GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

HPS

Seminar Report

MPI-based Creation and Benchmarking of a Dynamic Elasticsearch Cluster

Lars Quentin

MatrNr: 21774184

Supervisor: Hendrik Nolte

Georg-August-Universität Göttingen
Institute of Computer Science

July 12, 2024

Abstract

Due to current developments in simulation, data science, and machine learning, research has become ever more data-driven and compute-intensive. Especially, data lakes are evermore important, due to the decreased prices in storage with less specialized and more heterogeneous hardware support. To manage the sheer amount of raw, unprocessed data, a both sufficient and performant metadata management solution is paramount.

After an internal performance evaluation of Elasticsearch for data lake metadata management at the GWDG, it was found out that rally [1], Elastic's official benchmarker, does not scale well enough to simulate a realistic High-Performance Computing (HPC) workload. This report developed an MPI-based, HPC-native benchmarking framework for evaluating Elasticsearchs performance. A distributed, I/O-optimized ingestion benchmarker was designed and implemented, which measures both the latency and write throughput when ingesting large corpora of documents. Furthermore, a distributed query benchmarker was designed, which allows for custom scenarios using a newly developed, JSON-based Domain Specific Language (DSL) embedding the Elastcisearch Search API query language. All syntax is conceptually similar to rally, which makes porting those battle-tested benchmarks trivial.

Additionally, an MPI-based, zero-configuration, stateful workflow to automatically create and (re-)spawn an Elasticsearch cluster in a dynamic SLURM environment. By dynamically patching the Elasticsearch configuration used in the Singularity containers with the new hostnames assigned by SLURM, the cluster becomes hardware-independent and can easily be reused in future jobs. Lastly, a full end-to-end benchmarking workflow was designed.

The benchmarker was tested on the Emmy HPC cluster using 3 nodes. For this, Elastic's `nyc_taxis` benchmark track was successfully ported and extended with further scenarios. It can be seen that the benchmark successfully records the expected scaling behaviour when increasing the number of load generators per cluster node.

In conclusion, the benchmarker successfully provides a large-scale alternative to the canonical rally benchmarker, allowing further research in HPC-native, high-throughput use case benchmarking for data lake applications.

Statement on the usage of ChatGPT and similar tools in the context of examinations

In this work I have used ChatGPT or a similar AI-system as follows:

- ☐ Not at all
- ☐ In brainstorming
- ☐ In the creation of the outline
- ☐ To create individual passages, altogether to the extent of 0% of the whole text
- ☐ For proofreading
- ☑ Other, namely: Github Copilot

I assure that I have stated all uses in full.
Missing or incorrect information will be considered as an attempt to cheat.

Contents

List of Tables

List of Figures

List of Listings

List of Abbreviations

API Application Programming Interface

CLI Command Line Interface

DSL Domain Specific Language

HPC High-Performance Computing

JSON JavaScript Object Notation

MPI Message Passing Interface

NDJSON Newline Delimited JSON

NIC Network Interface Card

SLURM Simple Linux Utility for Resource Management

1 Introduction

1.1 Motivation

In the space of HPC, a significant trend from more compute-intensive tasks to more data-intensive tasks is taking place. This change necessitates better data-management tooling such as data lakes or data warehouses. Given that HPC computes on raw data, data lakes are a more natural fit. Efficient metadata management requires a great metadata store as well as support for full-text searches.

In this report, the viability of Elasticsearch[1] for HPC load is being evaluated. Elasticsearch benefits from many years of tooling development, making it a more pragmatic choice than building an own application around full-text search engines like Apache Lucene[2] or Tantivy[3].

While elastic provides rally[4], an in-house benchmark suite used for performance regression testing [2], after rigorous internal testing it was found out that thespian[5], its internal actor framework, did not scale up to more than 60 nodes, which was not sufficient for previous large-scale stress testing. Thus, for this report a new benchmarking framework was designed, using reliable HPC native technologies such as Message Passing Interface (MPI).

In addition, the data lake related use cases at the GWDG include spawning Elasticsearch instances on-demand. For this dynamic deployment, a containerized Elasticsearch cluster based on the underlying Simple Linux Utility for Resource Management (SLURM)[6] and MPI are needed. It should auto-configure itself, in order to be IP-agnostic.

1.2 Goals and Contributions

The goals of this report are twofold: First, providing and implementing a stateful architecture to dynamically spawn and respawn an Elasticsearch cluster without any previous manual configuration. Second, design a highly-scalable, HPC-native, distributed Elasticsearch benchmarking framework for both ingestion and querying performance.

To achieve these goals, the following contributions were made:

- Design and implementation of a zero-configuration workflow to spawn a rootless, containerized Elasticsearch cluster of arbitrary size within a SLURM-allocated MPI environment.
- Design and implementation of a highly scalable, distributed benchmarker for ingestion performance
- Design and implementation of a highly scalable, distributed benchmarker for query performance with custom benchmark scenarios using a JavaScript Object Notation (JSON)-based DSL.

