Robinhood PolicyEngine

First steps (tutorial)

Temporary Filesystem Manager

v2.5.2

May 21st, 2014

Table of contents

The purpose of this document is to guide you for the first steps of Robinhood installation and configuration. We will first run a very simple instance of Robinhood for statistics and monitoring purpose only. Then we will configure it for purging files when disk space is missing. Finally we will deal with more advanced usage, by defining filesets and associating different purge policies to them.

1. Installation

1.1. Robinhood

1.1.1. Install from RPM

Pre-generated RPMs can be downloaded on sourceforge, for the following configurations:
- x86_64 architecture , RedHat 5/6 Linux families
- MySQL database 5.x
- Posix filesystems, Lustre 1.8, 2.1, 2.2, 2.3, 2.4, 2.5

Purpose specific RPM: `robinhood-tmpfs`
It must be installed on a filesystem client.
It includes:
- 'robinhood' daemon
- Reporting and other commands: 'rbh-report', 'rbh-find', 'rbh-du' and 'rbh-diff'
- configuration templates
- /etc/init.d/robinhood init script

admin RPM (all purposes): `robinhood-adm`
Includes 'rbh-config' configuration helper. It is helpful on the DB host and Lustre MDS.

`robinhood-webgui` RPM installs a web interface to visualize stats from Robinhood database.
It must be installed on a HTTP server.
See section 6 for more details about this interface.

1.1.2. Build and install from the source tarball

1.1.2.a. Requirements

Before building Robinhood, make sure the following packages are installed on your system:
- mysql-devel
- lustre API library (if Robinhood is to be run on a Lustre filesystem): '/usr/include/liblustreapi.h' and '/usr/lib*/liblustreapi.a' are installed by lustre rpm.

1.1.2.b. Build

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

Unzip and untar the sources:

```
tar zxvf  robinhood-2.5.2.tar.gz
cd robinhood-2.5.2
```

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

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

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

Other '`./configure`' options:

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

Finally, build the RPMs:

```
make rpm
```

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

1.2. MySQL database

Robinhood needs a MySQL database for storing its data. This database can run on a different machine than the Robinhood commands.

1.2.1. Requirements

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

Start the database engine:

```
service mysqld start
```

1.2.2. Creating database

With the helper script:

To easily create robinhood database, you can use the 'rbh-config' script. Run this script on the database host to check your system configuration and perform database creation steps:

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

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

Note: if no option is given to `rbh-config`, it prompts for configuration parameters interactively. Else, if you specify parameters on command line, it runs in batch mode.

Write the database password to a file with restricted access (root/600),
e.g. `/etc/robinhood.d/.dbpassword`

or manually:

Alternatively, if you want a better control on the database configuration and access rights, you can perform the following steps of your own:

- 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:

  - `GRANT USAGE ON robinhood_db_name.* TO 'robinhood'@'%'`
    `identified by 'password';`
  - `GRANT ALL PRIVILEGES ON robinhood_db_name.* TO 'robinhood'@'%'`
    `identified by 'password';`
  - The 'super' privilege is required for creating DB triggers (needed for accounting optimizations):
    `GRANT SUPER ON *.* TO 'robinhood'@'%' IDENTIFIED BY 'password'`
    `;`
  - Refresh server access settings:
    `FLUSH PRIVILEGES ;`

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

- For testing access to database, execute the following command on the machine where robinhood will be running :
  `mysql --user=robinhood --password=password --host=db_host`
  `robinhood_db_name`

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

At this time, the database schema is empty. Robinhood will automatically create it the first time it is launched.

2. Run your first Robinhood instance

Let's begin with a simple case: we want to monitor the content of `/tmp` ext3 filesystem, by scanning it periodically.

