

Robinhood v2

Temporary Filesystem Manager

Admin Guide

Thomas LEIBOVICI
CEA/DAM

<thomas.leibovici@cea.fr>

V2.0-beta

10 Feb. 09

Table of contents

1	Product description	3
1.1	Overview	3
1.2	Execution modes	4
1.3	What changed in Robinhood v2?	4
2	First steps with Robinhood v2.....	6
2.1	Compiling and installing	6
2.2	Robinhood service.....	6
2.3	Command line options	7
2.4	Creating the database	8
3	Writing configuration file.....	9
3.1	Syntax.....	9
3.2	Configuration template and default parameters	11
3.3	General parameters.....	12
3.4	Logging parameters.....	12
3.5	Specifying purge policy	13
3.5.1	Triggers	14
3.5.2	White-listed items	16
3.5.3	Penalty rules	16
3.6	Purge parameters.....	17
3.7	Specifying 'rmdir' policy.....	18
3.8	'Rmdir' parameters	18
3.9	Database parameters.....	18
3.10	Filesystem scan parameters.....	19
3.11	Entry processor pipeline options	20
4	Reporting tool.....	21
4.1	Overview	21
4.2	Command line	21
4.3	Reports	22
5	System and database tunings	25
5.1	Lustre tunings and workarounds	25
5.2	Linux kernel tunings.....	26
5.3	Database tunings	26
6	Known bugs	27

1 Product description

1.1 Overview

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

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

Temporary filesystem management mainly consists of the following aspects:

- Scanning a large filesystem quickly, to build/update a list of candidate files in a database. This persistent storage ensures that a list of candidates is always available if a purge is needed (for freeing space in filesystem). Scanning is done using a parallel algorithm for best efficiency;
- Monitoring filesystem and Lustre OST usage, in order to maintain them 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;
- Removing empty directories that have not been used for a long time, if the administrator wants so;
- Raising alerts, generating accounting information and all kind of statistics about the filesystem.

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

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

1.2 Execution modes

Robinhood can be executed in 2 modes:

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

In the daemon mode:

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

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

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

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

1.3 What changed in Robinhood v2?

Short answer

Quite all...

Medium answer

Robinhood v2 supports all Robinhood v1 features for temporary filesystems management, but its internal architecture has been redesigned around a database engine, which offers a lot of benefits.

Robinhood v2 is also more customizable: alerts and purge policies (whitelist rules, penalties...) can be defined using complex Boolean expression on file attributes.

Last but not least, it can be split into several daemons running on different nodes, which makes it more scalable by sharing CPU load and memory usage between several machines.

Longer answer

Robinhood v2 uses a database engine for managing its list of filesystem entries instead of a list in memory in Robinhood v1. This 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).

Parallel algorithm for scanning filesystem has been improved: in Robinhood v1, several operations were made 'on-the-fly' when scanning the filesystem (like removing empty directories, sending alerts, getting file stripe, building a sorted list...) which could slow-down the scanning. In Robinhood v2, a pool of threads is dedicated to scanning the namespace (they only perform readdir operations). Those threads then push listed entries to a pipeline, and 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 whitelist and penalty rules;
- Checking alert rules and sending alerts;
- Finally insert/update the entry to the database.

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

Purge triggering capabilities have been extended:

- Like in v1, purges can be triggered on OST or filesystem usage.
- A quota-like policy can also be specified: if a given user or group exceeds a specified volume of data, then a purge is triggered on its files.
- An external command can also be used for testing purge start condition.

