Rally Documentation

Release 0.6.2

Daniel Mitterdorfer

Jul 25, 2017

Getting Started with Rally

You want to benchmark Elasticsearch? Then Rally is for you. It can help you with the following tasks:

- Setup and teardown of an Elasticsearch cluster for benchmarking
- Management of benchmark data and specifications even across Elasticsearch versions
- Running benchmarks and recording results
- Finding performance problems by attaching so-called telemetry devices
- Comparing performance results

We have also put considerable effort in Rally to ensure that benchmarking data are reproducible.

CHAPTER 1

Getting Help or Contributing to Rally

Use our Discuss forum to provide feedback or ask questions about Rally. Please see our contribution guide on guidelines for contributors.

CHAPTER 2

Source Code

Rally's source code is available on Github. You can also check the changelog and the roadmap there.

Quickstart

Install

Install Python 3.4+ including `pip3`, JDK 8 and git 1.9+. Then run the following command, optionally prefixed by `sudo` if necessary:

```
pip3 install esrally
```

If you have any trouble or need more detailed instructions, please look in the *detailed installation guide*.

Configure

Just invoke `esrally configure`.

For more detailed instructions and a detailed walkthrough see the *configuration guide*.

Run your first race

Now we're ready to run our first race:

```
esrally --distribution-version=5.0.0
```

This will download Elasticsearch 5.0.0 and run Rally's default track against it. After the race, a *summary report* is written to the command line::

```
-------------------------------------------------------
    ________                __   ______
   / ____(_)___  _____ _/ /  / ___/_________  _________
  / /_  / / __ \/ __ `/ /   \__ \/ ___/ __ \/ ___/ _ \
 / __/ / / / / / /_/ / /   ___/ / /__/ /_/ / /  /  __/
/_/   /_/_/ /_/\__,_/_/   /_____/\___/\____/_/   \___/
-------------------------------------------------------

|                         Metric |            Operation |     Value |   Unit |
|--------------------------------:|---------------------:|----------:|-------:|
|                  Indexing time |                      |   28.0997 |    min |
|                     Merge time |                      |   6.84378 |    min |
|                   Refresh time |                      |   3.06045 |    min |
|                     Flush time |                      |  0.106517 |    min |
|            Merge throttle time |                      |   1.28193 |    min |
|               Median CPU usage |                      |     471.6 |      % |
|             Total Young Gen GC |                      |    16.237 |      s |
|               Total Old Gen GC |                      |     1.796 |      s |
|                     Index size |                      |   2.60124 |     GB |
|                Totally written |                      |   11.8144 |     GB |
|         Heap used for segments |                      |   14.7326 |     MB |
|       Heap used for doc values |                      |  0.115917 |     MB |
|            Heap used for terms |                      |   13.3203 |     MB |
|            Heap used for norms |                      | 0.0734253 |     MB |
|           Heap used for points |                      |    0.5793 |     MB |
|    Heap used for stored fields |                      |  0.643608 |     MB |
|                  Segment count |                      |        97 |        |
|                Min Throughput |         index-append |   31925.2 | docs/s |
|             Median Throughput |         index-append |   39137.5 | docs/s |
|                Max Throughput |         index-append |   39633.6 | docs/s |
|      50.0th percentile latency |         index-append |   872.513 |     ms |
|      90.0th percentile latency |         index-append |   1457.13 |     ms |
|      99.0th percentile latency |         index-append |   1874.89 |     ms |
|       100th percentile latency |         index-append |   2711.71 |     ms |
| 50.0th percentile service time |         index-append |   872.513 |     ms |
| 90.0th percentile service time |         index-append |   1457.13 |     ms |
| 99.0th percentile service time |         index-append |   1874.89 |     ms |
|  100th percentile service time |         index-append |   2711.71 |     ms |
|                            ... |                  ... |       ... |    ... |
|                            ... |                  ... |       ... |    ... |
|                Min Throughput |     painless_dynamic |   2.53292 |  ops/s |
|             Median Throughput |     painless_dynamic |   2.53813 |  ops/s |
|                Max Throughput |     painless_dynamic |   2.54401 |  ops/s |
|      50.0th percentile latency |     painless_dynamic |    172208 |     ms |
|      90.0th percentile latency |     painless_dynamic |    310401 |     ms |
|      99.0th percentile latency |     painless_dynamic |    341341 |     ms |
|      99.9th percentile latency |     painless_dynamic |    344404 |     ms |
|       100th percentile latency |     painless_dynamic |    344754 |     ms |
| 50.0th percentile service time |     painless_dynamic |    393.02 |     ms |
| 90.0th percentile service time |     painless_dynamic |   407.579 |     ms |
| 99.0th percentile service time |     painless_dynamic |   430.806 |     ms |
| 99.9th percentile service time |     painless_dynamic |   457.352 |     ms |
|  100th percentile service time |     painless_dynamic |   459.474 |     ms |

----------------------------------
[INFO] SUCCESS (took 2634 seconds)
----------------------------------
```

Next steps

Now you can check *how to run benchmarks*, get a better understanding how to interpret the numbers in the *summary report* or start to *create your own tracks*. Be sure to check also the available *recipes* to help you understand how to solve specific problems in Rally.

Also run `esrally --help` to see what options are available and keep the *command line reference* handy for more detailed explanations of each option.

Installation

This is the detailed installation guide for Rally. If you are in a hurry you can check the *quickstart guide*.

Prerequisites

Before installing Rally, please ensure that the following packages are installed:

- Python 3.4 or better available as *python3* on the path (verify with: `python3 --version` which should print `Python 3.4.0` or higher)
- Python3 dev (for header files) (`python3-dev` package)
- `pip3` available on the path (verify with `pip3 --version`)
- JDK 8
- git 1.9 or better

Rally does not support Windows and is only actively tested on Mac OS X and Linux.

Note: If you use RHEL, please ensure to install a recent version of git via the Red Hat Software Collections.

Installing Rally

Simply install Rally with pip: `pip3 install esrally`

Note: Depending on your system setup you may need to prepend this command with `sudo`.

If you get errors during installation, it is probably due to the installation of `psutil` which we use to gather system metrics like CPU utilization. Please check the installation instructions of psutil in this case. Keep in mind that Rally is based on Python 3 and you need to install the Python 3 header files instead of the Python 2 header files on Linux.

Non-sudo Install

If you don't want to use `sudo` when installing Rally, installation is still possible but a little more involved:

1. Specify the `--user` option when installing Rally (step 2 above), so the command to be issued is: `python3 setup.py develop --user`.
2. Check the output of the install script or lookup the Python documentation on the variable site.USER_BASE to find out where the script is located. On Linux, this is typically `~/.local/bin`.

You can now either add `~/.local/bin` to your path or invoke Rally via `~/.local/bin/esrally` instead of just `esrally`.

VirtualEnv Install

You can also use Virtualenv to install Rally into an isolated Python environment without sudo.

1. Set up a new virtualenv environment in a directory with `virtualenv --python=python3 .`
2. Activate the environment with `source /path/to/virtualenv/dir/bin/activate`
3. Install Rally with `pip install esrally`

Whenever you want to use Rally, run the activation script (step 2 above) first. When you are done, simply execute `deactivate` in the shell to exit the virtual environment.

Next Steps

After you have installed, you need to configure it. Just run `esrally configure` or follow the *configuration help page* for more guidance.

Configuration

Rally has to be configured once after installation. If you just run `esrally` after installing Rally, it will detect that the configuration file is missing and asks you a few questions.

If you want to reconfigure Rally at any later time, just run `esrally configure` again.

Simple Configuration

By default, Rally will run a simpler configuration routine and autodetect as much settings as possible or choose defaults for you. If you need more control you can run Rally with `esrally configure --advanced-config`.

Rally can build Elasticsearch either from sources or use an official binary distribution. If you have Rally build Elasticsearch from sources, it can only be used to benchmark Elasticsearch 5.0 and above. The reason is that with Elasticsearch 5.0 the build tool was switched from Maven to Gradle. As Rally only supports Gradle, it is limited to Elasticsearch 5.0 and above.

If you want to build Elasticsearch from sources, Gradle needs to be installed prior to running the configuration routine.

Let's go through an example step by step: First run `esrally`:

```
dm@io:~ $ esrally


    ____         ____
   / __ \_____ _/ / /_  __
  / /_/ / __ `/ / / / / / /
 / _, _/ /_/ / / / / /_/ /
/_/ |_|\__,_/_/_/_/\__, /
                  /____/

Running simple configuration. You can run the advanced configuration with:

  esrally configure --advanced-config
```

```
* Autodetecting available third-party software
  git    : [OK]
  gradle : [OK]
  JDK 8  : [OK]

* Setting up benchmark data directory in [/Users/dm/.rally/benchmarks] (needs several␣
↪GB).
```

As you can see above, Rally autodetects if git, Gradle and JDK 8 are installed. If you don't have Gradle, that's no problem, you are just not able to build Elasticsearch from sources. Let's assume you don't have Gradle installed:

```
dm@io:~ $ esrally


    ____             ____
   / __ \_____ _/ / /_  __
  / /_/ / __ `/ / / / / / /
 / _, _/ /_/ / / / / /_/ /
/_/ |_|\__,_/_/_/_/\__, /
                  /____/

Running simple configuration. You can run the advanced configuration with:

  esrally configure --advanced-config

* Autodetecting available third-party software
  git    : [OK]
  gradle : [MISSING]
  JDK 8  : [OK]

************************************************************************************
You don't have the necessary software to benchmark source builds of Elasticsearch.

You can still benchmark binary distributions with e.g.:

  esrally --distribution-version=5.0.0
************************************************************************************
```

As you can see, Rally tells you that you cannot build Elasticsearch from sources but you can still benchmark official binary distributions.

It's also possible that Rally cannot automatically find your JDK 8 home directory. In that case, it will ask you later in the configuration process.

After the initial detection, Rally will try to autodetect your Elasticsearch project directory (either in the current directory or in `../elasticsearch`). If all goes well, then you will see this:

```
Autodetected Elasticsearch project directory at [/Users/dm/elasticsearch].
```

Otherwise, Rally will choose a default directory and ask you for confirmation:

```
* Setting up benchmark data directory in [/Users/dm/.rally/benchmarks] (needs several␣
↪GB).
Enter your Elasticsearch project directory: [default: '/Users/dm/.rally/benchmarks/src
↪']:
```

If you are ok with this default, just press "Enter" and Rally will take care of the rest. Otherwise, provide your Elasticsearch project directory here. Please keep in mind that Rally will run builds with Gradle in this directory if you start a benchmark.

If Rally has not found Gradle in the first step, it will not ask you for a source directory and just go on.

Now Rally is done:

```
Configuration successfully written to [/Users/dm/.rally/rally.ini]. Happy␣
↪benchmarking!

To benchmark the latest version of Elasticsearch with the default benchmark run:

  esrally --revision=latest

For more help see the user documentation at https://esrally.readthedocs.io
```

Congratulations! Time to run your first benchmark.

Advanced Configuration

If you need more control over a few variables or want to store your metrics in a dedicated Elasticsearch metrics store, then you should run the advanced configuration routine. You can invoke it at any time with `esrally configure --advanced-config`.

Prerequisites

When using the advanced configuration, Rally stores its metrics not in-memory but in a dedicated Elasticsearch instance. Therefore, you will also need the following software installed:

- Elasticsearch: a dedicated Elasticsearch instance which acts as the metrics store for Rally. If you don't want to set it up yourself you can also use Elastic Cloud.
- Optional: Kibana (also included in Elastic Cloud).

Preparation

First install Elasticsearch 5.0 or higher. A simple out-of-the-box installation with a single node will suffice. Rally uses this instance to store metrics data. It will setup the necessary indices by itself. The configuration procedure of Rally will you ask for host and port of this cluster.

Note: Rally will choose the port range 39200-39300 (HTTP) and 39300-39400 (transport) for the benchmark cluster, so please ensure that this port range is not used by the metrics store.

Optional but recommended is to install also Kibana. However, note that Kibana will not be auto-configured by Rally.

Configuration Options

Rally will ask you a few more things in the advanced setup:

- Benchmark data directory: Rally stores all benchmark related data in this directory which can take up to several tens of GB. If you want to use a dedicated partition, you can specify a different data directory here.
- Elasticsearch project directory: This is the directory where the Elasticsearch sources are located. If you don't actively develop on Elasticsearch you can just leave the default but if you want to benchmark local changes you should point Rally to your project directory. Note that Rally will run builds with Gradle in this directory (it runs `gradle clean` and `gradle :distribution:tar:assemble`).

- JDK 8 root directory: Rally will only ask this if it could not autodetect the JDK 8 home by itself. Just enter the root directory of the JDK you want to use.
- Name for this benchmark environment: You can use the same metrics store for multiple environments (e.g. local, continuous integration etc.) so you can separate metrics from different environments by choosing a different name.
- metrics store settings: Provide the connection details to the Elasticsearch metrics store. This should be an instance that you use just for Rally but it can be a rather small one. A single node cluster with default setting should do it. There is currently no support for choosing the in-memory metrics store when you run the advanced configuration. If you really need it, please raise an issue on Github.
- whether or not Rally should keep the Elasticsearch benchmark candidate installation including all data by default. This will use lots of disk space so you should wipe `~/.rally/benchmarks/races` regularly.

Proxy Configuration

Rally downloads all necessary data automatically for you:

- Elasticsearch distributions from elastic.co if you specify `--distribution-version=SOME_VERSION_NUMBER`
- Elasticsearch source code from Github if you specify a revision number e.g. `--revision=952097b`
- Track meta-data from Github
- Track data from an S3 bucket

Hence, it needs to connect via http(s) to the outside world. If you are behind a corporate proxy you need to configure Rally and git. As many other Unix programs, Rally relies that the HTTP proxy URL is available in the environment variable `http_proxy` (note that this is in lower-case). Hence, you should add this line to your shell profile, e.g. `~/.bash_profile`:

```
export http_proxy=http://proxy.acme.org:8888/
```

Afterwards, source the shell profile with `source ~/.bash_profile` and verify that the proxy URL is correctly set with `echo $http_proxy`.

Finally, you can set up git:

```
git config --global http.proxy $http_proxy
```

For details, please refer to the Git config documentation.

Please verify that the proxy setup for git works correctly by cloning any repository, e.g. the `rally-tracks` repository:

```
git clone https://github.com/elastic/rally-tracks.git
```

If the configuration is correct, git will clone this repository. You can delete the folder `rally-tracks` after this verification step.

To verify that Rally will connect via the proxy server you can check the log file. If the proxy server is configured successfully, Rally will log the following line on startup:

```
Rally connects via proxy URL [http://proxy.acme.org:3128/] to the Internet (picked up␣
↪from the environment variable [http_proxy]).
```

Note: Rally will use this proxy server only for downloading benchmark-related data. It will not use this proxy for the actual benchmark.

Running Races

A "race" in Rally is the execution of a benchmarking experiment. You can use different data sets (which we call *tracks*) for your benchmarks.

List Tracks

Start by finding out which tracks are available:

```
esrally list tracks
```

This will show the following list:

```
Name        Description                                                                                 Documents  Compressed Size    Uncompressed Size    Default Challenge        All Challenges
----------  ------------------------------------------------------------------------------------------  -----------  -----------------  -------------------  -----------------------  ----------------------
geonames    Standard track for Rally (11.4M POIs from Geonames)                                          11396505  252.4 MB           3.3 GB               append-no-conflicts      append-no-conflicts,app...
geopoint    60.8M POIs from PlanetOSM                                                                    60844404  481.9 MB           2.3 GB               append-no-conflicts      append-no-conflicts,app...
logging     Logging benchmark                                                                           247249096  1.2 GB             31.1 GB              append-no-conflicts      append-no-conflicts,app...
nested      Nested query benchmark using up to 11,203,029 questions from StackOverflow                   11203029  663.1 MB           3.4 GB               nested-search-challenge  nested-search-challenge...
nyc_taxis   Trip records completed in yellow and green taxis in New York in 2015                         165346692  4.5 GB             74.3 GB              append-no-conflicts      append-no-conflicts,app...
percolator  Percolator benchmark based on 2M AOL queries                                                  2000000  102.7 kB           104.9 MB             append-no-conflicts      append-no-conflicts,app...
pmc         Full text benchmark containing 574.199 papers from PMC                                       574199  5.5 GB             21.7 GB              append-no-conflicts      append-no-conflicts,app...
```

The first two columns show the name and a short description of each track. A track also specifies one or more challenges which basically defines the operations that will be run.

Starting a Race

Note: Do not run Rally as root as Elasticsearch will refuse to start with root privileges.

