# Robinhood v2

# Lustre-HSM Policy Engine

# Admin Guide

Thomas LEIBOVICI
CEA/DAM

<thomas.leibovici@cea.fr>

V2.1.2

9th of February, 2010

# Table of contents

# 1 Product overview

"Robinhood FS Monitor" is Open-Source software developed at CEA/DAM. It can be used as a PolicyEngine for Lustre-HSM.

It runs as a user-space daemon, and collects information from Lustre to be aware of filesystem state, file status, to make policy decisions:

- It is notified of important HSM events by Lustre changelogs:
    - when a file becomes "dirty" in Lustre;
    - when a file is copied to/from the back-end storage (HSM);
    - when a file is removed from Lustre.

- It retrieves information about file, using POSIX and liblustreapi calls:
    - POSIX attributes
    - path
    - stripe information (OST, pool…)
    - Lustre-HSM specific flags (*archived*, *dirty*, *released*, …)

- File information is stored persistently in a transactional database.
    - Changelogs records are managed in DB transactions, so none of them can be missed or lost.
    - Scanning filesystem namespace is rarely needed: only at first setup time, or in case of a database disaster.

- It monitors OST space usage, and release disk space when needed. It has the ability to purge files per OST. It releases archived files from the least recently used, according to custom policies.

- It also makes archiving decisions, using customizable and flexible policies. It can also pass configurable hints to the archiving tool, to guide HSM specific decisions (e.g. select HPSS Class of Service…).

# 2 Quick start

## 2.1 Compilation and installation

It is advised to build the RPM on your target system, to ensure the compatibility with your Lustre and database versions.

**Requirements**

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

- mysql-devel
- lustre API library ('/usr/include/liblustreapi.h' and '/usr/lib/liblustreapi.a'), **including all Lustre-HSM specific functions**. It is installed by standard Lustre rpm.

**Compilation**

Unzip and untar the sources:

```
> tar zxvf  robinhood-2.1.2.tar.gz
> cd robinhood-2.1.2
```

Run the "configure" script to generate Makefiles:

- specify the `--with-purpose=LUSTRE_HSM` option for using it as Lustre-HSM PolicyEngine;
- set the prefix of installation path (default is /usr/local) with '`--prefix=<path>`'

```
> ./configure --with-purpose=LUSTRE_HSM --prefix=/usr
```

Other '`./configure`' options:

- On Lustre 2.0-alpha5/6: a fork() in liblustreapi results in 'defunc' robinhood process when reading MDT changelogs. Use the '`--enable-llapi-fork-support`' option to avoid this.

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:

- 'rh-hsm' and 'rh-hsm-report' binaries
- 'rh-config' script (configuration helper)
- Configuration templates
- /etc/init.d/robinhood-hsm 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
```

**Robinhood service**

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

```
> chkconfig robinhood-hsm on
```

This service starts one 'rh-hsm' instance for each configuration file it finds in '/etc/robinhood.d/hsm' directory. Thus, if you want to manage several Lustre-HSM bindings, 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>
```

## *2.2 Database configuration*

Before running Robinhood for the first time, you must create its database and configure its access rights.

- Install 'mysql' and 'mysql-server' packages on the machine where the database will be located (it can be different from the machine where Robinhood will run).

- Start the database engine :
  ```
  service mysqld start
  ```

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

  ```
  # check database requirements:
  rh-config precheck_db
  # create the database:
  rh-config create_db
  ```

Alternatively, if you want a better control of your database configuration, you can perform the following steps of your own (without using the 'rh-config' script):

- As superuser (root): create the database (one per filesystem) using the `mysqladmin` command:
  `mysqladmin create <`*`robinhood_db_name`*`>`

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

  Execute the following commands in the MySQL session:

  - Create a database user and set its access rights ('%' matches all host names. Your can replace it by the host where Robinhood will be running):
    ```
    GRANT USAGE ON robinhood_db_name.* TO 'robinhood'@'%'
    identified by 'your_password';
    GRANT ALL PRIVILEGES ON robinhood_db_name.* TO 'robinhood'@'%';
    ```

  - Refresh server access settings:
    ```
    FLUSH PRIVILEGES ;
    ```

  - You can check user privileges by executing:
    ```
    SHOW GRANTS FOR robinhood ;
    ```

- For testing access to database, execute the following command on the host 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.

