Everything is a virtual filesystem: libferris

Ben Martin
affiliation pending
monkeyiq@users.sf.net

Abstract

Libferris is a user address space virtual (Semantic) filesystem. Over the years it has grown to be able to mount relational databases, XML, and many applications including X Window. Rich support for index and search has been added. Recently the similarities between modern Linux kernel filesystem, Semantic Filesystems and the XML data model has been exploited in libferris. This leads to enabling XSLT filesystems and the evaluation of XQuery directly on a filesystem. XQueries are evaluated using both indexing and shortcut loading to allow things like db4 files or kernel filesystems with directory name caching to be used in XQuery evaluation so that large lookup tables can be efficiently queried. As the result of XQuery is usually XML—As the similarities between XML and filesystems are discussed, the option is there for queries to generate filesystems.

Prior knowledge of the existance of Extended Attributes as well as some familiarity with XML and XQuery will be of help to the reader.

1 Introduction

Libferris [2, 9, 11, 13, 14] is a user address space virtual filesystem [1]. The most similar projects to libferris are gnome-vfs and kio_slaves. However, the scope of libferris is extended both in terms of its capability to mount things, its indexing and its metadata handling.

Among its "non conventional" data sources, libferris is able to mount relational databases, XML, db4, Evolution, Emacs, Firefox and X Window.

The data model of libferris includes rich support for unifying metadata from many sources and presenting applicable metadata on a per filesystem object basis. Indexing and querying based on both fulltext and metadata predicates complements this data model allowing users to create virtual filesystems which satisfy their information need. It should be noted that metadata can be associated with any filesystem object, for example a tuple in a mounted database.

The paper now moves to discuss what semantic filesystems are and in particular what libferris is, and how it relates to the initial designs of a semantic filesystems. The similarities and differences between the libferris, traditional Linux kernel filesystem and XML data models is then discussed with mention of how issues with data model differences have been resolved where they arise. The focus is then turned to information indexing and search. The indexing section is more an overview of previous publications in the area to give the reader familiarity for the example in the XQuery section. The treatment of XQuery evaluation both directly on a filesystem and on its index then rounds out the paper.

2 Semantic Filesystems

The notion of a semantic filesystem was originally published by David K. Gifford et al. in 1991 [4].

A semantic file system differs from a traditional file system in two major ways:

- Interesting meta data from files is made available as key-value pairs.
- Allowing the results of a query to be presented as a virtual file system.

The notion of interesting meta data is similar to modern Linux kernel Extended Attributes. The main difference being that meta data in a Semantic Filesystem is inferred at run time whereas Linux kernel Extended Attributes are read/write persisted byte streams. In the context of libferris, the term Extended Attributes can refer to both persisted and inferred data. In this way the Extended

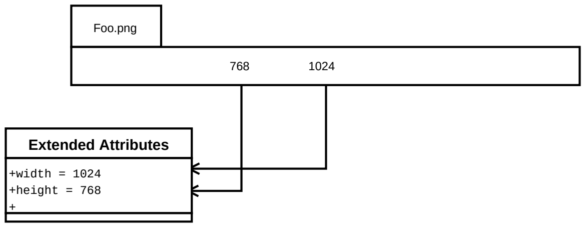

Figure 1: The image file Foo.png is shown with it's byte contents displayed from offset zero on the left extending to the right. The png image transducer knows how to find the metadata about the image file's width and height and when called on will extract or infer this information and return it through a metadata interface as an Extended Attribute.

Attributes in libferris have been virtualized along with the filesystem itself. The term Extended Attributes will be used in the libferris sense unless otherwise qualified.

In a Semantic filesystem interesting meta data is extracted from a file's byte content using what are referred to as transducers [4]. An example of a transducer would be a little bit of code that can extract the width of a specific image file format. A transducer to extract some image related metadata is shown conceptually in Fig.1. Image dimensions are normally available at specific offsets in the file's data depending on the image format. A transducer which understands the PNG image encoding will know how to find the location of the width and height information given a PNG image file.

Queries are submitted to the file system embedded in the path and the results of the query form the content of the virtual directory. For example, to find all documents that have been modified in the last week one might read the directory "/query/(mtime>=begin last week)/". The results of a query directory are calculated every time it is read. Any metadata which can be handled by the transducers [4] (metadata extraction process) can be used to form the query.