To start a race you have to define the track and challenge to run. For example:

```
esrally --distribution-version=5.0.0 --track=geopoint --challenge=append-fast-with-
↪conflicts
```

Rally will then start racing on this track. If you have never started Rally before, it should look similar to the following output:

```
dm@io:~ $ esrally --distribution-version=5.0.0 --track=geopoint --challenge=append-
↪fast-with-conflicts


    ____             ____
   / __ \____ _/ / /_  __
  / /_/ / __ `/ / / / / /
 / _, _/ /_/ / / / /_/ /
/_/ |_|\__,_/_/_/_/\__, /
                  /____/

[INFO] Racing on track [geopoint], challenge [append-fast-with-conflicts] and car␣
↪[defaults]
[INFO] Downloading Elasticsearch 5.0.0 ... [OK]
[INFO] Rally will delete the benchmark candidate after the benchmark
[INFO] Downloading data from [http://benchmarks.elasticsearch.org.s3.amazonaws.com/
↪corpora/geopoint/documents.json.bz2] (482 MB) to [/Users/dm/.rally/benchmarks/data/
↪geopoint/documents.json.bz2] ... [OK]
[INFO] Decompressing track data from [/Users/dm/.rally/benchmarks/data/geopoint/
↪documents.json.bz2] to [/Users/dm/.rally/benchmarks/data/geopoint/documents.json]␣
↪(resulting size: 2.28 GB) ... [OK]
[INFO] Preparing file offset table for [/Users/dm/.rally/benchmarks/data/geopoint/
↪documents.json] ... [OK]
Running index-update                                                           [  0%␣
↪done]
```

Please be patient as it will take a while to run the benchmark.

When the race has finished, Rally will show a summary on the command line:

```
|                            Metric |    Operation |     Value |   Unit |
|----------------------------------:|-------------:|----------:|-------:|
|                     Indexing time |              |   124.712 |    min |
|                        Merge time |              |   21.8604 |    min |
|                      Refresh time |              |   4.49527 |    min |
|                Merge throttle time |              |  0.120433 |    min |
|                   Median CPU usage |              |     546.5 |      % |
|                 Total Young Gen GC |              |    72.078 |      s |
|                   Total Old Gen GC |              |     3.426 |      s |
|                         Index size |              |   2.26661 |     GB |
|                      Totally written |              |    30.083 |     GB |
|           Heap used for segments |              |   10.7148 |     MB |
|         Heap used for doc values |              | 0.0135536 |     MB |
|              Heap used for terms |              |   9.22965 |     MB |
|             Heap used for points |              |   0.78789 |     MB |
|      Heap used for stored fields |              |  0.683708 |     MB |
|                    Segment count |              |       115 |        |
|                   Min Throughput | index-update |   59210.4 | docs/s |
|                Median Throughput | index-update |   65276.2 | docs/s |
|                   Max Throughput | index-update |   76516.6 | docs/s |
|       50.0th percentile latency | index-update |   556.269 |     ms |
|       90.0th percentile latency | index-update |   852.779 |     ms |
```

```
|       99.0th percentile latency | index-update |   1854.31 |     ms |
|       99.9th percentile latency | index-update |   2972.96 |     ms |
|      99.99th percentile latency | index-update |   4106.91 |     ms |
|        100th percentile latency | index-update |   4542.84 |     ms |
|  50.0th percentile service time | index-update |   556.269 |     ms |
|  90.0th percentile service time | index-update |   852.779 |     ms |
|  99.0th percentile service time | index-update |   1854.31 |     ms |
|  99.9th percentile service time | index-update |   2972.96 |     ms |
| 99.99th percentile service time | index-update |   4106.91 |     ms |
|   100th percentile service time | index-update |   4542.84 |     ms |
|                  Min Throughput |  force-merge |  0.221067 |  ops/s |
|               Median Throughput |  force-merge |  0.221067 |  ops/s |
|                  Max Throughput |  force-merge |  0.221067 |  ops/s |
|        100th percentile latency |  force-merge |   4523.52 |     ms |
|   100th percentile service time |  force-merge |   4523.52 |     ms |


----------------------------------
[INFO] SUCCESS (took 1624 seconds)
----------------------------------
```

Note: You can save this report also to a file by using `--report-file=/path/to/your/report.md` and save it as CSV with `--report-format=csv`.

What did Rally just do?

- It downloaded and started Elasticsearch 5.0.0
- It downloaded the relevant data for the geopoint track
- It ran the actual benchmark
- And finally it reported the results

If you are curious about the operations that Rally has run, please inspect the geopoint track specification or start to *write your own tracks*. You can also configure Rally to *store all data samples in Elasticsearch* so you can analyze the results with Kibana. Finally, you may want to *change the Elasticsearch configuration*.

Tournaments

Suppose, we want to analyze the impact of a performance improvement. First, we need a baseline measurement. We can use the command line parameter `--user-tag` to provide a key-value pair to document the intent of a race. After we've run both races, we want to know about the performance impact of a change. With Rally we can analyze differences of two given races easily. First of all, we need to find two races to compare by issuing `esrally list races`:

```
dm@io:~ $ esrally list races


    ____            ____
   / __ \____ _/ / /_  __
  / /_/ / __ `/ / / / / /
 / _, _/ /_/ / / / /_/ /
/_/ |_|\__,_/_/_/\__, /
                   /____/
Recent races:
```

```
Race Timestamp    Track    Challenge             Car       User Tag
----------------  -------  --------------------  --------  ----------------------------
↪--
20160518T122341Z  pmc      append-no-conflicts   defaults  intention:reduce_alloc_1234
20160518T112057Z  pmc      append-no-conflicts   defaults  intention:baseline_github_
↪1234
20160518T101957Z  pmc      append-no-conflicts   defaults
```

We can see that the user tag helps us to recognize races. We want to compare the two most recent races and have to provide the two race timestamps in the next step:

```
dm@io:~ $ esrally compare --baseline=20160518T112057Z --contender=20160518T112341Z


    ____             ____
   / __ \_____ _/ / /_  __
  / /_/ / __ `/ / / / / /
 / _, _/ /_/ / / / /_/ /
/_/ |_|\__,_/_/_/_/\__, /
                  /____/

Comparing baseline
  Race timestamp: 2016-05-18 11:20:57
  Challenge: append-no-conflicts
  Car: defaults

with contender
  Race timestamp: 2016-05-18 12:23:41
  Challenge: append-no-conflicts
  Car: defaults

--------------------------------------------------
    ________                __   _____
   / ____(_)___  ____ _/ /  / ___/_________  ________
  / /_  / / __ \/ __ `/ /   \__ \/ ___/ __ \/ ___/ _ \
 / __/ / / / / / /_/ / /   ___/ / /__/ /_/ / /  /  __/
/_/   /_/_/ /_/\__,_/_/   /____/\___/\____/_/   \___/
--------------------------------------------------
                                                       Metric    Baseline    Contender
↪          Diff
-------------------------------------------------------  ----------  -----------  ---
↪--------------
                         Min Indexing Throughput [docs/s]       19501        19118  -
↪383.00000
                      Median Indexing Throughput [docs/s]       20232      19927.5  -
↪304.45833
                         Max Indexing Throughput [docs/s]       21172        20849  -
↪323.00000
                                      Indexing time [min]     55.7989       56.335
↪+0.53603
                                         Merge time [min]     12.9766      13.3115
↪+0.33495
                                       Refresh time [min]     5.20067      5.20097
↪+0.00030
                                         Flush time [min]   0.0648667    0.0681833
↪+0.00332
                                Merge throttle time [min]    0.796417     0.879267
↪+0.08285
```

```
                Query latency term (50.0 percentile) [ms]      2.10049       2.15421
↪+0.05372
                Query latency term (90.0 percentile) [ms]      2.77537       2.84168
↪+0.06630
               Query latency term (100.0 percentile) [ms]      4.52081       5.15368
↪+0.63287
         Query latency country_agg (50.0 percentile) [ms]      112.049       110.385
↪1.66392
         Query latency country_agg (90.0 percentile) [ms]      128.426       124.005
↪4.42138
        Query latency country_agg (100.0 percentile) [ms]      155.989       133.797
↪22.19185
              Query latency scroll (50.0 percentile) [ms]      16.1226       14.4974
↪1.62519
              Query latency scroll (90.0 percentile) [ms]      17.2383       15.4079
↪1.83043
             Query latency scroll (100.0 percentile) [ms]      18.8419       18.4241
↪0.41784
 Query latency country_agg_cached (50.0 percentile) [ms]      1.70223       1.64502
↪0.05721
 Query latency country_agg_cached (90.0 percentile) [ms]      2.34819       2.04318
↪0.30500
Query latency country_agg_cached (100.0 percentile) [ms]      3.42547       2.86814
↪0.55732
             Query latency default (50.0 percentile) [ms]      5.89058       5.83409
↪0.05648
             Query latency default (90.0 percentile) [ms]      6.71282       6.64662
↪0.06620
            Query latency default (100.0 percentile) [ms]      7.65307        7.3701
↪0.28297
              Query latency phrase (50.0 percentile) [ms]      1.82687       1.83193
↪+0.00506
              Query latency phrase (90.0 percentile) [ms]      2.63714       2.46286
↪0.17428
             Query latency phrase (100.0 percentile) [ms]      5.39892       4.22367
↪1.17525
                           Median CPU usage (index) [%]      668.025        679.15
↪+11.12499
                           Median CPU usage (stats) [%]       143.75         162.4
↪+18.64999
                          Median CPU usage (search) [%]        223.1         229.2
↪+6.10000
                                 Total Young Gen GC [s]       39.447        40.456
↪+1.00900
                                   Total Old Gen GC [s]        7.108         7.703
↪+0.59500
                                        Index size [GB]      3.25475       3.25098
↪0.00377
                                   Totally written [GB]      17.8434       18.3143
↪+0.47083
                             Heap used for segments [MB]      21.7504       21.5901
↪0.16037
                          Heap used for doc values [MB]      0.16436       0.13905
↪0.02531
                               Heap used for terms [MB]      20.0293       19.9159
↪0.11345
                               Heap used for norms [MB]     0.105469     0.0935669
↪0.01190
```

```
                                Heap used for points [MB]     0.773487      0.772155     -
↪0.00133
                                Heap used for points [MB]     0.677795      0.669426     -
↪0.00837
                                             Segment count          136           121     -
↪15.00000
                          Indices Stats(90.0 percentile) [ms]     3.16053       3.21023     
↪+0.04969
                          Indices Stats(99.0 percentile) [ms]     5.29526       3.94132     -
↪1.35393
                         Indices Stats(100.0 percentile) [ms]     5.64971       7.02374     
↪+1.37403
                            Nodes Stats(90.0 percentile) [ms]     3.19611       3.15251     -
↪0.04360
                            Nodes Stats(99.0 percentile) [ms]     4.44111       4.87003     
↪+0.42892
                           Nodes Stats(100.0 percentile) [ms]     5.22527       5.66977     
↪+0.44450
```

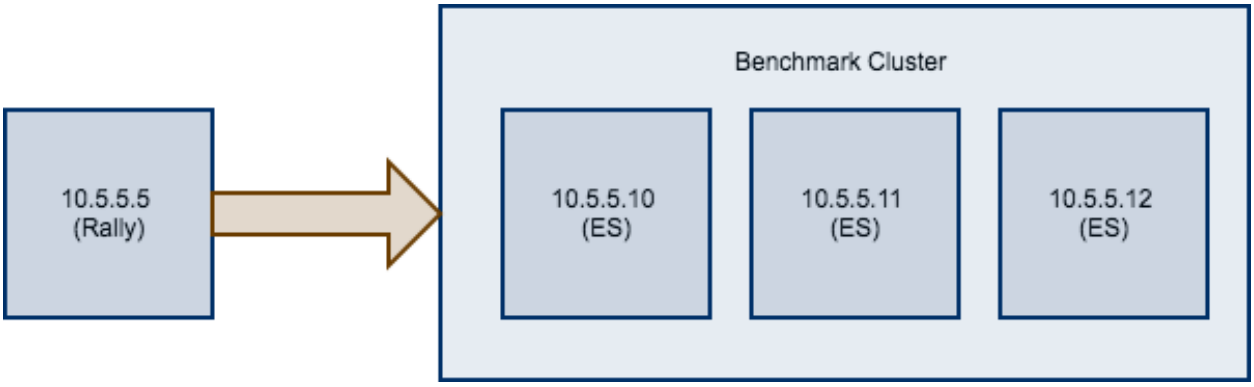
Recipes

Benchmarking an existing cluster

Warning: If you are just getting started with Rally and don't understand how it works, please do NOT run it against any production or production-like cluster. Besides, benchmarks should be executed in a dedicated environment anyway where no additional traffic skews results.

Note: We assume in this recipe, that Rally is already properly *configured*.

Consider the following configuration: You have an existing benchmarking cluster, that consists of three Elasticsearch nodes running on `10.5.5.10`, `10.5.5.11`, `10.5.5.12`. You've setup the cluster yourself and want to benchmark it with Rally. Rally is installed on `10.5.5.5`.



First of all, we need to decide on a track. So, we run `esrally list tracks`:

```
Name        Description
↪    Documents  Compressed Size    Uncompressed Size    Default Challenge        All␣
↪Challenges
----------  ------------------------------------------------------------------------
↪  -----------  -----------------  -------------------  ------------------------  ----
↪----------------------
geonames    Standard track in Rally (11.4M POIs from Geonames)
↪     11396505  252.4 MB           3.3 GB               append-no-conflicts      ␣
↪append-no-conflicts,app...
geopoint    60.8M POIs from PlanetOSM
↪     60844404  481.9 MB           2.3 GB               append-no-conflicts      ␣
↪append-no-conflicts,app...
logging     Logging benchmark
↪    247249096  1.2 GB             31.1 GB              append-no-conflicts      ␣
↪append-no-conflicts,app...
nested      Nested query benchmark using up to 11,203,029 questions from␣
↪StackOverflow     11203029  663.1 MB           3.4 GB               nested-search-
↪challenge  nested-search-challenge...
nyc_taxis   Trip records completed in yellow and green taxis in New York in 2015
↪    165346692  4.5 GB             74.3 GB              append-no-conflicts      ␣
↪append-no-conflicts,app...
percolator  Percolator benchmark based on 2M AOL queries
↪      2000000  102.7 kB           104.9 MB             append-no-conflicts      ␣
↪append-no-conflicts,app...
pmc         Full text benchmark containing 574.199 papers from PMC
↪       574199  5.5 GB             21.7 GB              append-no-conflicts      ␣
↪append-no-conflicts,app...
```

We're interested in a full text benchmark, so we'll choose to run `pmc`. If you have your own data that you want to use for benchmarks, then please *create your own track* instead; the metrics you'll gather which be representative and much more useful than some default track.

Next, we need to know which machines to target which is easy as we can see that from the diagram above.

Finally we need to check which *pipeline* to use. For this case, the `benchmark-only` pipeline is suitable as we don't want Rally to provision the cluster for us.

