Robinhood v2

Temporary Filesystem Manager

Admin Guide

Thomas LEIBOVICI
CEA/DAM

<thomas.leibovici@cea.fr>

V2.2.0

August 3rd, 2010

Table of contents

1 Product description 3
1.1 Overview 3
1.2 Execution modes 4
1.3 Internal architecture overview 4
2 First steps with Robinhood v2 5
2.1 Compiling and installing 5
2.2 Robinhood service 6
2.3 Command line options 6
2.4 Signals 7
2.5 Creating the database 8
2.6 Enabling Lustre Changelogs 9
3 Writing configuration file 10
3.1 Syntax 10
3.2 Configuration template and default parameters 13
3.3 General parameters 13
3.4 Log and alerts parameters 14
3.5 File classes 15
3.6 Purge policies 16
3.7 Purge triggers 17
3.8 Purge parameters 19
3.9 Specifying ‘rmdir’ policy 19
3.10 ‘Rmdir’ parameters 20
3.11 Periodic fileclass matching 20
3.12 Periodic information update when handling Lustre changelogs 21
3.13 Database parameters 21
3.14 Filesystem scan parameters 22
3.15 Lustre 2.x changelog parameters 23
3.16 Entry processor pipeline options 23
4 Reporting tool 25
4.1 Overview 25
4.2 Command line 25
4.3 Reports 26
5 System and database tunings 29
5.1 Lustre tunings and workarounds 29
5.2 Linux kernel tunings 30
5.3 Optimize Robinhood for reporting 30
5.4 Database tunings 30
6 Known bugs 31

1 Product description

1.1 Overview

“Robinhood FS Monitor” is Open-Source software developed at CEA/DAM for monitoring and purging temporary filesystems. It is designed in order to perform all its tasks in parallel, so it is particularly adapted for managing large file systems with millions of entries and petabytes of data. Moreover, it is Lustre capable i.e. it can monitor usage per OST and also purge files per OST.

A specific mode of Robinhood version 2 also makes it possible to synchronize a cluster filesystem with a HSM by applying admin-defined migration and purge policies. However, this document only deals with temporary filesystem management purpose.

Temporary filesystem management mainly consists of the following aspects:

- Scanning a large filesystem quickly, to build/update a list of candidate files in a database. This persistent storage ensures that a list of candidates is always available if a purge is needed (for freeing space in filesystem). Scanning is done using a parallel algorithm for best efficiency;

- Monitoring filesystem and Lustre OST usage, in order to maintain them bellow a given threshold. If a storage unit exceeds the threshold, it then takes the most recent list of files and purges them in the order of their last access/modification time. It can also only purge the files of a given OST. Purge operations are also performed in parallel so it can quickly free disk space;

- Removing empty directories that have not be used for a long time, if the administrator wants so;

- Raising alerts, generating accounting information and all kind of statistics about the filesystem.

All those actions are done according to very customizable policies. Thus, it makes it possible for you to preserve data of nice and responsible users, and penalize “abusers” and harmful behaviors… That’s why it is called Robinhood ;-)

Thanks to all of its features, Robinhood will help you to preserve the quality of service of your filesystem and avoid problematic situations.

1.2 Execution modes

Robinhood can be executed in 2 modes:

- As an ever-running daemon.
- As a “one-shot” command you can execute whenever you want.

In the daemon mode:

- Robinhood regularly refreshes its list of filesystem entries by scanning the filesystem it manages, and populating a database with this information.
- It constantly monitors filesystem and OST usage, and purge entries whenever needed;
- It also regularly checks empty, unused directories, if you enabled this feature.

In one-shot mode, each action is made once, and then the program exists:

- It scans the filesystem once and update its database;
- It checks filesystem and OST usage, and purge entries only if needed;
- It checks empty directories, if you enabled this feature.

Note that you can use any combination of actions in daemon and one-shot mode. For example, you can make a daily scan of the filesystem (as a one-shot scan) and have a Robinhood daemon that only checks for filesystem and OST usage and purge entries when needed.

1.3 Internal architecture overview

Robinhood v2 uses a database engine for managing its list of filesystem entries, which offers a lot of benefits:

- It can manage larger filesystems, because the size of its list is not limited by the memory of the machine.
- The list it builds is persistent, so it can immediately purge filesystem entries, even after the daemon restarted. This also makes ‘one-shot’ runs possible.
- Scan and purge actions can be run on different nodes: no direct communication is needed between them; they only need to be database clients.
- Administrator can collect custom and complex statistics about filesystem content using a very standard language (SQL).

Filesystem entry processing is highly parallelized: a pool of threads is dedicated to scanning the namespace (readdir operations). Those threads then push listed entries to a pipeline. Then, other operations are performed asynchronously by another pool of threads:

- Getting attributes;
- Checking if entry is up-to-date in database;
- Getting stripe info if it is not already known;
- Checking alert rules and sending alerts;
- Insert/update the entry in the database.

Thanks to its complex Boolean expression engine, Robinhood v2 policies and alert rules are flexible and easy to configure. A large range of attributes can be tested: name, path, type, owner, group, size, access or modification time, extended attributes, depth in namespace tree, number of entries in directory…

Various ways of triggering purges, to fit everyone's need:

- Purge triggers can be based on watermarks on OST usage or filesystem usage;
- Quota-like triggers can also be specified: if a given user or group exceeds a specified volume of data, then a purge is triggered on its files;
- Purge can be triggered when the inode count of a filesystem exceed a watermark.

2 First steps with Robinhood v2

2.1 Compiling and installing

It is advised to build a RPM from sources on your target system, so the program will have a better compatibility with your local Lustre and database version.

First, make sure the following packages are installed on your machine:

- mysql-devel
- lustre API library (if Robinhood is to be run on a Lustre filesystem): '/usr/include/liblustreapi.h' and '/usr/lib/liblustreapi.a' are installed by Lustre rpm.

Unzip and untar the sources:

```
tar zxvf  robinhood-2.2.0.tar.gz
cd robinhood-2.2.0
```

Then, use the "configure" script to generate Makefiles:

- use the `--with-purpose=TMP_FS_MGR` option for using it as a temporary filesystem manager;

```
./configure --with-purpose=TMP_FS_MGR
```

Other '`./configure`' options:

- You can change the default prefix of installation path (default is /usr) using: '`--prefix=<path>`'
- If you want to disable Lustre specific features (getting stripe info, purge by OST…), use the '`--disable-lustre`' option.

Finally, build the RPM:

```
> make rpm
```

A ready-to-install RPM is generated in the 'rpms/RPMS/<arch>' directory. The RPM is tagged with the lustre version it was built for.