2.1. Configuration file

We first need to write a very basic configuration file:

```
General
{
    fs_path = "/tmp";
    fs_type = ext3;
}
Log
{
    log_file       = "/var/log/robinhood/tmp_fs.log";
    report_file    = "/var/log/robinhood/reports.log";
    alert_file     = "/var/log/robinhood/alerts.log";
}

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

General section:
'fs_path' is the mount point of the filesystem we want to monitor.
'fs_type' is the type of filesystem (as returned by mount). This parameter is used for sanity checks.

Log section:
Make sure the log directory exists.
Note1: you can also specify special values 'stderr', 'stdout' or 'syslog' for log parameters.
Note2: robinhood is compliant with log rotation (if its log file is renamed, it will automatically switch to a new log file).

ListManager::MySQL section:
This section is for configuring database access.
Set the host name of the database server (*server* parameter), the database name (*db* parameter), the database user (*user* parameter) and specify a file where you wrote the password for connecting to the database (*password_file* parameter).
/!\ Make sure the password file cannot be read by any user, by setting it a '600' mode for example.
If you don't care about security, you can directly specify the password in the configuration file, by setting the *password* parameter.
E.g.: `password = 'passw0rd' ;`

2.2. Running robinhood

For this first example, we just want to scan the filesystem once and exit, so we can get stats about current content of /tmp.

Thus, we are going to run robinhood command with the '--scan' option, and the '--once' option so it will exit when the scan is finished.
You can specify the configuration file using the '–f' option, else it will use the first file in directory '`/etc/robinhood.d/tmpfs`'.
You can use a short name for a config file located in '`/etc/robinhood.d/tmpfs`', e.g. '`test`' for '`/etc/robinhood.d/tmpfs/test.conf`'.
If you want to override configuration values for log file, use the '–L' option. For example, let's use '-L stdout'

```
robinhood –f test –L stdout --scan ––once
```
or just:
```
robinhood –L stdout --scan ––once
```
(if your config file is the only one in /etc/robinhood.d/tmpfs)

You should get something like this:

```
2009/07/17 13:49:06: FS Scan | Starting scan of /tmp
2009/07/17 13:49:06: FS Scan | Full scan of /tmp completed, 7130 entries
found. Duration = 0.07s
2009/07/17 13:49:06: FS Scan | File list of /tmp has been updated
2009/07/17 13:49:06: Main | All tasks done! Exiting.
```

2.3. Getting stats on filesystem content

2.3.1. rbh-report

Now we performed a scan, we can get stats about users, files, directories, etc. using `rbh-report`:

- Get stats for a user: -u option

`rbh-report –u foo`

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

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

- Split user's usage per group: -S option

`rbh-report –u bar –S`

```
user      ,      group,     type,      count,   spc_used,   avg_size
bar       ,      proj1,     file,          4,    40.00 MB,   10.00 MB
bar       ,      proj2,     file,       3296,   947.80 MB,  273.30 KB
bar       ,      proj3,     file,     259781,   781.21 GB,    3.08 MB
```

- Get largest files: --top-size option

`rbh-report ––top-size`

```
rank,          path,      size, user, group,         last_access,            last_mod,  purge class
```

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

- Get top space consumers: --top-users option

  rbh-report –-top-users

```
rank, user      ,   spc_used,      count,  avg_size
   1, usr0021   ,   11.14 TB,     116396,  100.34 MB
   2, usr3562   ,    5.54 TB,        575,    9.86 GB
   3, usr2189   ,    5.52 TB,       9888,  585.50 MB
   4, usr2672   ,    3.21 TB,     238016,   14.49 MB
   5, usr7267   ,    2.09 TB,       8230,  266.17 MB
...
```

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

- Filesystem content summary: -i option

  rbh-report –i

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

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

- fileclasses summary: --class-info option

  rbh-report --class-info

```
Class:        important_files
Nb entries:               879
Space used:          9.00 GB    (18874368 blks)
Size min:                 0     (0 bytes)
Size max:             1.00 GB    (19327352832 bytes)
Size avg:            10.48 MB    (10993664 bytes)

Class:              project_B
Nb entries:               878
Space used:          2.23 GB    (4666481 blks)
Size min:                 0     (0 bytes)
Size max:            95.37 MB    (100000000 bytes)
Size avg:             2.60 MB    (2727157 bytes)
```

- [new 2.5] entry information: -e option

  rbh-report -e /mnt/lustre/dir1/file.1

```
id             :      [0x200000400:0x16a94:0x0]
parent_id      :      [0x200000007:0x1:0x0]
name           :      file.1
path updt      :      2013/10/30 10:25:30
path           :      /mnt/lustre/dir1/file.1
depth          :      1
```