Now we can invoke Rally:

```
esrally --track=pmc --target-hosts=10.5.5.10:9200,10.5.5.11:9200,10.5.5.12:9200 --
↪pipeline=benchmark-only
```

If you have X-Pack Security enabled, then you'll also need to specify another parameter to use https and to pass credentials:

```
esrally --track=pmc --target-hosts=10.5.5.10:9243,10.5.5.11:9243,10.5.5.12:9243 --
↪pipeline=benchmark-only --client-options="basic_auth_user:'elastic',basic_auth_
↪password:'changeme'"
```

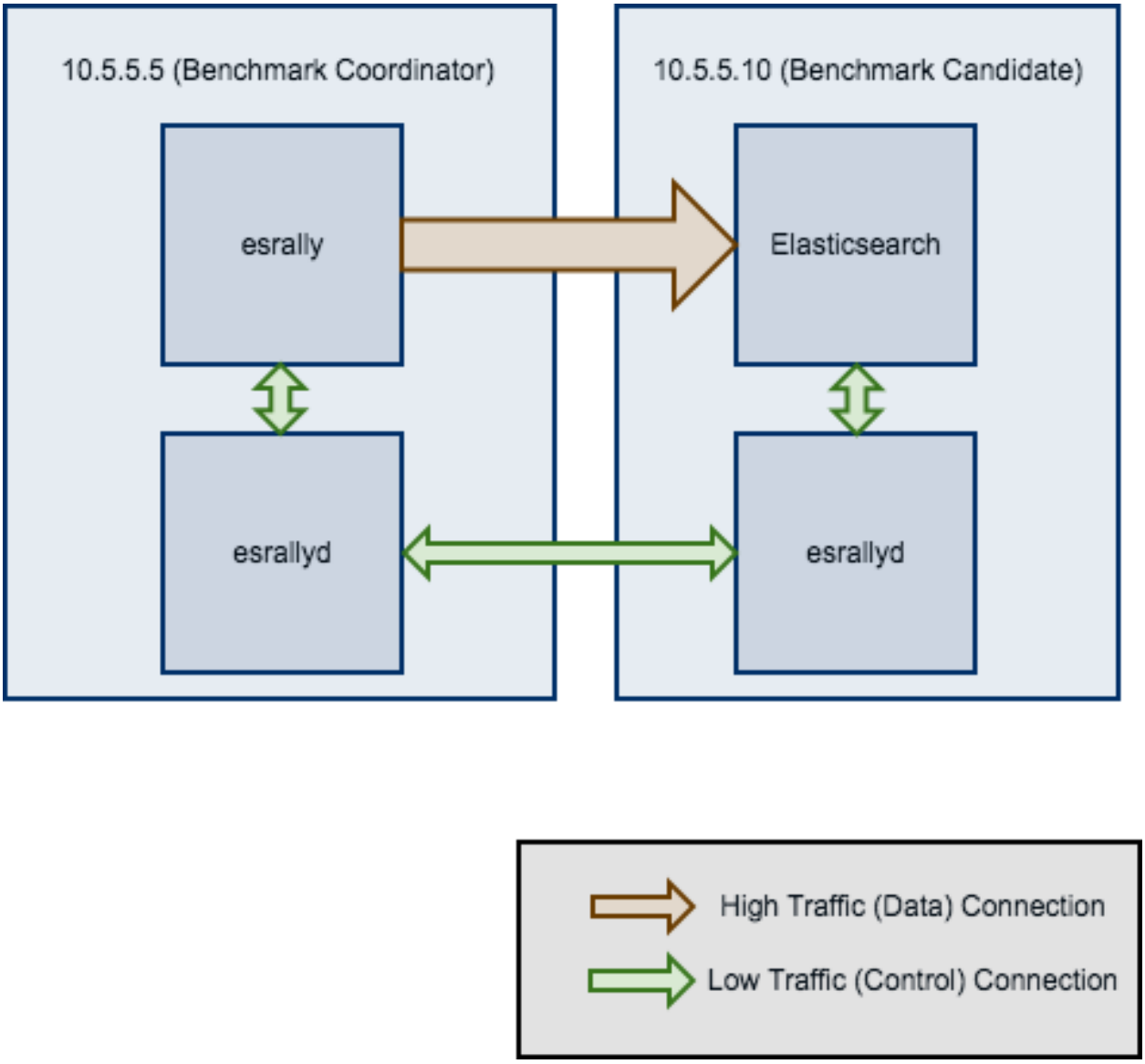
Benchmarking a remote cluster

Contrary to the previous recipe, you want Rally to provision all cluster nodes.

Note: At the moment, Rally can only provision a single remote node but support for provisioning of multi-node clusters is a high priority topic on our roadmap.

We will use the following configuration for the example:

- You will start Rally on `10.5.5.5`. We will call this machine the "benchmark coordinator".
- Your Elasticsearch cluster will run on `10.5.5.10`. We will call this machine the "benchmark candidate".



To run a benchmark for this scenario follow these steps:

1. *Install* and *configure* Rally on all machines. Be sure that the same version is installed on all of them.
2. Start the Rally daemon on each machine. The Rally daemon allows Rally to communicate with all remote machines. On the benchmark coordinator run `esrallyd start --node-ip=10.5.5.5 --coordinator-ip=10.5.5.5` and on the benchmark candidate run `esrallyd start --node-ip=10.5.5.10 --coordinator-ip=10.5.5.5`. The `--node-ip` parameter tells Rally the IP of the machine on which it is running. As some machines have more than one network interface, Rally will not attempt to auto-detect the machine IP. The `--coordinator-ip` parameter tells Rally the IP of the benchmark coordinator node.
3. Start the benchmark by invoking Rally as usual on the benchmark coordinator, for example: `esrally --distribution-version=5.0.0 --target-hosts=10.5.5.10:9200`. Rally will derive from

the `--target-hosts` parameter that it should provision the node `10.5.5.10`.

4. After the benchmark has finished you can stop the Rally daemon again. On the benchmark coordinator and on the benchmark candidate run `esrallyd stop`.

Note: Logs are managed per machine, so all relevant log files and also telemetry output is stored on the benchmark candidate but not on the benchmark coordinator.

Now you might ask yourself what the differences to benchmarks of existing clusters are. In general you should aim to give Rally as much control as possible as benchmark are easier reproducible and you get more metrics. The following table provides some guidance on when to choose which option:

Your requirement	Recommendation
You want to use Rally's telemetry devices	Use Rally daemon, as it can provision the remote node for you
You want to benchmark a source build of Elasticsearch	Use Rally daemon, as it can build Elasticsearch for you
You want to tweak the cluster configuration yourself	Set up the cluster by yourself and use `--pipeline=benchmark-only`
You need to run a benchmark with plugins	Set up the cluster by yourself and use `--pipeline=benchmark-only`
You need to run a benchmark against multiple nodes	Set up the cluster by yourself and use `--pipeline=benchmark-only`

Rally daemon will be able to cover most of the cases described above in the future so there should be almost no case where you need to use the `benchmark-only` pipeline.

Changing the default track repository

Rally supports multiple track repositories. This allows you for example to have a separate company-internal repository for your own tracks that is separate from Rally's default track repository. However, you always need to define `--track-repository=my-custom-repository` which can be cumbersome. If you want to avoid that and want Rally to use your own track repository by default you can just replace the default track repository definition in `~./rally/rally.ini`. Consider this example:

```
...
[tracks]
default.url = git@github.com:elastic/rally-tracks.git
teamtrackrepo.url = git@example.org/myteam/my-tracks.git
```

If `teamtrackrepo` should be the default track repository, just define it as `default.url`. E.g.:

```
...
[tracks]
default.url = git@example.org/myteam/my-tracks.git
old-rally-default.url=git@github.com:elastic/rally-tracks.git
```

Also don't forget to rename the folder of your local working copy as Rally will search for a track repository with the name `default`:

```
cd ~/.rally/benchmarks/tracks/
mv default old-rally-default
mv teamtrackrepo default
```

From now on, Rally will treat your repository as default and you need to run Rally with `--track-repository=old-rally-default` if you want to use the out-of-the-box Rally tracks.

Developing Rally

Prerequisites

Please ensure that the following packages are installed before installing Rally in development mode:

- Python 3.4 or better available as *python3* on the path (verify with: `python3 --version` which should print `Python 3.4.0` (or higher))
- `pip3` available on the path (verify with `pip3 --version`)
- JDK 8
- git 1.9 or better
- Gradle 3.3 or better

Rally does not support Windows and is only actively tested on Mac OS X and Linux.

Installation Instructions for Development

```
git clone https://github.com/elastic/rally.git
cd rally
./rally
```

If you get errors during installation, it is probably due to the installation of `psutil` which we use to gather system metrics like CPU utilization. Please check the installation instructions of psutil in this case. Keep in mind that Rally is based on Python 3 and you need to install the Python 3 header files instead of the Python 2 header files on Linux.

Configuring Rally

Before we can run our first benchmark, we have to configure Rally. Just invoke `./rally configure` and Rally will automatically detect that its configuration file is missing and prompt you for some values and write them to `~/.rally/rally.ini`. After you've configured Rally, it will exit.

For more information see *configuration help page*.

Key Components of Rally

To get a rough understanding of Rally, it makes sense to get to know its key components:

- *Race Control*: is responsible for proper execution of the race. It sets up all components and acts as a high-level controller.
- *Mechanic*: can build and prepare a benchmark candidate for the race. It checks out the source, builds Elasticsearch, provisions and starts the cluster.
- *Track*: is a concrete benchmarking scenario, e.g. the logging benchmark. It defines the data set to use.
- *Challenge*: is the specification on what benchmarks should be run and its configuration (e.g. index, then run a search benchmark with 1000 iterations)
- *Car*: is a concrete system configuration for a benchmark, e.g. an Elasticsearch single-node cluster with default settings.
- *Driver*: drives the race, i.e. it is executing the benchmark according to the track specification.
- *Reporter*: A reporter tells us how the race went (currently only after the fact).

There is a dedicated *tutorial on how to add new tracks to Rally*.

How to contribute code

First of all, please read the contributors guide.

We strive to be PEP-8 compliant but don't follow it to the letter.

Creating custom tracks

Note: Please see the *track reference* for more information on the structure of a track.

Example track

Let's create an example track step by step. We will call this track "tutorial". The track consists of two components: the data set and the actual track which describes what Rally should do with the data set.

First of all, we need some data. There are a lot of public data sets available which are interesting for new benchmarks and we also have a backlog of benchmarks to add.

Geonames provides geo data under a creative commons license. We will download allCountries.zip (around 300MB), extract it and inspect `allCountries.txt`.

You will note that the file is tab-delimited but we need JSON to bulk-index data with Elasticsearch. So we can use a small script to do the conversion for us:

```
import json

cols = (("geonameid", "int"),
        ("name", "string"),
        ("asciiname", "string"),
        ("alternatenames", "string"),
        ("latitude", "double"),
        ("longitude", "double"),
        ("feature_class", "string"),
        ("feature_code", "string"),
        ("country_code", "string"),
        ("cc2", "string"),
        ("admin1_code", "string"),
        ("admin2_code", "string"),
        ("admin3_code", "string"),
        ("admin4_code", "string"),
        ("population", "long"),
        ("elevation", "int"),
        ("dem", "string"),
        ("timezone", "string"))


def main():
    with open("allCountries.txt", "rt", encoding="UTF-8") as f:
        for line in f:
            tup = line.strip().split("\t")
            record = {}
```

```
            for i in range(len(cols)):
                name, type = cols[i]
                if tup[i] != "":
                    if type in ("int", "long"):
                        record[name] = int(tup[i])
                    elif type == "double":
                        record[name] = float(tup[i])
                    else:
                        record[name] = tup[i]
            print(json.dumps(record, ensure_ascii=False))


if __name__ == "__main__":
    main()
```

Go to `~/.rally/benchmarks/data` and create a folder "tutorial" there. Then invoke the script with `python3 toJSON.py > ~/.rally/benchmarks/data/tutorial/documents.json`.

Next we need to compress the JSON file with `bzip2 -9 -c documents.json > documents.json.bz2`. If you want other people to run the benchmark too, upload the data file to a place where it is publicly available. We choose `http://benchmarks.elasticsearch.org.s3.amazonaws.com/corpora/tutorial` for this example. If you don't want to share your track, just don't specify a `data-url` below.

Next we need a mapping file for our documents. For details on how to write a mapping file, see the Elasticsearch documentation on mappings and look at an example mapping file. Place the mapping file in your `rally-tracks` repository in a dedicated folder. This repository is located in `~/.rally/benchmarks/tracks/default` and we place the mapping file in `~/.rally/benchmarks/tracks/default/tutorial` for this track.

The track repository is managed by git. The reason is that you can run Rally with any version of Elasticsearch from 1.0 until the latest master version. As mappings change over time, we need to have a possibility to define different versions of our track. Therefore, Rally uses branches for that. For example, if you create a track on a branch named `5`, Rally will checkout this branch. Please see the README of the rally-track repository for details on the versioning scheme.

Ensure that you are on the `master` branch by running `git checkout master`. Then add a new JSON file right next to the mapping file. The file has to be called "track.json" and is the actual track specification

```
{
  "short-description": "Tutorial benchmark for Rally",
  "description": "This test indexes 8.6M documents (POIs from Geonames, total 2.8 GB␣
↪json) using 8 client threads and 5000 docs per bulk request against Elasticsearch",
  "data-url": "http://benchmarks.elasticsearch.org.s3.amazonaws.com/corpora/tutorial",
  "indices": [
    {
      "name": "geonames",
      "types": [
        {
          "name": "type",
          "mapping": "mappings.json",
          "documents": "documents.json.bz2",
          "document-count": 8647880,
          "compressed-bytes": 197857614,
          "uncompressed-bytes": 2790927196
        }
      ]
    }
  ],
  "operations": [
```

```
    {
      "name": "index",
      "operation-type": "index",
      "bulk-size": 5000
    },
    {
      "name": "force-merge",
      "operation-type": "force-merge"
    },
    {
      "name": "query-match-all",
      "operation-type": "search",
      "body": {
        "query": {
          "match_all": {}
        }
      }
    }
  ],
  "challenges": [
    {
      "name": "index-and-query",
      "description": "",
      "default": true,
      "index-settings": {
        "index.number_of_replicas": 0
      },
      "schedule": [
        {
          "operation": "index",
          "warmup-time-period": 120,
          "clients": 8
        },
        {
          "operation": "force-merge",
          "clients": 1
        },
        {
          "operation": "query-match-all",
          "clients": 8,
          "warmup-iterations": 1000,
          "iterations": 1000,
          "target-throughput": 100
        }
      ]
    }
  ]
}
```

Finally, you need to commit your changes: `git commit -a -m "Add tutorial track"`.

A few things to note:

- If you define multiple challenges, Rally will run the challenge where `default` is set to `true`. If you want to run a different challenge, provide the command line option `--challenge=YOUR_CHALLENGE_NAME`.
- You can add as many queries as you want. We use the official Python Elasticsearch client to issue queries.
- The numbers below the `types` property are needed to verify integrity and provide progress reports.

Note: You can store any supporting scripts along with your track. However, you need to place them in a directory starting with "_", e.g. "_support". Rally loads track plugins (see below) from any directory but will ignore directories starting with "_".

Note: We have defined a JSON schema for tracks which you can use to check how to define your track. You should also check the tracks provided by Rally for inspiration.

When you invoke `esrally list tracks`, the new track should now appear:

```
dm@io:~ $ esrally list tracks


    ____             ____
   / __ \_____ _/ / /_  __
  / /_/ / __ `/ / / / / / /
 / _, _/ /_/ / / / / /_/ /
/_/ |_|\__,_/_/_/_/\__, /
                  /____/
Available tracks:

Name        Description                    Documents    Compressed Size  Uncompressed␣
↪Size  Default Challenge  All Challenges
----------  ------------------------------ -----------  ---------------  --------------
↪---  -----------------  ---------------
tutorial    Tutorial benchmark for Rally      8647880  188.7 MB         2.6 GB       ␣
↪     index-and-query    index-and-query
```

Congratulations, you have created your first track! You can test it with `esrally --track=tutorial --offline` and run specific challenges with `esrally --track=tutorial --challenge=index-and-query --offline`.

Adding support for test mode

When you invoke Rally with `--test-mode`, it switches to a mode that allows you to check your track very quickly for syntax errors. To achieve that, it will postprocess its internal track representation after loading it:

- Iteration-based tasks will run at most one warmup iteration and one measurement iteration.
- Time-period-based task will run for at most 10 seconds without any warmup.

To avoid downloading a lot of data, Rally will postprocess all data file names of a track. So instead of `documents.json.bz2`, Rally will attempt to download `documents-1k.json.bz2` and will assume it contains 1.000 documents. However, you need to prepare these data files otherwise this test mode is not supported.

The preparation is very easy and requires these two steps:

1. Pick 1.000 documents from your data set. We choose the first 1.000 here but it does not matter usually which part you choose: `head -n 1000 documents.json > documents-1k.json`.
2. Compress it: `bzip2 -9 -c documents-1k.json > documents-1k.json.bz2`

You have to repeat these steps for all data files of your track.

Structuring your track

`track.json` is just the entry point to a track but you can split your track as you see fit. Suppose you want to add more challenges to the track above but you want to keep them in a separate files. Let's start by storing our challenge in a separate file, e.g in `challenges/index-and-query.json`. Create the directory and store the following in `index-and-query.json`:

```
{
      "name": "index-and-query",
      "description": "",
      "default": true,
      "index-settings": {
        "index.number_of_replicas": 0
      },
      "schedule": [
        {
          "operation": "index",
          "warmup-time-period": 120,
          "clients": 8
        },
        {
          "operation": "force-merge",
          "clients": 1
        },
        {
          "operation": "query-match-all",
          "clients": 8,
          "warmup-iterations": 1000,
          "iterations": 1000,
          "target-throughput": 100
        }
      ]
    }
```

Now modify `track.json` so it knows about your new file:

```
{
  "short-description": "Tutorial benchmark for Rally",
  "description": "This test indexes 8.6M documents (POIs from Geonames, total 2.8 GB␣
↪json) using 8 client threads and 5000 docs per bulk request against Elasticsearch",
  "data-url": "http://benchmarks.elasticsearch.org.s3.amazonaws.com/corpora/tutorial",
  "indices": [
    {
      "name": "geonames",
      "types": [
        {
          "name": "type",
          "mapping": "mappings.json",
          "documents": "documents.json.bz2",
          "document-count": 8647880,
          "compressed-bytes": 197857614,
          "uncompressed-bytes": 2790927196
        }
      ]
    }
  ],
  "operations": [
    {
```

```
      "name": "index",
      "operation-type": "index",
      "bulk-size": 5000
    },
    {
      "name": "force-merge",
      "operation-type": "force-merge"
    },
    {
      "name": "query-match-all",
      "operation-type": "search",
      "body": {
        "query": {
          "match_all": {}
        }
      }
    }
  ],
  "challenges": [
    {% include "challenges/index-and-query.json" %}
  ]
}
```

We replaced the challenge content with `{% include "challenges/index-and-query.json" %}` which tells Rally to include the challenge from the provided file. You can use `include` on arbitrary parts of your track.

