

# Constructing Virtual Filesystems with Query Semantics

Michael Raskin\*

Moscow State University

**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.

**Key words:** filesystems, search, domain-specific languages

## 1 Introduction

If someone wants to store and retrieve a lot of data, modern filesystems allow to do it. For some special cases SQL database will do the job better. In both cases, there are many available tools implementing the same interface. Applications may use the same interface to access multiple storage implementations; and applications developed before some technology improvement still benefit from it in most cases. For example, compiler knows nothing about RAID5, but still gets increased IO speed. An example more relevant to this paper is an image viewer happily showing images shared via NFS or SMB.

If the question is about searching data, things get more complicated. There are many tools for searching data in storage. Some of them use previously created

---

\*

indices, some process the data after getting the request. User can usually search by file name and size. Many programs can also take into account popular types of metadata.

On the other hand, most of the search tools have limited abilities for interacting with random third-party programs unaware of the search API — and each tool may have its own API. Saving a search query may be hard or impossible. Query language may have very limited expressiveness — for example, it can list the conditions and allow to apply either conjunction or disjunction.

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 Filesystems Based on Searches or Queries

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* caused 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 Basic Design of 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 search 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).

*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.

## 4 Implementation of the QueryFS core

For implementing *QueryFS* I chose Common Lisp. 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 direc-

tory); 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.

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.

## 5 Currently Implemented QueryFS plugins

Currently, there are only a few *QueryFS* plugins. First, there is “generic-readers” plugin that provides some parser components used by other plugins. Three plugins (“lisp-queries”, “sexp-queries” and “sql”) define query parsers.

“sexp-queries” plugin is the simplest of them all. It uses no keywords. It reads an s-expression and tries to represent it as a directory structure. For example, (`"doc" ("README" "This is a sample README") ("TODO" "Submit the paper")`) describes a directory named “doc” with two files, “README” and “TODO”.

“lisp-queries” plugin uses keywords and some additional syntax notations to illustrate the usage of query parsers. The queries written using it do not differ too much from queries that would be written directly using “mk-file” etc. Some example queries use this syntax to provide filesystem control interface via special files.

“sql” uses SQL queries to fill and update parts of filesystem. It uses special notation to avoid the need for inserting Lisp code into queries whenever it can be avoided. Use of *QueryFS* layout code allows to describe multi-level classifications step by step, without putting all the logic into one big SQL query. The SQL query results are cached; this gives significant speed improvement for commands like “ls -l”.

Two other plugins, “filenames” and “password-store”, just provide functions for use in queries.

## 6 Some of the Currently Implemented QueryFS Queries

This section contains descriptions of some sample *QueryFS* queries.

“commands” query from the distribution can be used to control *QueryFS*. It allows to reload all queries, shutdown *QueryFS*, execute snippets of code inside *QueryFS*, view the last error captured during query execution. It is implemented using verbatim Lisp handlers for writing to file. Arguably, some of this code could be moved to a plugin.

A more interesting query using Lisp is “lisp-fs”. It represents the Lisp memory image as a filesystem. The filesystem allows browsing namespaces, reading documentation, examining global variables and even changing them.

Among SQL-based *QueryFS* queries there are “password-store” and “tags”. “password-store” implements encrypted password storage. AES encryption uses a third-party Lisp cryptography library. New entries are created by “mkdir” to create a directory for all accounts on some service and subsequent echoing of the password in to a file with the same name as the account. SQL queries use some minimal amount of embedded Lisp code for calling wrapped encryption routines.

“tags” implements primitive file tagging support. Its interface allows creating a tag via “mkdir” and tagging a file by echoing its name into a special file inside tagging query result.

## 7 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 privileges, 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.

## 8 Practical Testing and Further Directions

As *QueryFS* shows, implementing all the infrastructure for a query-based filesystem on top of *FUSE* requires a relatively small amount of code. To measure *QueryFS* performance, data about around 3000 files were put into an SQL table, and a query was written to return symbolic links to these files. The links were split in directories according to the first letter of the name. Running “ls -color=always” on all the directories in the query result took around 7 seconds on a 1.8 GHz Celeron.