```
user           :      root
group          :      root
size           :      1.42 MB
spc_used       :      1.42 MB
creation       :      2013/10/30 10:07:17
last_access    :      2013/10/30 10:15:28
last_mod       :      2013/10/30 10:10:52
type           :      file
mode           :      rw-r--r--
nlink          :      1
md updt        :      2013/10/30 10:25:30
stripe_cnt, stripe_size,      pool:           2,     1.00 MB,
stripes        :      ost#1: 30515, ost#0: 30520
```

- …and more:
  you can also generate reports, or dump files per directory, per OST, etc…
  To get more details about available reports, run '`rbh-report --help`'.

2.3.2. rbh-find

'find' clone using robinhood database.
Example:

```
rbh-find /tmp/dir -u root -size +32M -mtime +1h -ls
```

2.3.3. rbh-du

'du' clone using robinhood database.
It provides extra features like filtering on a given user, group or type…
Example:

```
rbh-du –H -u foo /tmp

45.0G   /tmp
```

3. Scan options and alerts

In the first example, we have used Robinhood as a 'one-shot' command. It can also be used as a daemon that will periodically scan the filesystem for updating its database. You can also configure alerts when entries in the filesystem match a condition you specified.

Let's configure Robinhood to scan the filesystem once a day.
Add a FS_Scan section to the config file:

```
FS_Scan
{
    scan_interval      =   1d ;
    nb_threads_scan    =   2 ;

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

```
    Ignore
    {
        # ignore the content of /tmp/dir1
        tree == "/tmp/dir1"
    }
}
```

For a daily scan, we set `scan_interval` to 1 day (`1d`).

Robinhood uses a parallel algorithm for scanning large filesystems efficiently. You can set the number of threads used for scanning using the `nb_threads_scan` parameter.

You may need to ignore some parts of the filesystem (like .snapshot dirs etc…). For this, you can use a "ignore" sub-block, like in the example above, and specify complex Boolean expressions on file properties (refer to Robinhood admin guide for more details about available attributes).

To create alerts about filesystem entries, add an EntryProcessor section to configuration file:

```
EntryProcessor
{
    Alert Large_Directory
    {
        type == directory
        and
        dircount > 10000
        and
        last_mod < 1d
    }

    # Raise alerts for large files
    Alert Large_file
    {
        type == file
        and
        size > 100GB
        and
        last_mod < 1d
    }
}
```

In this example, we want to raise alerts for directories with more that 10.000 entries and file larger than 100GB.
Note that these rules are matched at each scan. If you don't want to be alerted about a given file at every scan, it is advised to specify a condition on `last_mod`, so you will only be alerted for recently modified entries.

You can now specify a name for alerts, so you can easily identify the issue in the alert title (in e-mail title).
Those alerts are written to the `alert_file` specified in Log section. They can also be sent by mail by specifying a mail address in the `alert_mail` parameter of Log section.
You can now batch alert e-mails as a single summary (instead of sending 1 mail per file). This is driven by the `batch_alert_max` parameter of Log section (0=no batch limit, 1=no batching).

Now, let's run Robinhood as a daemon. Use the '--detach' option to start it in background and detach it from your terminal:

```
robinhood --scan --detach
```

4. Resource monitoring and purges

One of the main purposes of robinhood is to monitor disk space usage and trigger purges when disk space runs low.
Purge triggering is based on high/low thresholds and files removal is based on a LRU list: when the used space reaches a high threshold, Robinhood will build a list of least recently accessed files (from its database) and purge them until the disk space is back to the low threshold.

4.1. Defining purge triggers

First of all, let's specify thresholds for triggering purge.
This can be done on several criteria:

- Global filesystem usage
- OST usage (for Lustre filesystems)
- User or group usage (quota-like purge)
- Periodic purge (whatever the filesystem usage is)

Write the following block to configuration file for triggering purge on global filesystem usage:

```
Purge_Trigger
{
    trigger_on         = global_usage ;
    high_threshold_pct = 90% ;
    low_threshold_pct  = 85% ;
    check_interval     = 5min ;
}
```

trigger_on specifies the type of trigger.
high_threshold_pct indicates the disk usage that must be reached for starting purge.
low_threshold_pct indicates the disk usage that must be reached for stopping purge.
check_interval is the interval for checking disk usage.

