Robinhood PolicyEngine

Temporary Filesystem Manager

Admin Guide

Thomas LEIBOVICI
CEA/DAM

<thomas.leibovici@cea.fr>

V2.4.0

October 31st, 2012

Table of contents

1 Product description

1.1 Overview

"Robinhood Policy Engine" is Open-Source software developed at CEA/DAM for monitoring and purging large 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 and/or PST pools.

A specific mode of Robinhood 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:

- Scan 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;
  With Lustre v2, scanning is no more needed thanks to Lustre MDT Changelog mechanism. **Robinhood's database is updated in soft real-time and you can use 'rbh-find' and 'rbh-du' commands to query your file system faster than ever!**

- Monitor filesystem and Lustre OST usage, in order to maintain them below 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;

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

- Raise alerts, search for entries, generate 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.
  If you run it on a Lustre v2 file system, it continuously reads MDT changelogs to update its database in soft real-time.
- It regularly 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 exits:

- It scans the filesystem once and update its database;
  With Lustre v2, it reads MDT Changelogs currently stacked;
- 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 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 thresholds 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 threshold.

2 First steps with Robinhood

2.1 Compiling and installing

2.1.1 Install from RPM

Pre-generated RPMs can be downloaded on sourceforge, for the following configurations:

- x86_64 architecture , RedHat 5/6 Linux family
- MySQL database 5.x
- Posix filesystems, Lustre 1.8, 2.0, 2.1, 2.2, 2.3

Purpose specific RPM: `robinhood-tmpfs`
/!\ this RPM's name changed in v2.4 (robinhood-tmp_fs_mgr → robinhood-tmpfs)
It includes:

- 'robinhood' daemon
- Reporting commands: 'rbh-report', 'rbh-find' and 'rbh-du'
- configuration templates
- /etc/init.d/robinhood init script

[new 2.4] admin RPM (all purposes): `robinhood-adm`
Includes 'rbh-config' configuration helper

[new 2.3.2] `robinhood-webgui` RPM installs a web interface to visualize stats from Robinhood database. See section 7 for more details about this interface.

2.1.2 Build and install from the source tarball

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.

Retrieve Robinhood tarball from sourceforge: http://sourceforge.net/projects/robinhood

Unzip and untar the sources:

```
tar zxvf  robinhood-2.4.0.tar.gz
cd robinhood-2.4.0
```

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

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

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

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 RPMs:

```
make rpm
```

RPMs are generated in the 'rpms/RPMS/<arch>' directory. RPM is eventually tagged with the lustre version it was built for.

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. If there are common configuration blocks for those filesystems, you can use the '`%include`' directive in configuration files.

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

```
(* = new feature in robinhood 2.4)

Usage: robinhood [options]

Action switches:
    -S, --scan
        Scan filesystem namespace.
    -P, --purge
        Purge non-directory entries according to policy.
    -C, --check-thresholds
        Only check thresholds of purge triggers without purging.
    -R, --rmdir
        Remove directories according to policy.
    -r, --read-log  (Lustre 2+ only)
        Handle events from Lustre MDT ChangeLog.
*   --partial-scan=dir
        Scan a subset of the filesystem namespace.

    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.
    --purge-class=fileclass
        Apply purge policies to files in the given class.

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 from parent process).

Config file options:
    -f file, --config-file=file
*       Path to configuration file (or short name).
    -T file, --template=file
        Write a configuration file template to the specified file.
    -D, --defaults
        Display default configuration values.
    --test-syntax
        Check configuration file and exit.

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).
```

Notes about the '-f' option:

- If no '-f' option is specified, robinhood searches for a config file in directory '/etc/robinhood.d/tmpfs';
- **[new 2.4]** If you specify a short name (e.g. 'foo'), it will first search for files named 'foo.conf'or 'foo.cfg' in '/etc/robinhood.d/tmpfs'. If none is found, it will search it in the current directory;
- You can specify a full path to the config file (eg. '–f /etc/robinhood.d/tmpfs/foo.conf')

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;
- **[new 2.3.3]** SIGUSR1: dump process stats to its log 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'@'%';
    ```
  - The 'super' privilege is required for creating DB triggers (needed for accounting optimizations):
    ```
    GRANT SUPER ON *.* TO 'robinhood'@'%' IDENTIFIED BY 'password'
    ;
    ```
  - 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.