The RPM includes:

- 'robinhood' and 'rbh-report' binaries
- 'rbh-config' command (configuration helper)
- Configuration templates
- /etc/init.d/robinhood script

NOTE: rpmbuild compatibility

Robinhood spec file (used for generating the RPM) is written for recent Linux distributions (RH5 and later). If you have troubles generating robinhood RPM (e.g. undefined rpm macros), you can switch to the older spec file (provided in the distribution tarball):

```
> mv robinhood.old_spec.in robinhood.spec.in
> ./configure ….
> make rpm
```

2.2 Robinhood service

Installing the rpm creates a 'robinhood' service. You can enable it like this:

```
> chkconfig robinhood on
```

This service starts one 'robinhood' instance for each configuration file it finds in '/etc/robinhood.d/tmpfs' directory.
Thus, if you want to monitor several filesystems, create one configuration file for each of them.

NOTE: Suze Linux operating system

On SLES systems, the default dependency for boot scheduling is on "mysql" service.
However, in many cases, it could be too early for starting robinhood daemon, especially if the filesystem it manages is not yet mounted. In such case, you have to modify the following lines in `scripts/robinhood.init.sles.in` before you run `./configure`:

```
# Required-Start:    <required service>
```

2.3 Command line options

```
Usage: robinhood [options]

Action switches:
    -S, --scan
        Scan filesystem namespace.
    -P, --purge
        Purge non-directory entries according to policy.
    -R, --rmdir
        Remove directories according to policy.
    -r, --read-log  (Lustre 2+ only)
        Handle events from Lustre MDT ChangeLog.

    Default mode is: --scan --purge –-rmdir

    On Lustre 2:
    Default mode is: --read-log --purge –-rmdir

Manual purge actions:
    --purge-ost=ost_index,target_usage_pct
        Purge files on the OST specified by ost_index until it reaches the
        specified usage.
```

```
    --purge-fs=target_usage_pct
        Purge files until the filesystem usage reaches the specified value.

Behavior options:
    --dry-run
        Only report actions that would be performed (rmdir, purge)
        without really doing them.
    -i, --ignore-policies
        Force purging all eligible files, ignoring policy conditions.
    -O, --once
        Perform only one pass of the specified action and exit.
    -d, --detach
        Daemonize the process (detach prom parent process).

Config file options:
    -f file, --config-file=file
        Specifies path to configuration file. By default, takes the file
        file it finds in /etc/robinhood.d/tmpfs.
    -T file, --template=file
        Write a configuration file template to the specified file.
    -D, --defaults
        Display default configuration values.

Filesystem options:
    -F path, --fs-path=path
        Force the path of the filesystem to be managed
        (overides configuration value).
    -t type, --fs-type=type
        Force the type of filesystem to be managed
        (overides configuration value).

Log options:
    -L logfile, --log-file=logfile
        Force the path to the log file (overides configuration value).
        Special values “stdout” and “stderr” can be used.
    -l level, --log-level=level
        Force the log verbosity level (overides configuration value).
        Allowed values: CRIT, MAJOR, EVENT, VERB, DEBUG, FULL.

Miscellaneous options:
    -h, --help
        Display a short help about command line options.
    -V, --version
        Display version info
    -p pidfile, --pid-file=pidfile
        Pid file (used for service management).
```

2.4 Signals

Robinhood traps the following signals:

- SIGTERM (kill <pid>) and SIGINT: perform a clean shutdown;
- SIGHUP (kill –HUP <pid>): reload dynamic parameters from config file.

2.5 Creating the database

Before running Robinhood for the first time, you must create its database.

- Install MySQL (mysql and mysql-server packages) on the node where you want to run the database engine.

- Start the database engine :
  `service mysqld start`

- Use the 'rbh-config' command to check your configuration and create Robinhood database:

  ```
  # check database requirements:
  rbh-config precheck_db

  # create the database:
  rbh-config create_db
  ```

- If no option is given to `rbh-config`, it prompts for configuration parameters (interactive mode). Else, if you specify parameters on command line, it runs in batch mode.

Alternatively, you can perform the following steps of your own, without using the 'rbh-config' script:

- Create the database (one per filesystem) using the `mysqladmin` command:
  `mysqladmin create <robinhood_db_name>`

- Connect to the database:
  `mysql <robinhood_db_name>`

  Then execute the following commands in the MySQL session:

  - Create a robinhood user and set its password (MySQL 5+ only):
    `create user robinhood identified by 'password';`
  - Give access rights on database to this user (you can restrict client host access by replacing '%' by the node where robinhood will be running):
    Mysql 5:
    ```
    GRANT USAGE ON robinhood_db_name.* TO 'robinhood'@'%' ;
    GRANT ALL PRIVILEGES ON robinhood_db_name.* TO 'robinhood'@'%';
    ```
    Mysql 4.1:
    ```
    GRANT USAGE ON robinhood_db_name.* TO 'robinhood'@'%'
    identified by 'password';
    GRANT ALL PRIVILEGES ON robinhood_db_name.* TO 'robinhood'@'%';
    ```
  - Refresh server access settings:
    `FLUSH PRIVILEGES ;`
  - You can check user privileges using:
    `SHOW GRANTS FOR robinhood ;`

- For testing access to database, execute the following command on the machine where robinhood will be running :

```
mysql --user=robinhood --password=password --host=db_host
robinhood_db_name
```

If the command is successful, a SQL shell is started. Else, you will get a ‘permission denied’ error.

- For now, the database schema is empty. Robinhood will automatically create it the first time it is launched.

2.6 Enabling Lustre Changelogs

With Lustre 2.x, file system scans are no more required to update robinhood’s database: it can collect events from Lustre using Lustre’s ChangeLog mechanism. This avoids over-loading the filesystem with namespace scans!

You can simply enable this feature by running ‘rbh-config’ on the MDS:

```
> rbh-config enable_chglogs
```

Alternatively, if you want to do it by yourself, perform the following actions on Lustre MDS

- Enable all changelog events:
  ```
  lctl set_param  mdd.*.changelog_mask all
  ```

- Changelogs consumers must be registered to Lustre to manage log records transactions properly. To do this, get a changelog reader id with the ‘lctl’ command:
  ```
  >lctl
  lctl > device lustre-MDT0000
  lctl > changelog_register
  lustre-MDT0000: Registered changelog userid 'cl1'
  ```

  Remember this id; it will be needed for writing PolicyEngine configuration file.

Note: in Lustre 2.0 release candidate, reading ChangeLogs in daemon mode causes a lot of “defunc” processes (bugzilla #23120). As a result, until this bug is fixed, ChangeLogs can only be processed in a one-shot command.