We can also do the same for users, by specifying a kind of "quota":

```
Purge_Trigger
{
    trigger_on         = user_usage ;
    high_threshold_vol = 10GB ;
    low_threshold_vol  = 9GB ;
    check_interval     = 12h ;
}
```

Every 12h, the daemon will check the space used by users. If a user uses more than 10GB, his files will be purged from the least recently accessed until the space he uses decrease to 9GB.

You can also specify quotas on inode count using ‘high_threshokld_cnt’ and ‘low_threshold_cnt’:

```
Purge_Trigger
{
    trigger_on         = user_usage ;
    high_threshold_cnt = 1M ; # i.e: user inode usage > 1 million
    low_threshold_cnt  = 900k ;
    check_interval     = 1h ;
}
```

A list of users (separated by coma) can also be specified for quotas (wildcards are allowed), e.g:

```
Purge_Trigger {
    trigger_on         = user_usage(foo,charlie,project*) ;
    …
}
```

Periodic trigger periodically purge all entries that match purge policies (no threshold is expected in this case):

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

To receive a mail notification each time a high threshold (or quota) is reached, add this parameter to a trigger:

```
      alert_high = true ;
```

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

```
      alert_low = false ;
```

4.2. Minimal purge policy

Robinhood makes it possible to define different purge policies for several file classes.
In this example, we will only define a single purge policy for all files.
This can be done in a ‘Purge_Policies’ section of config file:

```
Purge_policies {
      Policy default    {
            Condition { last_access > 1h }
      }
}
```

‘default’ policy is a special policy that applies to files that are not in a file class.
In a policy, you must specify a condition for allowing entries to be purged. In this example, we don’t want recently accessed entries (read or written within the last hour) to be purged.

4.3. Space monitoring and purges

Robinhood is now able to monitor disk usage and purge entries when needed.
Start the daemon with the '--purge' option. If your are not sure of your configuration, you can specify the '--dry-run' option, so it won't really purge files.

```
robinhood --purge --detach –-dry-run
```

You will get something like this in the log file:

```
Main | Resource Monitor successfully initialized
ResMonitor | Filesystem usage: 92.45% (786901 blocks) / high threshold:
90.00% (764426 blocks)
ResMonitor | 34465 blocks (x512) must be purged on Filesystem (used=786901,
target=734426, block size=4096)
Purge | Building a purge list from last full FS Scan: 2009/07/17 13:49:06
Purge | Starting purge on global filesystem
ResMonitor | Global filesystem purge summary: 34465 blocks purged/34465
blocks needed in /tmp
```

The list of purged files is written in the report file.

Note 1: you can use the same daemon for performing scans periodically and monitoring resource, by combining options on command line.
E.g.: `robinhood --purge --scan –-detach`
By default, if you start Robinhood with no parameter, it will run all configured actions.
Note 2: if you do not want to have a daemon on your system, you can perform resource monitoring with a 'one-shot' command that you can launch in cron, for example:
`robinhood --purge --once`
It will check the disk usage: it will exit immediately if disk usage does not exceed the high threshold; else, it will purge entries.

Only check thresholds/triggers without purging data:
if robinhood is started with the '`--check-thresholds`' option instead of '`--purge`', it will only check for trigger conditions without purging data.

4.4. Purge parameters

You can add a "Purge_parameters" block to the configuration file, to have a better control on purges:

```
Purge_Parameters
{
    nb_threads_purge      = 4;
    post_purge_df_latency = 1min;
    db_result_size_max    = 10000;
    recheck_ignored_classes = true;
}
```

Purge actions are performed in parallel. You can specify the number of purge threads by setting the *nb_threads_purge* parameter.
On filesystems where releasing data is asynchronous, the 'df' command may take a few minutes before returning an up-to-date value after purging a lot of files. Thus, Robinhood must wait before checking disk usage again after a purge. This is driven by the *post_purge_df_latency* parameter.

If purge policies application looks too slow, you can speed-up it by disabling *recheck_ignored_classes*. This with result in not rechecking previously ignored entries when applying a policy. It is recommended re-enable it after you changed policy definitions.

4.5. Using file classes

Robinhood makes it possible to apply different 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 section of the tutorial, we will define 3 classes and apply different policies to them:

- We don't want "*.log" files that belong to 'root' to be purged;
- We want to keep files from directory /tmp/A to be kept longer on disk than files from other directories.