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 '**set definitions'** below).
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 is owned by 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.

- **[new 2.4] creation:** condition based on file creation time. With scanning mode, this value is an estimation based on the time when robinhood sees a file for the first time and ctime.
  E.g.: creation > 1h matches files created more than 1 hour ago.

- **[new 2.3] ost_index:** condition on OSTs where a file is stored. A file can be striped on several OSTs, thus:
  “ost_index == *N*” is true if <u>at least one part</u> of the file is stored in OST index #*N*.
  “ost_index != *N*” is true if <u>no</u> part of the file is stored in OST index #*N*.
  (Note: < and > are not allowed for this criteria).

- **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**", "**inter**" and "**not**" keyworks. Such expressions can be encapsulated between parenthesis.

Example:

```
FileClass my_set_union {
      definition { ( Class1 union Class2 ) inter ( not 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";`

- **[new 2.4] fs_key**: this indicates the filesystem property used as unique and persistent file system identifier. Possible values are: 'fsname', 'devid' or 'fsid' ('fsid' is NOT recommended as it may change at each *mount*).
  E.g.: `fs_key = fsname;`

- **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;`

- **lock_file** (string): robinhood suspends its activity when this file exists.
  E.g.: `lock_file = "/var/lock/robinhood.lock";`

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 about **log_file** and **report_file**:
- ⇨ Make sure the log directory exists.
- ⇨ robinhood is compliant with log rotation (if its log file is renamed, it will automatically open a new file).
- ⇨ The following special values can be used as log files:
  - o '**stdout**': log to standard output
  - o '**stderr**': log to standard error
  - o **[new 2.2.1]** '**syslog**': log using syslog
- ⇨ For syslog, you can select the syslog facility using the '**syslog_facility**' parameter.
  E.g:
  ```
  log_file = syslog ;
  syslog_facility = local1.info ;
  ```

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 }
        }

}
```

Note: the target fileclasses are matched in the order they appear in the purge_policies block, so make sure to specify the more restrictive classes first.
Thus, in the following example, the second policy can't never be matched, because A already matches all entries in A_inter_B:

```
Filesets {
        Fileclass A {  … }
        Fileclass B {  … }
        Fileclass A_inter_B { definition { A inter B } }
}

purge_policies
{
        policy purge_1
        {
                target_fileclass = A; # all entries of fileclass A
                …
        }
        policy purge_2
        {
                target_fileclass = A_inter_B; # never matched!!!
                …
        }
}
```

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.
- Notification options

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 are owned by a user over the limit 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 are owned by a group over the limit 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].
- **[new 2.2.3] periodic:** purge runs at scheduled interval, with no condition on filesystem usage.

Start condition

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

- **high_threshold_pct** (percentage): specifies a percentage of space used over which a purge is launched.
- **high_threshold_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_threshold_cnt** (count): condition based on the number of inodes in the filesystem. It can be used for **global_usage**, **user_usage** and **group_usage** triggers. The value can be suffixed by K, M, …

No threshold is expected for ‘periodic’ triggers.

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_threshold_pct:** specifies a percentage of space used under which a purge stops.
- **low_threshold_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_threshold_cnt** (count): condition based on the number of inodes in the filesystem. It can be used for **global_usage**, **user_usage** and **group_usage** triggers. The value can be suffixed by K, M, …

No threshold is expected for ‘periodic’ triggers.

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).

[new 2.2] Raise alert when high threshold is reached

Optionally, an alert can be raised each time the high threshold is reached. This can be done by setting the "**alert_high**" parameter (Boolean) in the trigger:

```
alert_high = TRUE;
```

[new 2.2.3] Don't raise alert when low threshold cannot be reached

By default, robinhood raises an alert if it can't purge enough data to reach the low threshold. You can disable these alerts by adding this in a trigger definition:

```
alert_low = FALSE ;
```

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_threshold_pct = 85% ;
        low_threshold_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_threshold_pct = 90% ;
        low_threshold_pct = 85% ;
        check_interval = 5min ;
}
```

Daily check the space used by each user of a given list. If one of them uses more than 1TB, its files are purged until it uses less than 800GB. Also send an alert in this case.