[1] `https://www.elastic.co/elasticsearch`
[2] `https://lucene.apache.org/`
[3] `https://github.com/quickwit-oss/tantivy`
[4] `https://github.com/elastic/rally`
[5] `https://thespianpy.com/doc/`
[6] `https://slurm.schedmd.com/documentation.html`

- Benchmark a dynamically spawned Elasticsearch cluster on HPC using a canonical dataset common in existing literature.

1.3 Structure

This report is structured as follows: In section 2, Elasticsearch, the full-text search engine and NoSQL database used, is introduced. After that, section 3 covers the related work starting with classical load generation before focussing on the literature around Elasticsearch benchmarking. In section 4, the methodology and design of all components are covered. First, it covers the design and internal workflow of the aforementioned cluster spawning mechanism. Then, it covers both the ingestion and query benchmarkers. Lastly, it focuses on the benchmark design by showing the steps required to perform the benchmarks and the reasoning behind the corpus and query design used in this specific benchmark. Lastly, in section 5 the results of the benchmark are shown, before describing the open problems and challenges in Chapter 6. Concluding in Chapter 7, the results are summarized and possible future work is shown.

2 Background

2.1 Elasticsearch

Elasticsearch is a distributed search engine initially developed in 2010. Since it stores its data in a document model, it can also be seen as a NoSQL database. The full-text search internally relies on the Apache Lucene library. It is mainly used for its full-text fuzzy search capabilities and its JSON-based REST interface. It is used by many large websites such as Wikipedia[7], Netflix, Stackoverflow, and LinkedIn.

In Elasticsearch, a collection of *documents* are stored in an *index*. Documents are equivalent to, and can be sent in the form of, JSON objects. They can be nested. Each index has a *schema*, which is a type mapping for each of the key-value pairs contained in the index[8]. This mapping can either be statically preconfigured or dynamically guessed at ingestion time. The index data can be *queried* using Elasticsearchs own DSL, again relying on JSON and REST as the foundational technology.

In 2021, due to a license change from Apache 2.0 to a more permissive license, OpenSearch was created as an Elasticsearch fork, which is maintained by several companies such as AWS.

3 Related Work

While the topic of Elasticsearch benchmarking is more sparsely covered, a lot of previous work around general HTTP API benchmarking exists. The most used load generator is Apache JMeter [3], a sophisticated graphical load tester that supports many protocols such as HTTP(S), SOAP, or LDAP. Only relying on the Command Line Interface (CLI)

[7] Wikipedia also used Lucene beforehand.
[8] Akin to a SQL database definition.

interface, wrk [4] provides a simpler popular alternative. For a more scriptable alternative, the Javascript-based Grafana k6 [5] gained a lot of popularity over recent years.

For benchmarking Elasticsearch, the main tool is the aforementioned rally [1], a microbenchmarking framework developed by Elastic. While it can be run on a single node, it also supports distributed benchmarking through the Thespian actor system.

Different benchmarking scenarios are defined as so-called *tracks*. Every track contains one or more *corpora*, containing Newline Delimited JSON (NDJSON) objects as documents. All tracks are available on GitHub [6]. Each track contains many *operations* such as ingestion or specific queries, which are then structured into a *challenge's schedule* in a fork-join model. This means that Rally can be extended without editing the source code.

The rally framework is actively used in literature for benchmarking Elasticsearch clusters [7] [8] [9]. As mentioned in the introduction, it was not a viable choice for this paper due to the underlying actor framework not scaling into more than 128 nodes in previous experiments.

Furthermore, most of the benchmark comparisons between NoSQL databases such as tsbs [10] are done by database vendors themselves, resulting in conflicting financial interests.

4 Methodology and Design

The Methodology and Design section is split into four parts: First, the autoconfigured Elasticsearch cluster spawner is presented. After that, the next two subsections cover the distributed ingestion and query benchmarker respectively, including underlying reasoning as well as some technical details. Lastly, the overall benchmark design used for this report is discussed, focusing mainly on the high-level benchmark workflow as well as the corpus and query design.

4.1 Dynamic Cluster Creation Based on MPI Communicator

This section presents an automated approach to configuring and spawning a multi-node Elasticsearch cluster based solely on the MPI environment set up by SLURM. It dynamically fetches the different hosts, i.e. it is not required to know the hostnames or IPs beforehand, making it easily embeddable in any kind of job system. The cluster is very portable since it is using Singularity [11] containers as an Elasticsearch host. Any cluster size larger or equal to two nodes is supported. The code can be found on GitHub [12].

Furthermore, it supports *statefulness*, which means that the same cluster can be respawned on other nodes[9] without requiring a re-ingest, i.e. keeping the complete cluster state and configuration. Due to the aforementioned containerization, this also works while changing both the hardware and IP addresses. The only limitation is that it only supports one Elasticsearch node per host OS. This is by design, as we use hostname-based resolution instead of IP-based resolution to be agnostic to the Network Interface Card (NIC) used[10].

This automatic containerization has the big advantage that it can be embedded into any kind of job pipeline. While most web services require a continuously running search engine, it is common in HPC that applications are spawned on demand only when needed

[9] With the same cluster size.

[10] Being NIC agnostic implies that it supports both ethernet and InfiniBand.

and torn down once the computation is performed. More importantly, it allows for Elasticsearch to be implicitly spawned as a dependency for other, more cloud-native applications running in HPC.

