QueryFS, a virtual filesystem based on queries, and related tools.

Michael A. Raskin

December 29, 2018

Abstract

Modern hardware and software allow users to store and transmit huge data collections. Applications can rely on filesystem (or database) interface for most tasks. Unfortunately, indexing and searching these data collections has to be done using specialized tools with limited interoperability with existing software. This paper describes a tool providing a unified POSIX filesystem interface (using FUSE) to the results of search queries. The search queries themselves may be expressed using high-level languages, including SQL and specialized Common Lisp API.

1 Introduction

Modern filesystems allow users to store large volumes of data. When the data has some special structure, an SQL database may better suited for the task. In both cases there are many software packages implementing the same interface. Applications use the same interface to access multiple storage implementations; and many applications developed before a technology improvement becomes available still benefit from it. For example, SBCL has no need to know about RAID0 to get improved write speed. Neither it needs to know about SSH to read source from server using SSHFS-FUSE.

There are also many tools to find data in the storage. Some of them traverse all the storage to find the needed piece of information, some create and maintain indices, some expect user to explicitly add the data into indexed area.

Unfortunately, making these tools interact with unsuspecting applications is often hard and the query language may have limited expressive power.

This paper describes how QueryFS project tries to solve the problems of using query results in applications unaware of any special API, saving queries for future use and expressing complicated conditions with queries.

2 Existing Projects

To refine the goals and to give general information about previous work in this area this section contains a list of some projects or products working on similar

problems.

The problem of finding a file in the storage is probably as old as the very notion of file. Even modern systems have utilities which can be traced to the very time when the word "file" got a meaning related to computers. POSIX requires utility called "find" to be present. If we only consider interaction with programs following the original Unix design principles, "find" satisfies all the conditions outlined in the previous section; its command line can be easily saved to a text file, it generates output convenient to feed to programs with command-line interface, and it allows arbitrary logical combinations of basic conditions in queries. Unfortunately, modern GUI programs will not let user easily feed "find" output to a file selection dialog.

Another old example is feeding of search results to the UI element intended for directory view in many file managers. In current versions of Gnome Nautilus, Windows Explorer or MacOS X Finder user can save such a search query and interact with it as if it was a folder. The main problem is that applications unaware of this feature cannot use such directories. Even *WinFS* project by Microsoft was going to require applications to use special API to access such search folders.

Inability of some applications to access virtual directories and use plain text file lists can be mitigated by using *FUSE*. It allows mounting special filesystems and processing of the filesystem operation in the userspace.

For example, *beaglefs*, uses indices created by Beagle desktop search system. User can mount a directory filled with symbolic links to all files matching a Beagle query by a single invocation of "beaglefs" command, which makes use of search results in other applications trivial. Saving queries is as easy as creating a shell script. Unfortunately, the expressive power of Beagle queries is relatively limited.

Some of the filesystems emphasize user-entered metadata. For example, *tagfs* and *movemetafs* support marking each file with tags (arbitrary strings) instead of building file hierarchies. User can then go into a virtual directory which contains only the files having all the tags from some list. The full path of the directory can easily be saved as a symbolic link. Unfortunately, even file size cannot be taken into account in such queries.

The *libferris* project (together with *ferris-fuse*) provides means to access many different types of metadata found inside common file types. *libferris* on its own requires use of special API or utilities to access the data, but allows complicated queries in query languages like XPath and SQL. The FUSE filesystem, *ferris-fuse* only allows browsing the data. Another project, *BaseX*, uses XQuery language and has GUI and command-line tools for browsing indexed data. Currently, *BaseX* lacks *FUSE* support.

The *RelFS* project has its focus on representing SQL queries as directories. A *RelFS* filesystem can store files and symbolic links like an ordinary filesystem. It also allows going into a directory with name starting with "#" symbol, which is interpreted as an SQL query. Running "find" on such a directory returns approximately the same result as running the SQL query put into directory name. *RelFS* uses SQL query language, allows queries to return complicated directory

trees, and allows saving queries as symbolic links. Unfortunately, *RelFS* queries process only files and symbolic links stored on the *RelFS* filesystem itself, and storing large files on *RelFS* causes performance problems.

Two projects with the most radical goals, *dbfs* and *Hyppocampus*, store all the files inside the DB and have no hierarchical structure. All available ways to access files create special SQL queries.

3 Wishlist for QueryFS

QueryFS project started as an attempt to reimplement *RelFS* and remove some of its weaknesses.

