Elasticsearch Microbenchmark Suite

This directory contains the microbenchmark suite of Elasticsearch. It relies on JMH.

Purpose

We do not want to microbenchmark everything but the kitchen sink and should typically rely on our macrobenchmarks with Rally. Microbenchmarks are intended to spot performance regressions in performance-critical components. The microbenchmark suite is also handy for ad-hoc microbenchmarks, but please remove them again before merging your PR.

Getting Started

Just run `gradlew -p benchmarks run` from the project root directory. It will build all microbenchmarks, execute them and print the result.

Running Microbenchmarks

Running via an IDE is not supported as the results are meaningless because we have no control over the JVM running the benchmarks.

If you want to run a specific benchmark class like, say, `MemoryStatsBenchmark`, you can use `--args`:

```
gradlew -p benchmarks run --args 'MemoryStatsBenchmark'
```

Everything in the `'` gets sent on the command line to JMH.

You can set benchmark parameters with `-p`:

```
gradlew -p benchmarks/ run --args 'RoundingBenchmark.round -prounder=es -prange="2000-10-01 to 2000-11-01" -pzone=America/New_York -pinterval=10d -pcount=1000000'
```

The benchmark code defines default values for the parameters, so if you leave any out JMH will run with each default value, one after the other. This will run with `interval` set to `calendar year` then `calendar hour` then `10d` then `5d` then `1h`:

```
gradlew -p benchmarks/ run --args 'RoundingBenchmark.round -prounder=es -prange="2000-10-01 to 2000-11-01" -pzone=America/New_York -pcount=1000000'
```

Adding Microbenchmarks

Before adding a new microbenchmark, make yourself familiar with the JMH API. You can check our existing microbenchmarks and also the JMH samples.

In contrast to tests, the actual name of the benchmark class is not relevant to JMH. However, stick to the naming convention and end the class name of a benchmark with `Benchmark`. To have JMH execute a benchmark, annotate the respective methods with `@Benchmark`.

Tips and Best Practices

To get realistic results, you should exercise care when running benchmarks. Here are a few tips:

Do

- Ensure that the system executing your microbenchmarks has as little load as possible. Shutdown every process that can cause unnecessary runtime jitter. Watch the `Error` column in the benchmark results to see the run-to-run variance.
- Ensure to run enough warmup iterations to get the benchmark into a stable state. If you are unsure, don't change the defaults.
- Avoid CPU migrations by pinning your benchmarks to specific CPU cores. On Linux you can use `taskset`.
- Fix the CPU frequency to avoid Turbo Boost from kicking in and skewing your results. On Linux you can use `cpufreq-set` and the `performance` CPU governor.
- Vary the problem input size with `@Param`.
- Use the integrated profilers in JMH to dig deeper if benchmark results do not match your hypotheses:
  - Add `-prof gc` to the options to check whether the garbage collector runs during a microbenchmark and skews your results. If so, try to force a GC between runs (`-gc true`) but watch out for the caveats.
  - Add `-prof perf` or `-prof perfasm` (both only available on Linux, see Disassembling below) to see hotspots.
  - Add `-prof async` to see hotspots.
- Have your benchmarks peer-reviewed.

Don't

- Blindly believe the numbers that your microbenchmark produces but verify them by measuring e.g. with `-prof perfasm`.
- Run more threads than your number of CPU cores (in case you run multi-threaded microbenchmarks).
- Look only at the `Score` column and ignore `Error`. Instead, take countermeasures to keep `Error` low / variance explainable.

Disassembling

NOTE: Linux only. Sorry Mac and Windows.

Disassembling is fun! Maybe not always useful, but always fun! Generally, you'll want to install `perf` and FCML's `hsdis`. `perf` is generally available via `apg-get install perf` or `pacman -S perf`. FCML is a little more involved. This worked on 2020-08-01:

```
wget https://github.com/swojtasiak/fcml-lib/releases/download/v1.2.2/fcml-1.2.2.tar.gz
tar xf fcml*
cd fcml*
./configure
make
cd example/hsdis
make
sudo cp .libs/libhsdis.so.0.0.0 /usr/lib/jvm/java-14-adoptopenjdk/lib/hsdis-amd64.so
```

If you want to disassemble a single method do something like this:

```
gradlew -p benchmarks run --args ' MemoryStatsBenchmark -jvmArgs "-
XX:+UnlockDiagnosticVMOptions -XX:CompileCommand=print,*.yourMethodName -
XX:PrintAssemblyOptions=intel"
```

If you want `perf` to find the hot methods for you, then do add `-prof perfasm`.

Async Profiler

Note: Linux and Mac only. Sorry Windows.

IMPORTANT: The 2.0 version of the profiler doesn't seem to be compatible with JMH as of 2021-04-30.

The async profiler is neat because it does not suffer from the safepoint bias problem. And because it makes pretty flame graphs!

Let user processes read performance stuff:

```
sudo bash
echo 0 > /proc/sys/kernel/kptr_restrict
echo 1 > /proc/sys/kernel/perf_event_paranoid
exit
```

Grab the async profiler from https://github.com/jvm-profiling-tools/async-profiler and run `prof async` like so:

```
gradlew -p benchmarks/ run --args 'LongKeyedBucketOrdsBenchmark.multiBucket -prof
"async:libPath=/home/nik9000/Downloads/tmp/async-profiler-1.8.3-linux-
x64/build/libasyncProfiler.so;dir=/tmp/prof;output=flamegraph"'
```

If you are on Mac, this'll warn you that you downloaded the shared library from the internet. You'll need to go to settings and allow it to run.

The profiler tells you it'll be more accurate if you install debug symbols with the JVM. I didn't, and the results looked pretty good to me. (2021-02-01)