The high-level idea is as follows: For discovery and information communication, MPI is used. Furthermore, based on the world size, the number of master eligible nodes is decided[11]. After that, each node creates a config and runtime environment for itself. Lastly, each process spawns its container using its newly generated config.

For better portability and reproducibility, the cluster generator uses Singularity containers internally[12]. The container is based on the official Ubuntu 22.04 docker image[13], only adding packages for debugging and maintenance. Elasticsearch itself is not part of the image and is completely bind-mounted in; there are multiple reasons for this: First, unlike docker, Singularity containers are always immutable, so the program state itself has to always be bind-mounted in. Furthermore, Elasticsearch expects to be the owner of its folders. If Elasticsearch is not the folder owner, it disables multiple features such as internal auto-configuration. Since Singularity is rootless, the correct uid for the folders can't be enforced. Therefore, the most straightforward and stable solution is bind-mounting it in.

The workflow has the following steps:

- First, check if the Elasticsearch index data from the last run can be reused. This is the case if the number of nodes stayed the same. If not, create a new Elasticsearch into a temporary directory.
- `MPI_GATHER` a list of tuples `(rank, hostname)` into the root rank 0.
- For each node, the root creates/updates the config; the other nodes are waiting. Note that through updating the config instead of creating a new one the cluster stays intact, which is how the statefulness is implemented. An example config can be found in the appendix.
- Lastly, each rank starts the immutable singularity container with the new config and previously created Elasticsearch mounted in.

[11] If $N \leq 3$ then 1 master eligible node, otherwise 3 master eligible nodes. Note that only one master is active at a time. An odd number was chosen to prevent a split brain.

[12] Note that, due to Elasticsearch JDK problems, Singularity has to be started with `--cleanenv`. Therefore environment variables get ignored within the container.

[13] `https://hub.docker.com/_/ubuntu`

4.2 Distributed Ingestion Benchmarker

This section introduces the distributed, MPI-based ingestion benchmarker [13]. It is used both for providing a fast way to ingest a JSON-formatted corpus into an Elasticsearch cluster as well as measuring the performance of write operations in throughput as well as latency. The benchmarker itself is very I/O optimized, using so-called offset caching for reducing redundant operations between workers. It supports statically typed index definitions, configurable bulk size as well as a configurable number of shards. It supports all corpora designed for Elastic's rally benchmarker by using NDJSON as an input format[14]. The CLI interface with all its features can be found in the appendix.

The benchmark can be split into three phases:

Setup: Note that those steps are done by only the root/rank 0.

- Create the offset cache.

  The offset cache is needed for the following reason: To do the ingestion in a distributed manner, the bulk ingest load has to be split evenly between all nodes. This is done by giving each rank (approximately) the same number of documents in the corpus file. Note that NDJSON implies one document per line. For N nodes and L lines, each rank i gets the range

  $$\left[\frac{i}{N} \cdot L, \frac{i+1}{N} \cdot L\right)$$

  But this requires that the number of lines have to be known, which implies reading the whole file at least once. After the range has been computed, the file has to be read a second time to find the starting byte to seek to. This is needed since the JSON documents have a variadic size in bytes; one can't just compute the i-th document through $doc\_size \cdot i$.

  The corpora are often huge; the `nyc_taxis` corpus used in this report is around 75GB in size with over 160,000,000 documents. This would create a lot of I/O load if every node would do this every time the benchmark runs.

  So instead, the root computes all offsets once and saves it into a `.offsets.json` file, which can be reused in future benchmarks, removing all redundant work. This optimization is possible whenever the number of load generators stays the same.

  On a technical level, this is done as follows:

  1. Iteration 1: Count the number of lines.
  2. Compute the starting and ending line for each rank using the total number of lines.
  3. Iteration 2: Find the byte offsets for each rank.
  4. Save everything into a `.offsets.json` file.
  5. Validate that, starting at the current byte, the line of each rank has a complete JSON document.

[14] Another big advantage of NDJSON is that downscaling the benchmark is trivial using `head -n <NEW_NUMBER> original.json > downscaled.json`

An example `.offsets.json` file can be found in the appendix.

- Create an (empty) Elasticsearch index. It is created using the following settings:
  - Strict type mappings for reproducibility. Elasticsearch allows for dynamic schemas, which means that the datatype is interpreted when the first data arrives. When using a distributed benchmarker, it is not clear which rank sends the first bulk ingest request. Thus, indeterministic or unexpected behaviour could occur. So instead, using Elasticsearchs own type system[15] and DSL[16] the type mapping can be defined statically and used by the benchmarker.
  - The number of shards is explicitly set, defaulting to one shard per Elasticsearch node. This means that every node gets data while still keeping the sharding complexity as trivial as possible. This is configurable via the CLI interface.
  - Explicitly disable caching through `requests.cache.enable`[17]

At the end of the setup phase, the offsets are sent to all workers using `MPI_BROADCAST`.

Benchmark: This work is done by each rank including the root.

- Each rank computes to which Elasticsearch node it should send the data.