2 First steps with Robinhood v2

2.1 *Compiling and installing*

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

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

- mysql-devel
- lustre (if Robinhood is to be run on a Lustre filesystem): ‘/usr/include/liblustreapi.h’ and ‘/usr/lib/liblustreapi.a’

Unzip and untar the sources:

```
> tar zxvf robinhood-2.0.0.tar.gz
> cd robinhood-2.0.0
```

Then, use the “configure” script to generate Makefiles:

- use the `--with-purpose=TMP_FS_MGR` option for using it as a temporary filesystem manager;
- specify Mysql for database type with: `--with-db=MYSQL`
- set the prefix of installation path (default is /usr/local) with ‘`--prefix=<path>`’

```
> ./configure --with-purpose=TMP_FS_MGR --with-db=MYSQL --prefix=/usr
```

Finally, build the RPM:

```
> make rpm
```

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

The RPM includes:

- ‘robinhood’ and ‘robinhood-report’ binaries
- Configuration templates
- Man pages
- /etc/init.d/robinhood script

2.2 *Robinhood service*

After installing the RPM, you can enable the ‘robinhood’ service:

```
> chkconfig --add robinhood
```

This service starts one ‘robinhood’ instance for each configuration file it finds in ‘/etc/robinhood.d’ directory.

Thus, if you want to monitor several filesystems, create one configuration file for each of them.

2.3 Command line options

Usage: robinhood [options]

Action switches:

- S, --scan**
Scan filesystem namespace.
- P, --purge**
Purge non-directory entries according to policy.
- R, --rmdir**
Remove empty directories according to policy.

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

Behavior options:

- O, --one-shot**
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**
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).

2.4 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`
- Create the database (one per filesystem) using the `mysqladmin` command:
`mysqladmin create <robinhood_db_name>`
- Connect to the database:
`mysql <robinhood_db_name>`

Then execute the following commands in the MySQL session:

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

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

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

3 Writing configuration file

3.1 Syntax

General structure

The configuration file consists of several blocks. Each of them is a set of key/value peers, Boolean expression or sub-blocks. Line breaks are not mandatory for separating expressions. Values may also be followed by options between brackets.

```
BLOCK_1 {
    Key = value
    Key = value(opt1, opt2)
    Key = value
    SUBBLOCK1 {
        Key=value
    }
}
BLOCK_2 {
    (Key > value)
    and
    ( key = value or key != value )
}
```

Type of values

A value can be:

- A string delimited by single or double quotes (' or ").
- A Boolean. 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 under a given path. Shell-like wildcards are allowed.
E.g: tree = "/tmp/subdir/*/dir1" matches entry "/tmp/subdir/foo/dir1/dir2/foo".
- **path:** entry exactly matches the path. Shell-like wildcards are allowed.
E.g: path = "/tmp/*/foo*" matches entry "/tmp/subdir/foo123".
- **name:** entry name matches the given regexp.
E.g: filename = "*.log" matches entry "/tmp/dir/foo/abc.log".
- **type:** entry has the given type (**directory**, **file**, **symlink**, **chr**, **blk**, **fifo** or **sock**).
E.g: type = "symlink".
- **owner:** entry has the given owner (owner name expected).
E.g: owner = "root".
- **group:** entry owns to the given group (group name expected).
- **size:** entry has the specified size. Value can be suffixed with KB, MB, GB...
E.g: size >= 100MB matches file whose size equals 100x1024x1024 bytes or more.
- **last_access:** condition based on the last access time to a file (for reading or writing). This is the difference between current time and max(atime, mtime, ctime). Value can be suffixed by 'sec', 'min', 'hour', 'day', 'week'...
E.g: last_access < 1h matches files that have been read or written within the last hour.
- **last_mod:** condition based on the last modification time to a file. This is the difference between current time and max(mtime, ctime).
E.g: last_mod > 1d matches files that have not been modified for more than a day.
- **dircount** (for directories only): the directory has the specified number of entries (except '.' and '..').
E.g: dircount > 10000 matches directories with more than 10 thousand child entries.
- **external_command:** custom script for testing if an entry matches. Must return 1 if entry matches, 0 else. Note: the special '{}' parameter is replaced by entry full path.
E.g: external_command = "/usr/bin/do_match {}".

Example of Boolean expression:

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

Comments

The '#' sign indicates the beginning of a comment (except if it is in a quoted string). The comment ends at the end of 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 following blocks are expected in a configuration file:

- **General** (mandatory): main robinhood parameters.
- **Log**: logging parameters (log files, log level...).
- **Purge_Policy**: defines purge policy.
- **Purge_Parameters**: options for purges.
- **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.
- **EntryProcessor**: configuration of entry processing pipeline (for FS scan).