However, if your track consists of multiple challenges it can be cumbersome to include them all explicitly. Therefore Rally brings a `collect` helper that collects all related files for you. Let's adapt our track to use it:

```
{% import "rally.helpers" as rally %}
{
  "short-description": "Standard benchmark in Rally (8.6M POIs from Geonames)",
  "description": "This test indexes 8.6M documents (POIs from Geonames, total 2.8 GB␣
↪json) using 8 client threads and 5000 docs per bulk request against Elasticsearch",
  "data-url": "http://benchmarks.elasticsearch.org.s3.amazonaws.com/corpora/tutorial",
  "indices": [
    {
      "name": "geonames",
      "types": [
        {
          "name": "type",
          "mapping": "mappings.json",
          "documents": "documents.json.bz2",
          "document-count": 8647880,
          "compressed-bytes": 197857614,
          "uncompressed-bytes": 2790927196
        }
      ]
    }
  ],
  "operations": [
    {
      "name": "index",
      "operation-type": "index",
      "bulk-size": 5000
    },
    {
      "name": "force-merge",
```

```
      "operation-type": "force-merge"
    },
    {
      "name": "query-match-all",
      "operation-type": "search",
      "body": {
        "query": {
          "match_all": {}
        }
      }
    }
  ],
  "challenges": [
    {{ rally.collect(parts="challenges/*.json") }}
  ]
}
```

We changed two things here. First, we imported helper functions from Rally by adding `{% import "rally.helpers" as rally %}` in line 1. Second, we used Rally's `collect` helper to find and include all JSON files in the "challenges" subdirectory with the statement `{{ rally.collect(parts="challenges/*.json") }}`. When you add new challenges in this directory, Rally will automatically pick them up.

Note: If you want to check the final result, please check Rally's log file. Rally will print the full rendered track there after it has loaded it successfully.

You can even use Jinja2 variables but you need to import the Rally helpers a bit differently then. You also need to declare all variables before the `import` statement:

```
{% set clients = 16 %}
{% import "rally.helpers" as rally with context %}
```

If you use this idiom you can then refer to variables inside your snippets with `{{ clients }}`.

You've now mastered the basics of track development for Rally. It's time to pat yourself on the back before you dive into the advanced topics!

How to contribute a track

First of all, please read Rally's contributors guide.

If you want to contribute your track, follow these steps:

1. Create a track JSON file and mapping files as described above and place them in a separate folder in the `rally-tracks` repository. Please also add a README file in this folder which contains licensing information (respecting the licensing terms of the source data). Note that pull requests for tracks without a license cannot be accepted.
2. Upload the associated data so they can be publicly downloaded via HTTP. The data should be compressed either as .bz2 (recommended) or as .zip. Also, don't forget to upload the "-1k" data files to support test mode properly.
3. Create a pull request in the rally-tracks Github repo.

Advanced topics

Template Language

Rally uses Jinja2 as template language. This allows you to use Jinja2 expressions in track files.

Extension Points

Rally also provides a few extension points to Jinja2:

- `now`: This is a global variable that represents the current date and time when the template is evaluated by Rally.
- `days_ago()`: This is a filter that you can use for date calculations.

You can find an example in the logging track:

```
{
  "name": "range",
    "index": "logs-*",
    "type": "type",
    "body": {
      "query": {
        "range": {
          "@timestamp": {
            "gte": "now-{{'15-05-1998' | days_ago(now)}}d/d",
            "lt": "now/d"
          }
        }
      }
    }
  }
}
```

The data set that is used in the logging track starts on 26-04-1998 but we want to ignore the first few days for this query, so we start on 15-05-1998. The expression `{{'15-05-1998' | days_ago(now)}}` yields the difference in days between now and the fixed start date and allows us to benchmark time range queries relative to now with a predetermined data set.

Custom parameter sources

Note: This is a rather new feature and the API may change! However, the effort to use custom parameter sources is very low.

Warning: Your parameter source is on a performance-critical code-path so please double-check with *Rally's profiling support* that you did not introduce any bottlenecks.

Consider the following operation definition:

```
{
  "name": "term",
  "operation-type": "search",
  "body": {
    "query": {
      "term": {
```

```
        "body": "physician"
      }
    }
  }
}
```

This query is defined statically in the track specification but sometimes you may want to vary parameters, e.g. search also for "mechanic" or "nurse". In this case, you can write your own "parameter source" with a little bit of Python code.

First, define the name of your parameter source in the operation definition:

```
{
  "name": "term",
  "operation-type": "search",
  "param-source": "my-custom-term-param-source"
  "professions": ["mechanic", "physician", "nurse"]
}
```

Rally will recognize the parameter source and looks then for a file `track.py` in the same directory as the corresponding JSON file. This file contains the implementation of the parameter source:

```
import random


def random_profession(indices, params):
    # you must provide all parameters that the runner expects
    return {
        "body": {
            "query": {
                "term": {
                    "body": "%s" % random.choice(params["professions"])
                }
            }
        },
        "index": None,
        "type": None,
        "use_request_cache": False
    }

def register(registry):
    registry.register_param_source("my-custom-term-param-source", random_profession)
```

The example above shows a simple case that is sufficient if the operation to which your parameter source is applied is idempotent and it does not matter whether two clients execute the same operation.

The function `random_profession` is the actual parameter source. Rally will bind the name "my-custom-term-param-source" to this function by calling `register`. `register` is called by Rally before the track is executed.

The parameter source function needs to declare the two parameters `indices` and `params`. *indices* contains all indices of this track and `params` contains all parameters that have been defined in the operation definition in `track.json`. We use it in the example to read the professions to choose.

If you need more control, you need to implement a class. The example above, implemented as a class looks as follows:

```
import random


class TermParamSource:
```

```
    def __init__(self, indices, params):
        self._indices = indices
        self._params = params

    def partition(self, partition_index, total_partitions):
        return self

    def size(self):
        return 1

    def params(self):
        # you must provide all parameters that the runner expects
        return {
            "body": {
                "query": {
                    "term": {
                        "body": "%s" % random.choice(self._params["professions"])
                    }
                }
            },
            "index": None,
            "type": None,
            "use_request_cache": False
        }


def register(registry):
    registry.register_param_source("my-custom-term-param-source", TermParamSource)
```

Let's walk through this code step by step:

- Note the method `register` where you need to bind the name in the track specification to your parameter source implementation class similar to the simple example.
- The class `TermParamSource` is the actual parameter source and needs to fulfill a few requirements:
  - It needs to have a constructor with the signature `__init__(self, indices, params)`. You don't need to store these parameters if you don't need them.
  - `partition(self, partition_index, total_partitions)` is called by Rally to "assign" the parameter source across multiple clients. Typically you can just return `self` but in certain cases you need to do something more sophisticated. If each clients needs to act differently then you can provide different parameter source instances here.
  - `size(self)`: This method is needed to help Rally provide a proper progress indication to users if you use a warmup time period. For bulk indexing, this would return the number of bulks (for a given client). As searches are typically executed with a pre-determined amount of iterations, just return `1` in this case.
  - `params(self)`: This method needs to return a dictionary with all parameters that the corresponding "runner" expects. For the standard case, Rally provides most of these parameters as a convenience, but here you need to define all of them yourself. This method will be invoked once for every iteration during the race. We can see that we randomly select a profession from a list which will be then be executed by the corresponding runner.

Note: Be aware that `params(self)` is called on a performance-critical path so don't do anything in this method that takes a lot of time (avoid any I/O). For searches, you should usually throttle throughput anyway and there it does not matter that much but if the corresponding operation is run without throughput throttling, please double-check that

you did not introduce a bottleneck in the load test driver with your custom parameter source.

In the implementation of custom parameter sources you can access the Python standard API. Using any additional libraries is not supported.

You can also implement your parameter sources and runners in multiple Python files but the main entry point is always `track.py`. The root package name of your plugin is the name of your track.

Custom runners

Warning: Your runner is on a performance-critical code-path so please double-check with *Rally's profiling support* that you did not introduce any bottlenecks.

You cannot only define custom parameter sources but also custom runners. Runners execute an operation against Elasticsearch. Out of the box, Rally supports the following operations:

- Bulk indexing
- Force merge
- Searches
- Index stats
- Nodes stats

If you want to use any other operation, you can define a custom runner. Consider, we want to use the percolate API with an older version of Elasticsearch (note that it has been replaced by the percolate query in Elasticsearch 5.0). To achieve this, we c

In track.json specify an operation with type "percolate" (you can choose this name freely):

```
{
  "name": "percolator_with_content_google",
  "operation-type": "percolate",
  "body": {
    "doc": {
      "body": "google"
    },
    "track_scores": true
  }
}
```

Then create a file `track.py` next to `track.json` and implement the following two functions:

```
def percolate(es, params):
    es.percolate(
        index="queries",
        doc_type="content",
        body=params["body"]
    )


def register(registry):
    registry.register_runner("percolate", percolate)
```

The function `percolate` is the actual runner and takes the following parameters:

- `es`, which is the Elasticsearch Python client
- `params` which is a dict of parameters provided by its corresponding parameter source. Treat this parameter as read only and do not attempt to write to it.

This function can return either:

- Nothing at all. Then Rally will assume that by default `1` and `"ops"` (see below)
- A tuple of `weight` and a `unit`, which is usually `1` and `"ops"`. If you run a bulk operation you might return the bulk size here, for example in number of documents or in MB. Then you'd return for example `(5000, "docs")` Rally will use these values to store throughput metrics.
- A `dict` with arbitrary keys. If the `dict` contains the key `weight` it is assumed to be numeric and chosen as weight as defined above. The key `unit` is treated similarly. All other keys are added to the `meta` section of the corresponding service time and latency metrics records.

Similar to a parameter source you also need to bind the name of your operation type to the function within `register`.

If you need more control, you can also implement a runner class. The example above, implemented as a class looks as follows:

```
class PercolateRunner:
    def __enter__(self):
        return self

    def __call__(self, es, params):
        es.percolate(
            index="queries",
            doc_type="content",
            body=params["body"]
        )

    def __repr__(self, *args, **kwargs):
        return "percolate"

def register(registry):
    registry.register_runner("percolate", PercolateRunner())
```

The actual runner is implemented in the method `__call__` and the same return value conventions apply as for functions. For debugging purposes you should also implement `__repr__` and provide a human-readable name for your runner. Finally, you need to register your runner in the `register` function. Runners also support Python's context manager interface. Rally uses a new context for each request. Implementing the context manager interface can be handy for cleanup of resources after executing an operation. Rally uses it for example to clear open scrolls.

Note: You need to implement `register` just once and register all parameter sources and runners there.

Custom schedulers

Warning: Your scheduler is on a performance-critical code-path so please double-check with *Rally's profiling support* that you did not introduce any bottlenecks.

If you want to rate-limit execution of tasks, you can specify a `target-throughput` (in operations per second). For example, Rally will attempt to run this term query 20 times per second:

```
{
  "operation": "term",
  "target-throughput": 20
}
```

By default, Rally will use a deterministic distribution to determine when to schedule the next operation. This means, that it will execute the term query at 0, 50ms, 100ms, 150ms and so on. Note that the scheduler is aware of the number of clients. Consider this example:

```
{
  "operation": "term",
  "target-throughput": 20,
  "clients": 4
}
```

If Rally would not take the number of clients into account and would still issue requests (from each of the four clients) at the same points in time (i.e. 0, 50ms, 100ms, 150ms, ...), it would run at a target throughput of 4 * 20 = 80 operations per second. Hence, Rally will automatically reduce the rate at which each client will execute requests. Each client will issue requests at 0, 200ms, 400ms, 600ms, 800ms, 1000ms and so on. Each client issues five requests per second but as there are four of them, we still have a target throughput of 20 operations per second. You should keep this in mind, when writing your own custom schedules.

If you want to create a custom scheduler, create a file `track.py` next to `track.json` and implement the following two functions:

```
import random

def random_schedule(current):
    return current + random.randint(10, 900) / 1000.0


def register(registry):
    registry.register_scheduler("my_random", random_schedule)
```

You can then use your custom scheduler as follows:

```
{
  "operation": "term",
  "schedule": "my_random"
}
```

The function `random_schedule` returns a floating point number which represents the next point in time when Rally should execute the given operation. This point in time is measured in seconds relative to the beginning of the execution of this task. The parameter `current` is the last return value of your function and is 0 for the first invocation. So, for example, this scheduler could return the following series: 0, 0.119, 0.622, 1.29, 1.343, 1.984, 2.233. Note that this implementation is usually not sufficient as it does not take into account the number of clients. Therefore, you will typically want to implement a full-blown scheduler which can also take parameters. Below is an example for our random scheduler:

```
import random

class RandomScheduler:
    def __init__(self, params):
        # assume one client by default
        clients = self.params.get("clients", 1)
        # scale accordingly with the number of clients!
        self.lower_bound = clients * self.params.get("lower-bound-millis", 10)
```

```
        self.upper_bound = clients * self.params.get("upper-bound-millis", 900)

    def next(self, current):
        return current + random.randint(self.lower_bound, self.upper_bound) / 1000.0


def register(registry):
    registry.register_scheduler("my_random", RandomScheduler)
```

This implementation will now achieve the same rate independent of the number of clients. Additionally, we can pass the lower and upper bound for the random function from our track:

```
{
    "operation": "term",
    "schedule": "my_random",
    "clients": 4,
    "lower-bound-millis": 50,
    "upper-bound-millis": 250
}
```

Running tasks in parallel

Rally supports running tasks in parallel with the `parallel` element. Below you find a few examples that show how it should be used:

In the simplest case, you let Rally decide the number of clients needed to run the parallel tasks:

```
    {
      "parallel": {
        "warmup-iterations": 1000,
        "iterations": 1000,
        "tasks": [
          {
            "operation": "default",
            "target-throughput": 50
          },
          {
            "operation": "term",
            "target-throughput": 200
          },
          {
            "operation": "phrase",
            "target-throughput": 200
          },
          {
            "operation": "country_agg_uncached",
            "target-throughput": 50
          }
        ]
      }
    }
  ]
}
```

Rally will determine that four clients are needed to run each task in a dedicated client.

However, you can also explicitly limit the number of clients:

```
{
  "parallel": {
    "clients": 2,
    "warmup-iterations": 1000,
    "iterations": 1000,
    "tasks": [
      {
        "operation": "default",
        "target-throughput": 50
      },
      {
        "operation": "term",
        "target-throughput": 200
      },
      {
        "operation": "phrase",
        "target-throughput": 200
      },
      {
        "operation": "country_agg_uncached",
        "target-throughput": 50
      }
    ]
  }
}
```

This will run the four tasks with just two clients. You could also specify more clients than there are tasks but these will then just idle.

You can also specify a number of clients on sub tasks explicitly (by default, one client is assumed per subtask). This allows to define a weight for each client operation. Note that you need to define the number of clients also on the `parallel` parent element, otherwise Rally would determine the number of totally needed clients again on its own:

```
{
  "parallel": {
    "clients": 3,
    "warmup-iterations": 1000,
    "iterations": 1000,
    "tasks": [
      {
        "operation": "default",
        "target-throughput": 50
      },
      {
        "operation": "term",
        "target-throughput": 200
      },
      {
        "operation": "phrase",
        "target-throughput": 200,
        "clients": 2
      },
      {
        "operation": "country_agg_uncached",
        "target-throughput": 50
      }
    ]
  }
```

```
}
```

This will ensure that the phrase query will be executed by two clients. All other ones are executed by one client.

Warning: You cannot nest parallel tasks.