  It is assumed that the user understands that this distribution is a problem that has to be solved. Thus, it is expected that the MPI world size is a multiple of the number of Elasticsearch cluster nodes[18]. With that in mind, the distribution is calculated by `rank % N` with `%` being the modulo operation and `N` the number of Elasticsearch cluster nodes

- Next, seek to the starting byte based on the rank.

- Create and send the requests blockingly, as fast as possible. Track each response time.

  The requests are sent in bulk using Elasticsearches Bulk API[19] with a default bulk size of 1000 documents, configurable via CLI parameter.

  The measurements are done with utmost care to be as precise as possible and minimize the overhead created by the benchmarker itself. Thus, the delta recorded only measures the time directly before and after the HTTP request, removing the query-building overhead.

- Once all data is sent, the workers wait at an MPI barrier for the other workers to finish.

- In the end, all data is gathered at the root process.

Teardown: Once the data is gathered, the root dumps it into a JSON file.

[15] `https://www.elastic.co/guide/en/elasticsearch/reference/current/mapping-types.html`
[16] `https://www.elastic.co/guide/en/elasticsearch/reference/current/mapping.html`
[17] `https://www.elastic.co/guide/en/elasticsearch/reference/current/shard-request-cache.html`
[18] It also works when this isn't the case, although the distribution is less optimal.
[19] `https://www.elastic.co/guide/en/elasticsearch/reference/current/docs-bulk.html`

4.3 Distributed Query Benchmarker

While the last section focused on the ingestion (write) performance, this section will focus on benchmarking the query (read) performance of an Elasticsearch cluster [14]. The benchmarker also generates its load in a distributed manner using the MPI environment provided by SLURM. It measures the documents per second as well as the request latency of a cluster when put under load. The benchmark is structured using a fork-join-like model.

The distributed query benchmarker has the following features:

- Fully configurable by a DSL-like JSON standard; no hard-coded scenarios.

  The DSL embeds the Elasticsearch query language internally, making it very accessible to all Elasticsearch users. Furthermore, this allows for extending the benchmarking scenarios without the need to edit the source code. Since the language is a simplification of rally's DSL, elastics official benchmarks can be ported easily. An example of a ported rally benchmark can be found in the accompanying repository.

  Also, an example input file using the custom DSL can be found in the appendix.

- Measuring raw performance by bypassing the cache[20].
- Supporting a test mode for verifying the correctness of custom benchmarks.
- Parsing the Elasticsearch response instead of just relying on the HTTP response like a normal HTTP benchmarker. This is done to count the number of returned documents.
- Supporting multiple, alternating queries in a single fork-join task for creating a more realistic load.

On a technical level, each benchmark starts with the following preparation.

- First, all CLI arguments and the benchmark description are parsed. The benchmark description is written in the aforementioned, JSON-based custom DSL with the Elasticsearch Search API Syntax embedded into it. An example query benchmark description as well as the CLI interface structure of both benchmarkers can be found in the appendix.
- After that, every rank calculates which Elasticsearch server it sends its load to. See the ingest benchmarker section for a more detailed description of how this selection algorithm works.

Once all ranks are ready, the actual benchmark starts. For each of the fully disjunct benchmark steps, the following actions are performed:

- First, the root node, i.e. MPI rank 0, waits for the Elasticsearch cluster health[21] to be green while the other nodes wait at an MPI barrier, ready to benchmark. This is done in case the data was just ingested or the cluster was just started on-demand.

[20] `https://www.elastic.co/guide/en/elasticsearch/reference/current/shard-request-cache.html#_enabling_and_disabling_caching_per_request`

[21] `https://www.elastic.co/guide/en/elasticsearch/reference/current/cluster-health.html`

- As mentioned above, each benchmark step can contain multiple queries, which will then be executed in an alternating manner to simulate a more varied load on the server. Since all ranks start executing the queries at the same time, all ranks might be in sync, sending the exact same query at the same time, which would not be a realistic load. To prevent this, each rank creates a random permutation of all queries for this step.
- If the warmup time is set:
  - Send the next query, and throw away the result. This is needed to get the caches filled before starting the measurements; therefore reducing the variance of the result data. The index is configured to not do any caching on an Elasticsearch level, however, the OS will still do caching, both on a CPU L1/L2/L3 level as well as I/O caching through the page and buffer cache.
  - Sleep between results if a waiting time is configured.
- Once the warmup is done, the following steps are done until the configured execution time is reached:
  - Select the next query.
  - Track the current time right before sending the request.
  - Do the request containing the query against the search API endpoint. Note that the cache is explicitly disabled, both on an index level and on a request level[22].
  - Track the time right after the response was received, before processing the result. This is done for two reasons: First, it minimizes the measurement overhead created by the benchmarker itself and isolates the time Elasticsearch needed for creating and sending the response. Second, it increases the likelihood of our operation being atomic, i.e. that the OS scheduler will not preempt the process by giving the CPU time to another process before the measurement is finished.
  - Next, it will process and record the result based on the HTTP response code. If it was successful[23], it will count the number of received documents and will save a tuple `(latency, docs count)` for the current query and rank. If the request is unsuccessful, it saves the HTTP code for the current query and rank. Note that this all happens in Memory, no slow I/O is done to minimize the benchmarker overhead.
  - Lastly, sleep between results if a waiting time is configured.