Libferris allows the filesystem itself to automatically chain together implementations. The filesystem implementation can be varied at any file or directory in the filesystem. For example, in Figure 2 because an XML file has a hierarchical structure it might also be seen as a filesystem. The ability to select a different implementation at any directory in a URL requires various filesystems to be overlaid on top of each other in order to present a uniform filesystem interface.

When the filesystem implementation is varied at a file or directory then two different filesystem handlers are active at once for that point. The left side of Figure 2 is shown with more details in Figure 3. In this case both the operating system kernel implementation and the XML stream filesystem implementation are active at the URL `file://tmp/order.xml`. The XML stream implementation relies on the kernel implementation to provide access to a byte stream which is the XML file's contents. The XML implementation knows how to interpret this byte stream and how to allow interaction with the XML structure though a filesystem interface.

Note that because in the above the XML implementation can interact with the operating system kernel implementation to complete its task this is subtly different to standard UNIX mounting where a filesystem completely overrides the mount point.

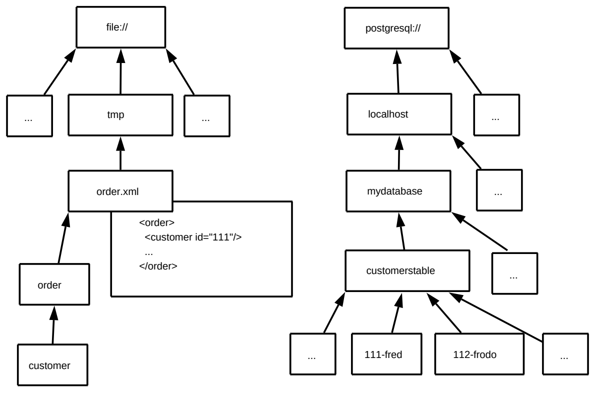

Figure 2: A partial view of a libferris filesystem. Arrows point from children to their parents, file names are shown inside each rectangle. Extended Attributes are not shown in the diagram. The box partially overlapped by `order.xml` is the contents of that file. On the left side, an XML file at path /tmp/order.xml has a filesystem overlaid to allow the hierarchical data inside the XML file to be seen as a virtual filesystem. On the right: Relational data can be accessed as one of the many data sources available though libferris.

The core abstractions in libferris can be seen as the ability to offer many filesystem implementations and select from among them automatically where appropriate for the user, the presentation of key-value attributes that files posses, a generic stream interface [6] for file and metadata content, indexing services and the creation of arbitrary new files.

Filesystem implementations allow one to drill into com-

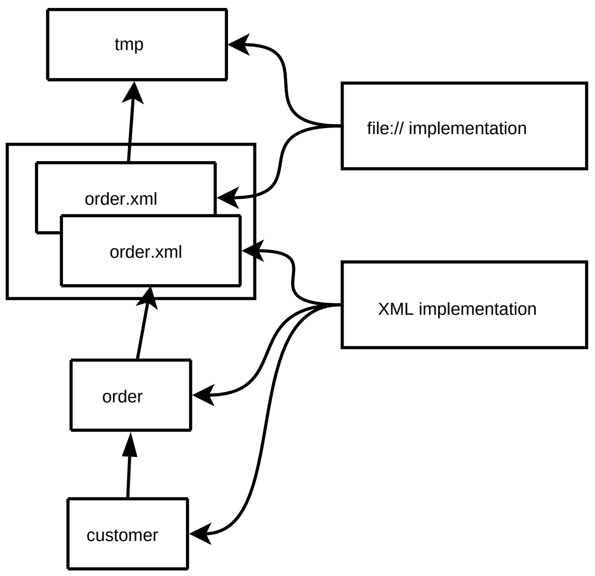

Figure 3: The filesystem implementation for an XML file is selected to allow the hierarchical structure of the XML to be exposed as a filesystem. Two different implementations exist at the "order.xml" file level: an implementation using the operating system's kernel IO interface and an implementation which knows how to present a stream of XML data as a filesystem. The XML implementation relies on the kernel IO implementation to provide the XML data itself.

posite files such as XML, ISAM[1] databases or tar files and view them as a file system. This is represented in Figure 2. Having the virtual filesystem able to select among filesystem implementations in this fashion allows libferris to provide a single file system model on top of a number of heterogeneous data sources.[2]

Presentation of key-value attributes is performed by either storing attributes on disk or by creating synthetic attributes who's values can be dynamically generated and can perform actions when their values are changed. Both stored and generated attributes in libferris are referred to simply as Extended Attributes (EAs). Examples of EAs that can be generated include the width and height of an image, the bit rate of an mp3 file or the MD5[3] hash of a file. This arrangement is shown in Figure 4.