```
Purge_Trigger
{
        trigger_on = user_usage(foo, charlie, roger, project*) ;
        high_threshold_vol = 1TB ;
        low_threshold_vol  = 800GB ;
        check_interval = 1day ;
        alert_high = TRUE ;
}
```

Check that user inode usage is less than 100k entries (and send a notification in this case):

```
Purge_Trigger
{
        trigger_on = user_usage ;
        high_threshold_cnt = 100k ;
        low_threshold_cnt  = 100k ;
        check_interval = 1day ;
        alert_high = TRUE ;
}
```

Apply purge policies twice a day (whatever the filesystem usage):

```
Purge_Trigger
{
      trigger_on = periodic;
      check_interval = 12h;
}
```

Note: check triggers conditions without purging
If robinhood is started with the '`--check-thresholds`' option instead of '`--purge`', it will only check for trigger conditions and eventually send notifications, without purging data.

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;
  ```

Accounting parameters (in ListManager):

- **[new 2.3] user_acct, group_acct** (Booleans): these parameters enable or disable optimized reports for user and group statistics.
  By default these parameters are enabled. If you disable them (or one of them), report generation will be slower, but it will make database operation faster during filesystem scans. For instance, if you only need reports by user, disable *group_acct* to optimize scan speed.
  E.g:
  ```
  user_acct = on ;
  group_acct = off ;
  ```
  See section 9.3 for more details.

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";`
- **[new 2.4] innodb** (Boolean): by default, Robinhood creates a database using InnoDB MySQL engine. Usually, InnoDB results in better performances, depending on your usage of robinhood.
  If you want to use MyISAM instead, disable this parameter:
  `innodb = disabled ;`
  Note: if Robinhood already created its database tables, you need to drop them (run `rbh-config empty_db`) and restart Robinhood to create the tables with the new DB engine.

3.14 Filesystem scan parameters

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

- **scan_interval** (duration): specifies a fix frequency for scanning the filesystem (daemon mode).

or

- **min_scan_interval**, **max_scan_interval** (durations): it is possible to adapt the scan frequency depending on the current filesystem usage. Indeed, it is not necessary to scan the filesystem frequently when it is empty (because no purge is needed). When the filesystem is full, robinhood will need a fresh list for purging files, so it is better to scan more frequently. For this, specify the interval between scans using:
  - **min_scan_interval**: the frequency for scanning when filesystem is full;
  - **max_scan_interval**: the frequency for scanning when filesystem is empty

  The interval between scans is computed according to this formula:
  `min + (100% - current usage)*(max-min)`
  Note : those parameters are not compatible with simple **scan_interval**.

- **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 filesystem operations are blocked more than this time, there are cancelled.
  It is recommended to enable 'exit_on_timeout' option in that case.
- **exit_on_timeout** (Boolean)**:** robinhood exits if filesystem operations are blocked for a long time (specified by **'**scan_op_timeout**').**
- **spooler_check_interval** (duration): interval for testing FS scans, deadlines and hangs.
- **nb_prealloc_tasks** (integer): number of pre-allocated task structures (advanced parameter).
- **[new 2.4] completion_command** (string): external completion command to be called when robinhood terminate a filesystem scan. The full path for the command must be specified. Command arguments can contain special values:
  - **{cfg}** for the config file
  - **{fspath}** for the path of the filesystem managed by robinhood

  Eg.: `completion_command = "/path/to/script.sh –f {cfg} –p {fspath}";`

- **Ignore** block (Boolean expression): robinhood will skip entries and directories that match the given expression. Several ignore blocks can be defined in FS_Scan.

  Examples:

```
Ignore {
      # ignore ".snapshot" directories (don't scan them)
      type == directory
      and
      name == ".snapshot"
}

Ignore {
      # ignore a whole part of the filesystem
      tree == "/mnt/lustre/dont_scan_me"
}
```

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, 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";
    }
}
```

You can also control the frequency of acknowledging chengelog records to Lustre (this reduces the number of filesystem calls), by specifying "**batch_ack_count**".
A zero value indicates that changelog records are acknowledged once they have all been read and processed.
E.g: clear changelog every 500 records

```
batch_ack_count = 500 ;
```

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 below 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 filesystem calls:
```
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 summaries.