Once all measurements are created, the results of all ranks get merged at the root using MPI gather as well as some data transformation. The resulting output format can be seen in the appendix. To finish the execution, the root dumps the JSON-formatted results at a path specified via a CLI parameter.

[22] `https://www.elastic.co/guide/en/elasticsearch/reference/current/shard-request-cache.html#_enabling_and_disabling_caching_per_request`

[23] i.e. a 2xx HTTP response code

4.3.1 Test Mode

Designing a database benchmark is complicated; it should be very clear both what the expected result is and whether the query actually produces that result. To verify the correctness of a given benchmark, a test mode was developed. This test mode can be run with `--test-mode`.

The purpose of the test mode is to give a short overview to check whether the raw Elasticsearch responses contain the results expected. Therefore it is sufficient that the test mode only runs on the root rank. The test mode itself is pretty trivial: It goes through each benchmark step, runs each query once, verifies that the response code is successful, and then prints the request input as well as the response output as prettified JSON into the terminal for further manual inspection.

While designing several benchmarks, it helped finding broken queries generating empty responses.

4.4 Benchmark Design

4.4.1 High-Level Benchmark Workflow

To create and run a benchmark, the following steps have to be done:

1. Choose a dataset or create a synthetic one. The dataset has to be formatted in the NDJSON format, which is defined as one JSON object per line. Datasets for many common Elasticsearch use cases can be found in Elastic's rally-tracks repository [6].

2. Define the data type mappings for each corpus attribute using the Elasticsearch index mapping syntax[24]. An example can be found in the repository.

3. Design the query document. This basically consists of embedding the Elasticsearch search API queries into a bigger JSON structure defining the benchmark steps. Note that the queries themselves do not have to be altered, thus they can also easily be constructed using cURL. Since Elastics rally also uses the search API syntax, their benchmarks can easily be ported. For each benchmark step, the warmup time, execution time, and sleep time between each request can be defined optionally. The query document format can be found in the appendix, and a ported benchmark from rally can be found in the repository.

4. Spawn up the automatically configured Elasticsearch cluster using the MPI-based cluster creator in a given SLURM environment using `mpirun`.

   Note that Elasticsearch requires a non-standard Kernel parameter to be set, a `vm.max_map_count` of at least 262144. This setting defines the maximum number of memory map areas per process. This is needed because Elasticsearch accesses the index data by mapping the files into memory using their so-called `mmapfs`[25], based on Lucenes `MMapDirectory`[26].

[24] `https://www.elastic.co/guide/en/elasticsearch/reference/current/properties.html`

[25] `https://www.elastic.co/guide/en/elasticsearch/reference/current/vm-max-map-count.html`

[26] `https://lucene.apache.org/core/6_3_0/core/org/apache/lucene/store/MMapDirectory.html`

5. Run the distributed ingest benchmarker with the previously created corpus and type mapping using `mpirun`.
6. Run the distributed query benchmarker with the previously created query document using `mpirun`.
7. Analyze all results. A Jupyter notebook in the repository can be used as a starting point.

4.4.2 Corpus and Query Design

The benchmark created for this report is a port of rallys `nyc_taxis` track [15]. Its corpus contains New York taxi data, more specifically all rides that have been performed in yellow taxis in New York in 2015. This data gets published every year by the NYC Taxi and Limousine Commission and can be freely downloaded [16]. An example corpus document can be found in the appendix.

This benchmark was chosen as it is one of the two rally benchmarks most often used in literature, `geonames`, and `nyc_taxis`. In the literature, both of these benchmarks have a specific purpose.

`nyc_taxis` is mainly used as a scaling test because of its big corpus size with over 165 million documents and comparatively big document size. The documents are very numeric, which makes the data set great for benchmarking range queries and aggregations such as histograms. It can't be used for thorough benchmarking of string-matching capabilities.

`geonames` is mostly used as a regression test for various features. It has a very balanced document structure, containing text, keywords, numbers, as well as geolocations. The main advantage of `geonames` is the number of queries designed for it, including text and keyword matching, several aggregations, scroll API, Elasticsearch's expression and painless scripting languages, and many more niche features of Elasticsearch.

Since the internal use case at the GWDG is mainly around fuzzy matching and range queries running on large corpora in an HPC environment, the `nyc_taxis` track is a better fit.

For the measurements done in this report, a full query benchmark was designed, loosely based on the one provided by Elastic. Note that the main purpose of this benchmark is to provide a starting point on how to design Elasticsearch queries for this benchmarker.

It contains the following benchmark steps:

- Simple, non-filtering `match_all`[27] requests, with response sizes of 10, 100, 1000, or 10000 documents. This is done to benchmark the base I/O performance and response size scaling.