For an example of a synthetic attribute that is writable consider an image file which has the key-value EA `width=800` attached to it. When one writes a value of `640` to the EA `width` for this image then the file's image data is scaled to be only 640 pixels wide. Having performed the scaling of image data the next time the `width` EA is read for this image it will generate the value `640` because the image data is `640` pixels wide. In this way the differences between generated and stored attributes are abstracted from applications.

Another way libferris extends the EA interface is by offering schema data for attributes. Such meta data allows for default sorting orders to be set for a datatype, filtering to use the correct comparison operator (integer vs. string comparison), and GUIs to present data in a format which the user will find intuitive.

3 Data models: XML and Semantic Filesystems

As the semantic filesystem is designed as an extension of the traditional Unix filesystem data model the two are very similar. Considering the relatively new adoption of Extended Attributes to kernel filesystems the data models between the two filesystem types are identical.

The main difference is the performance differences between deriving attributes (semantic filesystem and transducers) or storing attributes (Linux kernel Extended Attributes). Libferris extends the more traditional data model by allowing type information to be specified for its Extended Attributes and allowing many binary attributes to form part of a classification ontology [12, 7].

Type information is exposed using the same EA interface. For example an attribute `foo` would have `schema:foo` which contains the URL of the schema for the `foo` attribute. To avoid infinite nesting the schema for `schema:foo`, ie, `schema:schema:foo` will always have the type schema and there will be no `schema:schema:schema:foo`.

As the libferris data model is a superset of the standard Linux kernel filesystem data model one may ignore libferris specific features and consider the two data models in the same light. This is in fact what takes place when libferris is exposed as a FUSE filesystem [1].

[1]Indexed Sequential Access Method, e.g., B-Tree data stores such as Berkeley db.

[2]Some of the data sources that libferris currently handles include: http, ftp, db4, dvd, edb, eet, ssh, tar, gdbm, sysV shared memory, LDAP, mbox, sockets, mysql, tdb, and XML.

[3]MD5 hash function RFC, `http://www.ietf.org/rfc/rfc1321.txt`

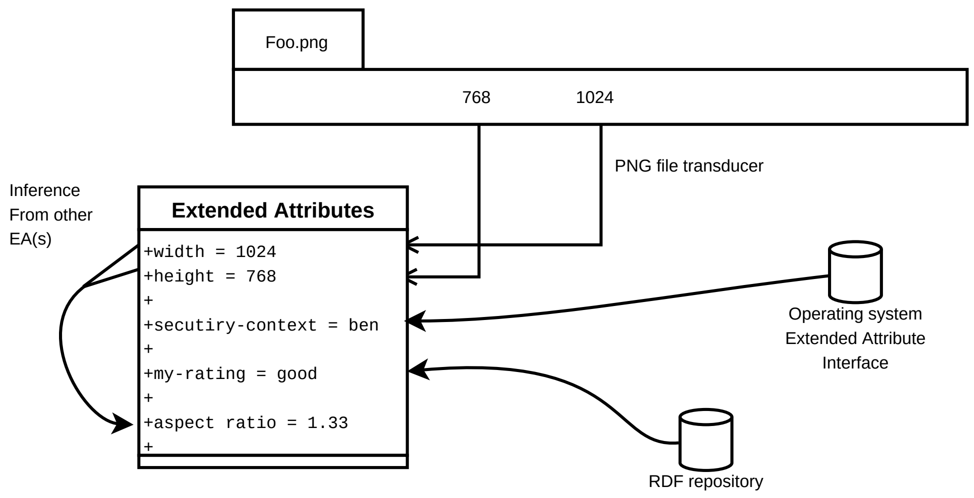

Figure 4: Metadata is presented via the same Extended Attribute (EA) interface. The values presented can be derived from the file itself, derived from the values of other EA, taken from the operating system's Extended Attribute interface or from an external RDF repository.

Taking an abstract view of the data model of libferris one arrives at: files with byte contents, files nested in a hierarchy and arbitrary attributes associated with each file. This is in many ways similar to the data model of XML: elements with an ordered list of byte content, elements nested in an explicit ordering with attributes possibly attached to each element.