3 Writing configuration file

3.1 Syntax

General structure

The configuration file consists of several blocks. They can contain key/value peers (separated by semi-colons), sub-blocks, Boolean expressions, or set definitions (see xxxx).
In some cases, blocks have an identifier.

```
BLOCK_1 bloc_id
{
      Key = value;
      Key = value(opt1, opt2);
      Key = value;
      SUBBLOCK1 {
            Key=value;
      }
}
BLOCK_2 {
      (Key > value)
      and
      ( key == value or key != value )
}
CLASS_DEF {
      Set1 union Set2
}
```

Type of values

A value can be:

- A **string** delimited by single or double quotes (‘ or “).
- A **Boolean** constant. Both of the following values are accepted and the case is not significant: TRUE, FALSE, YES, NO, 0, 1, ENABLED, DISABLED.
- A **numerical value** (decimal representation).
- A **duration**, i.e. a numerical value followed by one of those suffixes: ‘w’ for weeks, ‘d’ for days, ‘h’ for hours, ‘min’ for minutes, ‘s’ for seconds. E.g.: 1s ; 1min ; 3h ; …
  NB: if you do not specify a suffix, the duration is interpreted as seconds.
  E.g.: 60 will be interpreted at 60s, i.e.1 min.
- A **size**, i.e. a numerical value followed by one of those suffixes: PB for petabytes, TB for terabytes, GB for gigabytes, MB for megabytes, KB for kilobytes. No suffix is needed for bytes.
- A **percentage**: float value terminated by ‘%’. E.g.: 87.5%

Boolean expressions

Some blocks of configuration file are expected to be Boolean expressions on file attributes:

- AND, OR and NOT can be used in Boolean expressions.
- Brackets can be used for including sub-expressions.

- Conditions on attributes are specified with the following format:
  <attribute> <comparator> <value>.
- Allowed comparators are '==', '<>' or '!=', '>', '>=', '<', '<='.

The following properties can be used in Boolean expressions:

- **tree:** entry is in the given filesystem tree. Shell-like wildcards are allowed.
  Example:
  tree == "/tmp/subdir/*/dir1" matches entry "/tmp/subdir/foo/dir1/dir2/foo" because "/tmp/subdir/foo/dir1/dir2/foo" is in the "/tmp/subdir/foo/dir1" tree, that matches "/tmp/subdir/*/dir1" expression.

  A tree can be specified using an absolute path (recommanded) or a relative path (root path is the "fs_path" parameter in configuration).

  The special wildcard "**" matches any count of directory levels:
  E.g: tree == "**/.trash" matches any part of a filesystem under a ".trash" entry.

- **path:** entry exactly matches the path. Shell-like wildcards are allowed.
  E.g: path == "/tmp/*/foo*" matches entry "/tmp/subdir/foo123".

  A path can be absolute (recommanded) or relative (root is given by the "fs_path" parameter in configuration).

  The special wildcard "**" matches any count of directory levels:
  E.g: path == "**/.trash/**/file" matches any entry called "file" located somewhere under a ".trash" directory (at any depth).

- **name:** entry name matches the given regexp.
  E.g: name == "*.log" matches entry "/tmp/dir/foo/abc.log".

- **type:** entry has the given type (**directory**, **file**, **symlink**, **chr**, **blk**, **fifo** or **sock**).
  E.g: type == "symlink".

- **owner:** entry has the given owner (owner name expected).
  E.g: owner == "root".

- **group:** entry owns to the given group (group name expected).

- **size:** entry has the specified size. Value can be suffixed with KB, MB, GB…
  E.g: size >= 100MB matches file whose size equals 100x1024x1024 bytes or more.

- **last_access:** condition based on the last access time to a file (for reading or writing). This is the difference between current time and max(atime, mtime, ctime). Value can be suffixed by 'sec', 'min', 'hour', 'day', 'week'…
  E.g: last_access < 1h matches files that have been read or written within the last hour.

- **last_mod:** condition based on the last modification time to a file. This is the difference between current time and max(mtime, ctime).
  E.g: last_mod > 1d matches files that have not been modified for more than a day.

- **ost_pool:** condition about the OST pool name where the file was created. Wildcarded expressions are allowed.
  E.g. ost_pool == “pool*”.
- **xattr.**xxx: test the value of a user-defined extended attribute of the file.
  E.g: xattr.user.tag_no_purge == “1”
  - xattr values are interpreted as text string;
  - regular expressions can be used to match xattr values;
    E.g: xattr.user.foo == "abc.[1-5].*" matches file having xattr user.foo = "abc.2.xyz"
  - if an extended attribute is not set for a file, it matches empty string.
    Eg. xattr.user.foo == "" ⇔ xattr 'user.foo' is not defined

- **dircount** (for directories only): the directory has the specified number of entries (except ‘.’ and ‘..’).
  E.g: dircount > 10000 matches directories with more that 10 thousand child entries.

Example of Boolean expression:

```
ignore {
      ( name == “*.log”  and size < 15GB )
      or ( owner == “root” and last_access < 2d )
      or not tree == “/fs/dir”
}
```

Set definitions

In the case of FileClass definitions, you can define FileClasses as the union or intersection of other FileClasses previously defined. This can be done using “union” and “inter” keyworks. Such expressions can be encapsulated between parenthesis.

Example:

```
FileClass my_set_union {
      definition { ( Class1 union Class2 ) inter Class3 }
}
```

Comments

The ‘#’ and ‘//’ signs indicate the beginning of a comment (except if there are in a quoted string). The comment ends at the end of the line.
E.g.:

```
# this is only a comment line
x = 32 ; # a comment can also be placed after a significant line
```

Includes

A configuration file can be included from another file using the ‘`%include`’ directive. Both relative and absolute paths can be used.
E.g.:

```
      %include “subdir/common.conf”
```

Configuration blocks

The main blocks in a configuration file are:

- **General** (mandatory): main parameters.
- **Log**: log and alert parameters (log files, log level…).
- **Filesets**: definition of file classes
- **Purge_Policies**: defines purge policies.
- **Purge_Trigger**: specifies conditions for starting purges.
- **Purge_Parameters**: general options for purge.
- **Rmdir_Policy**: defines empty directory removal policy.
- **Rmdir_Parameters:** options about empty directory removal.
- **ListManager** (mandatory): database access configuration.
- **FS_Scan**: options about scanning the filesystem.
- **[new 2.2] Db_update_policy:** parameters about file class periodic matching, and entry information update interval.
- **ChangeLog**: parameters related to Lustre 2.x changelogs.
- **EntryProcessor**: configuration of entry processing pipeline (for FS scan).

Those blocks are described in the following sections.

3.2 Configuration template and default parameters

Template file

To easily create a configuration file, you can generate a documented template using the `--template` option of robinhood, and edit this file to set the values for your system:

```
robinhood --template=<template file>
```

Default configuration values

To know the default values of configuration parameters use the `--defaults` option:

```
robinhood --defaults
```

3.3 General parameters

General parameters are set in a configuration block whose name is '**General**'.

