Rally Documentation

Release 0.4.6

Daniel Mitterdorfer

December 07, 2016

Contents

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.

CHAPTER 3

Contents

3.1 Quickstart

3.1.1 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.

3.1.2 Configure

Just invoke `esrally configure`.

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

3.1.3 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)
----------------------------------
```

3.1.4 Next steps

Now you can check how to run benchmarks or how to define your own benchmarks.

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

3.2 Installation

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

3.2.1 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)
- `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.

3.2.2 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.

3.2.3 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`.

3.2.4 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 `/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.

3.2.5 Next Steps

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

3.3 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.

Note: If you get the error `UnicodeEncodeError: 'ascii' codec can't encode character`, please configure your shell so it supports UTF-8 encoding. You can check the output of `locale` which should show UTF-8 as sole encoding. If in doubt, add `export LC_ALL=en_US.UTF-8` to your shell init file (e.g. `~/.bashrc` if you use Bash) and relogin.

3.3.1 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 2.13 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    : []
  gradle : []
  JDK 8  : []

[] 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    : []
  gradle : []
  JDK 8  : []

************************************************************************************
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.

3.3.2 Advanced Configuration

If you need more control over a few variables or want to use advanced features like tournaments, 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 2.3 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:

- 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.

3.3.3 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 enviro
```

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

3.4 Running Races

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

3.4.1 List Tracks

Start by finding out which tracks are available:

```
esrally list tracks
```

This will show the following list:

```
Name        Description                                                          Challenges
----------  -------------------------------------------------------------------  -------------------
geonames    Standard benchmark in Rally (8.6M POIs from Geonames)                append-no-conflicts
geopoint    60.8M POIs from PlanetOSM                                            append-no-conflicts
logging     Logging benchmark                                                    append-no-conflicts
nyc_taxis   Trip records completed in yellow and green taxis in New York in 2015 append-no-conflicts
percolator  Percolator benchmark based on 2M AOL queries                         append-no-conflicts
pmc         Full text benchmark containing 574.199 papers from PMC               append-no-conflicts
tiny        First 2k documents of the geonames track for local tests             append-no-conflicts
```

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.

3.4.2 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-conflict


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

[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/d
[INFO] Decompressing track data from [/Users/dm/.rally/benchmarks/data/geopoint/documents.json.bz2] t
[INFO] Preparing file offset table for [/Users/dm/.rally/benchmarks/data/geopoint/documents.json] ...
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.

3.5 Creating custom tracks

3.5.1 Overview

First of all we need to clarify what a benchmark is. Rally has a few assumptions built-in:

1. Rally sets up a fresh Elasticsearch cluster, i.e. the cluster is entirely under Rally's control.

2. The first step of the benchmark is to index all required documents via the bulk API. Rally will measure metrics like indexing throughput in this phase.
3. An optional second step is to run one or more queries against the index. Rally will measure metrics like query latency in this phase.

This is called a "track" in Rally. The most important attributes of a track 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.

Separately from a track, we also have "cars" which define the settings of the benchmark candidate (Elasticsearch), like how much heap memory to use, the number of nodes to start and so on. Rally comes with a set of default tracks and cars which you can use for your own benchmarks (but you don't have to).

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.

3.5.2 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 parameterized.

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.

3.5.3 Example track

Let's create an example track step by step. 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
import csv

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'))

with open('allCountries.txt') as f:
 while True:
   line = f.readline()
   if line == '':
     break
   tup = line.strip().split('\t')
   d = {}
   for i in range(len(cols)):
     name, type = cols[i]
     if tup[i] != '':
       if type in ('int', 'long'):
         d[name] = int(tup[i])
       elif type == 'double':
         d[name] = float(tup[i])
       else:
         d[name] = tup[i]

   print(json.dumps(d))
```

We can invoke the script with `python3 toJSON.py > documents.json`.

Next we need to compress the JSON file with `bzip2 -9 -c documents.json > documents.json.bz2`. Upload the data file to a place where it is publicly available. We choose `http://benchmarks.elastic.co/corpora/geonames` for this example.

For initial local testing you can place the data file in Rally's data directory, which is located in `~/.rally/benchmarks/data`. For this example you need to place the data for the "geonames" track in `~/.rally/benchmarks/data/geonames` so Rally can pick it up. Additionally, you have to specify the `--offline` option when running Rally so it does not try to download any benchmark data.

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 the 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/geonames` for this track.

The track repository is managed by git, so 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

```
{
  "meta": {
    "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
    "data-url": "http://benchmarks.elasticsearch.org.s3.amazonaws.com/corpora/geonames"
  },
  "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",
      "type": "index",
      "bulk-size": 5000
    },
    {
      "name": "force-merge",
      "type": "force-merge"
    },
    {
      "name": "query-match-all",
      "operation-type": "search",
      "body": {
        "query": {
          "match_all": {}
        }
      }
    },
  ],
  "challenges": [
    {
      "name": "append-no-conflicts",
      "description": "",
      "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 geonames track"`.