The differences between the data models may raise issues and require explicit handling. The differences have been found to be:

- XML elements can contain multiple contiguous bytes serving as their "contents." Files may have many sections of contiguous bytes separated by holes. Holes serve to allow the many sections of contiguous bytes to appear in a single offset range. For example, I could have a file with the two worlds "alice" and "wonderland" logically separated by 10 bytes. The divergence of the data models in this respect is that the many sections of contiguous bytes in an XML element are not explicitly mapped into a single logical contiguous byte range.
- XML elements are explicitly ordered by their physical location in the document. For any two elements with a common parent element it will be apparent which element comes "before" the other. Normally files in a filesystem are ordered by an implementation detail—their inode. The inode is a unique number (across the filesystem itself) identifying that file. Many tools which interact with a filesystem will sort a directory by the file name to be more palatable to a human reader.
- The notions of file name and element name have different syntax requirements. A file name can contain any character apart from the "/" character. There are much more stringent requirements on XML element names—no leading numbers, a large range of characters which are forbidden.
- For all XML elements with a common parent it is not required that each child's name be unique. Any two files in a directory *must* have different names.

The differences are shown in Figure 5.

The identification of this link between data models and various means to address the issues where differences arise helps both the semantic filesystem and XML communities by bringing new possibilities to both. For example, the direct evaluation of XQuery on a semantic filesystem instead of on an XML document. The blurring of the filesystem and XML also allows modern Office suites to directly manipulate filesystems [14].

The file name uniqueness issue is only present if XML is being seen as a semantic filesystem. In this case it can be acceptable to modify the file name to include a unique number as a postfix. In cases such as resolution of XPath

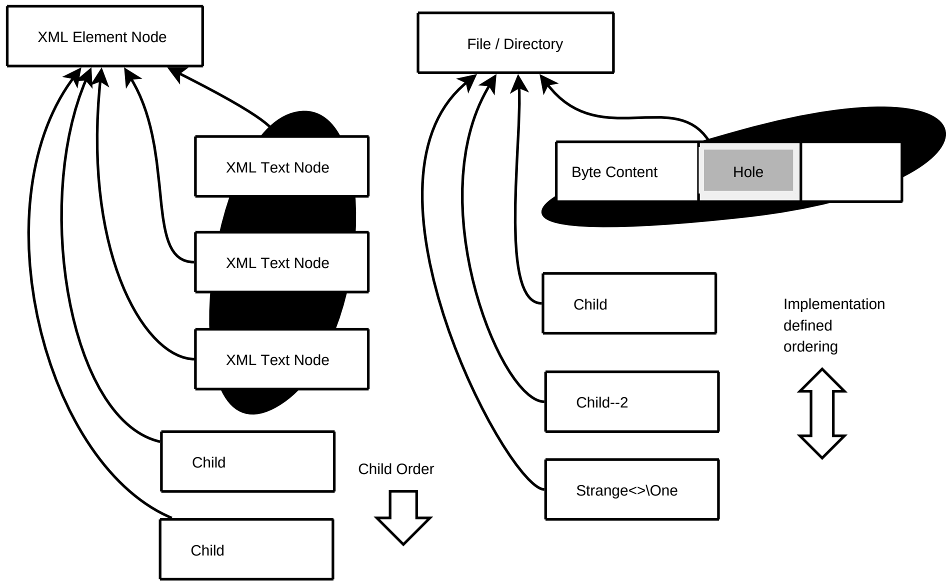

Figure 5: On the left an XML Element node is shown with some child nodes. On the right a filesystem node is shown with some similar child nodes. Note that XML Text nodes can be considered to provide the byte content of the synonymous filesystem abstraction but metadata about their arrangement can not be easily communicated. Child nodes in the XML side do not need to have unique names for the given parent node and maintain a strict document order. Child nodes on the filesystem side can contain more characters in their file names but the ordering is implementation defined by default.

or XQueries file names should be tested without consideration of the unique postfix so that query semantics are preserved.

As XML elements can not contain the "/" character exposing XML as a semantic filesystem poses no issue with mapping XML element names into file names. Unfortunately the heavy restrictions on XML element name syntax does present an issue. The most convenient solution has been found to be mapping illegal characters in file names into a more verbose description of the character itself. For example a file name "foo<bar>.txt" might be canonicalized to an XML element name of "foo-lt-bar-gt.txt". The original unmodified file name can be accessed through a name spaced XML attribute on the XML element.

XML element ordering can be handled by exposing the place that the XML element appeared in document order. For example, a document with "b" containing "c,d,e" in that order the "c" file would have a place of zero, and "e" would be two. With this attribute available the original document ordering can be obtained through the semantic filesystem by sorting the directory on that attribute. As there is no (useful) document ordering for a filesystem this is not an issue when exposing a filesystem as XML.