Custom Track Repositories

Rally provides a default track repository that is hosted on Github. You can also add your own track repositories although this requires a bit of additional work. First of all, track repositories need to be managed by git. The reason is that Rally can benchmark multiple versions of Elasticsearch and we use git branches in the track repository to determine the best match for each track. The versioning scheme is as follows:

- The *master* branch needs to work with the latest *master* branch of Elasticsearch.
- All other branches need to match the version scheme of Elasticsearch, i.e. `MAJOR.MINOR.PATCH-SUFFIX` where all parts except `MAJOR` are optional.

Rally implements a fallback logic so you don't need to define a branch for each patch release of Elasticsearch. For example:

- The branch *6.0.0-alpha1* will be chosen for the version `6.0.0-alpha1` of Elasticsearch.
- The branch *5* will be chosen for all versions for Elasticsearch with the major version 5, e.g. `5.0.0`, `5.1.3` (provided there is no specific branch).

Rally tries to use the branch with the best match to the benchmarked version of Elasticsearch.

Creating a new track repository

All track repositories are located in `~/.rally/benchmarks/tracks`. If you want to add a dedicated track repository, called `private` follow these steps:

```
cd ~/.rally/benchmarks/tracks
mkdir private
cd private
git init
# add your track now
git add .
git commit -m "Initial commit"
```

If you want to share your tracks with others you need to add a remote and push it:

```
git remote add origin git@git-repos.acme.com:acme/rally-tracks.git
git push -u origin master
```

If you have added a remote you should also add it in `~/.rally/rally.ini`, otherwise you can skip this step. Open the file in your editor of choice and add the following line in the section `tracks`:

```
private.url = <<URL_TO_YOUR_ORIGIN>>
```

Rally will then automatically update the local tracking branches before the benchmark starts.

You can now verify that everything works by listing all tracks in this track repository:

```
esrally list tracks --track-repository=private
```

This shows all tracks that are available on the `master` branch of this repository. Suppose you only created tracks on the branch `2` because you're interested in the performance of Elasticsearch 2.x, then you can specify also the distribution version:

```
esrally list tracks --track-repository=private --distribution-version=2.0.0
```

Rally will follow the same branch fallback logic as described above.

Adding an already existing track repository

If you want to add a track repository that already exists, just open `~/.rally/rally.ini` in your editor of choice and add the following line in the section `tracks`:

```
your_repo_name.url = <<URL_TO_YOUR_ORIGIN>>
```

After you have added this line, have Rally list the tracks in this repository:

```
esrally list tracks --track-repository=your_repo_name
```

Command Line Reference

You can control Rally with subcommands and command line flags:

- Subcommands determine which task Rally performs.
- Command line flags are used to change Rally's behavior but not all command line flags can be used for each subcommand. To find out which command line flags are supported by a specific subcommand, just run `esrally <<subcommand>> --help`.

Subcommands

race

The `race` subcommand is used to actually run a benchmark. It is the default one and chosen implicitly if none is given.

list

The `list` subcommand is used to list different configuration options:

- telemetry: Will show all *telemetry devices* that are supported by Rally.
- tracks: Will show all tracks that are supported by Rally. As this *may* depend on the Elasticsearch version that you want to benchmark, you can specify `--distribution-version` and also `--distribution-repository` as additional options.
- pipelines: Will show all *pipelines* that are supported by Rally.
- races: Will show all races that are currently stored. This is needed for the *tournament mode*.
- cars: Will show all cars that are supported by Rally (i.e. Elasticsearch configurations).

To list a specific configuration option, place it after the `list` subcommand. For example, `esrally list pipelines` will list all pipelines known to Rally.

compare

This subcommand is needed for *tournament mode* and its usage is described there.

configure

This subcommand is needed to *configure* Rally. It is implicitly chosen if you start Rally for the first time but you can rerun this command at any time.

Command Line Flags

track-repository

Selects the track repository that Rally should use to resolve tracks. By default the `default` track repository is used, which is available in the Github project rally-tracks. See *adding tracks* on how to add your own track repositories.

track

Selects the track that Rally should run. By default the `geonames` track is run. For more details on how tracks work, see *adding tracks*.

challenge

A track consists of one or more challenges. With this flag you can specify which challenge should be run. If you don't specify a challenge, Rally derives the default challenge itself. To see the default challenge of a track, run `esrally list tracks`.

team-repository

Selects the team repository that Rally should use to resolve cars. By default the `default` team repository is used, which is available in the Github project rally-teams. See the documentation about *cars* on how to add your own team repositories.

car

A car defines the Elasticsearch configuration that will be used for the benchmark.

pipeline

Selects the *pipeline* that Rally should run.

Rally can autodetect the pipeline in most cases. If you specify `--distribution-version` it will auto-select the pipeline `from-distribution` otherwise it will use `from-sources-complete`.

laps

Allows to run the benchmark for multiple laps (defaults to 1 lap). Each lap corresponds to one full execution of a track but note that the benchmark candidate is not restarted between laps.

enable-driver-profiling

This option enables a profiler on all operations that the load test driver performs. It is intended to help track authors spot accidental bottlenecks, especially if they implement their own runners or parameter sources. When this mode is enabled, Rally will enable a profiler in the load driver module. After each task and for each client, Rally will add the profile information to a dedicated profile log file. For example:

```
2017-02-09 08:23:24,35 rally.profile INFO
=== Profile START for client [0] and operation [index-append-1000] ===
   16052402 function calls (15794402 primitive calls) in 180.221 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      130    0.001    0.000  168.089    1.293 /Users/dm/Projects/rally/esrally/driver/
↪driver.py:908(time_period_based)
      129    0.260    0.002  168.088    1.303 /Users/dm/.rally/benchmarks/tracks/
↪develop/bottleneck/parameter_sources/bulk_source.py:79(params)
   129000    0.750    0.000  167.791    0.001 /Users/dm/.rally/benchmarks/tracks/
↪develop/bottleneck/parameter_sources/randomevent.py:142(generate_event)
   516000    0.387    0.000  160.485    0.000 /Users/dm/.rally/benchmarks/tracks/
↪develop/bottleneck/parameter_sources/weightedarray.py:20(get_random)
   516000    6.199    0.000  160.098    0.000 /Users/dm/.rally/benchmarks/tracks/
↪develop/bottleneck/parameter_sources/weightedarray.py:23(__random_index)
   516000    1.292    0.000  152.289    0.000 /usr/local/Cellar/python3/3.6.0/
↪Frameworks/Python.framework/Versions/3.6/lib/python3.6/random.py:96(seed)
   516000  150.783    0.000  150.783    0.000 {function Random.seed at 0x10b7fa2f0}
   129000    0.363    0.000   45.686    0.000 /Users/dm/.rally/benchmarks/tracks/
↪develop/bottleneck/parameter_sources/randomevent.py:48(add_fields)
   129000    0.181    0.000   41.742    0.000 /Users/dm/.rally/benchmarks/tracks/
↪develop/bottleneck/parameter_sources/randomevent.py:79(add_fields)
   ....

=== Profile END for client [0] and operation [index-append-1000] ===
```

In this example we can spot quickly that `Random.seed` is called excessively, causing an accidental bottleneck in the load test driver.

test-mode

Allows you to test a track without running it for the whole duration. This mode is only intended for quick sanity checks when creating a track. Please don't rely on these numbers at all (they are meaningless).

If you write your own track, please keep in mind that you need *prepare your track to support this mode*.

telemetry

Activates the provided *telemetry devices* for this race.

Example

```
esrally --telemetry=jfr,jit
```

This activates Java flight recorder and the JIT compiler telemetry devices.

revision

If you actively develop Elasticsearch and want to benchmark a source build of Elasticsearch (which will Rally create for you), you can specify the git revision of Elasticsearch that you want to benchmark. But note that Rally does only support Gradle as build tool which effectively means that it will only support this for Elasticsearch 5.0 or better. The default value is `current`.

You can specify the revision in different formats:

- `--revision=latest`: Use the HEAD revision from origin/master.
- `--revision=current`: Use the current revision (i.e. don't alter the local source tree).
- `--revision=abc123`: Where `abc123` is some git revision hash.
- `--revision=@2013-07-27T10:37:00Z`: Determines the revision that is closest to the provided date. Rally logs to which git revision hash the date has been resolved and if you use Elasticsearch as metrics store (instead of the default in-memory one), *each metric record will contain the git revision hash also in the meta-data section*.

Supported date format: If you specify a date, it has to be ISO-8601 conformant and must start with an `@` sign to make it easier for Rally to determine that you actually mean a date.

distribution-version

If you want to benchmark a binary distribution, you can specify the version here.

Example

```
esrally --distribution-version=2.3.3
```

Rally will then benchmark the official Elasticsearch 2.3.3 distribution.

Rally works with all releases of Elasticsearch that are supported by Elastic.

The following versions are already end-of-life:

- `0.x`: Rally is not tested, and not expected to work for this version; we will make no effort to make Rally work.
- `1.x`: Rally works on a best-effort basis with this version but support may be removed at any time.

Additionally, Rally will always work with the current development version of Elasticsearch (by using either a snapshot repository or by building Elasticsearch from sources).

distribution-repository

Rally does not only support benchmarking official distributions but can also benchmark snapshot builds. This is option is really just intended for our benchmarks that are run in continuous integration but if you want to, you can use it too. The only supported values are `release` (default) and `snapshot`.

Example

```
esrally --distribution-repository=snapshot --distribution-version=6.0.0-
↪SNAPSHOT
```

This will benchmark the latest 6.0.0 snapshot build of Elasticsearch that is available in the Sonatype repository.

report-format

The command line reporter in Rally displays a table with key metrics after a race. With this option you can specify whether this table should be in `markdown` format (default) or `csv`.

report-file

By default, the command line reporter will print the results only on standard output, but can also write it to a file.

Example

```
esrally --report-format=csv --report-file=~/benchmarks/result.csv
```

client-options

With this option you can customize Rally's internal Elasticsearch client.

It accepts a list of comma-separated key-value pairs. The key-value pairs have to be delimited by a colon. These options are passed directly to the Elasticsearch Python client API. See their documentation on a list of supported options.

We support the following data types:

- Strings: Have to be enclosed in single quotes. Example: `ca_certs:'/path/to/CA_certs'`
- Numbers: There is nothing special about numbers. Example: `sniffer_timeout:60`
- Booleans: Specify either `true` or `false`. Example: `use_ssl:true`

In addition to the options, supported by the Elasticsearch client, it is also possible to enable HTTP compression by specifying `compressed:true`

Default value: `timeout:60000,request_timeout:60000`

> **Warning:** If you provide your own client options, the default value will not be magically merged. You have to specify all client options explicitly. The only exceptions to this rule is `ca_cert` (see below).

Examples

Here are a few common examples:

- Enable HTTP compression: `--client-options="compressed:true"`
- Enable SSL (if you have Shield installed): `--client-options="use_ssl:true,verify_certs:true"`. Note that you don't need to set `ca_cert` (which defines the path to the root certificates). Rally does this automatically for you.
- Enable basic authentication: `--client-options="basic_auth_user:'user',basic_auth_password:'password'"`. Please avoid the characters `'`, `,` and `:` in user name and password as Rally's parsing of these options is currently really simple and there is no possibility to escape characters.

target-hosts

If you run the `benchmark-only` *pipeline*, then you can specify a comma-delimited list of hosts:port pairs to which Rally should connect. The default value is `127.0.0.1:9200`.

Example

```
esrally --pipeline=benchmark-only --target-hosts=10.17.0.5:9200,10.17.0.
↪6:9200
```

This will run the benchmark against the hosts 10.17.0.5 and 10.17.0.6 on port 9200. See `client-options` if you use Shield and need to authenticate or Rally should use https.

quiet

Suppresses some output on the command line.

offline

Tells Rally that it should assume it has no connection to the Internet when checking for track data. The default value is `false`. Note that Rally will only assume this for tracks but not for anything else, e.g. it will still try to download Elasticsearch distributions that are not locally cached or fetch the Elasticsearch source tree.

preserve-install

Rally usually installs and launches an Elasticsearch cluster internally and wipes the entire directory after the benchmark is done. Sometimes you want to keep this cluster including all data after the benchmark has finished and that's what you can do with this flag. Note that depending on the track that has been run, the cluster can eat up a very significant amount of disk space (at least dozens of GB). The default value is configurable in the advanced configuration but usually `false`.

Note: This option does only affect clusters that are provisioned by Rally. More specifically, if you use the pipeline `benchmark-only`, this option is ineffective as Rally does not provision a cluster in this case.

---

cluster-health

Rally checks whether the cluster health is "green" before it runs a benchmark against it. The main reason is that we don't want to benchmark a cluster which is shuffling shards around or might start doing so. If you really need to run a benchmark against a cluster that is "yellow" or "red", then you can explicitly override Rally's default behavior. But please think twice before doing so and rather eliminate the root cause.

Example

```
esrally --cluster-health=yellow
```

advanced-config

This flag determines whether Rally should present additional (advanced) configuration options. The default value is `false`.

Example

```
esrally configure --advanced-config
```

assume-defaults

This flag determines whether Rally should automatically accept all values for configuration options that provide a default. This is mainly intended to configure Rally automatically in CI runs. The default value is `false`.

Example

```
esrally configure --assume-defaults=true
```

user-tag

This is only relevant when you want to run *tournaments*. You can use this flag to attach an arbitrary text to the meta-data of each metric record and also the corresponding race. This will help you to recognize a race when you run `esrally list races` as you don't need to remember the concrete timestamp on which a race has been run but can instead use your own descriptive names.

The required format is `key` ":" `value`. You can choose `key` and `value` freely. Note that only one user tag is supported.

Example

```
esrally --user-tag="intention:github-issue-1234-baseline"
```

You can also specify multiple tags. They need to be separated by a comma:

```
**Example**
```

```
esrally --user-tag="disk:SSD,data_node_count:4"
```

When you run `esrally list races`, this will show up again:

```
dm@io:~ $ esrally list races


    ____             ____
   / __ \_____ _/ / /_  __
  / /_/ / __ `/ / / / / / /
 / _, _/ /_/ / / / / /_/ /
/_/ |_|\__,_/_/_/_/\__, /
                  /____/
Recent races:

Race Timestamp    Track    Challenge            Car       User Tag
----------------  -------  -------------------  --------  ----------------------------
↪--------
20160518T122341Z  pmc      append-no-conflicts  defaults  intention:github-issue-1234-
↪baseline
20160518T112341Z  pmc      append-no-conflicts  defaults  disk:SSD,data_node_count:4
```

This will help you recognize a specific race when running `esrally compare`.

Track Reference

Definition

A track is the description of one ore more benchmarking scenarios with a specific document corpus. It defines for example the involved indices, data files and which operations are invoked. Its most important attributes are:

- One or more indices, each with one or more types
- The queries to issue
- Source URL of the benchmark data
- A list of steps to run, which we'll call "challenge", for example indexing data with a specific number of documents per bulk request or running searches for a defined number of iterations.

Tracks are written as JSON files and are kept in a separate track repository, which is located at https://github.com/elastic/rally-tracks. This repository has separate branches for different Elasticsearch versions and Rally will check out the appropriate branch based on the command line parameter `--distribution-version`. If the parameter is missing, Rally will assume by default that you are benchmarking the latest version of Elasticsearch and will checkout the `master` branch of the track repository.

Anatomy of a track

A track JSON file consists of the following sections:

- indices
- operations
- challenges

In the `indices` section you describe the relevant indices. Rally can auto-manage them for you: it can download the associated data files, create and destroy the index and apply the relevant mappings. Sometimes, you may want to have full control over the index. Then you can specify `"auto-managed": false` on an index. Rally will then assume the index is already present. However, there are some disadvantages with this approach. First of all, this can only work if you set up the cluster by yourself and use the pipeline `benchmark-only`. Second, the index is out of control of Rally, which means that you need to keep track for yourself of the index configuration. Third, it does not play nice with the `laps` feature (which you can use to run multiple iterations). Usually, Rally will destroy and recreate all specified indices for each lap but if you use `"auto-managed": false`, it cannot do that. As a consequence it will produce bogus metrics if your track specifies that Rally should run bulk-index operations (as you'll just overwrite existing documents from lap 2 on). So please use extra care if you don't let Rally manage the track's indices.

In the `operations` section you describe which operations are available for this track and how they are parametrized.

In the `challenges` section you describe one or more execution schedules for the operations defined in the `operations` block. Think of it as different scenarios that you want to test for your data set. An example challenge is to index with 2 clients at maximum throughput while searching with another two clients with 10 operations per second.

Track elements

The track elements that are described here are defined in Rally's JSON schema for tracks. Rally uses this track schema to validate your tracks when it is loading them.

Each track defines a three info attributes:

- `description` (mandatory): A human-readable description of the track.