The following parameters can be specified in this block:

- **fs_path** (string, mandatory): the path of the file system to be managed. This must be an absolute path. This parameter can be overridden by "`--fs-path`" parameter on command line.
  E.g.: `fs_path = "/tmp_fs";`

- **fs_type** (string, mandatory): the type of the filesystem to be managed (as displayed by mount). This is mainly used for checking if the filesystem is mounted. This parameter can be overridden by "`--fs-type`" parameter on command line.
  E.g.: `fs_type = "lustre";`
- **lock_file** (string): robinhood suspends its activity when this file exists.
  E.g.: `lock_file = "/var/lock/robinhood.lock";`
- **stay_in_fs** (Boolean): if this parameter is TRUE, robinhood checks that the entries it handles are in the same device as *fs_path*, which prevents from traversing mount points.
  E.g.: `stay_in_fs = TRUE;`
- **check_mounted** (Boolean): if this parameter is TRUE, robinhood checks that the filesystem of *fs_path* is mounted.
  E.g.: `check_mounted = TRUE;`

3.4 Log and alerts parameters

Logging parameters are set in a configuration block whose name is '**Log**'.

The following parameters can be specified in this block:

- **debug_level** (string): verbosity level of logs. This parameter can be overridden by "`--log-level`" parameter on command line.
  Allowed values are :
  - FULL: highest level of verbosity. Trace everything.
  - DEBUG: trace information for debugging.
  - VERB: high level of traces (but usable in production).
  - EVENT: standard production log level.
  - MAJOR: only trace major events.
  - CRIT: only trace critical events.

  E.g.: `debug_level = VERB;`
- **log_file** (string): file where logs are written. This parameter can be overridden by "`--log-file`" parameter on command line.
  E.g.: `log_file = "/var/logs/robinhood/robinhood.log";`
- **report_file** (string): file where pruge and rmdir operations are logged.
  E.g.: `report_file = "/var/logs/robinhood/purge_report.log";`

Notes:
- ⇨ Make sure the log directory exists.
- ⇨ You can use special values 'stderr' or 'stdout' for log files, so you can directly read log messages in your terminal when testing your configuration.
- ⇨ robinhood is compliant with log rotation (if its log file is renamed, it will automatically open a new file).

Two methods can be used for raising alerts: sending a mail, writing to a file, or both.
This is set by the following parameters:

- **alert_file** (string): if this parameter is set, alerts are written to the specified file.
  E.g.: `alert_file = "/var/logs/robinhood/alerts.log";`

- **alert_mail** (string): if this parameter is set, mail alerts are sent to the specified recipient.
  E.g.: `alert_mail = “admin@localdomain”;`

[New 2.2] Alert parameters:

- **alert_show_attrs** (Boolean): If true, details entry attributes in alerts.
- **batch_alert_max** (integer): this controls alert batching (sending 1 alert summary instead of 1 per entry):
  - If the value is 0, there is no limit in batching alerts. 1 summary is send after each scan.
  - If the value is 1, alerts are not batched.
  - If the value is N > 1, a summary is sent every N alerts.

3.5 File classes

You may need to apply different purge policies depending on file properties. To do this, you can define file classes.
A file class is defined by a ‘FileClass’ block. All file class definitions must be grouped in the ‘Filesets’ block of the configuration file.
Each file class has an identifier (that you can use for addressing it in policies) and a definition (a condition for entries to be in this file class).
[new 2.2] FileClasses can be defined as the union or the intersection of other FileClasses, using ‘inter’ and ‘union’ keywords in fileclass definition.

File classes definition overview:

```
Filesets
{
        FileClass  my_class_1
        {
                Definition
                {
                        tree == “/fs/dir_A”
                        and
                        owner == root
                }
        }

        FileClass my_class_2
        {
                …
        }

        FileClass my_inter_class
        {
                Definition { my_class_1 inter my_class_3 }
        }

        …
}
```

Important note: if you modify fileclass definitions or target fileclasses of policies, you need to reset fileclass information in Robinhood database.
To do so, run the following command:
`rbh-config reset_classes`

3.6 Purge policies

In general, files are purged in the order of their last access time (LRU list). You can however specify conditions to allow/avoid entries to be purged, depending on their file class, and file properties.

To define purge policies, you can specify:

- Sets of entries that must never be purged (ignored).
- Purge policies to be applied to file classes.
- A default purge policy for entries that don't match any file class.

In configuration file, all those parameters are grouped in a '**Purge_Policies**' block that consists in:

- '**Ignore**' sub-blocks: Boolean expressions to "white-list" filesystem entries depending on their properties.
  E.g.: `Ignore { size == 0 or type == "symlink" }`

- **'Ignore_fileclass**': "white-list" all entries of a fileclass (see section 3.5 about defining 3.5File classes).
  E.g.: `Ignore_FileClass = my_class_1;`

- '**Policy**' sub-blocks: specify conditions for purging entries of file classes.
  A policy has a custom name, one or several target file classes, and a condition for purging files.
  E.g:

```
Policy purge_classes_2and3
{
        target_fileclass = class_2;
        target_fileclass = class_3;

        condition
        {
                Last_access > 1h
        }
}
```

- A default policy that applies to files that don't match any previous file class or 'ignore' directive. It is a special 'Policy' block whose name is 'default' and with no target_fileclass.
  E.g:

```
Policy default
{
        condition
        {
                last_access > 30min
        }
}
```

As a summary, the 'purge_policies' block will look like this:

```
purge_policies
{
        # don't purge symlinks and entries owned by root
        Ignore  { owner == "root" or type == symlink }

        # don't purge files of classes 'class_xxx' and 'class_yyy'
        Ignore_FileClass = class_xxx ;
        Ignore_FileClass = class_yyy ;

        # purge policy for files of 'my_class1' and 'my_class2'
```

```
        policy my_purge_policy1
        {
                target_fileclass = my_class1;
                target_fileclass = my_class2;
                condition { last_access > 1h and last_mod > 2h }
        }
        …
        # purge policy for other files
        policy default
        {
                condition { last_access > 10min }
        }

}
```

3.7 Purge triggers

Triggers describe conditions for starting/stopping purges. They are defined by 'purge_trigger' blocks. Each trigger consists of:

- The type of condition (on global filesystem usage, on OST usage, on volume used by a user or a group…);
- A purge start condition ;
- A purge target condition ;
- An interval for checking start condition.

Several triggers can be specified.