Those blocks are described in the following sections.

3.2 Configuration template and default parameters

Template file

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

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

Default configuration values

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

```
robinhood --defaults
```

3.3 General parameters

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

The following parameters can be specified in this block:

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

3.4 Logging 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"`

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"`

3.5 Specifying purge policy

Basically, files are purged in the order of their last access. However, complex attributes-based policies can be specified: entries can be white-listed, or penalized depending on their path, size, owner, filename...

So, a purge policy consists of:

- Sets of entries that must never be purged (whitelist).
- Sets of entries that must be penalized (i.e. purged earlier).

A purge policy is also determined by:

- A condition for triggering purge
- A purge target condition
- A runtime interval

In configuration file, all those parameters are grouped in a '**Purge_Policy**' block that contains 3 types of sub-blocks: 'trigger', 'whitelist' and 'penalty' described bellow.

This block looks like this:

```
Purge_policy
{
    Trigger
    {
        # trigger definition
    }
    Trigger
    {
        # another trigger definition
    }
    ...
    Whitelist
    {
        # condition for whitelisting entries
        #(complex Boolean expression on file properties)
    }
    Whitelist
    {
        # another condition for whitelisting entries
    }
    ...
    Penalty
    {
        # Definition of a penalty for a given set of files
    }
    ...
}
```

3.5.1 Triggers

Triggers describe conditions for starting/stopping purges. Each trigger consists of:

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

Several triggers can be specified.

Type of condition

The type of condition is specified by “**trigger_on**” parameter.

Possible values are:

- **global_usage**: purge start/stop condition is based on the space 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).
- **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).
- **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 *<to be specified>*.

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 parameter “**check_interval**”. The value for this parameter can be suffixed by ‘sec’, ‘min’, ‘hour’, ‘day’, ‘week’, ‘year’... (the value is interpreted as seconds if no suffix is specified).

Examples

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

```
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%:

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

Daily check the space used by each of the users “foo” and “charlie”. Start purging their files if they use more than 1TB, until they use less than 800GB:

```
Trigger
{
    trigger_on = user_usage(foo, charlie)
    high_watermark_vol = 1TB
    low_water_mark_vol = 800GB
    check_interval = 1day
}
```

Run “/usr/local/bin/custom_script.sh” every 6 hours and purge data depending on the value returned by the command:

```
Trigger
{
    trigger_on = external_command("/usr/local/custom_script.ksh")
    check_interval = 6h
}
```

3.5.2 White-listed items

Whitelist block syntax

“Whitelist” sub-blocks make it possible for administrator to define entries that will never be purged. This block consists of a Boolean expression on file properties. If a file matches the expression, it is white-listed (see section 3.1 for more details about writing Boolean expressions on file attributes).

Examples

Whitelist all entries whose name matches “*.log” or that owns to “root”, and that are not located in “/dir/foo” tree.

```
Whitelist
{
    ( filename = "*.log" or owner = "root" )
    and not tree = "/dir/foo"
}
```

Do not purge entries accessed within the last hour, or copied-out today:

```
Whitelist
{
    last_access < 1h
    or
    last_copy < 1d
}
```

Do not purge files in working directories of running jobs (indicated by an external command):

```
Whitelist
{
    tree = "/tmp/batch_job_dirs"
    and
    External_command = "/usr/local/bin/is_job_running {}"
}
```

3.5.3 Penalty rules

Penalty rules can make sets of entries to be considered older than they really are, so they are purged earlier. Each rule is defined by:

- A set of entries impacted by the rule;
- An amount of penalty for this kind of entry.

Penalty rules are defined with “**penalty**” blocks.

Sets of penalized entries

A set is defined by a Boolean expression on entry attributes (see section 3.1 for more details about writing Boolean expressions on file attributes). This must be written as in “**Applies_to**” sub-block.

Amount of penalty

As entries are purged in the order of their last access, the penalty amount is a duration that is added to the actual last access. The related parameter name is “penalty_amount”.

This can be constant (E.g. `penalty_amount = 1week`) or it can be based on a file attribute (for now, only file size is supported). Variable penalties have the following syntax:

`<duration>/<size>` (e.g. `penalty_amount = 1h/100GB` make entries older of 1h per 100GB).

Note

If several penalties apply to the same file, its penalty is the max of them (not the sum).

Examples

This penalty rule penalizes log files located in “/tmp/logs” tree and bigger than 100MB, and it applies to them a penalty of 1h per MB.