- `short-description` (mandatory): A shorter description of the track.
- `data-url` (optional): A http or https URL that points to the root path where Rally can obtain the corresponding data for this track. This element is not needed if data are only generated on the fly by a custom runner.

Example:

```
{
    "short-description": "Standard benchmark in Rally (8.6M POIs from Geonames)",
    "description": "This test indexes 8.6M documents (POIs from Geonames, total 2.8␣
↪GB json) using 8 client threads and 5000 docs per bulk request against Elasticsearch
↪",
    "data-url": "http://benchmarks.elasticsearch.org.s3.amazonaws.com/corpora/geonames
↪"
}
```

meta

For each track, an optional structure, called `meta` can be defined. You are free which properties this element should contain.

This element can also be defined on the following elements:

- `challenge`
- `operation`
- `task`

If the `meta` structure contains the same key on different elements, more specific ones will override the same key of more generic elements. The order from generic to most specific is:

1. track
2. challenge
3. operation
4. task

E.g. a key defined on a task, will override the same key defined on a challenge. All properties defined within the merged `meta` structure, will get copied into each metrics record.

indices

The `indices` section contains a list of all indices that are used by this track. By default Rally will assume that it can destroy and create these indices at will.

Each index in this list consists of the following properties:

- `name` (mandatory): The name of the index.
- `auto-managed` (optional, defaults to `true`): Controls whether Rally or the user takes care of creating / destroying the index. If this setting is `false`, Rally will neither create nor delete this index but just assume its presence.
- types (optional): A list of types in this index.

Each type consists of the following properties:

- `name` (mandatory): Name of the type.

- `mapping` (mandatory): File name of the corresponding mapping file.
- `documents` (optional): File name of the corresponding documents that should be indexed. If you are using parent-child, specify the number of parent documents. This file has to be compressed either as `.zip`, `.bz2`, `.gz`, `.tar`, `.tar.gz`, `.tgz` or `.tar.bz2` and must contain exactly one JSON file with the same name. The preferred file extension for our official tracks is .bz2.
- `document-count` (mandatory if `documents` is set): Number of documents in the documents file. This number is used by Rally to determine which client indexes which part of the document corpus (each of the N clients gets one N-th of the document corpus).
- `compressed-bytes` (optional but recommended if `documents` is set): The size in bytes of the compressed document file. This number is used to show users how much data will be downloaded by Rally and also to check whether the download is complete.
- `uncompressed-bytes` (optional but recommended if `documents` is set): The size in bytes of the documents file after decompression. This number is used by Rally to show users how much disk space the decompressed file will need and to check that the whole file could be decompressed successfully.

Example:

```
"indices": [
    {
      "name": "geonames",
      "types": [
        {
          "name": "type",
          "mapping": "mappings.json",
          "documents": "documents.json.bz2",
          "document-count": 8647880,
          "compressed-bytes": 197857614,
          "uncompressed-bytes": 2790927196
        }
      ]
    }
]
```

templates

The `indices` section contains a list of all index templates that Rally should create.

- `name` (mandatory): Index template name
- `index-pattern` (mandatory): Index pattern that matches the index template. This must match the definition in the index template file.
- `delete-matching-indices` (optional, defaults to `true`): Delete all indices that match the provided index pattern before start of the benchmark.
- `template` (mandatory): Index template file name

Example:

```
"templates": [
    {
        "name": "my-default-index-template",
        "index-pattern": "my-index-*",
        "delete-matching-indices": true,
        "template": "default-template.json"
```

```
    }
]
```

operations

The `operations` section contains a list of all operations that are available later when specifying challenges. Operations define the static properties of a request against Elasticsearch whereas the `schedule` element defines the dynamic properties (such as the target throughput).

Each operation consists of the following properties:

- `name` (mandatory): The name of this operation. You can choose this name freely. It is only needed to reference the operation when defining schedules.
- `operation-type` (mandatory): Type of this operation. Out of the box, Rally supports the following operation types: `index`, `force-merge`, `index-stats`, `node-stats` and `search`. You can run arbitrary operations however by defining *custom runners*.

Depending on the operation type a couple of further parameters can be specified.

index

With the operation type `index` you can execute bulk requests. It supports the following properties:

- `index` (optional): An index name that defines which indices should be targeted by this indexing operation. Only needed if the `index` section contains more than one index and you don't want to index all of them with this operation.
- `bulk-size` (mandatory): Defines the bulk size in number of documents.
- `batch-size` (optional): Defines how many documents Rally will read at once. This is an expert setting and only meant to avoid accidental bottlenecks for very small bulk sizes (e.g. if you want to benchmark with a bulk-size of 1, you should set batch-size higher).
- `pipeline` (optional): Defines the name of an (existing) ingest pipeline that should be used (only supported from Elasticsearch 5.0).
- `conflicts` (optional): Type of index conflicts to simulate. If not specified, no conflicts will be simulated. Valid values are: 'sequential' (A document id is replaced with a document id with a sequentially increasing id), 'random' (A document id is replaced with a document id with a random other id).
- `action-and-meta-data` (optional): Defines how Rally should handle the action and meta-data line for bulk indexing. Valid values are 'generate' (Rally will automatically generate an action and meta-data line), 'none' (Rally will not send an action and meta-data line) or 'sourcefile' (Rally will assume that the source file contains a valid action and meta-data line).

Example:

```
{
  "name": "index-append",
  "operation-type": "index",
  "bulk-size": 5000
}
```

Throughput will be reported as number of indexed documents per second.

force-merge

With the operation type `force-merge` you can call the force merge API. On older versions of Elasticsearch (prior to 2.1), Rally will use the `optimize API` instead. It supports the following parameter:

- `max_num_segments` (optional) The number of segments the index should be merged into. Defaults to simply checking if a merge needs to execute, and if so, executes it.

Throughput metrics are not necessarily very useful but will be reported in the number of completed force-merge operations per second.

index-stats

With the operation type `index-stats` you can call the indices stats API. It does not support any parameters.

Throughput will be reported as number of completed *index-stats* operations per second.

node-stats

With the operation type `nodes-stats` you can execute nodes stats API. It does not support any parameters.

Throughput will be reported as number of completed *node-stats* operations per second.

search

With the operation type `search` you can execute request body searches. It supports the following properties:

- `index` (optional): An index pattern that defines which indices should be targeted by this query. Only needed if the `index` section contains more than one index. Otherwise, Rally will automatically derive the index to use. If you have defined multiple indices and want to query all of them, just specify `"index": "_all"`.
- `type` (optional): Defines the type within the specified index for this query.
- `cache` (optional): Whether to use the query request cache. By default, Rally will define no value thus the default depends on the benchmark candidate settings and Elasticsearch version.
- `request-params` (optional): A structure containing arbitrary request parameters. The supported parameters names are documented in the Python ES client API docs. Parameters that are implicitly set by Rally (e.g. *body* or *request_cache*) are not supported (i.e. you should not try to set them and if so expect unspecified behavior).
- `body` (mandatory): The query body.
- `pages` (optional): Number of pages to retrieve. If this parameter is present, a scroll query will be executed. If you want to retrieve all result pages, use the value "all".
- `results-per-page` (optional): Number of documents to retrieve per page for scroll queries.

Example:

```
{
  "name": "default",
  "operation-type": "search",
  "body": {
    "query": {
      "match_all": {}
    }
  },
```

```
  "request-params": {
    "_source_include": "some_field",
    "analyze_wildcard": false
  }
}
```

For scroll queries, throughput will be reported as number of retrieved scroll pages per second. The unit is ops/s, where one op(eration) is one page that has been retrieved. The rationale is that each HTTP request corresponds to one operation and we need to issue one HTTP request per result page. Note that if you use a dedicated Elasticsearch metrics store, you can also use other request-level meta-data such as the number of hits for your own analyses.

For other queries, throughput will be reported as number of search requests per second, also measured as ops/s.

challenges

The `challenges` section contains a list of challenges which describe the benchmark scenarios for this data set. It can reference all operations that are defined in the `operations` section.

Each challenge consists of the following properties:

- `name` (mandatory): A descriptive name of the challenge. Should not contain spaces in order to simplify handling on the command line for users.
- `description` (mandatory): A human readable description of the challenge.
- `default` (optional): If true, Rally selects this challenge by default if the user did not specify a challenge on the command line. If your track only defines one challenge, it is implicitly selected as default, otherwise you need define `"default": true` on exactly one challenge.
- `index-settings` (optional): Defines the index settings of the benchmark candidate when an index is created. Note that these settings are only applied if the index is auto-managed.
- `schedule` (mandatory): Defines the concrete execution order of operations. It is described in more detail below.

schedule

The `schedule` element contains a list of tasks that are executed by Rally. Each task consists of the following properties:

- `clients` (optional, defaults to 1): The number of clients that should execute a task concurrently.
- `warmup-iterations` (optional, defaults to 0): Number of iterations that Rally should execute to warmup the benchmark candidate. Warmup iterations will not show up in the measurement results.
- `iterations` (optional, defaults to 1): Number of measurement iterations that Rally executes. The command line report will automatically adjust the percentile numbers based on this number (i.e. if you just run 5 iterations you will not get a 99.9th percentile because we need at least 1000 iterations to determine this value precisely).
- `warmup-time-period` (optional, defaults to 0): A time period in seconds that Rally considers for warmup of the benchmark candidate. All response data captured during warmup will not show up in the measurement results.
- `time-period` (optional): A time period in seconds that Rally considers for measurement. Note that for bulk indexing you should usually not define this time period. Rally will just bulk index all documents and consider every sample after the warmup time period as measurement sample.

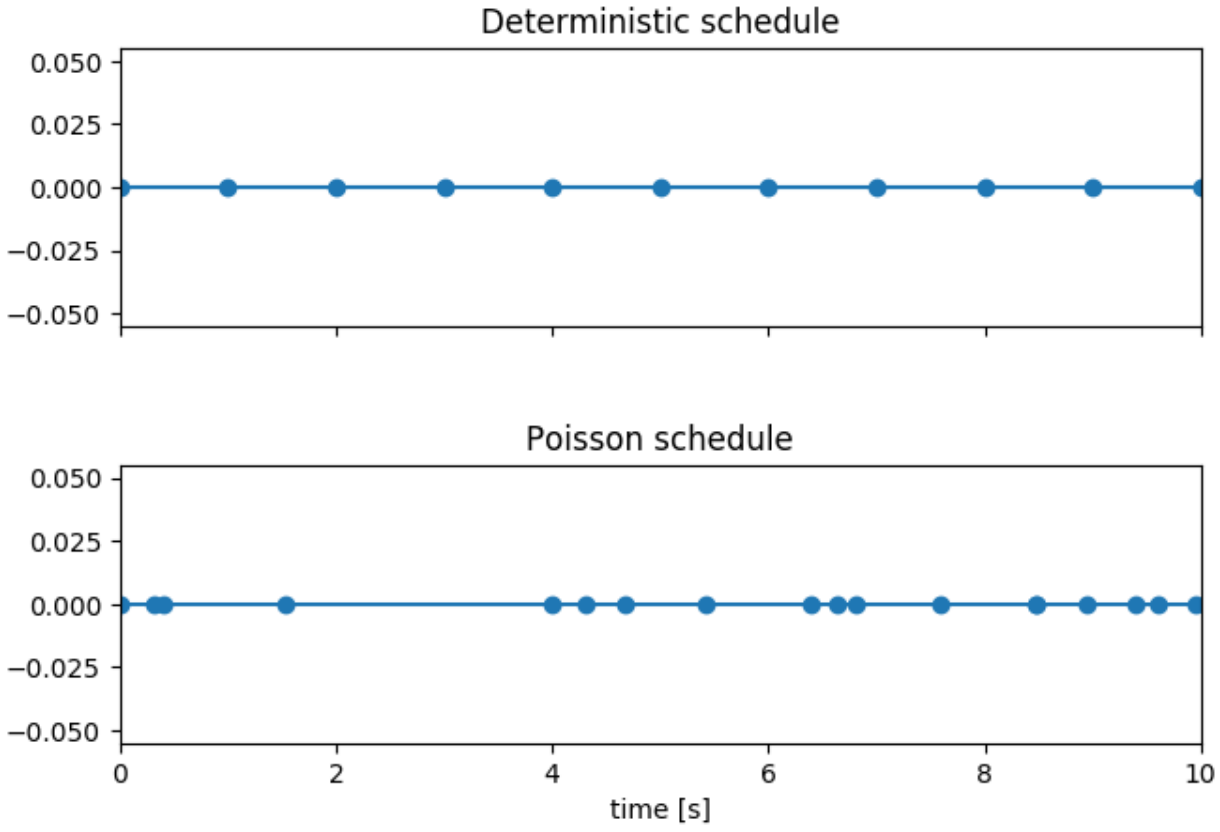
- `schedule` (optional, defaults to `deterministic`): Defines the schedule for this task, i.e. it defines at which point in time during the benchmark an operation should be executed. For example, if you specify a `deterministic` schedule and a target-interval of 5 (seconds), Rally will attempt to execute the corresponding operation at second 0, 5, 10, 15 ... . Out of the box, Rally supports `deterministic` and `poisson` but you can define your own *custom schedules*.
- `target-throughput` (optional): Defines the benchmark mode. If it is not defined, Rally assumes this is a throughput benchmark and will run the task as fast as it can. This is mostly needed for batch-style operations where it is more important to achieve the best throughput instead of an acceptable latency. If it is defined, it specifies the number of requests per second over all clients. E.g. if you specify `target-throughput: 1000` with 8 clients, it means that each client will issue 125 (= 1000 / 8) requests per second. In total, all clients will issue 1000 requests each second. If Rally reports less than the specified throughput then Elasticsearch simply cannot reach it.
- `target-interval` (optional): This is just `1 / target-throughput` (in seconds) and may be more convenient for cases where the throughput is less than one operation per second. Define either `target-throughput` or `target-interval` but not both (otherwise Rally will raise an error).

Choosing a schedule

Rally allows you to choose between the following schedules to simulate traffic:

- deterministically distributed
- Poisson distributed

The diagram below shows how different schedules in Rally behave during the first ten seconds of a benchmark. Each schedule is configured for a (mean) target throughput of one operation per second.



If you want as much reproducibility as possible you can choose the *deterministic* schedule. A Poisson distribution models random independent arrivals of clients which on average match the expected arrival rate which makes it suitable for modelling the behaviour of multiple clients that decide independently when to issue a request. For this reason, Poisson processes play an important role in queueing theory.

If you have more complex needs on how to model traffic, you can also implement a *custom schedule*.

Time-based vs. iteration-based

You should usually use time periods for batch style operations and iterations for the rest. However, you can also choose to run a query for a certain time period.

All tasks in the `schedule` list are executed sequentially in the order in which they have been defined. However, it is also possible to execute multiple tasks concurrently, by wrapping them in a `parallel` element. The `parallel` element defines of the following properties:

- `clients` (optional): The number of clients that should execute the provided tasks. If you specify this property, Rally will only use as many clients as you have defined on the `parallel` element (see examples)!
- `warmup-time-period` (optional, defaults to 0): Allows to define a default value for all tasks of the `parallel` element.
- `time-period` (optional, no default value if not specified): Allows to define a default value for all tasks of the `parallel` element.
- `warmup-iterations` (optional, defaults to 0): Allows to define a default value for all tasks of the `parallel` element.
- `iterations` (optional, defaults to 1): Allows to define a default value for all tasks of the `parallel` element.
- `tasks` (mandatory): Defines a list of tasks that should be executed concurrently. Each task in the list can define the same properties as defined above.

Note: `parallel` elements cannot be nested.

Warning: Specify the number of clients on each task separately. If you specify this number on the `parallel` element instead, Rally will only use that many clients in total and you will only want to use this behavior in very rare cases (see examples)!

Examples

Note that we do not show the operation definition in the examples below but you should be able to infer from the operation name what it is doing.

In this example Rally will run a bulk index operation unthrottled for one hour:

```
"schedule": [
  {
    "operation": "bulk",
    "warmup-time-period": 120,
    "time-period": 3600,
    "clients": 8
  }
]
```

If we want to run a few queries concurrently, we can use the `parallel` element (note how we can define default values on the `parallel` element):

```
"schedule": [
  {
    "parallel": {
      "warmup-iterations": 50,
      "iterations": 100,
      "tasks": [
        {
          "operation": "match-all",
          "clients": 4,
          "target-throughput": 50
        },
        {
          "operation": "term",
          "clients": 2,
          "target-throughput": 200
        },
        {
          "operation": "phrase",
          "clients": 2,
          "target-throughput": 200
        }
      ]
    }
  }
]
```

This schedule will run a match all query, a term query and a phrase query concurrently. It will run with eight clients in total (four for the match all query and two each for the term and phrase query). You can also see that each task can have different settings.