First, we need to define those file classes, in a 'filesets' section of the configuration file. We associate a custom name to each FileClass, and specify the definition of the class:

```
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 classes 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
{
      # whitelist log files of 'root'
      ignore_fileclass = root_log_file;

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

      # The 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 in 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 used;
- If you modify a fileclass definition or target fileclasses of policies, you need to reset fileclass information in database. To do so, run this command:
  `rbh-config reset_classes`
- 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”
   }

   …
```

FileClasses can be defined as the union or the intersection of other FileClasses. To do so, use the special keywords “union”, “inter” and “not” in the fileclass definition:

```
FileClass root_log_A
{
        Definition {
                (root_log_files inter A_files)
                union (not B_files)
        }
}
```

5. Directory removal

Purge after purge, there will be more and more empty directories in the filesystem namespace. Robinhood provides a mechanism for removing directories that have been empty for a long time.
It is also possible to recursively remove entire parts of the filesystem (“rm –rf”), by specifying a condition on the top-level directory.

Directory removal is driven by a ‘rmdir_policy’ section in the configuration file:

- The condition for removing empty directories is given by the ‘age_rm_empty_dirs’ parameter, that indicates the duration after which an empty directory is removed.
- You can specify one or several ‘ignore’ condition for directories you never want to be removed.

- Use ‘recursive_rmdir’ sub-blocks to indicate what matching directories are to be removed recursively. /!\The whole directories content is removed without checking policies on their content (whitelist rules…).

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

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

      # whitelist directories matching the following condition
      ignore
      {
            depth < 2
            or
            owner == ‘foo’
            or
            tree == /fs/subdir/A
      }
}
```

You can specify advanced parameters by writing an ‘rmdir_parameters’ block:

```
rmdir_parameters
{
    # Interval for performing empty directory removal
    runtime_interval = 12h ;

    # Number of threads for performing rmdir operations
    nb_threads_rmdir = 4 ;
}
```

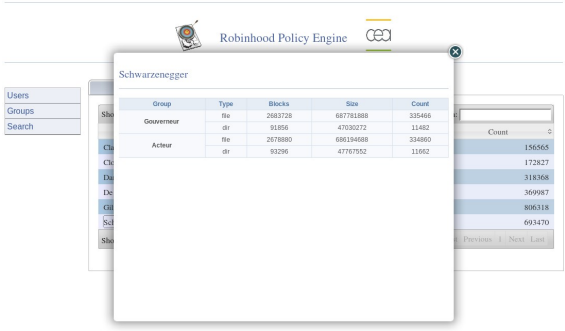
In this section, you can also specify the period for checking empty directories (*runtime_interval* parameter).

For running empty directory removal, start robinhood with the ‘`--rmdir`’ option:

```
robinhood –f /etc/robinhood.d/test.conf –rmdir
```

6. Setting up web interface

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



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

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

The following parameter must be set in httpd.conf:

```
AllowOverride All
```

Install robinhood interface:

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

or

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

Configuration:
In a web browser, enter the robinhood URL:
http://*yourserver*/robinhood
The first time you connect to this address, fill-in database parameters (host, login, password, …):

Those parameters are saved in:
`/var/www/http/robinhood/app/config/database.xml`

That’s done. You can enjoy statistics charts.

7. Robinhood on Lustre

Robinhood provides special features for Lustre filesystems:

7.1. Purge triggers on OST usage

Robinhood can monitor OST usage independently, and trigger purges only for the files of a given OST when it exceeds a threshold.

E.g.:

```
Purge_Trigger
{
      trigger_on = OST_usage ;
      high_threshold_pct = 85% ;
      low_threshold_pct = 80% ;
      check_interval = 5min ;
}
```

7.2. File classes and conditions on OST indexes

On Lustre filesystem, you can specify policy conditions and fileclasses based on ost indexes.

E.g.:

```
Filesets
{
      FileClass retired_ost
      {
            ost_index == 1 or ost_index == 4 or ost_index == 7
            and ost_index != 9
      }
}
```

"ost_index == *N*" is true if at least one part of the file is stored in OST index #*N*.
"ost_index != *N*" is true if no part of the file is stored in OST index #*N*.

(Note: '<' and '>' comparators are not allowed for this condition).

7.3. File classes/conditions on OST pool names

In Lustre 1.8 release and later, you can use OST pool names for specifying file classes.

E.g.:

```
Filesets
{
      FileClass da1x_storage
      {
            ost_pool == "da1*"
      }
}
```

7.4. Use Lustre changelogs instead of scanning the filesystem with Lustre 2!

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

If filesystem MDS and MGS are on the same host, you can simply enable this feature by running 'rbh-config' on this host (for other cases, see *Robinhood admin guide*):

```
rbh-config enable_chglogs
```

Then, set related information into robinhood's configuration:

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

By default, on Lustre v2, robinhood performs event handling when it is started with no option (instead of scanning).
You can start a robinhood instance for reading events only, using the '`--read-log`' option.

robinhood also maintains changelog record stats. You can get them using 'rbh-report –a':

```
Changelog stats:

        Last read record id:      71812545
        Last record read time:    2012/02/08 16:00:14
        Last committed record id: 71812545
        Changelog stats:
                 type            total  (diff)  (rate)
                 MARK:          135406  (+282)   (0.38/sec)
                CREAT:        16911356  (+824)   (0.91/sec)
                MKDIR:          629730
                HLINK:          128048
                SLINK:          601645  (+117)   (0.12/sec)
                MKNOD:               0
                UNLNK:        11150277  (+472)   (0.55/sec)
                RMDIR:          411672
                RNMFM:         1566093
                RNMTO:         1572679  (+268)   (0.43/sec)
                 OPEN:               0
                CLOSE:               0
                IOCTL:               0
                TRUNC:               0
                SATTR:         6674152  (+241)   (0.40/sec)
                XATTR:               0
                  HSM:               0
                MTIME:         8201315  (+1930)  (2.21/sec)
                CTIME:               0
                ATIME:               0
```

7.5. rbh-find '-ost' and '-pool' options

'rbh-find' provides specific options for Lustre:

- '-ost <idx>' filter entries in the specified OST
- '-pool <pool_name>' filter entries in the given pool name

Examples:

```
rbh-find /lustre/project1 -u foo -size +32M -mtime +1h -ost 2 -ls
rbh-find /lustre/ -pool fastdisk -ls
```

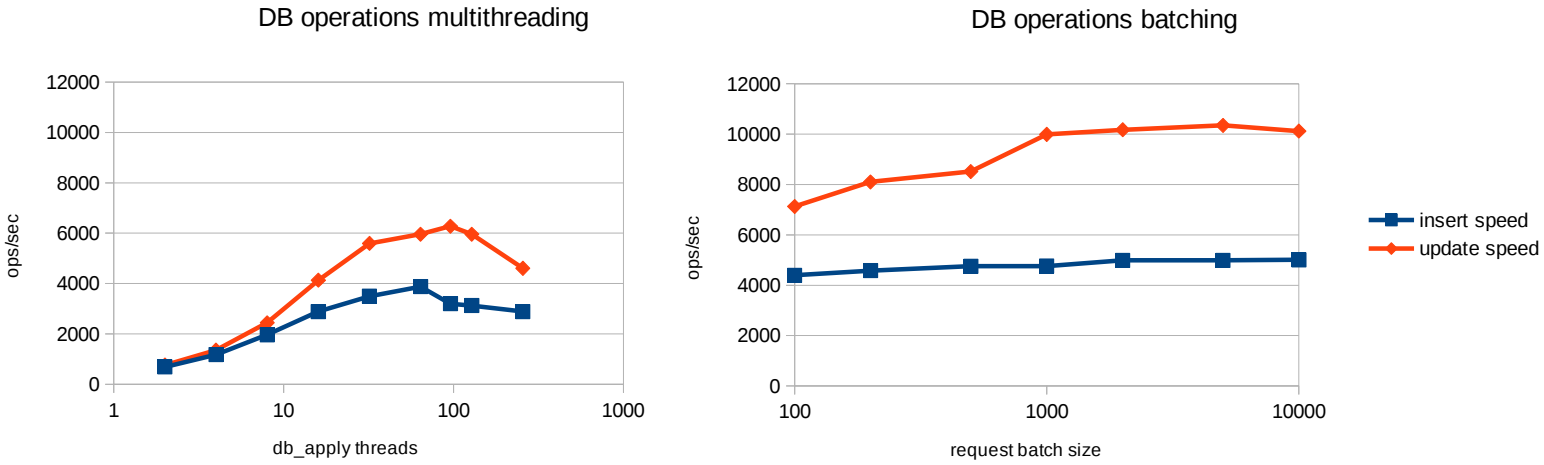
8. Optimizations and compatibility

8.1. [new 2.5] Performance strategy for DB operations

You have the choice between 2 strategies to maximize robinhood processing speed:

- *multi-threading*: perform multiple DB operations in parallel as independent transactions.
- *batching*: batch database operations (insert, update...) into a single transaction, which minimize the need for IOPS on the database backend. Batches are not executed in parallel as they may results in DB deadlocks.

The following benchmark evaluates the DB performance for each strategy. This benchmark ran on a simple test-bed using basic SATA disk as DB storage for innodb.



The results show that batching is more efficient than multi-threading whatever the thread count, so it has been made the defaut behavior for robinhood 2.5.

You can control batches size defining this parameter in the *EntryProcessor* configuration block (see section 3):

- **[new 2.5] max_batch_size** (positive integer): by default, the entry processor tries to batch similar database operations to speed them. This can be controlled by the max_batch_size parameter. The default max batch size is 1000.

If your DB storage backend is efficient enough (high IOPS) you may want to give a try to the other strategy. To switch from *batching* strategy to *multi-threading*, set **max_batch_size = 1**. This will automatically disable batching and enables multi-threading for DB operations. Consider increasing **nb_threads** parameter in this case (*EntryProcessor* configuration block):

- **nb_threads** (integer): total number of threads for performing pipeline tasks. Default is 4. Consider increasing it if you disable *batching*.

8.2. Database tunings

You can modify those parameters in `/etc/my.cnf` to speed-up database requests:
(tuning `innodb_buffer_pool_size` is strongly recommended)