There is no simple solution to the fact that XML elements can have multiple text nodes as children. In cases where XML with multiple child nodes exist they are merged into a single text node containing the concatenation in document order of the child text nodes. Files with holes are presented as though the hole contained null bytes.

4 Information Search

In recent years much emphasis has been placed on so called "Desktop search". Few machines exist as islands and as such the index and search capabilities of any non trivial system must allow seamless integration of Internet, Intranet and desktop sources. The term "filesystem search" at least removes the connotations that search is limited to the desktop alone.

Details of indexing have been presented in prior publications [10, 9, 11]. Briefly the indexing capabilities in libferris are exposed through plugins. Much of the emphasis has been placed on indexing of metadata leaving full text indexing [17] to implementations such as Lucene and TSearch2. Two of the main metadata plugins use sorted Berkeley db4 files or a PostgreSQL database.

The chosen query syntax is designed based on the "The String Representation of LDAP Search Filters" [5]. This is a very simple syntax which provides a small set of comparative operators to make `key operator value` terms and a means to combine these terms with boolean `and`, `or` and `not`. All terms are contained in parenthesis with boolean combining operators located before the terms they operate on.

The comparative operators have been enhanced and in some cases modified from the original semantics [5]. Syntax changes include the use of == instead of = for equality testing. Approximate matching ~= was dropped in favor of regular expression matching using perl operator syntax =~. Operators which are specific to the LDAP data model have been removed. The operators and semantics are presented in Table 1. Coercion of `rvalue` is performed both for sizes and relative times. For example, "begin today" will be converted into the operating system's time representation for the start of the day that the query is executed.

OP	Semantics
=~	`lvalue` matches the regular expression in `rvalue`
==	`lvalue` is exactly equal to `rvalue`
<=	`lvalue` is less than or equal to `rvalue`
>=	`lvalue` is greater than or equal to `rvalue`

Table 1: Comparative operators supported by the libferris search syntax. The operators are used infix, there is a key on the left side and a value on the right. The key is used to determine which EA is being searched for. The `lvalue` is the name of the EA being queried. The `rvalue` is the value the user supplied in the query.

Resolution of the `and` and `or` is performed (conceptually) by merging the sets of matching file names using either an intersection or union operation respectively. The semantics of negation are like a set subtraction: the files matching the negated subquery are removed from the set of files matching the portion of the query that the negation will combine with. If negation is applied as the top level operation then the set of files to combine with is considered to be the set of all files. The nesting of `and`, `or` and `not` will define what files the negation subquery will combine with. As an example of negation resolution consider the `fquery` which combines a width search with a negated size search: `(&(width<=640)(!(size<=900)))`. The set of files which have a width satisfying the first subquery are found and we call this set A. The set of files which have a size matching the second part of the query, ie, `size<=900` are found and we call this set B. The result of the query is $A \setminus B$.

The eaq:// virtual filesystem takes a query as a directory name and will populate the virtual directory with files matching the query. Other closely related query filesystems are the eaquery:// tree. The eaquery:// filesystem is has slightly longer URLs but it allows you to set limits on the number of results returned and to set how conflicting file names are resolved. Some example queries are shown in Figure 6. Normally a file's URL is used as its file name for eaquery:// filesystems. The shortnames option uses just the file's name and when two results from different directories happen to have the exact same file name it appends a unique number to one of the result's file names. This is likely to happen for common file names such as README.

Full text queries can be evaluated using the `fulltextquery://` or `ftxq://` URL schemes. Both metadata and fulltext predicates can be evaluated to produce a single result filesystem [11].

One major area where the index and search in libferris diverges from similar tools is the application of Formal Concept Analysis (FCA) [3]. FCA can be seen as unsupervised machine learning and is a formal method for dealing with the natural groupings within a given set of data. The result of FCA is a Concept Lattice. A Concept Lattice has many formal mathematical properties but may be considered informally as a specialization hierarchy where the further down a lattice one goes the more attributes the files in each node have. Files can be in multiple nodes at the same time. For example, if there are two attributes (mtime>=begin last week) and (mtime>=begin last month) then a file with the first attribute will also have the latter.

Using the SELinux type and identity of the example 201,759 files the concept lattice shown in Figure 7 is generated. The concept 11 in the middle of the bottom row shows that user_u identity is only active for 3 fonts_t typed files. Many of the links to the lower concepts are caused by the root and system identities being mutually exclusive while the system identity combines with every attribute that the root identity does.