In this scenario, we run indexing and a few queries concurrently with a total of 14 clients:

```
"schedule": [
  {
    "parallel": {
      "tasks": [
        {
          "operation": "bulk",
          "warmup-time-period": 120,
          "time-period": 3600,
          "clients": 8,
          "target-throughput": 50
        },
        {
          "operation": "default",
          "clients": 2,
          "warmup-iterations": 50,
          "iterations": 100,
          "target-throughput": 50
        },
        {
          "operation": "term",
          "clients": 2,
          "warmup-iterations": 50,
          "iterations": 100,
          "target-throughput": 200
```

```
        },
        {
          "operation": "phrase",
          "clients": 2,
          "warmup-iterations": 50,
          "iterations": 100,
          "target-throughput": 200
        }
      ]
    }
  }
]
```

We can also mix sequential tasks with the `parallel` element. In this scenario we are indexing with 8 clients and continue querying with 6 clients after indexing has finished:

```
"schedule": [
  {
    "operation": "bulk",
    "warmup-time-period": 120,
    "time-period": 3600,
    "clients": 8,
    "target-throughput": 50
  },
  {
    "parallel": {
      "warmup-iterations": 50,
      "iterations": 100,
      "tasks": [
        {
          "operation": "default",
          "clients": 2,
          "target-throughput": 50
        },
        {
          "operation": "term",
          "clients": 2,
          "target-throughput": 200
        },
        {
          "operation": "phrase",
          "clients": 2,
          "target-throughput": 200
        }
      ]
    }
  }
]
```

Be aware of the following case where we explicitly define that we want to run only with two clients *in total*:

```
"schedule": [
  {
    "parallel": {
      "warmup-iterations": 50,
      "iterations": 100,
      "clients": 2,
      "tasks": [
```

```
        {
          "operation": "match-all",
          "target-throughput": 50
        },
        {
          "operation": "term",
          "target-throughput": 200
        },
        {
          "operation": "phrase",
          "target-throughput": 200
        }
      ]
    }
  }
]
```

Rally will *not* run all three tasks concurrently because you specified that you want only two clients in total. Hence, Rally will first run "match-all" and "term" concurrently (with one client each). After they have finished, Rally will run "phrase" with one client.

Cars

A rally "car" is a specific configuration of Elasticsearch. You can list the available cars with `esrally list cars`:

```
    _____           ____
   / __ \_____ _/ / /_  __
  / /_/ / __ `/ / / / / / /
 / _, _/ /_/ / / / / /_/ /
/_/ |_|\__,_/_/_/_/\__, /
                  /____/

Available cars:

Name
----------
1gheap
2gheap
4gheap
defaults
verbose_iw
```

You can specify the car that Rally should use with e.g. `--car=4gheap`.

Similar to *custom tracks*, you can also define your own cars.

Warning: This is a rather new feature (introduced in Rally 0.6.2). We want expand it so it can also install and configure plugins as well as multiple nodes the specific format and directory layout may change. Please keep this in mind when doing customizations.

The Anatomy of a car

The default car definitions of Rally are stored in `~/.rally/benchmarks/teams/default/cars`. There we find the following structure:

```
- 1gheap.ini
- 2gheap.ini
- 4gheap.ini
- defaults.ini
- vanilla
|   - config
|       - elasticsearch.yml
|       - jvm.options
|       - log4j2.properties
- verbose_iw
|   - config
|       - elasticsearch.yml
|       - jvm.options
|       - log4j2.properties
- verbose_iw.ini
```

Each `.ini` file in the top level directory defines a car. And each directory (`vanilla` or `verbose_iw`) contains templates for the config files.

Let's have a look at the `1gheap` car by inspecting `1gheap.ini`:

```
[config]
base=vanilla

[variables]
heap_size=1g
```

The name of the car is derived from the ini file name. In the `config` section we define that this definition is based on the `vanilla` configuration. We also define a variable `heap_size` and set it to `1g`.

Let's open `vanilla/config/jvm.options` to see how this variable is used (we'll only show the relevant part here):

```
# Xms represents the initial size of total heap space
# Xmx represents the maximum size of total heap space

-Xms{{heap_size}}
-Xmx{{heap_size}}
```

So Rally reads all variables and the template files and replaces the variables in the final configuration. Note that Rally does not know anything about `jvm.options` or `elasticsearch.yml`. For Rally, these are just plain text templates that need to be copied to the Elasticsearch directory before running a benchmark. Under the hood, Rally uses Jinja2 as template language. This allows you to use Jinja2 expressions in your carr configuration files.

If you open `vanilla/config/elasticsearch.yml` you will see a few variables that are not defined in the `.ini` file:

- `network_host`
- `http_port`
- `node_count_per_host`

These values are derived by Rally internally based on command line flage and you cannot override them in your car definition. You also cannot use these names as names for variables because Rally would simply override them.

Custom Team Repositories

Rally provides a default team repository that is hosted on Github. You can also add your own team repositories although this requires a bit of additional work. First of all, team repositories need to be managed by git. The reason is that Rally can benchmark multiple versions of Elasticsearch and we use git branches in the track repository to determine the best match. The versioning scheme is as follows:

- The *master* branch needs to work with the latest *master* branch of Elasticsearch.
- All other branches need to match the version scheme of Elasticsearch, i.e. `MAJOR.MINOR.PATCH-SUFFIX` where all parts except `MAJOR` are optional.

Rally implements a fallback logic so you don't need to define a branch for each patch release of Elasticsearch. For example:

- The branch *6.0.0-alpha1* will be chosen for the version `6.0.0-alpha1` of Elasticsearch.
- The branch *5* will be chosen for all versions for Elasticsearch with the major version 5, e.g. `5.0.0`, `5.1.3` (provided there is no specific branch).

Rally tries to use the branch with the best match to the benchmarked version of Elasticsearch.

Creating a new team repository

All team repositories are located in `~/.rally/benchmarks/teams`. If you want to add a dedicated team repository, called `private` follow these steps:

```
cd ~/.rally/benchmarks/teams
mkdir private
cd private
git init
# add your team now (don't forget to add the subdirectory "cars").
git add .
git commit -m "Initial commit"
```

If you want to share your teams with others (or you want to run remote benchmarks) you need to add a remote and push it:

```
git remote add origin git@git-repos.acme.com:acme/rally-teams.git
git push -u origin master
```

If you have added a remote you should also add it in `~/.rally/rally.ini`, otherwise you can skip this step. Open the file in your editor of choice and add the following line in the section `teams`:

```
private.url = <<URL_TO_YOUR_ORIGIN>>
```

Rally will then automatically update the local tracking branches before the benchmark starts.

Warning: If you run benchmarks against a remote machine that is under the control of Rally then you need to add the custom team configuration on every node!

You can now verify that everything works by listing all teams in this team repository:

```
esrally list cars --team-repository=private
```

This shows all teams that are available on the `master` branch of this repository. Suppose you only created tracks on the branch `2` because you're interested in the performance of Elasticsearch 2.x, then you can specify also the distribution version:

```
esrally list teams --team-repository=private --distribution-version=2.0.0
```

Rally will follow the same branch fallback logic as described above.

Adding an already existing team repository

If you want to add a team repository that already exists, just open `~/.rally/rally.ini` in your editor of choice and add the following line in the section `teams`:

```
your_repo_name.url = <<URL_TO_YOUR_ORIGIN>>
```

After you have added this line, have Rally list the tracks in this repository:

```
esrally list cars --team-repository=your_repo_name
```

Telemetry Devices

You probably want to gain additional insights from a race. Therefore, we have added telemetry devices to Rally. If you invoke `esrally list telemetry`, it will show which telemetry devices are available:

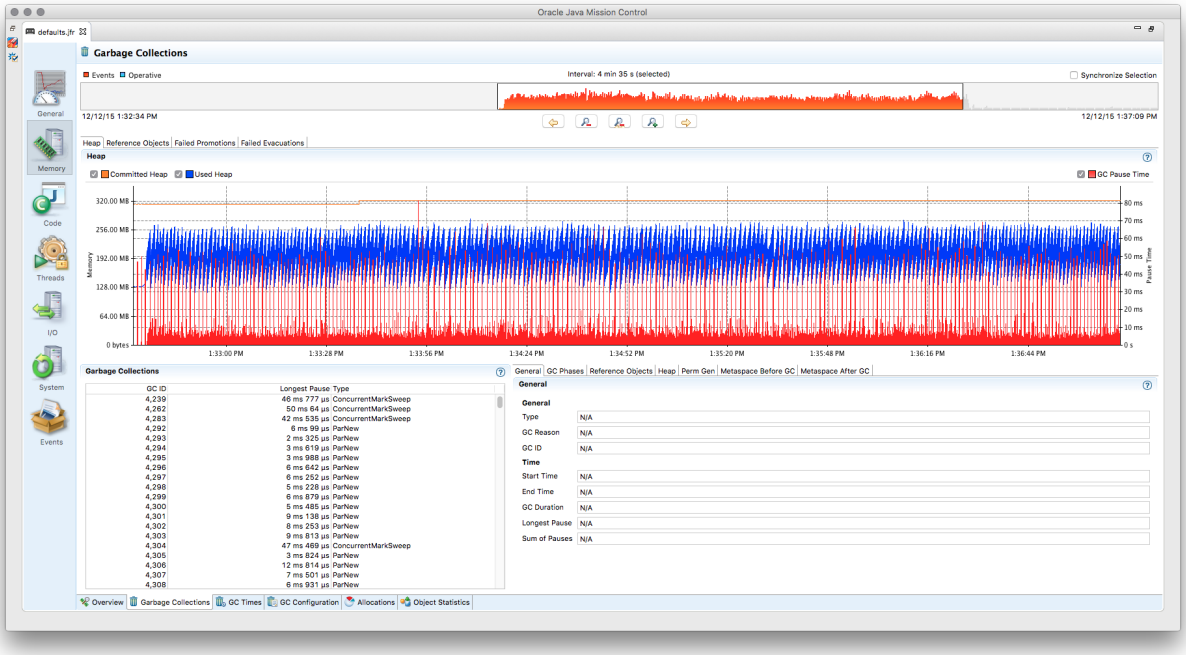
```
dm@io:Projects/rally ‹master*›$ esrally list telemetry


    ____        ____
   / __ \_____ _/ / /_  __
  / /_/ / __ `/ / / / / /
 / _, _/ /_/ / / / / /_/ /
/_/ |_|\__,_/_/_/_/\__, /
                  /____/


Available telemetry devices:

Command    Name                    Description
---------  ----------------------  ------------------------------------------------------
↪-
jit        JIT Compiler Profiler   Enables JIT compiler logs.
gc         GC log                  Enables GC logs.
jfr        Flight Recorder         Enables Java Flight Recorder (requires an Oracle␣
↪JDK)
perf       perf stat               Reads CPU PMU counters (requires Linux and perf)

Keep in mind that each telemetry device may incur a runtime overhead which can skew␣
↪results.
```

You can attach one or more of these telemetry devices to the benchmarked cluster. However, this only works if Rally provisions the cluster (i.e. it does not work with `--pipeline=benchmark-only`).

jfr

The `jfr` telemetry device enables the Java Flight Recorder on the benchmark candidate. Java Flight Recorder ships only with Oracle JDK, so Rally assumes that Oracle JDK is used for benchmarking.

To enable `jfr`, invoke Rally with `esrally --telemetry jfr`. `jfr` will then write a flight recording file which can be opened in Java Mission Control. Rally prints the location of the flight recording file on the command line.



Note: The licensing terms of Java flight recorder do not allow you to run it in production environments without a valid license (for details, please refer to the Oracle Java SE Advanced & Suite Products page). However, running in a QA environment is fine.

jit

The `jit` telemetry device enables JIT compiler logs for the benchmark candidate. If the HotSpot disassembler library is available, the logs will also contain the disassembled JIT compiler output which can be used for low-level analysis. We recommend to use JITWatch for analysis.

`hsdis` can be built for JDK 8 on Linux with (based on a description by Alex Blewitt):

```
curl -O -O -O https://raw.githubusercontent.com/dmlloyd/openjdk/jdk8u/jdk8u/
↪hotspot/src/share/tools/hsdis/{hsdis.c,hsdis.h,Makefile,README}
mkdir -p build/binutils
curl http://ftp.gnu.org/gnu/binutils/binutils-2.27.tar.gz | tar --strip-components=1 -
↪C build/binutils -z -x -f -
make BINUTILS=build/binutils ARCH=amd64
```

After it has been built, the binary needs to be copied to the JDK directory (see `README` of hsdis for details).

gc

The `gc` telemetry device enables GC logs for the benchmark candidate. You can use tools like GCViewer to analyze the GC logs.

perf

The `perf` telemetry device runs `perf stat` on each benchmarked node and writes the output to a log file. It can be used to capture low-level CPU statistics. Note that the perf tool, which is only available on Linux, must be installed before using this telemetry device.

Pipelines

A pipeline is a series of steps that are performed to get benchmark results. This is *not* intended to customize the actual benchmark but rather what happens before and after a benchmark.

An example will clarify the concept: If you want to benchmark a binary distribution of Elasticsearch, Rally has to download a distribution archive, decompress it, start Elasticsearch and then run the benchmark. However, if you want to benchmark a source build of Elasticsearch, it first has to build a distribution with Gradle. So, in both cases, different steps are involved and that's what pipelines are for.

You can get a list of all pipelines with `esrally list pipelines`:

```
Available pipelines:

Name                       Description
-------------------------  ---------------------------------------------------------------
↪---------------------------------
from-distribution          Downloads an Elasticsearch distribution, provisions it, runs␣
↪a benchmark and reports results.
from-sources-complete      Builds and provisions Elasticsearch, runs a benchmark and␣
↪reports results.
benchmark-only             Assumes an already running Elasticsearch instance, runs a␣
↪benchmark and reports results
from-sources-skip-build    Provisions Elasticsearch (skips the build), runs a benchmark␣
↪and reports results.
```

benchmark-only

This is intended if you want to provision a cluster by yourself. Do not use this pipeline unless you are absolutely sure you need to. As Rally has not provisioned the cluster, results are not easily reproducable and it also cannot gather a lot of metrics (like CPU usage).

To benchmark a cluster, you also have to specify the hosts to connect to. An example invocation:

```
esrally --pipeline=benchmark-only --target-hosts=search-node-a.intranet.acme.com:9200,
↪search-node-b.intranet.acme.com:9200
```

from-distribution

This pipeline allows to benchmark an official Elasticsearch distribution which will be automatically downloaded by Rally. The earliest supported version is Elasticsearch 1.7.0. An example invocation:

```
esrally --pipeline=from-distribution --distribution-version=1.7.5
```

The version numbers have to match the name in the download URL path.

You can also benchmark Elasticsearch snapshot versions by specifying the snapshot repository:

```
esrally --pipeline=from-distribution --distribution-version=5.0.0-SNAPSHOT --
↪distribution-repository=snapshot
```

However, this feature is mainly intended for continuous integration environments and by default you should just benchmark official distributions.

Note: This pipeline is just mentioned for completeness but Rally will autoselect it for you. All you need to do is to define the `--distribution-version` flag.

---

from-sources-complete

You should use this pipeline when you want to build and benchmark Elasticsearch from sources. Remember that you also need to install git and Gradle before and Rally needs to be configured for building for sources. If that's not the case you'll get an error and have to run `esrally configure` first. An example invocation:

```
esrally --pipeline=from-sources-complete --revision=latest
```

You have to specify a *revision*.

Note: This pipeline is just mentioned for completeness but Rally will autoselect it for you. All you need to do is to define the `--revision` flag.

---

from-sources-skip-build

This pipeline is similar to `from-sources-complete` except that it assumes you have built the binary once. It saves time if you want to run a benchmark twice for the exact same version of Elasticsearch. Obviously it doesn't make sense to provide a revision: It is always the previously built revision. An example invocation:

```
esrally --pipeline=from-sources-skip-build
```

Metrics

Metrics Records

At the end of a race, Rally stores all metrics records in its metrics store, which is a dedicated Elasticsearch cluster. Rally store the metrics in the indices `rally-metrics-*` and it will create a new index for each month.