[27] `https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-match-all-query.html`

- Next a `range` query[28], again scaling up the number of documents between each step. The range query is the exact same one also used by Rally.
- After that, it alternates the `range` and `match_all` query in the same benchmark step. This is done to simulate a more varied and realistic load.
- In order to measure the latency change given less load, the `match_all` query gets executed with 0.02, 0.05, 0.1, 0.2, 0.5, and 1-second sleep between each request.
- Lastly, multiple histogram aggregations, either with automatic or manual bucketing, are being executed, all designed by Elastic. This allows us to measure aggregation performance while also being comparable to other benchmarks made with Rally.

5 Test Run and Analysis

In this section, an example benchmark is shown in order to confirm that works as expected as well as show how this benchmarker could possibly be used in a real scientific benchmarking scenario. This section will be split into three parts: The setup, ingestion results, and query results. Note that all raw data as well as the analysis scripts can be found in the accompanying GitHub repository.

5.1 Setup

The aforementioned ported `nyc_taxis` benchmark was run on the Emmy[29] supercomputer using 3 dedicated standard96 partition compute nodes to create a 3 node Elasticsearch cluster. The nodes communicated using an ethernet interconnect instead of Intel Omni-Path, although both were tested to work properly. The Singularity container is based on the Ubuntu 22.04 dockerhub image. It runs Elasticsearch 8.11.0, which shipps its own OpenJDK 21.0.1. Elasticsearch indices were configured to not cache at all. The benchmarker used Python 3.9 as well as OpenMPI 4.1.

Note that for these benchmarks a slightly modified version of this benchmarker was used. The main difference was that the benchmarker provided in this report does not support to run the cluster spawner and benchmarker on the same server since both expect to use `MPI_COMM_WORLD`. But for the purposes of this example benchmark, in order to save computing resources, it is okay to execute the load generators and to-be-benchmarked process on the same machine.

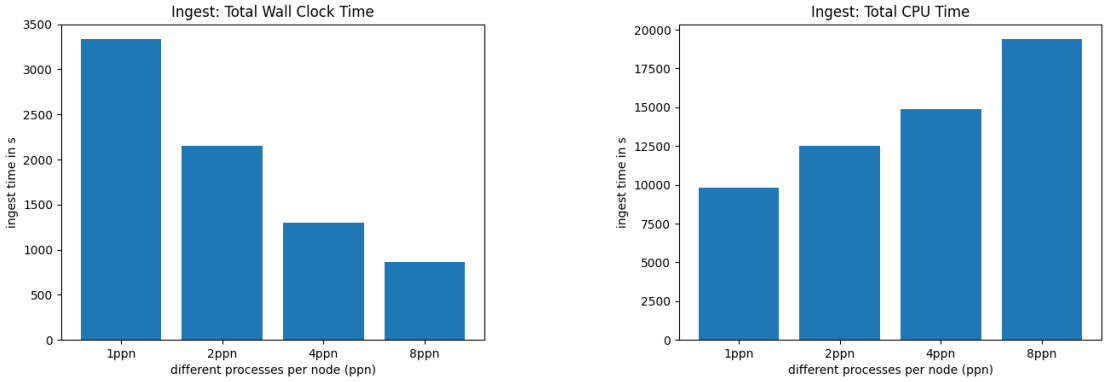
5.2 Ingestion

For the ingestion benchmarks, the 3 node cluster was ingested with different processes per node (ppn); A ppn of 4 means that for 3 Elasticsearch nodes $3 \cdot 4 = 12$ load generators were spawned. Note that the load generators choose a random Elasticsearch instance to send the data to. This is good, because it imitates the network overhead of a real benchmark.

Here are the results:

[28] `https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-range-query.html`

[29] `https://gwdg.de/hpc/systems/emmy/`



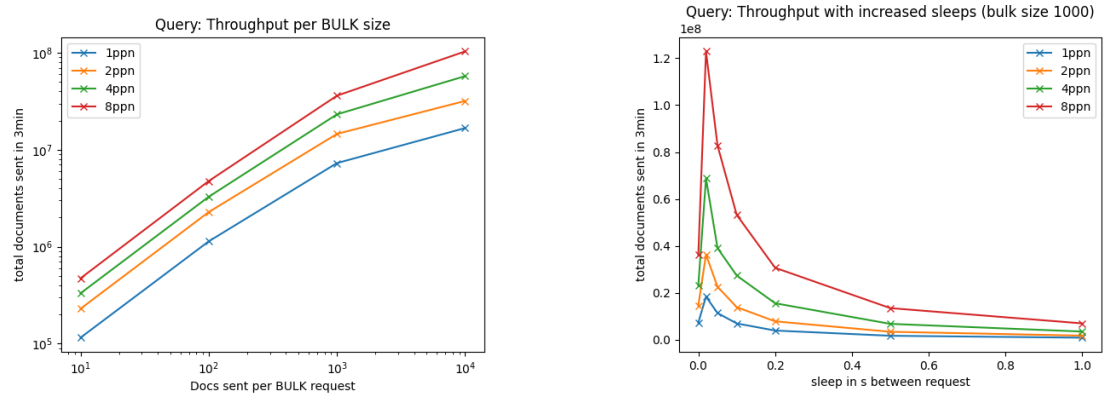
(a) (b)

Figure 1: Results showing wall clock time (a) (maximum execution time) and CPU time (b) (sum of all execution times) for different ppn.