- Initially, the database schema is empty. Robinhood will automatically create it the first time it starts.

## *2.3 Lustre changelogs setup*

Enabling Lustre changelogs is required for Lustre-HSM.

You can simply achieve this by running 'rh-config' on the MDS:

```
> rh-config enable_chglogs
```

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

- Enable (at least) the following changelog record types: HSM, CREAT, UNLNK, TRUNC, TIME, SATTR:
  ```
  lctl set_param  mdd.*.changelog_mask "HSM CREAT UNLNK TRUNC TIME
  SATTR"
  ```

- If your policy rules are massively based on file paths or file names, and if files are frequently renamed or moved from one directory to another, you should also activate the following events: RNMFM, RNMTO:
  ```
  lctl set_param  mdd.*.changelog_mask "+RNMFM +RNMTO"
  ```

- 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.

## 2.4 PolicyEngine (basic) configuration

**Bootstrap**

You can easily start your PolicyEngine configuration from a template file:

- Two templates are installed in the ‘/etc/robinhood.d/hsm/templates’ directory (installed by the RPM): a basic example, and a more detailed.
- You can also generate a documented example using the ‘--template’ or ‘-T’ option of the Robinhood command:
  ```
  rh-hsm -T template_file.conf
  ```

**General, Log parameters**

First, set the general options: the filesystem to be managed and log files (make sure the log directory exists).

```
General
{
    fs_path = "/mnt/lustre";
}

Log
{
    log_file       = "/var/log/robinhood/lustre_hsm.log";
    report_file    = "/var/log/robinhood/reports.log";
    alert_file     = "/var/log/robinhood/alerts.log";
}
```

**Database parameters**

Then, set database parameters : database host, database name, database user, and a file that contains password.

```
ListManager
{
    MySQL
    {
        server = db_host;
        db = robinhood_test;
        user = robinhood;
        password_file = /etc/robinhood.d/.dbpass;
    }
}
```

/!\ Make sure the password file cannot be read by any user (use mode ‘600’ for example).
If you don’t care about security, you can directly specify the password in the configuration file, by specifying a *password* parameter instead:
Example: `password = ‘passw0rd’;`

**Changelog parameters**

Specify MDT name and the changelog reader id you got at step 2.3 *(Lustre changelogs setup)*:

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

    force_polling = ON;
}
```

About ‘`force_polling = ON`’: on Lustre 2.0-alpha5/6, changelog readers need to perform active polling to get new events from MDT. So, on these lustre versions, you need to activate polling with this option.

## *2.5 First run*

The first time you run the policy engine, you need to perform a full filesystem scan to collect information about previously existing entries. By security, this scan is required even if your filesystem is actually empty.

To perform this scan, start Robinhood using the configuration file you just wrote. Specify the “--scan” option and the “--once” option, so it will exit when the scan is completed.

```
rh-hsm  –f your_config.conf --scan --once
```

Note 1: by default, robinhood uses the first config file it finds in the ‘/etc/robinhood.d/hsm’ directory. So, your config file is the only one in this directory, you can omit the ‘`-f`’ parameter:

```
rh-hsm  --scan --once
```

Note 2: to get traces inline, you can change log file with the ‘–L’ option (this overrides ‘log_file’ parameter in configuration file). Special values ‘stderr’ and ‘stdout’ can be used:

```
rh-hsm  --scan -–once –L stderr
```

You get something like this:

```
2009/07/17 13:49:06: FS Scan | Starting scan of /mnt/lustre
...
2009/07/17 13:49:06: FS Scan | Full scan of /mnt/lustre completed,
                               7130 entries found. Duration = 0.12s
2009/07/17 13:49:06: FS Scan | File list of /mnt/lustre has been updated
```

```
2009/07/17 13:49:06: Main | All tasks done! Exiting.
```

Note 3: if your filesystem is accessed during this scan, you should handle Changelog records at the same time, so the PolicyEngine database will be kept up-to-date.
Unlike the scan operation, changelog processing must be done continuously (not only once). So we start it as a separated command. Additionally, we can specify the '--detach' option to execute it in background.