```
Penalty
{
    Applies_to
    {
        tree = "/tmp/logs"
        and filename = "*.log"
        and size > 100MB
    }
    Penalty_amount = 1h/1MB
}
```

This second rule penalizes files that own to “charlie” and applies 1day of penalty to them:

```
Penalty
{
    Applies_to
    {
        owner = charlie
    }
    penalty_amount = 1d
}
```

3.6 Purge parameters

Purge parameters are specified in a ‘purge_parameters’ block.

The following options can be set:

- **simulation_mode** (Boolean): if this parameter is TRUE, entries are not really purged. The daemon however reports the purge operations it would have done.
E.g.: `simulation_mode = FALSE`
- **nb_threads_purge** (integer): this determine the number of purge operations that can be performed in parallel.
E.g.: `nb_threads_purge = 8`
- **purge_op_timeout** (duration): this specifies the timeout for purge operations. If a thread is stuck in a filesystem operation during this time, it is cancelled.
E.g.: `purge_op_timeout = 15min`
- **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.

3.7 Specifying 'rmdir' policy

'Rmdir' policy indicates what empty directories are to be removed, and when. It is defined by a 'Rmdir_policy' block.

This policy is simpler than purge policy: it is only based on the duration a directory has been empty:

- **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 also white-list directories that must not be removed, by specifying '**Whitelist**' sub-blocks.

3.8 'Rmdir' parameters

Empty directory removal parameters are specified in a 'rmdir_parameters' block.

The following options can be set:

- **simulation_mode** (Boolean): if this parameter is TRUE, empty directories are not really removed. The daemon however reports the 'rmdir' operations it would have done.
- **runtime_interval** (duration): interval for performing empty directory removal.
- **nb_threads_rmdir** (integer): this determine 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.

3.9 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).
 - **periodical(<nbr_transactions>)**: operations are packed in large transactions before they are committed. 'Commit' is done every *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 = periodical(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 or 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"`

3.10 Filesystem scan parameters

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

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

- **min_audit_period**: the frequency for scanning when filesystem is full;
- **max_audit_period**: the frequency for scanning when filesystem is empty

The interval between scans is computed according to this formula:

$\text{min} + (100 \times \text{current usage}) * (\text{max} - \text{min})$

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

3.11 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_CHECK_EXIST_threads_max** (integer): this limits the number of threads that simultaneously check if an entry already exists in database.
- **STAGE_GET_EXTRA_INFO_threads_max** (integer): this limits the number of threads that simultaneously retrieve extra information about filesystem entries (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/update entries in the database.

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

`STAGE_GET_EXTRA_INFO_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).

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