Here is a typical metrics record:

```
{
      "environment": "nightly",
      "track": "geonames",
```

```
    "challenge": "append-no-conflicts",
    "car": "defaults",
    "sample-type": "normal",
    "trial-timestamp": "20160421T042749Z",
    "@timestamp": 1461213093093,
    "relative-time": 10507328,
    "name": "throughput",
    "value": 27385,
    "unit": "docs/s",
    "operation": "index-append-no-conflicts",
    "operation-type": "Index",
    "lap": 1,
    "meta": {
      "cpu_physical_cores": 36,
      "cpu_logical_cores": 72,
      "cpu_model": "Intel(R) Xeon(R) CPU E5-2699 v3 @ 2.30GHz",
      "os_name": "Linux",
      "os_version": "3.19.0-21-generic",
      "host_name": "beast2",
      "node_name": "rally-node0",
      "source_revision": "a6c0a81",
      "distribution_version": "5.0.0-SNAPSHOT",
      "tag_reference": "Github ticket 1234",
    }
  }
```

As you can see, we do not only store the metrics name and its value but lots of meta-information. This allows you to create different visualizations and reports in Kibana.

Below we describe each field in more detail.

environment

The environment describes the origin of a metric record. You define this value in the initial configuration of Rally. The intention is to clearly separate different benchmarking environments but still allow to store them in the same index.

track, challenge, car

This is the track, challenge and car for which the metrics record has been produced.

sample-type

Rally runs warmup trials but records all samples. Normally, we are just interested in "normal" samples but for a full picture we might want to look also at "warmup" samples.

trial-timestamp

A constant timestamp (always in UTC) that is determined when Rally is invoked. It is intended to group all samples of a benchmark trial.

@timestamp

The timestamp in milliseconds since epoch determined when the sample was taken.

relative-time

The relative time in microseconds since the start of the benchmark. This is useful for comparing time-series graphs over multiple trials, e.g. you might want to compare the indexing throughput over time across multiple benchmark trials. Obviously, they should always start at the same (relative) point in time and absolute timestamps are useless for that.

name, value, unit

This is the actual metric name and value with an optional unit (counter metrics don't have a unit). Depending on the nature of a metric, it is either sampled periodically by Rally, e.g. the CPU utilization or query latency or just measured once like the final size of the index.

operation, operation-type

`operation` is the name of the operation (as specified in the track file) that ran when this metric has been gathered. It will only be set for metrics with name `latency` and `throughput`.

`operation-type` is the more abstract type of an operation. During a race, multiple queries may be issued which are different `operation``s but they all have the same ``operation-type` (Search). For some metrics, only the operation type matters, e.g. it does not make any sense to attribute the CPU usage to an individual query but instead attribute it just to the operation type.

lap

The lap number in which this metric was gathered. Laps start at 1. See the *command line reference* for more info on laps.

meta

Rally captures also some meta information for each metric record:

- CPU info: number of physical and logical cores and also the model name
- OS info: OS name and version
- Host name
- Node name: If Rally provisions the cluster, it will choose a unique name for each node.
- Source revision: We always record the git hash of the version of Elasticsearch that is benchmarked. This is even done if you benchmark an official binary release.
- Distribution version: We always record the distribution version of Elasticsearch that is benchmarked. This is even done if you benchmark a source release.
- Custom tag: You can define one custom tag with the command line flag `--user-tag`. The tag is prefixed by `tag_` in order to avoid accidental clashes with Rally internal tags.
- Operation-specific: The optional substructure `operation` contains additional information depending on the type of operation. For bulk requests, this may be the number of documents or for searches the number of hits.

Note that depending on the "level" of a metric record, certain meta information might be missing. It makes no sense to record host level meta info for a cluster wide metric record, like a query latency (as it cannot be attributed to a single node).

Metric Keys

Rally stores the following metrics:

- `latency`: Time period between submission of a request and receiving the complete response. It also includes wait time, i.e. the time the request spends waiting until it is ready to be serviced by Elasticsearch.
- `service_time` Time period between start of request processing and receiving the complete response. This metric can easily be mixed up with `latency` but does not include waiting time. This is what most load testing tools refer to as "latency" (although it is incorrect).
- `throughput`: Number of operations that Elasticsearch can perform within a certain time period, usually per second. See the *track reference* for a definition of what is meant by one "operation" for each operation type.
- `merge_parts_total_time_*`: Different merge times as reported by Lucene. Only available if Lucene index writer trace logging is enabled.
- `merge_parts_total_docs_*`: See `merge_parts_total_time_*`
- `disk_io_write_bytes`: number of bytes that have been written to disk during the benchmark. On Linux this metric reports only the bytes that have been written by Elasticsearch, on Mac OS X it reports the number of bytes written by all processes.
- `disk_io_read_bytes`: number of bytes that have been read from disk during the benchmark. The same caveats apply on Mac OS X as for `disk_io_write_bytes`.
- `cpu_utilization_1s`: CPU usage in percent of the Elasticsearch process based on a one second sample period. The maximum value is N * 100% where N is the number of CPU cores available.
- `node_total_old_gen_gc_time`: The total runtime of the old generation garbage collector across the whole cluster as reported by the node stats API.
- `node_total_young_gen_gc_time`: The total runtime of the young generation garbage collector across the whole cluster as reported by the node stats API.
- `segments_count`: Total number of segments as reported by the indices stats API.
- `segments_memory_in_bytes`: Number of bytes used for segments as reported by the indices stats API.
- `segments_doc_values_memory_in_bytes`: Number of bytes used for doc values as reported by the indices stats API.
- `segments_stored_fields_memory_in_bytes`: Number of bytes used for stored fields as reported by the indices stats API.
- `segments_terms_memory_in_bytes`: Number of bytes used for terms as reported by the indices stats API.
- `segments_norms_memory_in_bytes`: Number of bytes used for norms as reported by the indices stats API.
- `segments_points_memory_in_bytes`: Number of bytes used for points as reported by the indices stats API.
- `merges_total_time`: Total runtime of merges as reported by the indices stats API. Note that this is not Wall clock time (i.e. if M merge threads ran for N minutes, we will report M * N minutes, not N minutes).
- `merges_total_throttled_time`: Total time within merges have been throttled as reported by the indices stats API. Note that this is not Wall clock time.
- `indexing_total_time`: Total time used for indexing as reported by the indices stats API. Note that this is not Wall clock time.

- `refresh_total_time`: Total time used for index refresh as reported by the indices stats API. Note that this is not Wall clock time.
- `flush_total_time`: Total time used for index flush as reported by the indices stats API. Note that this is not Wall clock time.
- `final_index_size_bytes`: Final resulting index size after the benchmark.

Summary Report

At the end of each *race*, Rally shows a summary report. Below we'll explain the meaning of each line including a reference to its corresponding *metrics key* which can be helpful if you want to build your own reports in Kibana. Note that not every summary report will show all lines.

Indexing time

- **Definition**: Total time used for indexing as reported by the indices stats API. Note that this is not Wall clock time (i.e. if M indexing threads ran for N minutes, we will report M * N minutes, not N minutes).
- **Corresponding metrics key**: `indexing_total_time`

Merge time

- **Definition**: Total runtime of merges as reported by the indices stats API. Note that this is not Wall clock time.
- **Corresponding metrics key**: `merges_total_time`

Refresh time

- **Definition**: Total time used for index refresh as reported by the indices stats API. Note that this is not Wall clock time.
- **Corresponding metrics key**: `refresh_total_time`

Flush time

- **Definition**: Total time used for index flush as reported by the indices stats API. Note that this is not Wall clock time.
- **Corresponding metrics key**: `flush_total_time`

Merge throttle time

- **Definition**: Total time within merges have been throttled as reported by the indices stats API. Note that this is not Wall clock time.
- **Corresponding metrics key**: `merges_total_throttled_time`

Merge time (x)

Where `x` is one of:

- postings
- stored fields
- doc values
- norms
- vectors
- points
- **Definition**: Different merge times as reported by Lucene. Only available if Lucene index writer trace logging is enabled (use the car *verbose_iw* for that).
- **Corresponding metrics keys**: `merge_parts_total_time_*`

Median CPU usage

- **Definition**: Median CPU usage in percent of the Elasticsearch process during the whole race based on a one second sample period. The maximum value is N * 100% where N is the number of CPU cores available
- **Corresponding metrics key**: `cpu_utilization_1s`

Total Young Gen GC

- **Definition**: The total runtime of the young generation garbage collector across the whole cluster as reported by the node stats API.
- **Corresponding metrics key**: `node_total_young_gen_gc_time`

Total Old Gen GC

- **Definition**: The total runtime of the old generation garbage collector across the whole cluster as reported by the node stats API.
- **Corresponding metrics key**: `node_total_old_gen_gc_time`

Index size

- **Definition**: Final resulting index size after the benchmark.
- **Corresponding metrics key**: `final_index_size_bytes`

Totally written

- **Definition**: number of bytes that have been written to disk during the benchmark. On Linux this metric reports only the bytes that have been written by Elasticsearch, on Mac OS X it reports the number of bytes written by all processes.
- **Corresponding metrics key**: `disk_io_write_bytes`

Heap used for x

Where `X` is one of:

- doc values
- terms
- norms
- points
- stored fields
- **Definition**: Number of bytes used for the corresponding item as reported by the indices stats API.
- **Corresponding metrics keys**: `segments_*_in_bytes`

Segment count

- **Definition**: Total number of segments as reported by the indices stats API.
- **Corresponding metrics key**: `segments_count`

Throughput

Rally reports the minimum, median and maximum throughput for each operation.

- **Definition**: Number of operations that Elasticsearch can perform within a certain time period, usually per second.
- **Corresponding metrics key**: `throughput`

Latency

Rally reports several percentile numbers for each operation. Which percentiles are shown depends on how many requests Rally could capture (i.e. Rally will not show a 99.99th percentile if it could only capture five samples because that would be a vanity metric).

- **Definition**: Time period between submission of a request and receiving the complete response. It also includes wait time, i.e. the time the request spends waiting until it is ready to be serviced by Elasticsearch.
- **Corresponding metrics key**: `latency`

Service time

Rally reports several percentile numbers for each operation. Which percentiles are shown depends on how many requests Rally could capture (i.e. Rally will not show a 99.99th percentile if it could only capture five samples because that would be a vanity metric).

- **Definition**: Time period between start of request processing and receiving the complete response. This metric can easily be mixed up with `latency` but does not include waiting time. This is what most load testing tools refer to as "latency" (although it is incorrect).
- **Corresponding metrics key**: `service_time`

Error rate

- **Definition**: The ratio of erroneous responses relative to the total number of responses. Any exception thrown by the Python Elasticsearch client is considered erroneous (e.g. HTTP response codes 4xx, 5xx or network errors (network unreachable)). For specific details, please check the reference documentation of the Elasticsearch client. Usually any error rate greater than zero is alerting. You should investigate the root cause by inspecting Rally and Elasticearch logs and rerun the benchmark.
- **Corresponding metrics key**: `service_time`. Each `service_time` record has a `meta.success` flag. Rally simply counts how often this flag is `true` and `false` respectively.

Frequently Asked Questions (FAQ)

A benchmark aborts with `Couldn't find a tar.gz distribution`. What's the problem?

This error occurs when Rally cannot build an Elasticsearch distribution from source code. The most likely cause is that there is some problem in the build setup.

To see what's the problem, try building Elasticsearch yourself. First, find out where the source code is located (run `grep local.src.dir ~/.rally/rally.ini`). Then change to this directory and run the following commands:

```
gradle clean
gradle :distribution:tar:assemble
```

By that you are mimicking what Rally does. Fix any errors that show up here and then retry.

Where does Rally get the benchmark data from?

Rally comes with a set of tracks out of the box which we maintain in the rally-tracks repository on Github. This repository contains the track descriptions. The actual data are stored as compressed files in an S3 bucket.

Will Rally destroy my existing indices?

First of all: Please (please, please) do NOT run Rally against your production cluster if you are just getting started with it. You have been warned.

Depending on the track, Rally will delete and create one or more indices. For example, the geonames track specifies that Rally should create an index named "geonames" and Rally will assume it can do to this index whatever it wants. Specifically, Rally will check at the beginning of a race if the index "geonames" exists and delete it. After that it creates a new empty "geonames" index and runs the benchmark. So if you benchmark against your own cluster (by specifying the `benchmark-only` *pipeline*) and this cluster contains an index that is called "geonames" you will lose (all) data if you run Rally against it. Rally will neither read nor write (or delete) any other index. So if you apply the usual care nothing bad can happen.

What does *latency* and *service_time* mean and how do they related to the *took* field that Elasticsearch returns?

Let's start with the *took* field of Elasticsearch. *took* is the time needed by Elasticsearch to process a request. As it is determined on the server, it can neither include the time it took the client to send the data to Elasticsearch nor the time

it took Elasticsearch to send it to the client. This time is captured by *service_time*, i.e. it is the time period from the start of a request (on the client) until it has received the response.

The explanation of *latency* is a bit more involved. First of all, Rally defines two benchmarking modes:

- Throughput benchmarking mode: In this mode, Rally will issue requests as fast as it can, i.e. as soon as it receives a response, it will issue the next request. This is ideal for benchmarking indexing. In this mode `latency == service_time`.
- Throughput-throttled mode: If you define a specific target throughput rate in a track, for example 100 requests per second (you should choose this number based on the traffic pattern that you experience in your production environment), then Rally will define a schedule internally and will issue requests according to this schedule regardless how fast Elasticsearch can respond. To put it differently: Imagine you want to grab a coffee on your way to work. You make this decision independently of all the other people going to the coffee shop so it is possible that you need to wait before you can tell the barista which coffee you want. The time it takes the barista to make your coffee is the service time. The service time is independent of the number of customers in the coffee shop. However, you as a customer also care about the length of the waiting line which depends on the number of customers in the coffee shop. The time it takes between you entering the coffee shop and taking your first sip of coffee is latency.

If you are interested in latency measurement, we recommend you watch the following talks:

"How NOT to Measure Latency" by Gil Tene:

Benchmarking Elasticsearch with Rally by Daniel Mitterdorfer:

Where and how long does Rally keep its data?

Rally stores a lot of data (this is just the nature of a benchmark) so you should keep an eye on disk usage. All data are kept in `~/.rally` and Rally does not implicitly delete them. These are the most important directories:

- `~/.rally/logs`: Contains all log files. Logs are rotated daily. If you don't need the logs anymore, you can safely wipe this directory.
- `~/.rally/benchmarks/races`: telemetry data, Elasticsearch logs and even complete Elasticsearch installations including the data directory if a benchmark failed. If you don't need the data anymore, you can safely wipe this directory.
- `~/.rally/benchmarks/src`: the Elasticsearch Github repository (only if you had Rally build Elasticsearch from sources at least once).
- `~/.rally/benchmarks/data`: the benchmark data sets. This directory can get very huge (way more than 100 GB if you want to try all default tracks). You can delete the files in this directory but keep in mind that Rally may needs to download them again.
- `~/.rally/benchmarks/distributions`: Contains all downloaded Elasticsearch distributions.

There are a few more directories but the ones above are the most disk-hogging ones.

Does Rally spy on me?

No. Rally does not collect or send any usage data and also the complete source code is open. We do value your feedback a lot though and if you got any ideas for improvements, found a bug or have any other feedback, please head over to Rally's Discuss forum or raise an issue on Github.

Do I need an Internet connection?

You do NOT need Internet access on any node of your Elasticsearch cluster but the machine where you start Rally needs an Internet connection to download track data sets and Elasticsearch distributions. After it has downloaded all data, an Internet connection is not required anymore and you can specify `--offline`. If Rally detects no active Internet connection, it will automatically enable offline mode and warn you.

As a workaround you can also download all data on a different machine and transfer it the the target machine but this is not actively supported.

Glossary

track A *track* is the description of one ore more benchmarking scenarios with a specific document corpus. It defines for example the involved indices, data files and which operations are invoked. List the available tracks with `esrally list tracks`. Although Rally ships with some tracks out of the box, you should usually *create your own track* based on your own data.

challenge A challenge describes one benchmarking scenario, for example indexing documents at maximum throughput with 4 clients while issuing term and phrase queries from another two clients rate-limited at 10 queries per second each. It is always specified in the context of a track. See the available challenges by listing the corresponding tracks with `esrally list tracks`.

car A *car* is a specific configuration of an Elasticsearch cluster that is benchmarked, for example the out-of-the-box configuration, a configuration with a specific heap size or a custom logging configuration. List the available cars with `esrally list cars`.

telemetry *Telemetry* is used in Rally to gather metrics about the car, for example CPU usage or index size.

race A *race* is one invocation of the Rally binary. Another name for that is one "benchmarking trial". During a race, Rally runs one challenge on a track with the given car.

tournament A tournament is a comparison of two races. You can use Rally's *tournament mode* for that.

Community Resources

Below are a few community resources about Rally. If you find an interesting article, talk or custom tracks, please raise an issue or open a pull request.

Talks

Articles

Using Rally to benchmark Elasticsearch queries by Darren Smith

CHAPTER 3

License

Index