```
rh-hsm  –-handle-events --detach
```

## 2.6 Minimal migration policy

With the basic configuration we wrote, the PolicyEngine is only able to process changelogs and update its database.

Let's now add a basic migration policy to the configuration file:

```
migration_policies
{
    policy default
    {
        condition
        {
            last_mod > 6h
        }
    }
}
```

This definition only consists of a special 'default' policy that applies to all files. It specifies a condition for archiving files. In this example, we want to migrate "dirty" files that have not be modified for 6 hours.

You can also specify the following migration parameters: number of threads for performing migration queries, migration runtime interval, and the maximum number (and/or volume) of copies to be requested at each pass.

```
Migration_parameters
{
    nb_threads_migration = 4;
    runtime_interval = 15min;
    max_migration_count = 10000;
    max_migration_volume = 10TB;
}
```

To test this migration policy inline, let's run it once with the '--dry-run' option, so the PolicyEngine doesn't really request file copy:

```
rh-hsm  –-migrate –-once –-dry-run –L stderr
```

## 2.7 Minimal purge policy

The basic purge policy is very similar to the migration policy. It consists of a ‘default’ case, with a condition for releasing files in Lustre (after they have successfully been archived).
Note that files are not released immediately when they match this condition: this is just a minimal condition for being included to the LRU list for purge. When disk space is needed, files are then purged in the order of this list.

```
purge_policies
{
    policy default
    {
        condition
        {
            last_access > 1d
        }
    }
}
```

In this example, we never want to release files accessed during the last hour.
All other archived files can be considered in purge LRU list.

You also need to specify a condition for triggering a purge, and for stopping it. This is done with a ‘purge_trigger’. The most common (and secure) trigger is on OST usage:

```
purge_trigger
{
    trigger_on         = OST_usage ;
    high_watermark_pct = 85% ;
    low_watermark_pct  = 80% ;
    check_interval     = 5min ;
}
```

In this trigger, OST usage is checked every 5 minutes. A purge operation is performed on an OST if its usage exceeds 85%. Files on this OST are then released until the OST usage reaches the low watermark (80% in this example).

Then, you can test this purge policy by executing:
```
rh-hsm –-purge --once –-dry-run –L stderr
```

## 2.8 Using file classes

Robinhood makes it possible to apply different migration/purge policies to files, depending on their properties (path, posix attributes, …). This can be done by defining file classes that will be addressed in policies.

In this example, we define 3 classes and apply different policies to them:
- We never want “*.log” files that owns to ‘root’ to be released;
- We want files in directory /tmp/A to stay longer on disk than other directories.

To do this, we define those classes in the ‘filesets’ section of the configuration file. We associate a custom name to each FileClass, and give its definition:

```
Filesets
{
      # log files owned by root
      FileClass root_log_files
      {
            definition
            {
                  owner == root
                  and
                  name == “*.log”
            }
      }

      # files in filesystem tree /fs/A
      FileClass A_files
      {
            definition
            {
                  tree == “/fs/A”
            }
      }
}
```

Then, those class names can be  used in policies:

- entries can be white-listed using a ‘ignore_fileclass’ statement;
- they can be targeted in a policy, using a ‘target_fileclass’ directive.

```
Purge_Policies
{
      # don’t purge log files of ‘root’
      ignore_fileclass = root_log_file;

      # keep files in /fs/A at least 12h after their last access time
      Policy purge_A_files
      {
            target_fileclass = A_files;
            condition
            {
                  last_access > 12h
            }
      }

      # ‘default’ policy applies to all other files
      # (files not in /fs/A and that don’t own to root)
      Policy default
      {
            condition
            {
                  last_access > 1h
            }
      }
}
```

Notes:

- a given FileClass cannot be targeted simultaneously by several purge policies;
- policies are matched in the order they appear in configuration file. In particular, if 2 policy targets overlap, the first matching policy will be considered;
- For ignoring entries, you can directly specify a condition in the ‘purge_policies’ section, using a ‘ignore’ block:

```
      Purge_Policies
      {
         Ignore
         {
            owner == root
            and
            name == “*.log”
         }

         …
```

Note: this example is for purge policies, but filesets can also be used for migration policies.

## 2.9 Minimal ‘hsm_remove’ policy

When files are definitively removed from Lustre, you may want to clean the related data in the HSM, to free tape space. However, it can be interesting to defer the removal in the HSM, to prevent from accidental ‘rm’. This is the purpose of the ‘hsm_remove’ policy.
It only consists of 2 parameters:

- **no_hsm_remove**: if TRUE, this disables object removal in the HSM (objects are never cleaned in the HSM when they are removed from Lustre).
- **deferred_remove_delay**: delay for deleting an object in the HSM after it has been removed from Lustre.