Readers interested in FCA with libferris should see [8, 16, 15].

4.1 XQuery

Being able to view an entire filesystem as an XML data model allows the evaluation of XQuery directly on the filesystem.

Libferris implements the native XQilla data model and attempts to offer optimizations to XQuery evaluation. Some of the possibilities of this are very nice, bringing together the db4, Postgresql, XML and file:// in a single XQuery.

There are also many efficiency gains that are available by having multiple data sources (filesystems) available. In an XML only query if you are looking up a user by a key things can be very slow. If you copy the users info into a db4 file and use libferris to evaluate your XQuery then a user lookup becomes only a handful of db4 btree or hash lookups.

Mounted postgresql functions allow efficient access to relational data from XQuery. Postgresql function arguments are passed through the file path from XQuery variables. This is a reasonable stop gap to being able to use prepared SQL statements with arguments as a filesystem. If the result is only a hand full of tuples with it will be very quick for libferris to make available to the xquery as a native document model.

A postgresql function is setup as shown in Figure 8. This can subsequently be used as any other read only filesystem as shown in Figure 9. A simple XQuery is shown in Figure 10 and its evaluation in Figure 11.

A major area which is currently optimized in the evaluation of XQuery with libferris is the evaluation of XPath expressions. This is done in the `Node::getAxisResult()` method specifically when the axis is `XQStep::CHILD`. Both files and directories are represented as Context objects in libferris. When seeking a specific child node `Context::isSubContextBound()` is used to quickly test if such a child exists. The `isSubContextBound()` method indirectly calls `Context::priv_getSubContext()` which is where filesystem plugins can offer the ability to read a directory piecewise.

The normal opendir(3), readdir(3) sequence of events to read a directory can be preempted by calls to `Context::priv_getSubContext()` to discover partial directory contents. As libferris is a virtual filesystem some other filesystem implementations also implement `Context::priv_getSubContext()` and piecewise directory reading. A specific example is the db4 filesystem which allows very efficient loading of a hand full of files from a large directory.

Where the direct evaluation on a filesystem as shown above becomes too slow the filesystem indexes can also be used in an XQuery by making an XPath that uses the filesystem interface to queries shown in Fig. 6.

A more complete example is shown in Figure 12. This uses the filesystem index and search along with XQuery variables to search for files which contain a person in a given location as a boolean `AND` style full text query. Note that multiple use of indexes, in particular the use of federated filesystem indexes [11] is possible together with immediate evaluation of other queries on db4 or RDF files to generate a combined result.

5 The Future

Closer integration of XML and libferris and in particular the ability to arbitrarily stack the two in any order. For example, being able to run an XQuery and take its results as the input to `xsltfs://` to generate an office document to edit with Open Office. As `xsltfs://` does not enforce a strict isomorphism between filesystems the resulting document when edited and saved could effect changes on both the underlying filesystem objects that the XQuery dealt with as well as any other desired side effects.

More efficient and user friendly access to the Formal Concept Analysis in libferris. There are still some complex persistence and processing tasks which need to be improved before the use of FCA on filesystems will see broad adoption.

References

[1] Fuse, http://fuse.sf.net. Visited Fed 2007.

```
# All files modified recently
$ ferrisls -lh "eaq://(mtime>=begin last week)"

# Same as above but limited to 100 results
# as an XML file
$ ferrisls --xml \
"eaquery://filter-100/(mtime>=begin last week)"

# limit of 10,
# resolve conflicts with version numbers
# include the desired metadata in the XML result
$ ferrisls --xml \
  --show-ea=mtime-display,url,size-human-readable \
"eaquery://filter-shortnames-10/(mtime>=blast week)"
```

Figure 6: Query results as a filesystem.