Type of condition

The type of condition is specified by "**trigger_on**" parameter.
Possible values are:

- **global_usage:** purge start/stop condition is based on the spaced used in the whole filesystem (based on *df* return). All entries in filesystem are considered for such a purge.
- **OST_usage:** purge start/stop condition is based on the space used on each OST (based on *lfs df*). Only files stored in an OST are considered for such a purge.
- **user_usage[(user1, user2…)]:** purge start/stop condition is based on the space used by a user (kind of quota). Only files that own to a user are considered for such a purge. If it is used with no arguments, all users will be affected by this policy. A list of users can also be specified for restricting the policy to a given set of users (coma-separated list of users between brackets). [Fully implemented since Robinhood 2.2].
- **group_usage[(grp1, grp2…)]:** purge start/stop condition is based on the space used by a group (kind of quota). Only files that own to a group are considered for purge. If it is used with no arguments, all groups will be affected by this policy. A list of groups can also be specified for restricting the policy to a given set of groups (coma-separated list of groups between brackets). [Fully implemented since Robinhood 2.2].

Start condition

This is mandatory for all types of conditions.
A purge start condition can be specified by two ways: percentage or volume.

- **high_watermark_pct** (percentage): specifies a percentage of space used over which a purge is launched.
- **high_watermark_vol (**size): specifies a volume of space used over which a purge is launched. The value for this parameter can be suffixed by KB, MB, GB, TB…
- **[New 2.2] high_watermark_cnt** (count): condition based on the number of inodes in the filesystem. It only applies to **global_usage** triggers. The value can be suffixed by K, M, …

Stop condition

This is mandatory for all types of conditions.
A purge stop condition can also be specified by two ways: percentage or volume.

- **low_watermark_pct:** specifies a percentage of space used under which a purge stops.
- **low_watermark_vol:** specifies a volume of space used under which a purge stops. The value for this parameter can be suffixed by KB, MB, TB… (the value is interpreted as bytes if no suffix is specified).
- **[New 2.2] low_watermark_cnt** (count): condition based on the number of inodes in the filesystem. It only applies to **global_usage** triggers. The value can be suffixed by K, M, …

Runtime interval

The time interval for checking a condition is set by parameter "**check_interval**". The value for this parameter can be suffixed by 'sec', 'min', 'hour', 'day', 'week', 'year'… (the value is interpreted as seconds if no suffix is specified).

Examples

Check 'df' every 5 minutes, start a purge if space used > 85% of filesystem and stop purging when space used reaches 84.5%:

```
Purge_Trigger
{
       trigger_on = global_usage ;
       high_watermark_pct = 85% ;
       low_water_mark_pct = 84.5% ;
       check_interval = 5min ;
}
```

Check OST usage every 5 minutes, start a purge of files on an OST if it space used is over 90% and stop purging when space used on the OST falls to 85%:

```
Purge_Trigger
{
       trigger_on = OST_usage ;
       high_watermark_pct = 90% ;
       low_water_mark_pct = 85% ;
       check_interval = 5min ;
}
```

Daily check the space used by each user. If one of them uses more than 1TB, its files are purged until it uses less than 800GB:

```
Purge_Trigger
{
       trigger_on = user_usage ;
```

```
        high_watermark_vol = 1TB ;
        low_water_mark_vol = 800GB ;
        check_interval = 1day ;
}
```

3.8 Purge parameters

Purge parameters are specified in a 'purge_parameters' block.
The following options can be set:

- **nb_threads_purge** (integer): this determines the number of purge operations that can be performed in parallel.
  E.g.: `nb_threads_purge = 8 ;`
- **post_purge_df_latency** (duration): immediately after purging data, *df* and *ost df* may return a wrong value, especially if freeing disk space is asynchronous. So, it is necessary to wait for a while before issuing a new *df* or *ost df* command after a purge. This duration is set by this parameter.
  E.g.: `post_purge_df_latency = 1min ;`
- **purge_queue_size** (integer): this advanced parameter is for leveraging purge thread load.
- **db_result_size_max** (integer): this impacts memory usage of MySQL server and Robinhood daemon. The higher it is, the more memory they will use, but less DB requests will be needed.

3.9 Specifying 'rmdir' policy

Directory removal is driven by the 'rmdir_policy' section in the configuration file:

- **age_rm_empty_dirs** (duration): indicates the time after which an empty directory is removed. If set to 0, empty directory removal is disabled.
- You can specify one or several '**ignore**' condition for directories you never want to be removed.
- **'recursive_rmdir'** sub-blocks indicates that the matching directories must be removed recursively. /!\In this case, the whole directories content is removed without checking policies on their content (whitelist rules…).

Example:

```
rmdir_policy
{
      # remove empty directories after 15 days
      age_rm_empty_dirs = 15d;


      # recursively remove ".trash" directories after 15 days
      recursive_rmdir
      {
            name == ".trash" and last_mod > 15d
      }

      # whitelist directories matching the following condition
```

```
        ignore
        {
                depth < 2
                or
                owner == 'foo'
                or
                tree == /fs/subdir/A
        }
}
```

3.10 'Rmdir' parameters

Directory removal parameters are specified in the 'rmdir_parameters' block.

The following options can be set:

- **runtime_interval** (duration): interval for performing empty directory removal.
- **nb_threads_rmdir** (integer): this determines the number of 'rmdir' operations that can be performed in parallel.
  E.g.: `nb_threads_rmdir = 4;`
- **rmdir_op_timeout** (duration): this specifies the timeout for 'rmdir' operations. If a thread is stuck in a filesystem operation during this time, it is cancelled.
  E.g.: `rmdir_op_timeout = 15min;`
- **rmdir_queue_size** (integer): this advanced parameter is for leveraging rmdir thread load.

3.11 Periodic fileclass matching

[new 2.2] In previous versions, robinhood matched fileclasses every time it applied policies. This resulted in performing a lot of filesystem calls at this time. Now, it can match fileclasses at regular interval, and only the policy condition for the fileclass is checked when performing policy application. Additionally, the fileclass of each file is now stored in the database, so you can see it using the `rbh-report` command.

The fileclass matching interval is set using the '`fileclass_update`' parameter in the new '`db_update_policy`' section:

```
db_update_policy
{
        fileclass_update = periodic( 1h );
}
```

Possible values are:

- `never`: match file class once, and never again
- `always`: always re-match file class when applying policies (this is the old behavior)
- `periodic(<period>)`: periodically re-match fileclasses

3.12 Periodic information update when handling Lustre changelogs

[new 2.2] This is similar to fileclass periodic matching, but it is for updating file metadata and path in the database when processing Lustre changelogs.
Indeed, if robinhood receives a lot of events for a given entry, it can be very loud to update entry information when processing each event.
You can specifies the way entry path and entry metadata is updated in the database using 'md_update' and 'path_update' parameters in the new '`db_update_policy`' section:

```
db_update_policy
{
      md_update = on_event_periodic(1sec,1min);
      path_update = on_event;
}
```

Possible values for those parameters are:

- `never`: retrieve it once, then never update the information
- `always`: always update the information when receiving an event for the entry
- `periodic(<period>)`: update the information only if it was not updated for a while
- `on_event`: update the information every time the event is related to it (eg. update entry path when the event is a 'rename' on the entry).
- `on_event_periodic(<interval_min>,<interval_max>)`: this is the smarter one

3.13 Database parameters

The 'ListManager' block is the configuration for accessing the database.