```
hsm_remove_policy
{
      deferred_remove_delay = 7d;
}
```

You can also specify the following parameters: numbers of threads for performing ‘remove’ requests, runtime interval, and the maximum number of requests per pass:

```
hsm_remove_parameters
{
    runtime_interval = 5min;
    nb_threads_rm = 4;
    max_rm_count = 10000;
}
```

To test this policy inline, run it once with the ‘--dry-run’ option, so the PolicyEngine doesn’t really request file removal:

```
rh-hsm –-hsm-remove –-once –-dry-run –L stderr
```

## 2.10 Running in production

In previous steps, we executed each feature independently (changelog processing, migration, purge, removal…). In production, you can run all of them in the same process without specifying any feature on command line (use ‘--detach’ or ‘-d’ option to run the process in background):

```
rh-hsm –d
```

You can also combine several features:

```
rh-hsm –d –-migrate –-purge
```

# 3 Configuration reference

In the previous sections, we got familiar with main features of Robinhood PolicyEngine.
This section now describes in details all parameters of configuration files.

## *3.1 Syntax*

**General structure**

The configuration file consists of several blocks.
A block can contain:
- a set of key/value peers (separated by semi-colons)
- sub-blocks
- a Boolean expression.

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 )
}
```

**Types**

Parameter values can be:

- A **string** delimited by single or double quotes ( ‘ or “ ).
- A **Boolean** constant. All the following values are accepted (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 specifying sub-expressions.
- Conditions on attributes are specified with the following syntax:
  <attribute> <comparator> <value>.
- Allowed comparators are ‘==’, ‘<>’ or ‘!=’, ‘>’, ‘>=’, ‘<’, ‘<=’.

The following properties can be used in Boolean expressions:

- **tree:** entry is under a given path. Shell-like wildcards are allowed.
  E.g: tree == “/fs/subdir/*/dir1” matches entry “/fs/subdir/foo/dir1/dir2/foo”.

- **fullpath:** entry exactly matches the path. Shell-like wildcards are allowed.
  E.g: fullpath == “/fs/*/foo*” matches entry “/fs/subdir/foo123” but it doesn’t match “/fs/subdir/foo4/file”.

- **name:** entry name matches the given regular expression.
  E.g: name == “*.log” matches entry “/fs/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 (name expected).
  E.g: owner == “root”.

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

- **size:** entry has the specified size. You can use suffixes like 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 of a file (for reading or writing). This is the difference between current time and max(atime, mtime). The 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 mtime.
  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.yyy*: 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

- **external_command** [not yet implemented]: custom script for testing if an entry matches. Must return 0 if the entry matches, a non-null value else.
  Note: special parameters can be used for defining the command (see section XXX for more details).
  E.g: external_command( "/usr/bin/do_match {fullpath}" )

Example of Boolean expression:

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

**Comments**

The '#' and '//' signs indicate the beginning of a comment (except if there are in a quoted string). A 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 definition 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**: logging 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.
- **Migration_Policies**: specifies conditions for archiving files
- **Migration_Parameters:** options about migrations.
- **ListManager** (mandatory): database access configuration.
- **FS_Scan**: options about scanning the filesystem.
- **EntryProcessor**: parameters for entry processing pipeline (for FS scan and changelog processing).

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 appropriate values for your system:

```
rh-hsm --template=<template file>
```

**Default configuration values**

To display the default values for configuration parameters, use the `--defaults` option:

```
rh-hsm --defaults
```

## 3.3 General parameters

General parameters are specified 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";`
- **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 specified by *fs_path* is mounted.
  E.g.: `check_mounted = TRUE;`

## 3.4 Logging parameters

Logging parameters are specified 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`" or "`-l`" 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 migration and purge operations are reported.
  E.g.: `report_file = "/var/logs/robinhood/purge_report.log";`

Two methods can be used for raising alerts: sending a mail, writing to a file, or both.
This is specified 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";`

## *3.5 File class definition*

You may need to apply different migration/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 are grouped in the 'Filesets' block of the configuration file.
Each file class has an identifier (used for addressing it in policies) and a definition (a condition for entries to be in this file class).