```
Alert
{
  type = directory
  and
  dircount > 10000
}
```

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

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

4 Reporting tool

4.1 Overview

The content of Robinhood's database can be very useful for building detailed reports about filesystem. For example, you can know how many entries of each type (directory, file, symlink...) exist in the filesystem, the min/max/average size of files, the min/max/average count of entries in directories, the space used by a given user, etc...

All those statistics can easily be retrieved using Robinhood reporting tool: **robinhood-report**.

4.2 Command line

Usage: robinhood-report [options]

Stats switches:

- a, --activity**
Display stats about daemon's activity.
- i, --fsinfo**
Display filesystem content statistics.
- u user, --userinfo [user]**
Display user statistics. Use optional parameter user for retrieving stats about a single user.
- g group, --groupinfo [group]**
Display group statistics. Use optional parameter group for retrieving stats about a single group.
- d count, --topdirs [count]**
Display largest directories. Optional argument indicates the number of directories to be returned (default: 20).
- s count, --topsize [count]**

- Display largest files. Optional argument indicates the number of files to be returned (default: 20).
- p** count, **--toppurge** [count]
Display oldest entries eligible for purge. Optional argument indicates the number of entries to be returned (default: 20).
 - r** count, **--toprmdir** [count]
Display oldest empty directories eligible for rmdir. Optional argument indicates the number of dirs to be returned (default: 20).
 - U** count, **--topusers** [count]
Display largest disk space consumers. Optional argument indicate the number of users to be returned (default: 20).

Config file options:

- f** file, **--config-file**=file
Specifies path to configuration file.

Output format options:

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

Miscellaneous options:

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

4.3 Reports

This command can provide the following reports:

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

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

Example of output:

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

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

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

User info report (--userinfo option)

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

Example of output:

```
User:                root

  Type:              directory
  Count:             8426
  Space used:        34.29 MB   (70232 blks)
  Dircount min:      0
  Dircount max:      4525
  Dircount avg:      13

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

Group info report (--groupinfo option)

Same thing as 'userinfo', by for each group (or the specified one).

Top directories (--topdirs option)

This option displays directories with the highest number of child entries.

Useful information is given for each of them: path, owner, number of entries, last modification time.

Example of output:

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

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

...
```

Top file size (--topsize option)

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

Example of output:

```
Rank:                1
Path:                /ptmp/group1/toto/opt/lib/gcj-4.3.2-9/classmap.db
Size:                69.57 GB   (74700554240 bytes)
```

```

Last access:      2009/01/13 07:24:56
Last modification: 2009/01/13 07:22:04
Owner/Group:     toto/group1
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 purged first, if disk space is needed. Also, this command gives an overview of oldest entries of filesystem. Note that this is only an estimation, and those entries may not be purged if they have been moved or accessed since they were scanned. This returns entry path and type, last access and modification time, the penalty applied to the entry (if it matches a penalty rule in purge policy), and storage information (size, blocks, stripe info...).

```

Rank:            1
Path:            /ptmp/vm/benchs/bonnie+-1.03a/zcav.8
Type:            file
Last access:     2009/01/13 08:26:31
Last modification: 2009/01/13 08:26:31
Penalty:         1.0d
Size:            2.20 KB   (2253 bytes)
Space used:     4.00 KB   (8 blocks)
Stripe count:    2
Stripe size:     4.00 MB   (4194304 bytes)
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
Penalty:         1.0d
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 'rmdir' candidates (--toprmdir option)

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

Example of output:

```

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

```

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

Top disk space consumers (--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 filesystem scan, last purge...

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

Storage unit usage max:  55.33%

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

5 System and database tunings

5.1 Lustre tunings and workarounds

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

- If your system "Oops" in statahead, disable this feature:
`echo 0 > /proc/fs/lustre/llite/*/statahead_max`
- CPU overload and client performance drop when free memory is low (bug #17282):
in this case, `lru_size` must be set at `CPU_count * 100`:
`lctl set_param ldlm.namespaces.*.lru_size=800`

5.2 Linux kernel tunings

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

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

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

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

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

E.g:

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

5.3 Database tunings

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

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

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

Test bed 1:

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

With the following parameters for MySQL:

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

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

We get the following resource usage:

For purge action only:

- mysqld uses 560MB
- robinhood uses 300MB

For all actions (scan + purge + rmdir) performed simultaneously:

- mysqld uses 650MB
- robinhood uses 500MB

6 Known bugs

- **Process terminates with SEGFault when MySQL server restarts**
 - Cause:
This is due to a bad resilience of MySQL client API to server crash when using prepared statements. This is known as bug #33384 in MySQL tracker (check current bug status here: <http://bugs.mysql.com/bug.php?id=33384>).
 - Workaround:
Disable prepared statements in Robinhood: to do so, use '--disable-prep-stmts' argument to ./configure before building the RPM.