```
innodb_file_per_table

# 50% to 90% of the physical memory
innodb_buffer_pool_size=16G

# 2*nbr_cpu_cores
innodb_thread_concurrency=32
```

```
# memory cache tuning
innodb_max_dirty_pages_pct=15

# robinhood is massively multithreaded: set enough connections
# for its threads, and its multiple instances
max_connections=256

# If you get DB connection failures, increase this parameter:
connect_timeout=60

# This parameter appears to have a significant impact on performances:
# see this article to tune it appropriately:
#http://www.mysqlperformanceblog.com/2008/11/21/how-to-calculate-a-good-innodb-log-file-size
innodb_log_file_size=500M
```

To manage transactions efficiently, innodb needs a storage backend with high IOPS performances. You can monitor you disk stress by running "`sar -d`" on your DB storage device: if '%util' field is close to 100%, your database rate is limited by disk IOPS.
In this case you have the choice between these 2 solutions, depending on how critical is your robinhood DB content:

- Safe (needs specific hardware): put your DB on a SSD device, or use a write-back capable storage that is protected against power-failures. In this case, no DB operation can be lost.
- Cheap (and unsafe): add this tuning to `/etc/my.cnf`:
  `innodb_flush_log_at_trx_commit=2`
  This results in flushing transactions to disk only every second, which dramatically reduce the required IO rate. The risk is to loose the last second of recorded information if the DB host crashes. This is affordable if you use to scan your filesystem (the missing information will be added to the database during the next scan). If you read Lustre changelogs, then you will need to scan your filesystem after a DB server failure.

This little script is also very convenient to analyze your database performance and it often suggests relevant tunings:

http://mysqltuner.pl

8.3. Optimize scanning vs reporting speed

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

8.4. SLES init depedencies

On SLES systems, the default dependency for boot scheduling is on "mysql" service. However, in many cases, it should 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>
```

8.5. Lustre troubles

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

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

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

9. Getting more information

In-line help
‘--help’ options of ‘robinhood’ and ‘rbh-report’ commands provide you with detailed descriptions of command-line parameters.

Admin guide
For more details about robinhood configuration, you can refer to the admin guide.
It is usually located in the `doc/admin_guides` directory of the distribution tarball.

Project website and wiki
You can get extra information on the website, especially in the wiki: http://robinhood.sf.net

Mailing list

- You can get support by posting your issue to robinhood-support mailing list: robinhood-support@lists.sourceforge.net

- To keep updated about new versions of robinhood, subscribe to: robinhood-news@lists.sourceforge.net

- Robinhood is Open Source. To participate to its development, join: robinhood-devel@lists.sourceforge.net