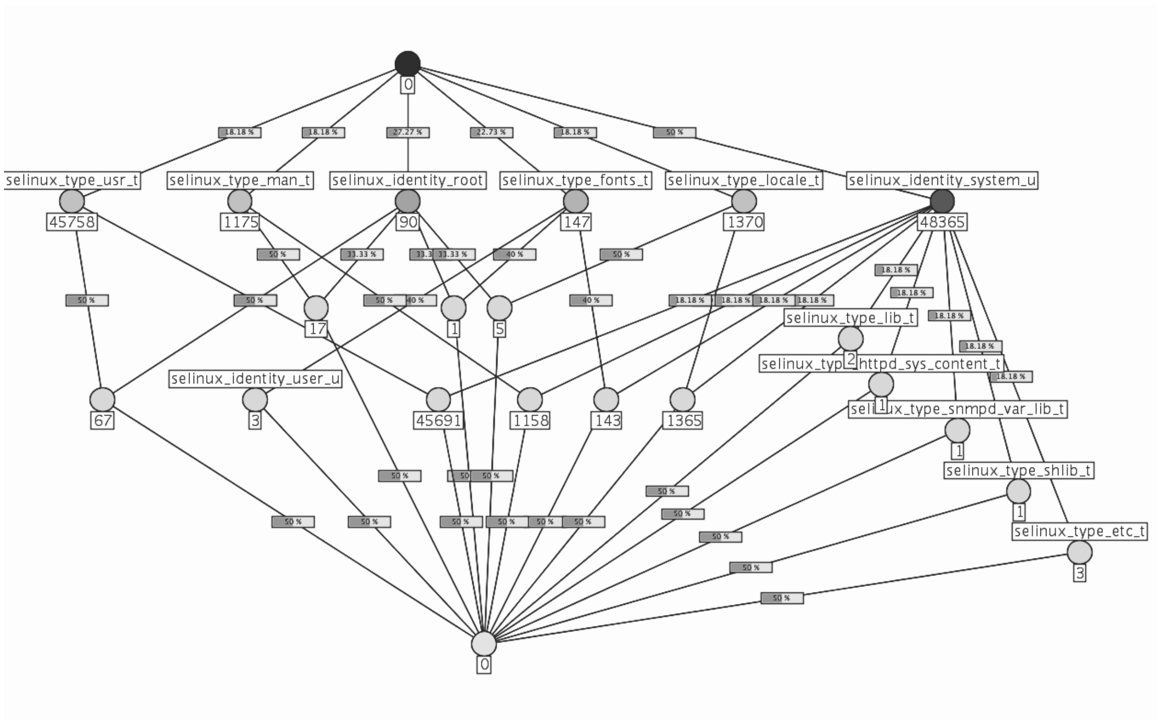

Figure 7: Concept lattice for SELinux type and identity of files in /usr/share/ on a Fedora Core 4 Linux installation. The Hasse diagram is arranged with three major sections; direct parents of the root are in a row across the top, refinements of selinux_identity_system_u are down the right side with combinations of the top row in the middle and left of the diagram. Attribute are shown above a node and they apply transitively to all nodes reachable downwards. The number of files in each node is shown below it.

```
$ psql junkdb
# CREATE TYPE junk_result
    AS (f1 int, f2 text);
# drop function junk( int, int );
# CREATE FUNCTION junk( int, int )
   returns setof junk_result
AS
$BODY$
DECLARE
  iter int;
rec junk_result;
BEGIN
  iter = $1;
for rec in select $1*10,$2*100 union
   select $1 * 100, $2 * 1000
LOOP
        return next rec;
END LOOP;
return;
END;
$BODY$
  LANGUAGE 'plpgsql' ;
# exit
```

Figure 8: Setting up a PostgreSQL function to be mounted by libferris

```
$ ferrisls --show-ea=f1,f2,name  --xml
"postgresql://localhost/play/junk(1,2)"
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<ferrisls>

  <ferrisls f1="" f2="" name="junk(1,2)"
url="postgresql:///localhost/play/junk(1,2)">
    <context f1="10" f2="200" name="10-200"/>
    <context f1="100" f2="2000" name="100-2000"/>
  </ferrisls>

</ferrisls>

$ fcat postgresql://localhost/play/junk(1,2)/10-200
...

$ ferriscp \
    postgresql://localhost/play/junk(1,2)/10-200 \
    /tmp/plan8.xml
```

Figure 9: Viewing the output of a PostgreSQL function with ferrisls

```
$ cat fdoc-pg.xq
<data>
 {
  for $b in ferris-doc("postgresql://localhost/play/junk(1,2)")
  return $b
 }
</data>
```

Figure 10: A Trivial XQuery to show the output of calling a PostgreSQL function through libferris

```
$ ferris-xqilla --show-ea=f1,f2,name fdoc-pg.xq
<?xml version="1.0"?>
<data>
  <junk_oper_1_comma_2_cper_ name="junk(1,2)">10,200
<number_10_dash_200 f1="10" f2="200" name="10-200">
&lt;context  f1="10"  f2="200"  /&gt;
</number_10_dash_200>
<number_100_dash_2000 f1="100" f2="2000" name="100-2000">
&lt;context  f1="100"  f2="2000"  /&gt;
</number_100_dash_2000></junk_oper_1_comma_2_cper_>
</data>
```