You can also specify **migration hints** for each file class: this information is passed to the copytool for guiding HSM-specific decisions. If several "migration_hints" are specified, they are appended using coma as delimiter. See section 3.11 for more details.
E.g: migration_hints = "fileclass={fileclass}";

Lustre-HSM can handle several storage backends, identified by a unique "**archive number**".
You can assign a target backend to each fileclass specifying "archive_num = <n>;".

File classes definition overview:

```
Filesets {
        FileClass  my_class_1 {
                Definition {
                        tree == "/fs/dir_A"
                        and
                        owner == root
                }
                migration_hints = “cos=18”;
                archive_num = 2;
        }

        FileClass my_class_2
        {
                …
        }
        …
}
```

## 3.6 Purge policies

Normally, 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 of:

- ‘**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.
  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
          }
  }
  ```
- The default policy 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 looks like this:

```
purge_policies {
        # don’t purge entries owned by root
```

```
        Ignore { owner == “root” }

        # 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 all 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 trigger types can be used:

**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) - [Not fully implemented in robinhood 2.1.1].
- **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) - [Not fully implemented in robinhood 2.1.1].

- **external_command(“<cmd line>”):** purge start/stop condition is based on an external command. Command output must have a specific syntax, to specify the kind and the amount of files to be purged. [Not implemented in Robinhood 2.1.2].

**Start condition**

This is mandatory for all types of conditions, except “external_command”.
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, TB…

**Stop condition**

This is mandatory for all types of conditions, except “external_command”.
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).

**Runtime interval**

The time interval for checking a condition is set by the “**check_interval**” parameter. 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, release its files 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 the memory usage, but fewer DB requests are performed.

## *3.9 Migration policies*

Normally, files are archived in the order of their last modification time. You can however specify conditions to allow/avoid entries to be archived, depending on their file class, and file properties.

To define migration policies, you can specify:

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

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

- '**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.
  E.g.: `Ignore_FileClass = my_class_1;`
- '**Policy**' sub-blocks: specify conditions for archiving entries of file classes.
  A policy has a custom name, one or several target file classes, and a condition for archiving files.
- You can specify "**migration_hints**" to guide copytool decisions. If a file class has "migration_hints", they are appended to policy "migration_hints" (see section 3.11).

- You can indicate an “**archive_num**”, which will be the target storage backend used for file migration. If archive_num is specified in both fileclass definition and policy definition, policy archive_num overrides fileclass archive_num.

  Example of policy:

```
Policy archive_classes_2and3 {
        target_fileclass = class_2;
        target_fileclass = class_3;

        condition
        {
                Last_mod > 1h
        }
        migration_hints = “cos=1,PE_policy={policy}”;
        archive_num = 1;
}
```

- The default policy 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_mod > 30min
        }
        archive_num = 3;
}
```

As a summary, the ‘migration_policies’ block looks like this:

```
migration_policies
{
        # don’t archive logs
        Ignore { tree == “/fs/logs” and name=“*.log” }

        # don’t archive files of classes ‘class_xxx’ and ‘class_yyy’
        Ignore_FileClass = class_xxx ;
        Ignore_FileClass = class_yyy ;