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.

[New 2.2] FileClass matching at scan time

By default in version 2.2, entry fileclass are matched immediately when entries are discovered (basically at scan time). This make it possible for the reporting command to display the fileclass information.
If the filesystem is overloaded, this can be disabled using the '**match_classes**' parameter in the '**EntryProcessor**' block. In this case, entry fileclasses will only be matched when applying policies.

```
match_classes = FALSE;
```

[New 2.4] Detecting "fake" mtime

You may have robinhood policies based on file modification time. If users change their file modification times using 'touch' command, 'utime' call, or copying files using 'rsync' or 'cp –p', it may result in unexpected policy decisions (like purging recently modified files because the user set a mtime in the past).
To avoid such situation, enable the '**detect_fake_mtime**' parameter in the '**EntryProcessor**' block:

```
detect_fake_mtime = TRUE;
```

Note: robinhood traces the fake mtime it detects with 'debug' trace level.

4 Reporting tool (rbh-report)

4.1 Overview

The content of Robinhood's database can be very useful for building detailed reports about filesystem content. 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

(* = new feature in robinhood 2.4)

Usage: `rbh-report [options]`

Stats switches:

--activity, **-a**
Display stats abount daemon's activity.

--fs-info, **-i**
Display filesystem content statistics.

--class-info [=fileclass]
Display Fileclasses summary. Use optional parameter fileclass for retrieving stats about a given fileclass (wildcards allowed).

--user-info [=user], **-u** user
Display user statistics. Use optional parameter user for retrieving stats about a single user.

--group-info [=group], **-g** group
Display group statistics. Use optional parameter group for retrieving stats about a single group.

--top-dirs [=count], **-d** count
Display largest directories. Optional argument indicates the number of directories to be returned (default: 20).

--top-size [=count], **-s** count
Display largest files. Optional argument indicates the number of files to be returned (default: 20).

--top-purge [=count], **-p** count
Display oldest entries eligible for purge. Optional argument indicates the number of entries to be returned (default: 20).

--top-rmdir [=count], **-r** count
Display oldest empty directories eligible for rmdir. Optional argument indicates the number of dirs to be returned (default: 20).

--top-users [=count], **-U** count
Display top disk space consumers. Optional argument indicates the number of users to be returned (default: 20).

--dump, **-D**
Dump all filesystem entries.

--dump-user user
Dump all entries for the given user.

--dump-group group
Dump all entries for the given group.

--dump-ost ost_index
Dump 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.

-C class, **--filter-class** class
Report only entries in the given FileClass.

--count-min cnt
Display only topuser/userinfo with at least cnt entries

Accounting report options:

* **--size-profile**, **--szprof**
Display size profile statistics

* **--by-size-ratio** range, **--by-szratio** range
Sort on the ratio of files in the given size-range
range: <val><sep><val>- or <val><sep><val-1> or <val><sep>inf
<val>: 0, 1, 32, 1K 32K, 1M, 32M, 1G, 32G, 1T
<sep>: ~ or ..
e.g: 1G..inf, 1..1K-, 0..31M

--by-count
Sort top users by count instead of sorting by volume

--by-avgsize
Sort users by average file size

--reverse

```
        Reverse sort order
    -S, --split-user-groups
        Display the report by user AND group
    -F, --force-no-acct
        Generate the report without using accounting table

Config file options:
    --config-file=file, -f file
        Path to configuration file (or short name).

Output format options:
    -c , --csv
        Output stats in a csv-like format for parsing
    -q , --no-header
        Don't display column headers/footers

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

Example of output:

```
type    ,      count,     volume,   avg_size
dir     ,    1780074,    8.02 GB,    4.72 KB
file    ,   21366275,   91.15 TB,    4.47 MB
symlink ,     496142,   24.92 MB,         53

Total: 23475376 entries, 100399805708329 bytes (91.31 TB)
```

Fileclasses summary (`--class-info` option)

This generates a summary of fileclasses used for purge policy.
Example of output:

```
purge class         ,      count,   spc_used,     volume,   min_size,   max_size,   avg_size
BigFiles            ,       1103,   19.66 TB,   20.76 TB,    8.00 GB,  512.00 GB,   19.28 GB
EmptyFiles          ,    1048697,    7.92 GB,    4.15 GB,          0,    1.96 GB,    4.15 KB
SmallFiles          ,   20218577,    9.63 TB,    9.67 TB,          0,   95.71 MB,  513.79 KB
ImportantFiles      ,     426427,   60.75 TB,   60.86 TB,   16.00 MB,    7.84 GB,  149.66 MB
```

Note: if you need to list all entries of a given fileclass, use the '`--dump-all`' and the '`--filter-class`' options together:

```
rbh-report –-dump-all --filter-class project_B
```

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

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

Example of output:

```
user      ,     type,      count,   spc_used,   avg_size
foo       ,      dir,      75450,  306.10 MB,    4.15 KB
foo       ,     file,     116396,   11.14 TB,  100.34 MB

Total: 191846 entries, 12248033808384 bytes used (11.14 TB)
```

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

Same report as 'user-info', for groups.

Info per user <u>and</u> group (`--split-user-groups` option, or briefly: `-S`)

This option splits user stats by group.
E.g. : `rbh-report -u john -S`

```
user ,      group,   type,    count,    spc_used,   avg_size
john ,     group1,   dir,      208,    106.49 KB,          0
john ,     group1,  file,   259781,    781.21 GB,   3.08 MB
john ,     group2,   dir,      125,     64.00 KB,          0
john ,     group2,  file,    34525,      4.26 GB, 123.46 KB
```

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, avg size of dir entries, last modification time.

Example of output:

```
rank,            path, dircount,    avgsize,    user,      group,            last_mod
   1,      /lustre/90k,    89957,          0,    john,       pr01,  2012/03/21 21:50:14
   2,    /lustre/foooo,    80939,   79.13 KB,     tod,       pr32,  2011/05/25 13:33:05
   3,  /lustre/results,    73983,  197.49 KB,    eddy,       pr32,  2011/06/15 19:23:40
   4,      /lustre/bar,    73524,    1.72 MB, charlie,       pr87,  2012/09/15 19:36:17
   5,      /lustre/log,    53693,  350.94 KB,    lily,       pr14,  2012/10/22 18:32:31
```

- By default, directories are sorted by entry count
- [new 2.4] '--by-avgsize' option sorts dirs by average file size
- [new 2.3.3] '--reverse' option reverses sort order (e.g. smallest first)
- [new 2.3.3] Use '--count-min *N*' option to only display directories with at least *N* entries.

Example of use: find directories with plenty of small files
(directories with at least 1000 entries, sorted by average file size, from the smaller avgsize to the bigger)

```
rbh-report --topdirs --count-min=1000 --by-avgsize –-reverse
```

```
rank,        path, dircount,    avgsize,  user,   group,            last_mod
  1, /lustre/dir1,     1121,    6.01 KB,   foo,     gp1,  2012/04/29 18:20:34
  2, /lustre/dir2,     1543,    7.75 KB,   foo,     gp1,  2012/04/17 15:34:55
  3, /lustre/x,        1029,   12.88 KB,   bar,     gp2,  2011/02/16 01:22:57
  4, /lustre/y,        1019,   12.99 KB,   bar,     gp2,  2011/01/21 11:16:06
...
```

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

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

Example of output:

```
rank,           path,       size, user, group,          last_access,             last_mod,  purge class
   1, /tmp/file.big1,  512.00 GB, foo1,   p01,  2012/10/14 17:41:38,  2011/05/25 14:22:41,     BigFiles
   2, /tmp/file2.tar,  380.53 GB, foo2,   p01,  2012/10/14 21:38:07,  2012/02/01 14:30:48,     BigFiles
   3, /tmp/big.1,      379.92 GB, foo1,   p02,  2012/10/14 20:24:20,  2012/05/17 17:40:57,     BigFiles
...
```

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

Display users who consume the larger disk space.