ListManager parameters:

- **commit_behavior**: this is the method for committing information to database.
  The following values are allowed:
  - **autocommit**: weak transactions. In this mode, each operation on database is committed immediately, and multiple operations on the same entry are not grouped in transactions (more efficient, but database inconsistencies may appear).
  - **transaction**: group operations in transactions (best consistency, lower performance).
  - **periodic(<nbr_transactions>)**: operations are packed in large transactions before they are committed. 'Commit' is done evry *n* transactions. This method is more efficient for in-file databases like SQLite. This causes no database inconsistency, but more operations are lost in case of a crash.

  E.g: `commit_behavior = periodic(1000);`

- **connect_retry_interval_min**, **connect_retry_interval_max** (durations):
  'connect_retry_interval_min' is the time (in seconds) to wait before re-establishing a lost connection to database. If reconnection fails, this time is doubled at each retry, until 'connect_retry_interval_max'.
  E.g: `connect_retry_interval_min = 1;`

```
connect_retry_interval_max = 30;
```

MySQL specific configuration is set in a '**MySQL**' sub-block, with the following parameters:

- **server**: machine where MySQL server is running. Both server name and IP address can be specified.
  E.g.: `server = "mydbhost.localnetwork.net";`
- **db** (string, mandatory): name of the database.
  E.g.: `db = "robinhood_db";`
- **user** (string): name of the database user.
  E.g.: `user = "robinhood";`
- **password** or **password_file** (string, mandatory): there are two methods for specifying the password for connecting to the database, depending of the security level you want. You can directly write it in the configuration file, by setting the '**password**' parameter. You can also write the password in a distinct file (with more restrictive rights) and give the path to this file by setting '**password_file**' parameter. This makes it possible to have different access rights for config file and password file.
  E.g.: `password_file = "/etc/robinhood/.dbpass";`

3.14 Filesystem scan parameters

Parameters for scanning the filesystem are set in the '**FS_Scan**' block.
It can contain the following parameters:

- **min_scan_interval**, **max_scan_interval** (durations): robinhood adapts its frequency for scanning the filesystem to the current filesystem usage. Indeed, it is not necessary to scan filesystem frequently when it is empty (because no purge will be needed for a long time). However, the more the filesystem is used, the more it needs a fresh list for purging files. Thus, specify delay between filesystem scans using:
  - **min_audit_period**: the frequency for scanning when filesystem is full;
  - **max_audit_period**: the frequency for scanning when filesystem is empty

  The interval between scans is computed according to this formula:
  `min + (100 x current usage)*(max-min)`

- **nb_threads_scan** (integer): number of threads used for scanning the filesystem in parallel.
- **scan_retry_delay** (duration): if a scan fails, this is the delay before starting another.
- **scan_op_timeout** (duration): this specifies the timeout for readdir/getattr operations. If a thread is stuck in a filesystem operation during this time, it is cancelled.
- **spooler_check_interval** (duration): interval for testing FS scans, deadlines and hangs.
- **nb_prealloc_tasks** (integer): number of pre-allocated task structures (advanced parameter).

3.15 Lustre 2.x changelog parameters

With Lustre 2.x, FS scan are no more required to update robinhood's database.
Reading Lustre's changelog is much more efficient, because this does not load the filesystem as much as a full namespace scan.
Accessing the chngelog is driven by the '**ChangeLog**' block of configuration.

It contains one 'MDT' block for each MDT, with the following information:

- **mdt_name** (string): name of the MDT to read ChangeLog from (basically "MDT0000").
- **reader_id** (string): log reader identifier, returned by '**lctl changelog_register**' (see section 2.6: "Enabling Lustre Changelogs").

On Lustre 2.0-alpha5, changelog readers need to perform active polling to get new events from MDT. So, on this lustre version, you need to activate polling:
force_polling = ON;

You can also specify a polling interval:
polling_interval = 1s;

Finally, the "ChangeLog" block looks like this:

```
ChangeLog
{
    MDT
    {
        mdt_name  = "MDT0000";
        reader_id = "cl1";
    }
}
```

Note: in Lustre 2.0 release candidate, reading ChangeLogs in daemon mode causes a lot of "defunc" processes (bugzilla #23120). As a result, until this bug is fixed, ChangeLogs can only be processed in a one-shot command.

3.16 Entry processor pipeline options

When scanning the filesystem, entries are handled by a pool of threads, with a pipeline model. Options for this pipeline are set in a '**EntryProcessor**' block, with the following parameters:

- **nb_threads** (integer): number of threads for performing pipeline tasks.
- **max_pending_operations** (integer): this parameter limits the number of pending operations in the pipeline, so this prevents from using too much memory. When the number of queued entries reaches this value, the scanning process is slowed-down to keep the pending operation count bellow this value.

Pipeline processing is divided in several stages. It is possible to limit the number of threads working simultaneously on a given stage by setting a '<stage_name>_threads_max' parameter. Thus, the following parameters can be set:

- **STAGE_GET_FID_threads_max** (integer): when scanning a filesystem, this indicates the maximum number of threads that can perform a llapi_path2fid operation simultaneously.
- **STAGE_GET_INFO_DB_threads_max** (integer): this limits the number of threads that simultaneously check if an entry already exists in database.
- **STAGE_GET_INFO_FS_threads_max** (integer): this limits the number of threads that simultaneously retrieve information from filesystem (getstripe…).
- **STAGE_INFER_ATTRS_threads_max** (integer): this limits the number of threads that simultaneously check white-list and penalty rules on entries.
- **STAGE_REPORTING_threads_max** (integer): this limits the number of threads that simultaneously check and raise alerts about filesystem entries.
- **STAGE_DB_APPLY_threads_max** (integer): this limits the number of threads that simultaneously insert/upate entries in the database.

E.g.: for limiting the number of simultaneous 'getstripe' operation:

```
STAGE_GET_INFO_FS_threads_max = 1;
```

Alerts

One of the tasks of the Entry Processor is to check alert rules and raise alerts. For defining an alert, simply write an '**Alert**' sub-block with a Boolean expression that describes the condition for raising an alert (see section 3.1 for more details about writing Boolean expressions on file attributes).
[New 2.2] Alerts can be named, to easily identify/distinguish them in alert summuries.

E.g.: raise an alert if a directory contains more that 10 thousand entries:

```
Alert   Large_flat_directory
{
    type == directory
    and
    dircount > 10000
    and
    last_mod < 2h
}
```

Another example: raise an alert if a file is larger that 100GB (except for user 'foo'):

```
Alert   Big_File
{
    type == file
    and
    size > 100GB
    and
    owner != 'foo'
    and
    last_mod < 2h
}
```

Tip: alert rules are matched at every scan. If you don't want to be alerted about a given file at each scan, it is advised to specify a condition on `last_mod`, so you will only be alerted for recently modified entries.