        # migration policy for files of ‘my_class1’ and ‘my_class2’
        policy my_migr_policy1
        {
                target_fileclass = my_class1;
                target_fileclass = my_class2;
                condition { last_mod > 2h or last_copyout > 1d }
                archive_num = 1;
        }
        …
        # migration policy for all other files
        policy default
        {
                condition { last_mod > 6h or last_copyout > 7d }
                archive_num = 2;
        }
}
```

## 3.10 Migration parameters

Migration parameters are specified in a ‘migration_parameters’ block.
The following options can be specified:

- **runtime_interval** (duration): interval for checking migration policies.
  E.g.: `runtime_interval = 10min ;`
- **max_migration_count** (integer): maximum number of migration requests that can be performed per pass.
  E.g.: max_migration_count `= 1000 ;`
- **max_migration_volume** (size): maximum volume of migration requests that can be performed per pass.
  E.g.: max_migration_volume `= 1TB ;`
- **nb_threads_migration** (integer): this determines the number of ‘archive’ requests that can be performed in parallel.
  E.g.: `nb_threads_migration = 8 ;`
- **backup_new_files** (Boolean): specifies if newly created files with no data must be archived.
- **check_copy_status_on_startup** (Boolean): indicates if previously running copies must be checked when the PolicyEngine restarts.
- **migration_queue_size** (integer): this advanced parameter is for leveraging migration thread load.
- **db_result_size_max** (integer): this impacts memory usage of MySQL server and Robinhood daemon. The higher it is, the more the memory usage, but fewer DB requests are performed.

## 3.11 About migration hints

Migration hints can be used to pass specific information to the transfer tool or to the storage backend.
They can be specified in a file class definition or in a migration policy definition. If both are specified, fileclass hints and policy hints are appended and separated by a coma.

The following special variables can be used in migration_hints:
- “`{policy}`”: stands for the policy name
- “`{fileclass}`”: stands for the file class name
- “`{path}`”: stands for full posix path of the entry
- “`{name}`”: stands for the file name
- “`{ost_pool}`”: stands for the pool name where the file is stored (if any).

Example:

```
Filesets {
        FileClass  my_class_1 {
                Definition {
                        tree == “/fs/dir_A”
                }
                migration_hints = “cos=18”;
                migration_hints = “class={fileclass}”;
        }
}

migration_policies
{

        policy my_migr_policy1
        {
                target_fileclass = my_class_1;
                condition { last_mod > 2h or last_copyout > 1d }
                migration_hints = “pol={policy}”;
                migration_hints = “fullpath=’{path}’”;
        }
        …
}
```

For a file “`/fd/dir_A/dir.1/dir.2/file.x`”, the following hints string will be passed to the copytool:

```
cos=18,class=my_class_1,pol=my_migr_policy1,fullpath=’/fd/dir_A/dir.1/dir.2/file.x’
```

## *3.12 HSM_remove policy*

When files are definitively removed from Lustre, this policy defines when the related data is cleaned in the HSM.
This is specified by the following parameters, grouped in a ‘**hsm_remove_policy**’ block:

- **no_hsm_remove** (Boolean): if TRUE, this disables object removal in the HSM (objects are never cleaned in the HSM when they are removed from Lustre).
- **deferred_remove_delay** (duration): delay before deleting an object in the HSM when it has been removed from Lustre.

## *3.13 HSM_remove parameters*

‘HSM_remove’ parameters are specified in a ‘hsm_remove_parameters’ block.
The following options can be specified:

- **runtime_interval** (duration): interval for checking files to be removed.
  E.g.: `runtime_interval = 10min ;`
- **max_rm_count** (integer): maximum number of requests that can be sent per pass.
  E.g.: `max_rm_count = 1000 ;`
- **nb_threads_rm** (integer): this determines the number of ‘remove’ requests that can be performed in parallel.
  E.g.: `nb_threads_rm = 8 ;`
- **rm_queue_size** (integer): this advanced parameter is for leveraging thread load.

## *3.14 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";`

## *3.15 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): those parameters are usefully on systems on which frequent scans are needed. For Lustre-HSM purpose, you should only use the scan feature with the '--once' option, so you can ignore those parameters.
- **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 memory parameter).

## 3.16 Entry processor pipeline options

For handling Changelog records, or 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, we temporarily stop reading changelog records 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): this limits the number of threads that simultaneously perform "path2fid" operations, during a filesystem scan.
- **STAGE_CHECK_EXIST_threads_max** (integer): this limits the number of threads that simultaneously check if an entry already exists in database.
- **STAGE_GET_INFO_threads_max** (integer): this limits the number of threads that simultaneously retrieve extra information about filesystem entries (stat, get stripe, get HSM state flags…).
- **STAGE_HSM_REMOVE_threads_max** (integer): this limits the number of threads that simultaneously send HSM remove requests to coordinator [temporary mechanism that will be removed in next Robinhood versions].
- **STAGE_REPORTING_threads_max** (integer) : this limits the number of threads that simultaneously check and raise alerts about filesystem entries [not implemented in v2.1.2].
- **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 operations of retrieving entries info :
```
STAGE_GET_INFO_threads_max = 2;
```

**Alerts** [not implemented in v2.1.2]
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).

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