It is obvious that a *FUSE* filesystem will never beat a well-optimised kernel filesystem in storing large files, so *QueryFS* is not supposed to carry files of any significant size. Whenever such files should appear in earch results, symbolic links will be used.

This decision rules out maintaining up-to-date indices by monitoring access to the filesystem itself. Fortunately, there are a lot of other projects dedicated to indexing of data (e.g. aforementioned Beagle and BaseX). Some of them even use some kind of filesystem event notifications supported by recent operating system kernels to update information in real-time. That means that ease of adding an interface to such a "foreign" index is much more important than non-trivial indexing implemented inside *QueryFS*.

As the queries should be easy to use from other applications unaware of *QueryFS*, they should be represented as directories. Actually, all the content of a *QueryFS* instance is generated this way. Some of the queries may provide access to internals of the filesystem, e.g. allow loading new queries by writing to a special file.

One of the design goals is allowing user to save queries. *QueryFS* takes an extreme position by making it hard to use a query without saving it. Neither the core nor example plugins and queries support this. By default, *QueryFS* expects path to a directory having subdirectories "results" (future mountpoint), "queries" (user queries) and "plugins" (non-core code, expected to define ways of parsing queries in different format). Loading queries from files also eliminates syntactical problems related to fitting complex expressions into a command or even filesystem paths (*RelFS* and *tagfs* actually do put queries into filesystem paths).

4 Provided interface

QueryFS tries to make adding support for a new query type as easy as possible. To make query parsers easily replaceable, *QueryFS* is split into core code (FUSE interaction, path management, basic plugin and query management), plugins (query parsers and helper functions for queries) and queries (generators of filesystem contents).

In general, lifecycle of *QueryFS* instance looks like the following.

When *QueryFS* is launched, it loads plugins. Plugins can register query parsers. Afterwards queries are loaded. Query parser corresponding to a query file currently depends only on the file extension. The parser receives the query and returns source code which describes the resulting layout. Layout is described in declarative terms, all path processing is done in the core code. This code is labeled with the query file name (without extension) and saved. After all queries are parsed, all the generated layout code is compiled and executed as needed to answer filesystem requests. Execution results can be cached for a short amount of time (mainly to handle cases like "ls -l" command), but these caches are invalidated when a file or a directory is created or removed.

5 Implementation

QueryFS is written Common Lisp, because parsing queries have to be translated into the main implementation language and it is way simpler with Lisp language family. Some parts of *QueryFS* and *CL-FUSE* currently require *Steel Bank Common Lisp* to run.

The lowest two levels written in Lisp are a wrapper around *FUSE* library implemented using *CFFI* and a more Lisp-like API for implementing *FUSE* filesystem without constant use of *CFFI*.

Next abstraction level allows defining filesystem layout in a declarative syntax without reimplementing path processing each time. It is made easier by the fact that Lisp programs are represented by trees in the most explicit way. There are two ways to write such a description. User can specify a literal tree structure with attributes in nodes; Lisp macrosystem allowed to create a simpler syntax for creating "standard" nodes. For example, `(mk-file "README" "This is QueryFS")` and `` `(:TYPE :FILE :NAME "README" :CONTENTS ,(LAMBDA () "This is QueryFS") :WRITER NIL :REMOVER NIL) `` mean the same: a file named "README" with text "This is QueryFS" should be created in the directory described by containing expression, it should not be writeable or removable. Operations currently used in *QueryFS* plugins are: "mk-file", "mk-dir", "mk-symlink" for describe filesystem contents, "mk-creator" for describing entry addition and "mk-pair-generator" for easier generation of filesystem structure based on information retrieved from external sources or computed at run time. The first three operations just accept expressions that will be evaluated to retrieve their names and contents. For files there can be extra expressions to handle file modification or removal. "mk-creator" accepts expression that need to be evaluated to create a file or directory entry in containing directory.

The last operation, "mk-pair-generator", requires an expression returning list of contents and an expression with a free parameter which can give details about each entry. The first expression returns a list of lists, where first entry of each list is entry name and the rest should be used when evaluating entry details.

There is also a more general operation, "mk-generator", which allows use of

independent content lister and name parser. This allows special features like allowing to access HTTP URLs by accessing "http/www.example.org/path/to/file". Currently, no *QueryFS* plugin is able to generate code using this feature.

Next level is *QueryFS* itself. As described in the previous section, all the functionality of its core is related to handling of queries and plugins.

Loading plugins is done in a very straightforward way. There are some checks allowing to specify either full path or just the file name (if it is in the plugin directory); but basically it is one "load" call wrapped in error handling. Some of the query parsing code in the core is present only to be used by plugins. More specifically, there are two macros, "def-query-parser" and "def-linear-query-parser" to generate code that can be used in any plugin.