A few things to note:

- Rally assumes that the challenge that should be run by default is called "append-no-conflicts". 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: 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                                                 Challenges
----------   ---------------------------------------------------------   -------------------
geonames     Standard benchmark in Rally (8.6M POIs from Geonames)       append-no-conflicts
```

Congratulations, you have created your first track! You can test it with `esrally --track=geonames --offline` (or whatever the name of your track is) and run specific challenges with `esrally --track=geonames --challenge=append-fast-with-conflicts --offline`.

If you want to share your track with the community, please read on.

3.5.4 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.
3. Create a pull request in the rally-tracks Github repo.

3.5.5 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.

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.

Custom runners

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`.

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

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 commit -a -m "Initial commit"
```

If you also have a remote for this repository, open `~/.rally/rally.ini` in your editor of choice and add the following line in the section `tracks`, otherwise just skip this step:

```
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
```

3.6 Cars

A rally "car" is a specific configuration of Elasticsearch. At the moment, Rally ships with a fixed set of cars that you cannot change. You can list them with `esrally list cars`:

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

Available cars:

Name
----------
defaults
4gheap
two_nodes
verbose_iw
```

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

3.7 Frequently Asked Questions (FAQ)

3.7.1 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.

3.7.2 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.

3.7.3 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.

3.7.4 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/benchmarks/races`: logs, telemetry data or even complete Elasticsearch installations including the data directory if a benchmark failed. If you don't need the logs 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 four above are the most disk-hogging ones.

3.7.5 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.

3.7.6 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.

3.8 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:

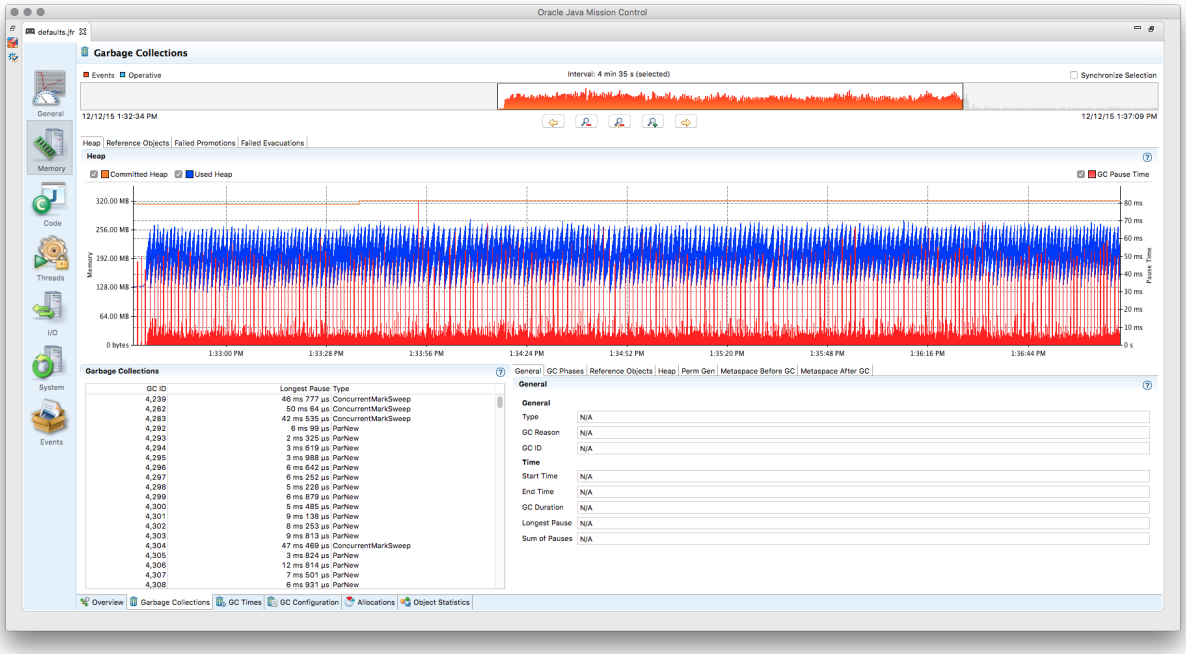
```
dm@io:Projects/rally ‹master*›$ esrally list telemetry


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


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`).