```
Alert
{
    type == file
    and
    size > 100GB
      and
      owner != 'foo'
}
```

# 4 The ‘rh-hsm’ command

## 4.1 Execution modes

**Automatic mode**

The ‘rh-hsm’ command runs the PolicyEngine daemon, which automatically applies policies according to its configuration.

By default, it makes all PolicyEngine tasks in a single process:

- Processing changelog records;
- Applying migration policies;
- Monitoring space usage and applying purge policies when needed.
- Remove copies in HSM if files have been deleted in Lustre.

Basically, the following command starts the PolicyEngine as a daemon, using the first configuration file in ‘`/etc/robinhood.d/hsm`’:

```
rh-hsm -d
```

It can also be split in several instances, running on several nodes, to get a better load balancing. This is done by using the following options:

- --handle-events: only process changelogs;
- --migrate: only apply migration policies;
- --purge: only monitor space usage and apply purge policies.
- --hsm-remove: only remove orphan copies in the HSM.

These options can also be used together.
E.g:

```
rh-hsm --migrate --purge
```

**Manual actions**

You can also use the ‘rh-hsm’ command to perform custom actions manually.
Normally, manual actions are started with the ‘--once’ option, so the program exists when it is done. Otherwise, it will loop on performing the action, until the program is killed.

Custom manual actions:

- Before a risky operation on your Lustre filesystem, you may want to backup all modified files immediately, whatever the policy conditions. This is done by executing:
  `rh-hsm –-sync`
  (which is equivalent to `--migrate --once --ignore-policies).`

- Release files in a given OST until its space usage is below the specified percentage:
  `--purge-ost=<ost_index>,<usage_pct_target>`
  E.g.: released file in OST #2 until its space usage is under 80.5%
  `--purge-ost=2,80.5`

- Release files in the entire filesystem until its space usage is below the specified percentage:
  `--purge-fs=<usage_pct_target>`
  E.g.: release files until filesystem usage is below 75%
  `--purge-fs=75`

- Archive files in a given OST, according to migration policies:
  `--migrate-ost=<ost_index>`

- Archive files of a given user, according to migration policies:
  `--migrate-user=<user_name>`

- Archive files of a given group, according to migration policies:
  `--migrate-group=<group_name>`

You can also force migration/purge of all eligible entries, ignoring policy conditions, using the '--ignore-policies' option:

- For migration actions (--migrate, --migrate-ost, --migrate-user, --migrate-group) , this option results in archiving all modified files, even if they are currently being modified.
  /!\ This may generate a huge amount of copy operations.

- For purge actions (--purge, --purge-fs, --purge-os), this option results is releasing the data for all files archived in the HSM and not modified in Lustre.
  /!\ If the target space usage is low, this may result in releasing files that are currently opened.

## 4.2 Command line options

Several command line options have been described in the previous section.
Here is the list of all available options:

```
Usage: rh-hsm [options]

Action switches:
    -S, --scan
        Scan filesystem namespace.
    -P, --purge
        Release file data according to purge policies.
    -M, --migrate
        Copy "dirty" entries to HSM.
    -H, --handle-events
        Handle events from MDT ChangeLog.
    -R, --hsm-remove
        Perform deferred removals in the HSM.
    Default is: --handle-events --purge –-migrate –-hsm-remove

Manual migration actions:
    -s, --sync
        Immediately migrate all modified files, ignoring policy conditions.
        It is equivalent to "--migrate --ignore-policies --once".
    --migrate-ost=ost_index
        Apply migration policies to files on the given OST ost_index.
    --migrate-user=user_name
        Apply migration policies to files owned by user_name.
    --migrate-group=grp_name
        Apply migration policies to files of group grp_name.