The first macro, "def-query-parser", acts in a way similar to Lisp function definition syntax. It simply defines a function with the specified body and registers it as query parser for specified type of queries. The second one assumes that query can be parsed by reading first "word" in Lisp sense and looking for it in a list of actions. It generates an invocation of the first macro and additional code to do the matching. This approach allows reusing the parser components in plugins.

Plugins are loaded as is and can select their own namespace to use. They are supposed to use the same namespace as the core *QueryFS* code.

To describe query grammar for a parser, one can use "Parser Expression Grammars". *Esrap-PEG*, a wrapper around existing *Esrap* packrat parser supporting stadard "PEG" syntax, was developed for the needs of *QueryFS*.

Loading queries is only a bit more complicated. Each query is processed by one of a few query parsers; it also gets loaded into its own namespace.

Currently, the most polished plugin is SQL2. It provides a syntax based on bash and SQL to represent results of SQL queries as directories.

```
 transient master_password "" setter "::password"
master_password

for p in "select username || '@' || service, password from passwords where
username is not null and ${master_password} <> ''" encrypt-by $master_password
$(with-file $name do on-read $p[1]; done)

mkdir "by-service" do

  on-create-dir name "insert into passwords (service) values (${name})"

  grouped-for srv in "select distinct service from passwords where
  ${master_password} <> ''" do on-create-file name "insert into passwords
  (service, username) values (${srv[0]}, ${name})"

  with-file "::remove::" do on-read "" on-write data "delete from
  passwords where service = ${srv[0]}" done
```

```
for un in "select username, password from passwords where service =
${srv[0]} and username is not null" encrypt-by $master_password $(
with-file $name do on-read $un[1] on-write data "update passwords set
password = ${data} where username = ${name} and service = ${srv[0]}"
on-remove "delete from passwords where username = ${name} and service
= ${srv[0]}" done) done done
```

6 Stability and security

QueryFS uses *CL-FUSE* functionality to catch errors in run time. So a query with a mistake should not easily take down the entire filesystem instance. On the other hand, both malicious query and malicious plugin amount to arbitrary code running with user priviliges, so untrusted plugins and queries should not be run.

As the queries are processed with plugins, plugins may limit code generation to exclude unsafe function calls. Unfortunately, doing this well requires developing a security model that can allow loading "safe" external libraries and has correct definition of "safe". For queries this may be worked around if plugins wrap the library calls they consider safe. Anyway, if a query uses SQL and it is supposed to issue "DELETE FROM" sometimes, it can just drop the database unless a powerful query analyzer is used. Security model for plugins is an even more complicated task, because they can do whatever queries can, but they are also the natural place to put wrapper over foreign code; and even well-intentioned third-party code may be useful to a malicious plugin if such third-party code can write to a file.

7 Reusable libraries written

Regardless of your opinion about *QueryFS* project, you may find one of its library useful.

7.1 CL-Fuse

A wrapper around *FUSE* libraries for Common Lisp.

7.2 Esrap-PEG

A library to support standard programming language independent "PEG" syntax for parser generation.

8 Future plans

One of the long-term ideas of *RelFS* project was storing metadata in a SQLite database in the directory containing relevant data. For example, a USB HDD

could contain a picture archive and a database with metadata for all the pictures. User could attach the database to *QueryFS* instance and select files by metadata.

Another feature *QueryFS* could eventually support is helping user to manage DB schema for metadata if user wants to create metadata manually or using scripts. Currently setting up the tables has to be done manually.

A "SPARQL" plugin for *QueryFS* would allow convenient means of experimentation with NoSQL metadata storage.

From the point of view of concrete applications, creating a convenient schema for storing email would be a good demonstration. Currently I read all my email using special *QueryFS* queries, but there is much space for improvement to make search more convenient.

References

[1] FUSE project, `http://fuse.sf.net`

[2] RelFS project, `http://relfs.sf.net`

[3] Holupirek, Grün, Scholl: BaseX and DeepFS — Joint Storage for Filesystem and Database. EDBT 2009 (Demo Track), March 2009. `http://www.inf.uni-konstanz.de/dbis/publications/download/joint_storage.pdf`

[4] libferris project, `http://libferris.com`

[5] Gorter, O.: Database File System — An Alternative to Hierarchy Based File Systems. `http://tech.inhelsinki.nl/dbfs/dbfs-screen.pdf`