Figure 11: The evaluation of the XQuery in Figure 10. The embedded `&lt;` etc. shown below come from the "content" of the file which in this case is the same as the above ferrisls command.

```
$ cat xquery-index.xq
declare variable $qtype    := "boolean";
declare variable $person   := "alice";
declare variable $location := "wonderland";
<data>
 {
  for $idx in ferris-doc( concat("fulltextquery://", $qtype, "/",
           $person, " ", $location))
    for $res in $idx/*
       return
    <match
            name="{ $res/@name }" url="{ $res/@url }"
            modification-time="{ $res/@mtime-display }"
           >
    </match>
 }
</data>

$ ferris-xqilla xquery-index.xq
<?xml version="1.0"?>
<data>
  <match modification-time="99 Jul 27 12:53"
     name="file:///.../doc/CommandLine/command.txt ...>
  <match modification-time="00 Mar 11 06:58"
     name="file:///.../doc/Gimp/Grokking-the-GIMP-v1.0/node8.html
     ...>
...</data>
```

Figure 12: Running an XQuery which uses filesystem index and search. The `idx` XQuery variable will be the virtual directory containing the query results and for the sake of clarity the `idx` is then looped over explicitly in the XQuery. The person and location can easily be obtained from other sources making the fulltext query portion complement a larger information goal.

[2] libferris, http://witme.sf.net/libferris.web/. Visited Nov 2005.

[3] Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis — Mathematical Foundations*. Springer–Verlag, Berlin Heidelberg, 1999.

[4] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and James W. Jr O'Toole. Semantic file systems. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, ACM SIGOPS, pages 16–25, 1991.

[5] Network Working Group. Rfc 2254 - the string representation of ldap search filters, http://www.faqs.org/rfcs/rfc2254.html. Visited Sep 2003.

[6] Angelike Langer and Klaus Kreft. *Standard C++ IOStreams and Locales: Advanced programmer's Guide and Reference*. Addison Wesley, Reading, Massachusetts 01867, 2000.

[7] Ben Martin. File system wide file classification with agents. In *Australian Document Computing Symposium (ADCS03)*. University of Queensland, 2003.

[8] Ben Martin. Formal concept analysis and semantic file systems. In Peter W. Eklund, editor, *Concept Lattices, Second International Conference on Formal Concept Analysis, ICFCA 2004, Sydney, Australia, Proceedings*, volume 2961 of *Lecture Notes in Computer Science*, pages 88–95. Springer, 2004.

[9] Ben Martin. Filesystem indexing with libferris. *Linux Journal*, 2005(130):7, 2005.

[10] Ben Martin. A virtual filesystem on steroids: Mount anything, index and search it. In *Proceedings of the 12th International Linux System Technology Conference (Linux-Kongress 2005)*. GUUG e.V. / Lehmanns / Ralf Spenneberg, 2005.

[11] Ben Martin. Federated desktop and file server search with libferris. *Linux Journal*, 2006(152):8, 2006.

[12] Ben Martin. Geotagging files with libferris and google earth (linux.com), April 2006.

[13] Ben Martin. The world is a libferris filesystem. *Linux Journal*, 2006(146):7, 2006.

[14] Ben Martin. Virtual filesystems are virtual office documents. *Linux Journal*, 2007(154):8, 2007.

[15] Ben Martin and Peter W. Eklund. Spatial indexing for scalability in fca. In Rokia Missaoui and Jürg Schmid, editors, *ICFCA*, volume 3874 of *Lecture Notes in Computer Science*, pages 205–220. Springer, 2006.

[16] Ben Martin and Peter W. Eklund. Custom asymmetric page split generalized index search trees and formal concept analysis. In *ICFCA*, 2007.

[17] Ian H. Witten, Alistar Moffat, and Timothy C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, 340 Pine Street, San Francisco, CA 94104-3205, USA, 1999.

Proceedings of the Linux Symposium

Volume One

June 27th–30th, 2007
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
C. Craig Ross, *Linux Symposium*

Review Committee

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
Dirk Hohndel, *Intel*
Martin Bligh, *Google*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat, Inc.*
C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
Gurhan Ozen, *Red Hat, Inc.*
John Feeney, *Red Hat, Inc.*
Len DiMaggio, *Red Hat, Inc.*
John Poelstra, *Red Hat, Inc.*