Manual purge actions:
    --purge-ost=ost_index,target_usage_pct
        Purge files on the OST specified by ost_index until it reaches the
        specified space 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.
    -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
        (overrides configuration value).
    -t type, --fs-type=type
        Force the type of filesystem to be managed
        (overrides configuration value).

Log options:
    -L logfile, --log-file=logfile
        Force the path to the log file (overrides configuration value).
        Special values “stdout” and “stderr” can be used.
    -l level, --log-level=level
        Force the log verbosity level (overrides 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).
```

### 4.3 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.

# 5 Reporting tool

## 5.1 Overview

Robinhood's database content is very useful for building detailed reports about filesystem. For example, you can get the min/max/average size of files, proportion of archived/released/dirty files, the space used by a given user, etc…
Those statistics can be retrieved using Robinhood reporting command: **rh-hsm-report**.

## 5.2 Command line

```
Usage: rh-hsm-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.
    -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).
    -U count, --topusers[=count]
        Display largest disk space consumers. Optional argument indicate
        the number of users to be returned (default: 20).
    -R, --deferred-rm
        Display all files to be removed from HSM.
    -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.
    --dump-status status
        List all entries with the given status (new, modified|dirty,
        retrieving|restoring, archiving, synchro, released, ...).
```

```
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.
```

## 5.3 Reports

This command can generate the following reports:

**Filesystem content report** (`--fsinfo` option)

This displays HSM status of filesystem entries.

Example of output:

```
Status: new files (no HSM status)
   Count:          1393
   Volume:    176.25 MB   (184815609 bytes)

Status: dirty (must be archived)
   Count:         12450
   Volume:    133.62 GB   (143473382523 bytes)

Status: being retrieved
   Count:           255
   Volume:     25.42 GB   (27294517166 bytes)

Status: being archived
   Count:           564
   Volume:     68.16 GB   (73186242724 bytes)

Status: up-to-date (can be purged)
   Count:       1253766
   Volume:    987.63 GB   (1060459637637 bytes)
```

**User info report** (`--userinfo` option)

This displays user file statistics (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** (`--groupinfo` option)

Same report as ‘userinfo’, for groups.

**Top file size** (`--topsize` 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:               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
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
Stripe count:      2
Stripe size:       4.00 MB   (4194304 bytes)
Storage units:     OST #2, OST #1
…
```

**Top purge candidates** (`--toppurge` option)

This displays files that are likely to be released 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, 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
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
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
...
```

### **Top disk space consumers** (`--topusers` 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 migration, last purge…

```
Last Filesystem scan:      2009/09/07 12:59:01

Storage unit usage max:    82.55%

Last migration:            2009/09/07 13:04:36
    Status:                running
    Migration info:        migrate all matching entries

Last purge:                2009/09/01 15:33:34
    Target:                OST #5
    Status:                OK
```

**Deferred removals** (`--deferred-rm` option)

Lists files that must be deleted in the HSM, their last known path and when they will be effectively removed:

```
Rank:                       1
fid:                        [0x20000400:0x1A8:0x0]
Last known path:            /mnt/lustre/tmp_dir/file.1
Lustre rm time:             2009/09/09 15:57:44
HSM rm time:                2009/09/17 15:57:44
```

…

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

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

Example 1: listing all entries on OST #14:

```
# rh-hsm-report --dump-ost 14

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

Example 2: listing all modified entries under `/mnt/lustre/dir1`:

```
# rh-hsm-report --dump-status=mod –-filter-path=/mnt/lustre/dir1

   type,     status,       size,      owner,      group, path
   file,   modified,   41.39 KB,       root,       root, /mnt/lustre/dir1/data.1
   file,   modified,    4.21 GB,       root,       root, /mnt/lustre/dir1/data.2
   file,   modified,   16.94 MB,       root,       root, /mnt/lustre/dir1/subdir/data.3
```

## 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 at compilation time using '`./configure --disable-prep-stmts`' before building Robinhood RPM.

- **Many** <defunc> **'rh-hsm' process when handling Changelogs**

  - Cause:
    In Lustre 2.0-alpha5/6 release, liblustreapi forks a process each time the changelog is reopened, but robinhood doesn't trap SIGCHLD.

  - Workaround:
    Make robihood trap SIGCHLD, by specifying the following option to configure: "`--enable-llapi-fork-support`".