4 Reporting tool

4.1 Overview

The content of Robinhood's database can be very useful for building detailed reports about filesystem. For example, you can know how many entries of each type (directory, file, symlink…) exist in the filesystem, the min/max/average size of files, the min/max/average count of entries in directories, the space used by a given user, etc…
All those statistics can easily be retrieved using Robinhood reporting tool: **rbh-report**.

4.2 Command line

```
Usage: rbh-report [options]

Stats switches:
    -a, --activity
        Display stats about daemon's activity.
    -i, --fsinfo
        Display filesystem content statistics.
    -u user, --userinfo[=user]
        Display user statistics. Use optional parameter user
        for retrieving stats about a single user.
    -g group, --groupinfo[=group]
        Display group statistics. Use optional parameter group
        for retrieving stats about a single group.
    -d count, --topdirs[=count]
        Display largest directories. Optional argument indicates the number
        of directories to be returned (default: 20).
    -s count, --topsize[=count]
        Display largest files. Optional argument indicates the number
        of files to be returned (default: 20).
    -p count, --toppurge[=count]
        Display oldest entries eligible for purge. Optional argument
        indicates the number of entries to be returned (default: 20).
    -r count, --toprmdir[=count]
        Display oldest empty directories eligible for rmdir.
        Optional argument indicates the number of dirs to be returned
        (default: 20).
    -U count, --topusers[=count]
        Display largest disk space consumers. Optional argument indicate
        the number of users to be returned (default: 20).
    -D, --dump-all
        List all filesystem entries.
    --dump-user user
        List all entries for the given user.
    --dump-group group
        List all entries for the given group.
    --dump-ost ost_index
        List all entries on the given OST.

Filter options:
The following filters can be speficied for reports:
    -P path, --filter-path=path
        Display the report only for objects in the given path.
```

```
Config file options:
    -f file, --config-file=file
        Specifies path to configuration file.

Output format options:
    -c , --csv
        Output stats in a csv-like format for parsing

Miscellaneous options:
    -l level, --log-level=level
        Force the log verbosity level (overrides configuration value).
        Allowed values: CRIT, MAJOR, EVENT, VERB, DEBUG, FULL.
    -h, --help
        Display a short help about command line options.
    -V, --version
        Display version info.
```

4.3 Reports

This command can provide the following reports:

Filesystem content report (`--fs-info` option)

This displays the number of entries of each type. For directories, it gives the min, max and average number of entries in directories (in the size column). For other types of entries, it gives their size, as returned by POSIX ‘stat()’ call.

Example of output:

```
Type:              directory
Count:                 31676
Dircount min:              0
Dircount max:           4525
Dircount avg:             26

Type:                   file
Count:                773285
Size min:                  0
Size max:         74700554240
Size avg:             4993864

Type:                 symlink
Count:                   4306
Size min:                   2
Size max:                 153
Size avg:                  42
```

User info report (`--user-info` option)

This displays about the same statistics as ‘fs-info’ for each user (or only the user given in parameter).

Example of output:

```
User:                   root

    Type:              directory
    Count:                  8426
    Space used:         34.29 MB    (70232 blks)
    Dircount min:              0
```

```
    Dircount max:            4525
    Dircount avg:              13

    Type:                    file
    Count:                 101398
    Space used:           3.44 TB    (7382951640 blks)
    Size min:                   0    (0 bytes)
    Size max:             8.00 GB    (8589934592 bytes)
    Size avg:            35.56 MB    (37286674 bytes)
```

Group info report (`--group-info` option)

Same report as ‘user-info’, for groups.

Top directories (`--top-dirs` option)

This option displays directories with the highest number of child entries.
Useful information is given for each of them: path, owner, number of entries, last modification time.

Example of output:

```
Rank:               1
Path:               /ptmp/diskless/root_fortoy153/usr/share/man/man3
Dircount:           4525
Last modification:  2009/01/13 15:13:34
Owner/Group:        root/root

Rank:               2
Path:               /ptmp/toto/compile/gcc/gcc-4.3.2/gcc/testsuite/gcc.target
Dircount:           1872
Last modification:  2009/01/13 08:26:31
Owner/Group:        toto/group1

...
```

Top file size (`--top-size` option)

This options displays a list of largest files, with useful information: path, size, last access time, last modification time, owner, stripe information.

Example of output:

```
Rank:               1
Path:               /ptmp/group1/toto/opt/lib/gcj-4.3.2-9/classmap.db
Size:               69.57 GB   (74700554240 bytes)
Last access:        2009/01/13 07:24:56
Last modification:  2009/01/13 07:22:04
Owner/Group:        toto/group1
*Purge class:       toto_file_class
Stripe count:       2
Stripe size:        4.00 MB   (4194304 bytes)
Storage units:      OST #16, OST #15

Rank:              2
Path:              /ptmp/vm/Fortoy578
Size:              8.00 GB   (8589934592 bytes)
Last access:       2009/01/14 17:28:20
Last modification: 2009/01/14 17:28:20
Owner/Group:       root/root
*Purge class:      vm_file_class
Stripe count:      2
Stripe size:       4.00 MB   (4194304 bytes)
```

```
Storage units:     OST #2, OST #1
...

* [new version 2.2]
```

Top purge candidates (`--top-purge` option)

This displays files that are likely to be purged first, if disk space is needed. Also, this command gives an overview of oldest entries of filesystem. Note that this is only an estimation, and those entries may not be purged if they have been moved or accessed since they were scanned. This returns entry path and type, last access and modification time, the penalty applied to the entry (if it matches a penalty rule in purge policy), and storage information (size, blocks, stripe info…).

```
Rank:               1
Path:               /ptmp/vm/benchs/bonnie++-1.03a/zcav.8
Type:               file
Last access:        2009/01/13 08:26:31
Last modification:  2009/01/13 08:26:31
*Purge class:       file_class_1
Size:               2.20 KB   (2253 bytes)
Space used:         4.00 KB   (8 blocks)
Stripe count:       2
Stripe size:        4.00 MB   (4194304 bytes)
Pool:               array1
Storage units:      OST #2, OST #3

Rank:               2
Path:               /ptmp/vm/benchs/bonnie++-1.03a/bonnie.h.in
Type:               file
Last access:        2009/01/13 08:26:31
Last modification:  2009/01/13 08:26:31
*Purge class:       file_class_1
Size:               1.36 KB   (1391 bytes)
Space used:         4.00 KB   (8 blocks)
Stripe count:       2
Stripe size:        4.00 MB   (4194304 bytes)
Storage units:      OST #7, OST #8
...

* [new version 2.2]
```

Top ‘rmdir’ candidates (`--top-rmdir` option)

This displays empty directories that are likely to be removed first. Note that this is only an estimation, and those directories may not be removed if they have been modifed since they were scanned. This returns directory path, owner, last modification time, and the estimated time before they are removed (or ‘expired’ if this delay is over).