3.8.1 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.

3.8.2 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 -O https://raw.githubusercontent.com/dmlloyd/openjdk/jdk8u/jdk8u/hotspot/src/share/tool
mkdir -p build/binutils
curl http://ftp.gnu.org/gnu/binutils/binutils-2.27.tar.gz | tar --strip-components=1 -C build/binutil
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).

3.8.3 gc

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

3.8.4 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.

3.9 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
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 repo
from-sources-skip-build    Provisions Elasticsearch (skips the build), runs a benchmark and reports res
```

3.9.1 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.i
```

3.9.2 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=
```

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.

3.9.3 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.

3.9.4 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
```

3.10 Metrics

3.10.1 Metrics Records

At the end of a race, Rally stores all metrics records in its metrics store, which is a dedicated Elasticsearch cluster.

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).

3.10.2 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.
- `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.

3.11 Tournaments

Warning: If you want to use tournaments, then Rally requires a dedicated metrics store as it needs to store data across multiple races. So ensure to run `esrally configure --advanced-config` first. For details please see the configuration help page.

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

3.12 Recipes

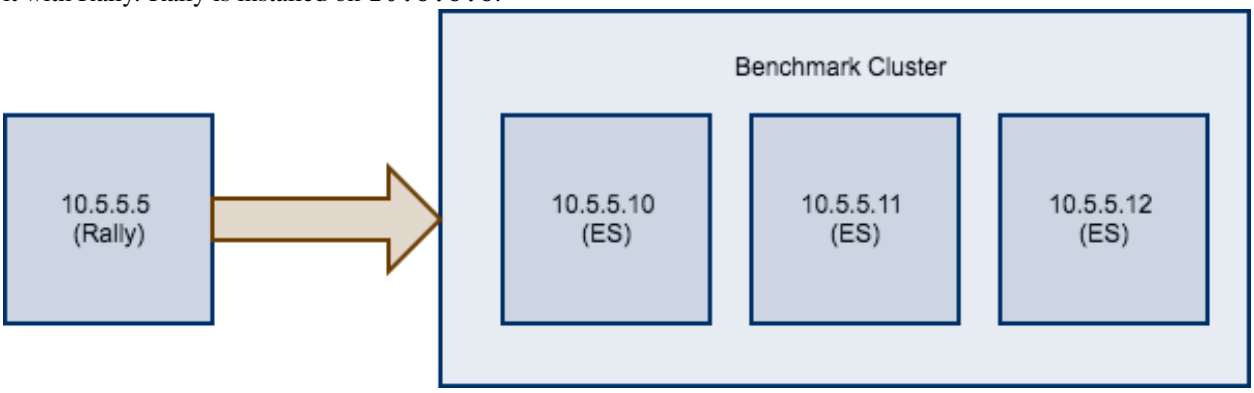
3.12.1 Benchmarking an existing cluster

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                                                        Challenges
----------  -----------------------------------------------------------------  ----------
geonames    Standard benchmark in Rally (8.6M POIs from Geonames)              append-no-conflicts
geopoint    60.8M POIs from PlanetOSM                                          append-no-conflicts
logging     Logging benchmark                                                  append-no-conflicts
nyc_taxis   Trip records completed in yellow and green taxis in New York in 2015  append-no-conflicts
percolator  Percolator benchmark based on 2M AOL queries                       append-no-conflicts
pmc         Full text benchmark containing 574.199 papers from PMC             append-no-conflicts
```

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-
```

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-
```

3.13 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`.

3.13.1 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 only needed for the tournament mode and it will also only work if you have setup Rally so it supports tournaments.
- 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.

3.13.2 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 on Github. 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.

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.

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.

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:'passw`
  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.

---

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.

Example

```
esrally --user-tag=github-issue-1234-baseline
```

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  github-issue-1234-baseline
```

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

3.14 Developing Rally

3.14.1 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 2.13

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

3.14.2 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.

3.14.3 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.

3.14.4 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.

3.15 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.

CHAPTER 4

License

C

R

T