```
rank, user      ,   spc_used,      count,  avg_size
   1, usr0021   ,   11.14 TB,     116396,  100.34 MB
   2, usr3562   ,    5.54 TB,        575,    9.86 GB
   3, usr2189   ,    5.52 TB,       9888,  585.50 MB
...
```

Useful options:

- [new 2.3.2] '--by-count' option sorts users by entry count
- [new 2.3.3] '--by-avgsize' option sorts users by average file size
- [new 2.3.3] '--reverse' option reverses sort order (e.g. smallest first)
- [new 2.3.3] Use '--count-min *N*' option to only display users with at least *N* entries.
- [new 2.4] '--by-size-ratio' option makes it possible to sort users using the percentage of files in the given range.

Example of use: find users with the highest percentage of files < 32MB
(users sorted by file percentage in the range [0-32M[, with at least 1000 entries)

```
rbh-report --topusers --by-szratio=0..31M --count-min=1000
```

```
rank, user  ,   spc_used,  count,   avg_size,  ... , ratio(0..31M)
   1, foo1  ,    1.23 GB,    2915,  440.57 KB,  ... , 100.00%
   2, bar2  ,    1.60 MB,    3876,   42.50 KB,  ... , 100.00%
   3, foo2  ,  511.36 MB,   11298,   47.79 KB,  ... , 99.95%
   4, bar3  ,   40.06 GB,    5247,   39.35 MB,  ... , 99.77%
   5, usr202,    8.72 GB,   58152,    5.10 MB,  ... , 99.75%
```

Size profile (`--size-profile` option)

This option can be used to display the file size profile in the filesystem (with with `--fs-info`), for a user (with `--user-info`), or a group (with `--group-info`).

```
rbh-report -u foo --szprof
user,  type,  count, spc_used,  avg_size,  0, 1~31, 32~1K-, 1K~31K, 32K~1M-, 1M~31M, 32M~1G-, 1G~31G, 32G~1T-, +1T
foo ,  file, 116396, 11.14 TB, 100.34 MB, 56, 2536,  28661,  11054,   16286,  14443,   43291,     68,       1,   0
```

Note: such information is also available as chart via the robinhood's web UI

robinhood activity (`--activity` option)

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

```
Filesystem scan activity:

    Current scan interval:   2.8d
```

```
    Previous filesystem scan:
            start:           2012/02/07 11:16:12
            duration:        2h 28min 50s

    Last filesystem scan:
            status:          running
            start:           2012/02/08 15:00:26 (1h 13min 16s ago)
            last action:     2012/02/08 16:00:27 (1min 15s ago)

         Statistics:
            entries scanned: 3757494
            errors:          0
            timeouts:        0
            # threads:       1
            average speed:   4177 entries/sec
        >>> current speed:   1985 entries/sec



*Changelog stats:

        Last read record id:      71812545
        Last record read time:    2012/02/08 16:00:14
        Last committed record id: 71812545
        Changelog stats:
                 type            total  (diff)  (rate)
                 MARK:           135406  (+28)   (0.03/sec)
                CREAT:         16911356  (+82)   (0.09/sec)
                MKDIR:           629730
                HLINK:           128048
                SLINK:           601645  (+11)   (0.01/sec)
                MKNOD:                0
                UNLNK:         11150277  (+47)   (0.05/sec)
                RMDIR:           411672
                RNMFM:          1566093
                RNMTO:          1572679  (+26)   (0.03/sec)
                 OPEN:                0
                CLOSE:                0
                IOCTL:                0
                TRUNC:                0
                SATTR:          6674152  (+24)   (0.03/sec)
                XATTR:                0
                  HSM:                0
                MTIME:          8201315  (+193)  (0.21/sec)
                CTIME:                0
                ATIME:                0

Storage unit usage max: 55.33%

Last purge: 2011/05/04 13:45:44
       Target: OST #16
       Status: OK (83 files purged, 958MB)

* [new in 2.3.3]
```

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

These options can be used for listing entries with a given criteria.
They can be used with filtering options on path (–P) or fileclass (-C).

Example: listing all entries of user ‘foo’ in the ‘BigFiles’ class:

```
# rbh-report --dump-user foo –C BigFiles

type,      size,    user,    group,  purge class, path
file,  32.00 GB,     foo,  grp2191,     BigFiles, /lustre/foo/job2389/data.1123
file, 135.32 GB,     foo,  grp2191,     BigFiles, /lustre/foo/job1223/data.2332


Total: 2 entries, 179658481991 bytes (167.32 GB)
```

5 [new 2.4] rbh-find

5.1 Overview

This is a clone of the standard ‘find’ command, much faster, as is it based on robinhood’s database (a database is much more adapted for performing queries based on criteria, than a filesystem which is more adapted for IOs).
If you are using Lustre v2 Changelogs, you will get an even more fresh result as the database is fed in soft real-time, whereas the long time of running standard ‘find’ results in an out-of-date result at the end.
Note: this command may require an access to the filesystem (not only a connection to the DB).

5.2 Syntax

```
Usage: rbh-find [options] [path|fid]...

Filters:
    -user user
    -group group
    -type type
       'f' (file), 'd' (dir), 'l' (symlink), 'b' (block), 'c' (char), 'p' (named pipe/FIFO),
       's' (socket)
    -size size_crit
       [-|+]<val>[K|M|G|T]
    -name filename
    -mtime time_crit
       [-|+]<val>[s|m|h|d|y] (s: sec, m: min, h: hour, d:day, y:year. default unit is days)
    -mmin minute_crit
        same as '-mtime Nm'
    -msec second_crit
        same as '-mtime Ns'
    -ost ost_index

Output options:
    -ls         : display attributes

Program options:
    -f config_file
    -d log_level
       CRIT, MAJOR, EVENT, VERB, DEBUG, FULL
    -h, --help
        Display a short help about command line options.
    -V, --version
        Display version info
```

Note:

- The path or fid argument is optional. If not specified, the command will run on the entire filesystem specified in robinhood’s config file.
- Size can be specified with a suffix (eg. 10M for 10 MB). No suffix means ‘bytes’.
- Time criteria can be suffixed (e.g. 10h for 10 hours). No suffix means ‘days’, like the standard *find* syntax).
- Specifying ‘+’ before a value matches values strictly higher than the value (e.g. +10M matches files whose size is > 10MB).
- Specifying ‘-’ before a value matches values strictly smaller than the value (e.g. -10M matches files whose size is < 10MB).

5.3 Performance

Performance comparison on a 1 million entry Lustre filesystem:

find

```
find /lustre -user foo -type f -size -32M -ls
```

(no possible criteria on OST index)
> **58m13s**

lfs find

```
lfs find /lustre -user foo -type f --obd lustre-OST0001
```

(no possible criteria on size)
> **20m46s**

rbh-find

```
rbh-find /lustre -user foo -type f -size -32M -ost 1 –ls
```

(criteria on both size and OST index)
> **1.2s**

⇨ **~3000 times faster than find, ~1000 times faster than lfs find**

6 [new 2.4] rbh-du

6.1 Overview

This is a clone of the standard ‘du’ command, querying robinhood’s DB, with additional features and enhancements:

- filtering per user (**-u** option), per group (**-g** option) or per type (**-t** option)
- more detailed outputs (**-d** option): display entry count, size and disk usage per type.

Note: this command requires an access to the filesystem (not only a connection to the DB).

6.2 Syntax

```
Usage: rbh-du [options] [path|fid]

Filters:
    -u user
    -g group
    -t type
       'f' (file), 'd' (dir), 'l' (symlink), 'b' (block), 'c' (char),
       'p' (named pipe/FIFO), 's' (socket)

Output options:
    -s, --sum
       display total instead of stats per argument
    -c, --count
```

```
      display entry count instead of disk usage
   -b, --bytes
      display size instead of disk usage (display in bytes)
   -k, --kilo
      display disk usage in blocks of 1K (default)
   -m, --mega
      display disk usage in blocks of 1M
   -H, --human-readable
      display in human readable format (e.g 512K 123.7M)
   -d, --details
      show detailed stats: type, count, size, disk usage
      (display in bytes by default)

Program options:
   -f config_file
   -l log_level
   -h, --help
       Display a short help about command line options.
   -V, --version
       Display version info
```

6.3 Performances

du