Further directions include getting feedback about query syntax and improving query syntax according to this feedback; writing plugins to interact with existing data indexing and retrieving projects; designing some security model

for *QueryFS* queries; developing applications that can generate queries based on user input.

Currently, the project can be found at <http://mtn-host.prjek.net/projects/cl-fuse/>.

## 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](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>

## Appendix A. Brief Description of SQL Plugin Query

### Keywords

lisp-escape	read an s-expression and execute it as Lisp code
end	close one level and go up
transient	define a variable and set initial value
db	set db settings (specify parameter name, way of obtaining value — set directly or read from mentioned file, and value or file name)
local-db	after that point db reconfiguration stays local to current query
file	create an immutable file
symlink	create a symlink
dir	create a directory
sink	create a writeable file
sink-removable	create a writeable file which can be deleted
create, create-dir	specify handlers for file creation and directory creation
with, list	create directories according to the query result (or supplied list and put something into them)
for, query-data	create an object for each row in query result

## Appendix B. Lisp-FS Query Source

```
with pkg from (mapcar (lambda (s) (list (package-name s) s))
                    (list-all-packages))
file "::nicknames" (fmt "~{~a~%~}" (package-nicknames pkg))
file "::uses" (fmt "~{~a~%~}" (package-use-list pkg))
file "::used-by" (fmt "~{~a~%~}" (package-used-by-list pkg))
with symb from (loop for s being each present-symbol in pkg collect
                 (list (symbol-name s) s))
file "::documentation" (with-output-to-string (s) (describe symb s))
file "::package" (package-name (symbol-package symb))
file "::boundp" (fmt "~a~%" (boundp symb))
file "::fboundp" (fmt "~a~%" (fboundp symb))
sink "::contents" (fmt "~s~%" (ignore-errors (symbol-value symb)))
```

```
(data
  (let* ((str (ignore-errors
              (sb-ext:octets-to-string data
                :external-format :utf-8)))
        (code (ignore-errors (with-input-from-string (s str)
                                                       (read s))))))
    )
  (set symb code)))
```

## Appendix C. Tags Query Source

```
dir "tags"
create-dir data ("insert into tags (tag) values (~a)" data)
sink data "::untag-file::" ""
("delete from tags where path = ~a" data)
with tag name-position 0 from
"select distinct tag from tags where
tag is not null and tag <> ''"
sink data "::tag-file::" ""
("insert into tags (tag, basename, path)
values (~a, ~a, ~a)" (tag 0)
(progn
  (setf cl-fuse::*last-wrapper-error*
        (format nil "~s ~s" data (basename data)))
  (basename data) data)
sink data "::forget-tag::" ""
("delete from tags where tag = ~a" (tag 0))
with filename name-position 0 from
("select distinct basename from tags where tag = ~a and
basename is not null and basename <> ''"
```

```

(tag 0))
for obj name-position 0 from ("select id, path from tags
where tag = ~a and basename = ~a" (tag 0) (filename 0))
symlink name (obj 1)
sink data "::untag-file::" ""
("delete from tags where path = ~a and tag = ~a" data
(tag 0))
sink data "::untag-all::" ""
("delete from tags where tag = ~a and basename = ~a"
(tag 0) (filename 0))
end
end
end

```

Result listing:

```

queries/tags/tags:
my-articles  ::untag-file::

queries/tags/tags/my-articles:
::forget-tag:: query-fs.ltx  ::tag-file::

queries/tags/tags/my-articles/query-fs.ltx:
35  ::untag-all::  ::untag-file::

```

## Appendix D. An Implementation Note About CL-FUSE

*CL-FUSE* uses event loop translated from the *FUSE* source, and runs it in a fresh pthread, just like *FUSE* does. The reason for that is that “fuse\_main” creates its own threads and sets signal handlers for them. This behavior is incompatible with *SBCL* use of signals for organizing garbage collection. That’s why event loop has to be run inside Lisp thread. Running *CL-FUSE* not in a fresh pthread

is possible, but attempts to use some system calls lead to unpredictable results. As even single-threaded *FUSE* runs its event loop inside a fresh pthread, this behavior was also copied.