One can see that, as expected, by increasing the amount of load generators the wall clock time to ingest a given set of data decreases, although it unfortunately scales sub-linear. But when looking at the CPU time, which is the sum of all wall clock times, the ingestion becomes more inefficient with more load generators. Note that they logically would have stayed the same if the efficiency in (a) increased linarly with regard to the ppn. This is most likely due to the increased load on Elasticsearch, since the load generators themselves do not communicate between each other while ingesting, thus no additional MPI overhead should be measured.

5.3 Queries

The query benchmarks were also done for different ppn, here are some results based on the aforementioned range queries:



(a) (b)

Figure 2: Scaling range queries along the BULK request size (a) (number of documents) and along the sleep between requests (b)

The first plot shows scaling along the BULK request size. This means increasing the number of documents sent per request. It was measured in powers of 10, i.e. 10, 100,

1000, 10000 documents per request. The results are expected, since increasing the number of documents per request decreases the percentage wise network overhead, thus resulting in higher throughput. Furthermore it can be seen that it does not scale linearly[30] which shows that the Elasticsearch actually slows down from the increased overall load. More processes per node increase the throughput, implying that the server can still handle the number of processes well.

The second plot does the same query, but with increased waiting/sleep times off each load generator between the requests. Fascinatingly, it can be seen that a sleep time of $0.02s$ increases the throughput significantly over not sleeping at all. One can also see that sleepint $0.2s$ does approximately create the same throughput as not sleeping at al! A possible guess is that the server was overloaded, resulting in dropped TCP packages, which then results in reduction of sending due to additive increase, multiplicative decrease. But after that, can be seen that increasing the sleep time decreases the overhead, which is to be expected since the nodes were not completely utilized while benchmarking.

6 Challenges

The design and implementation of the benchmarker still leaves two challenges open, both of which are not trivial to solve optimally.

Response Size: If not specified otherwise, the Elasticsearch Search API will only return the 10 elements, even if more matching documents exist in the index. This number can be increased by the `size` parameter[31].

Unfortunately, due to performance reasons, this number can not be increased indefinitely. The maximum count of documents returnable by Elasticsearch is limited by the `index.max_result_window`[32] config setting, which defaults to 10000 documents. This means that only the first 10000 elements can be benchmarked. Theoretically, one can offset the result using the `from` parameter, practically Elasticsearch enforces the condition that `from+size <= index.max_window`.

There are at least three solutions to this problem:

1. Design the benchmarks around this constraint. For `nyc_taxis`, this is what was done by Elastic; this is also the approach in most of the literature.

2. Use the `search_after` parameter for the search API[33]. This parameter takes the `sort` offset from a previous query and computes the new result starting from that offset.

   This approach has several big disadvantages. First, it blows up the complexity of both the benchmark design as well as the benchmarker logic by requiring statefulness

[30] Note that both axis are logarithmic.

[31] `https://www.elastic.co/guide/en/elasticsearch/reference/current/paginate-search-results.html`

[32] `https://www.elastic.co/guide/en/elasticsearch/reference/current/index-modules.html`

[33] `https://www.elastic.co/guide/en/elasticsearch/reference/current/paginate-search-results.html#search-after`

between requests. The requests can't be done in a random manner since they always have to keep the offsets from previous responses. Additionally, it would make the benchmarking results harder to understand, as it may not be obvious how many requests were needed to get a specific amount of documents. Lastly, the `search_after` parameter does not increase the complexity of each request, since it performs the same amount of work, except that it first seeks to a specific byte offset.

3. Use this Scroll API[34]. This has the same problem as the `search_after` API as it requires keeping the state of a previous search query. Furthermore, according to Elastics documentation, its usage is discouraged for deep pagination of over 10000 elements.

Load Generator to Cluster Node Mapping As described in the methodology, each load generator sends its request to the same Elasticsearch cluster node every time. To provide roughly even load distribution, the node number is calculated by `MPI_rank % N` with `%` being the modulo operation and `N` the number of Elasticsearch cluster nodes.

This node selection algorithm could result in a suboptimal usage of the network topology, resulting in longer package round trip times, for example requiring many hops in a fat tree structure.

