indeed, the answer of this operation does not depend on the updating of the internal tables.

5.3 Quality

We ran several experiments with different datas: audio files, man pages and computer science documents (book references, articles).

For audio files, the use of LFS can be compared with music files stored on a hierarchical file system. With the latter, we are forced to choose one hierarchy to classify the files, for example by artists. This makes every search on another criterion difficult. With LFS, all criteria are taken into account on a par. Being able to use negation, disjunction, ...over either the *genre*, *year*, *artist*, makes LFS very convenient to choose a set of music files to play.

For man pages, the use of LFS can be compared with the Unix command *apropos*. Finding the appropriate man page seems to be faster with LFS. Indeed, classically one has to do first `apropos`, then to locate the corresponding entry, which needs some time since the result can be big, and then to call the user command

`man` with the appropriate entry. With LFS, we used the words of the synopsis of the man page as its description. One has just to do `cd` *keyword*.

Note that since the shell is a standard one, all its commodities combine with LFS. For instance, if the shell performs name completion, it will work with LFS.

Finally, we compare managing a bibliography via LFS with managing a large BibTeX file. Being able to use completion when interactively classifying a new file makes the process faster than manually entering all the data in the BibTeX file. Moreover, as opposed to BibTeX, we are able to create a DAG of the different computer science fields, which means that when we say a document talks about $neuralNetwork$, it implicitly talks about ai, and so future queries such as `cd ai` will find this document. To obtain the same service with a single BibTeX file requires to mention in the corresponding BibTeX entry the keyword $neuralNetwork$, but also the redundant keyword ai, which is tedious and error-prone.

6 Related work

The Semantic File System (SFS [GJSJ91]) was the first file system to combine querying and navigation. It remains 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 4.2). More importantly, querying and navigation were two separated operation modes; one could not navigate in a virtual directory that results from a query.

Another file system that combines querying and navigation is the Be file system (BeFS [Gia99]). 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 LFS where one can use classical shells). Then, a new interface was designed, to be used by specialised tools. 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 [BDB+94] 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 organised in a hierarchy, where subviews refine parents 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.

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 [Lin95], a corpus of keywords is extracted from man pages, and via formal concept analysis [GW99], 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 [Ped93], a similar approach is applied to bibliographic information retrieval. In fact, the querying mechanism of these applications is completely encompassed by LFS; the answers of LFS *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 [BEH99, MTW95]). 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 LFS. E.g., commands `touch` and `ls` are too expensive. Tricks such as grouping of commands (as in Xwindow), amortization, lazy structures or use of idle time will certainly improve the performance of `touch`. Statistics about sets of objects could help in designing heuristics for the `LS` algorithms. For example, with valued attributes (where `ls` is quite slow), knowing whether the different attribute extensions are disjoint, and knowing their cardinalities could lead to a more efficient `LS` algorithm. Furthermore, using idle time makes it possible to predict and to pre-calculate future user queries. Indeed, users find future queries among the increments provided by LFS. 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 LFS. Since logic formulas are ordered by the $\models$ relation (e.g., $a \wedge b$ is more specific than a), they can be inserted in the taxonomy DAG, which causes no change on the `LS` algorithm. Still, this requires to change `mkdir` to accurately find where to insert new formulas in the taxonomy DAG.

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. We could even imagine a way to generate a new file from the current query (i.e., a *view* of the program), and we could even allow the user to modify this view and to back-propagate the modifications to the original file. Note that there should be no *view update problem* here, because answers to a query are actually parts of the source file. This direction of research can certainly contribute to the software engineering domain, where aspect oriented programming, and multi-view interactions are popular issues.

8 Conclusion

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*. 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 LFS with encouraging performances. Experiments show that though the prototype LFS is slower than a more classical one, it passes usual benchmarks with reasonable performances: `create`-intensive benchmarks show a performance ratio of about 7 against LFS, but `ls`-intensive benchmarks show almost no penalty. The current implementation of LFS 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 confirm the validity of LFS.

The challenge is now to improve LFS efficiency, to allow arbitrary logic formulas in descriptions of objects and in queries, to make the system reflexive by allowing attributes to have attributes, and object to have relations with other objects, and finally to permit navigating in files as well as in directories.

References

[BDB+94] 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.

[BEH99] 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.

[Fit96] J. Fitzhardinge. Userfs: A file system for linux. In *unpublished software package documentation*, ftp://sunsite.unc.edu/pub/linux/alpha/userfs/userfs-0.9.tar.gz, 1996.

[FR00] 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.

[FR01] S. Ferré and O. Ridoux. Searching for objects and properties with logical concept analysis. In *Int. Conf. Conceptual Structures*, LNCS 2120. Springer, 2001.

[Gia99] D. Giampaolo. *Practical File System Design with the Be File System.* Morgan Kaufmann Publishers, 1999.

[GJSJ91] 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.

[GM99] 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.

[GW99] B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations.* Springer, 1999.

[Kle86] S.R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *USENIX Summer*, pages 238–247, 1986.

[KMN+88] 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.

[Lin95] C. Lindig. Concept-based component retrieval. In *IJCAI95 Workshop on Formal Approaches to the Reuse of Plans, Proofs, and Programs*, 1995.

[MTW95] R. Miller, O. Tsatalos, and J. Williams. Integrating hierarchical navigation and querying: A user customizable solution, 1995.

[OBS99] M.A. Olson, K. Bostic, and M. Seltzer. Berkeley db. In *FREENIX Track: 1999 USENIX Annual Technical Conference*, pages 183–192, 1999.

[Ped93] G.S. Pedersen. A browser for bibliographic information retrieval based on an application of lattice theory. In *ACM-SIGIR'93*, pages 270–279, 1993.

[RO92] 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.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399