Example of output:

```
Rank:               1
Path:               /ptmp/grp1/foo/BENCH/CSC_SVN_REPO/benchmark/utils/gprof/.svn/prop-base
Rmdir deadline:     expired
Last modification:  2009/01/15 15:47:28
Owner/Group:        foo/grp1

Rank:               2
Path:               /ptmp/grp2/foo/BENCH/CSC_SVN_REPO/benchmark/applications/run/tmp
Rmdir deadline:     3.7d
Last modification:  2009/01/20 09:55:15
Owner/Group:        foo/grp2
...
```

Top disk space consumers (`--top-users` option)

Display users who consume the larger disk space.

```
Rank:                     1
User:                   tom
Space used:          3.45 TB    (7409880032 blks)
Nb entries:           223746
Size min:                0    (0 bytes)
Size max:             8.00 GB    (8589934592 bytes)
Size avg:            16.17 MB    (16958395 bytes)

Rank:                     2
User:                 charly
Space used:          71.80 GB    (150570152 blks)
Nb entries:             73721
Size min:                0    (0 bytes)
Size max:            69.57 GB    (74700554240 bytes)
Size avg:          1018.71 KB    (1043154 bytes)
...
```

Daemon's activity (`--activity` option)

This reports the last actions Robinhood did, and their status: last filesystem scan, last purge…

```
Last Filesystem scan:     2009/02/09 13:19:07

Storage unit usage max:   55.33%

Last purge:               2009/02/09 13:45:44
    Target:               OST #16
    Status:               OK
```

Dump commands (`--dump-all, --dump-user, --dump-group, --dump-ost`)

These options can be used for listing entries with a given critera.

Example: listing all entries on OST #14:

```
# rbh-report --dump-ost 14

   type,         size,      owner,      group, path
   file,     16.26 KB,       root,       root, /mnt/lustre/config.h.in
   file,           48,       root,       root, /mnt/lustre/ChangeLog
   file,    186.55 KB,       root,       root, /mnt/lustre/aclocal.m4
   file,     42.40 KB,       root,       root, /mnt/lustre/config.guess
   file,     35.23 KB,       root,       root, /mnt/lustre/libsysio/Makefile.in
   file,     29.77 KB,       root,       root, /mnt/lustre/libsysio/config.sub
   file,    705.89 KB,       root,       root, /mnt/lustre/configure
```

5 System and database tunings

5.1 Lustre tunings and workarounds

Several bugs or bad behaviours in Lustre can make your node crash or use a lot of memory when Robinhood is scanning or massively purging entries in the FileSystem. Here are some workarounds we had to apply on our system for making it stable:

- If your system “Oops” in `statahead` function, disable this feature:
  `echo 0 > /proc/fs/lustre/llite/*/statahead_max`

- CPU overload and client performance drop when free memory is low (bug #17282):
  in this case, lru_size must be set at CPU_count * 100:
  `lctl set_param ldlm.namespaces.*.lru_size=800`

5.2 Linux kernel tunings

Robinhood daemon retrieves attributes for large sets of entries in filesystem:

- When scanning, it needs to retrieve attributes of all objects in the filesystem;
- When purging, its checks the attributes of entries before purging them.

This results in loading many inodes and direntries in Linux VFS cache.

If the free memory of the machine is low, or if the machine swaps, this may be due to this cache. To check this, you can read the ‘`/proc/slabinfo`’ pseudo-file that reports how many objects are allocated by the system, and their size.

For reducing the size of this cache, you can make VFS garbage collection more aggressive by setting the `/proc/sys/vm/vfs_cache_pressure` parameter. By default, its value is 100. If you increase it, garbage collection will be more aggressive and VFS cache will use less memory.

E.g:

```
echo 1000 > /proc/sys/vm/vfs_cache_pressure
```

5.3 Optimize Robinhood for reporting

By default, robinhood database is optimized for insertion and update, to get the best performances when scanning large filesystems and processing high streams of events.
On the other side, this makes it longer to generate reports: it may take a couple of minutes generating a summary for filesystems with tens of millions of entries.
If you don’t care about the time it takes for scanning, and if its event processing pipeline does not appear to be busy, you may want to optimize Robinhood for generating reports faster. There is not a single universal optimization for this: you have to define specific indexes on the database, depending on the reports you need.
Don’t hesitate asking on robinhood-support mailing list to determine the good index for your need : robinhood-support@lists.sourceforge.net.

5.4 Database tunings

For managing very large filesystems, some tuning is needed for optimizing database performance and fit with available memory. Of course, using large buffers and memory caches will make DB requests faster, but if buffers are oversized, the DB engine and the client may use too much memory, slow-down filesystem performances or make the machine swap.

MySQL server tuning is to be done in `/etc/my.cnf`.

Tuning is often empirical, and depends on many parameters (CPU, memory, number of entries in filesystem…). So the best we can do here is to give parameters we set on several systems.

Test bed 1:
- 12M entries filesystem
- 8GB of total memory
- Linux CentOS 5.2
- Robinhood and MySQL server running on the same node

With the following parameters for MySQL:

```
key_buffer_size             =   128M
table_cache                 =   2000
max_allowed_packet          =   1M
myisam_sort_buffer_size     =   16M
sort_buffer_size            =   16M
read_buffer_size            =   16M
read_rnd_buffer_size        =   4M
thread_cache_size           =   128
query_cache_size            =   40M
query_cache_limit           =   1M
tmp_table_size              =   64M
```

And in the 'purge_parameters' block of Robinhood config:
```
db_result_size_max          =   10000
```

We get the following resource usage:

For purge action only:
- mysqld uses 560MB
- robinhood uses 300MB

For all actions (scan + purge + rmdir) performed simultaneously:
- mysqld uses 650MB
- robinhood uses 500MB

6 Known bugs

- **Process terminates with SEGFAULT when MySQL server restarts**

  - Cause:
    This is due to a bad resilience of MySQL client API to server crash when using prepared statements. This is known as bug #33384 in MySQL tracker (check current bug status here: http://bugs.mysql.com/bug.php?id=33384).

  - Workaround:
    Disable prepared statements in Robinhood: to do so, use '`--disable-prep-stmts`' argument to `./configure` before building the RPM.

- **If a given inode has several hardlinks, and if different policy cases apply to them (due to their different paths), the policy case that is effectively used is undetermined**

  - Cause:
    For now, robinhood only stores one path per entry (not all of its hardlinks). As a result, if an entry has several paths, one of them is arbitrary used for matching policies.