```
du –sh /lustre/grp1234/usr123
12T     /lustre/grp1234/usr123
```

> **3m22.108s**

rbh-du

```
rbh-du -Hd /lustre/grp1234/usr123
dir count:75448, size:305.8M, spc_used:306.1M
file count:116250, size:11.1T, spc_used:11.1T
```

> **16.815s**

⇨ **12 times faster than *du* (for 200k entries in user's directory)**

7 [new 2.3.2] Web interface

7.1 Overview

Web interface is a new feature in Robinhood 2.3.2. It makes it possible for administrator to visualize top disk space consumers (per user or per group), top inode consumers with fancy charts, details for each user. It also makes it possible to search for specific entries in the filesystem.

7.2 Installation and configuration

You can install it on any machine with a web server (not necessarily the robinhood or the database node). Of course, the web server must be able to contact the Robinhood database.

Requirements: php/mysql support must be enabled in the web server configuration.
The following packages must be installed on the web server: `php, php-mysql, php-xml, php-pdo, php-gd`

The following parameter must be set in httpd.conf:

```
AllowOverride All
```

Install robinhood interface:

- install robinhood-webgui RPM on the web server (it will install php files into /var/www/html/robinhood)

or

- untar the robinhood-webgui tarball in your web server root directory (e.g. /var/www/http)

Configuration:
In a web browser, enter the robinhood URL: http://*yourserver*/robinhood
The first time you connect to this address, fill-in database parameters (host, login, password, …).
Those parameters are saved in:
`/var/www/http/robinhood/app/config/database.xml`

That's done. You can enjoy statistics charts.

8 Configuration and admin helper (rbh-config)

rbh-config is a script (installed in /usr/sbin) that helps administrator for configuration and maintenance operations (mostly on robinhood database). It can be used as an interactive command, or in batch mode:

- Interactive mode: just specify an action, it prompts for additional parameters.
  E.g: `rbh-config test_db`

- Batch mode: specify all required parameters on command line.
  E.g.: `rbh-config test_db "robinhood_scratch" "passw0rd"`

Available actions:
- **precheck_db**: check database packages and service
- **create_db**: create robinhood database
- **empty_db**: clear robinhood database content
- **test_db**: test if the database exists and test login on it
- **repair_db**: check tables and fix them after a mysql server crash
- **reset_classes**: reset fileclasses after a change in config file
- **enable_chglogs**: enable ChangeLogs (Lustre 2.x only, must be executed on MDT)
- **[new 2.3] backup_db**: backup robinhood database

9 System and database tunings

9.1 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`

9.2 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
```

9.3 [new 2.3] Optimize reporting speed

By default, robinhood is optimized for speeding up common accounting reports (by user, by group, …), but this can slow database operations during filesystem scans. If you only need specific reports, you can disable some parameters to make scan faster.
For instance, if you only need usage reports by user, you had better disable *group_acct* parameter; this will improve scan performance. In this case, reports on groups will still be available, but their generation will be slower: if you request a *group-info* report and if *group_acct* is *off*, the program will iterate through all the entries (complexity: O(n) with n=number of entries). If *group_acct* is *on*, robinhood will directly access the data in its accounting table, which is quite instantaneous (complexity: O(1)).
Performance example: with *group_acct* parameter activated, *group-info* report is generated in 0.01sec for 1M entries. If *group_acct* is disabled, the same report takes about 10sec.

9.4 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

# This must be 2 * nbr_cpu_cores:
thread_concurrency=32
```

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

InnoDB specific tunings

If you use innoDB engine (see database parameters in 3.13), these tunings can significantly increase performances (in `/etc/my.cnf`):

```
# 2*nbr_cpu_cores
innodb_thread_concurrency=16
# half of the memory is quite good
innodb_buffer_pool_size=4000M
# memory cache tunning
innodb_max_dirty_pages_pct=15
```

If you want to parallelize scans across several nodes (using the `--partial-scan` option), you need to increase the maximum number of connections the database allows:

```
max_connections=2048
```

If you get many DB connection failures, increase this parameter:

```
connect_timeout=60
```

10 Known issues

- **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.