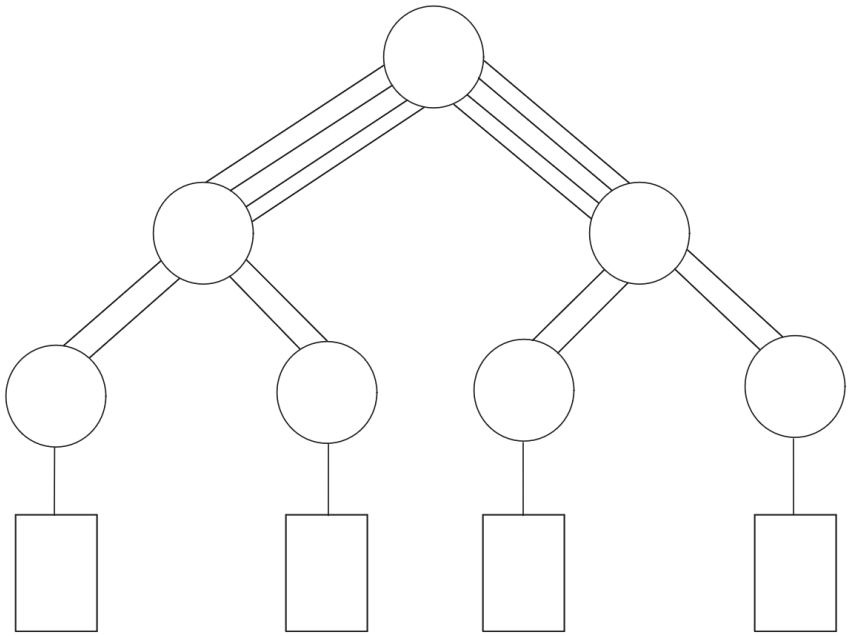


Figure 3: An example of a fat tree. The leafs are nodes, and the intermediate nodes are routers. The higher the router, the more throughput it gets to provide to not limit the throughput by the higher degree nodes.

A possible solution would be to compute the optimal mapping every time before starting the benchmark. This could be done by measuring the round trip time between any pair of load generator and cluster node several times and using its mean or median average. Note that the lowest number can't always be used since all load generators have to be divided up onto the cluster nodes equally. Instead, this could be solved as a linear

[34] `https://www.elastic.co/guide/en/elasticsearch/reference/current/scroll-api.html`

program[35].

Unfortunately, this solution would overall worsen the benchmark results since, based on the current overall network usage, the routing could change between benchmarks, making the configuration indeterministic and thus the results unreproducible.

7 Conclusion

For the completion of this report, a zero-configuration workflow for automatically spawning and re-spawning a stateful Elasticsearch cluster was developed, allowing for job-based on-demand metadata processing as part of sophisticated job pipelines. Furthermore, a complete distributed benchmarking workflow was developed, including an ingestion benchmarker and a query benchmarker, allowing for large-scale benchmarks in HPC environments. An example benchmark commonly used in the literature was successfully ported, executed, and analyzed. The code is fully accessible via GitHub, allowing it to be used as a platform for future performance research.

7.1 Future Work

As part of the data lake work at the GWDG, this benchmarker will be used to evaluate different encryption strategies based on their performance penalty. In general, most of the future work will be in the usage of the benchmarker for doing large-scale benchmarks, not in the extension of the benchmarker itself.

As described in the Challenges, the primary open problem is figuring out how to handle response sizes larger than `index.max_result_window` through the methods above. Furthermore, currently it is not supported to run both the spawner and load generators on the same nodes, since the spawner is blocking the MPI environment. Lastly, a collection of common benchmarks that can be used for this benchmarker has to still be created over time.

[35] After some discussion with Johann Carl Meyer, who is a fellow student, I learned that this can be solved even more efficiently if seen as an optimal transport problem.

References

[1] *elastic/rally: Macrobenchmarking framework for Elasticsearch*. URL: https://github.com/elastic/rally (visited on 01/18/2024).

[2] *Elasticsearch Benchmarks*. URL: https://elasticsearch-benchmarks.elastic.co/# (visited on 01/17/2024).

[3] *Apache JMeter - Apache JMeter™*. URL: https://jmeter.apache.org/ (visited on 01/18/2024).

[4] Will Glozer. *wg/wrk*. original-date: 2012-03-20T11:12:28Z. Jan. 18, 2024. URL: https://github.com/wg/wrk (visited on 01/18/2024).

[5] *Load testing for engineering teams | Grafana k6*. URL: https://k6.io (visited on 01/18/2024).

[6] *elastic/rally-tracks*. original-date: 2016-06-06T13:16:40Z. Jan. 18, 2024. URL: https://github.com/elastic/rally-tracks (visited on 01/18/2024).

[7] Hårek Haugerud, Mohamad Sobhie, and Anis Yazidi. "Tuning of Elasticsearch Configuration: Parameter Optimization Through Simultaneous Perturbation Stochastic Approximation". In: *Frontiers in Big Data* 5 (2022). ISSN: 2624-909X. URL: https://www.frontiersin.org/articles/10.3389/fdata.2022.686416 (visited on 01/18/2024).

[8] Wataru Takase et al. *A solution for secure use of Kibana and Elasticsearch in multi-user environment*. June 30, 2017. arXiv: 1706.10040[cs]. URL: http://arxiv.org/abs/1706.10040 (visited on 01/18/2024).

[9] Hui Dou, Pengfei Chen, and Zibin Zheng. "Hdconfigor: Automatically Tuning High Dimensional Configuration Parameters for Log Search Engines". In: *IEEE Access* 8 (2020), pp. 80638–80653. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.2990735. URL: https://ieeexplore.ieee.org/document/9079492/ (visited on 01/18/2024).

[10] *timescale/tsbs*. original-date: 2018-08-08T14:30:28Z. Jan. 17, 2024. URL: https://github.com/timescale/tsbs (visited on 01/18/2024).

[11] Gregory M. Kurtzer et al. *sylabs/singularity: SingularityCE 4.1.0 Release Candidate 1*. Version v4.1.0-rc.1. Jan. 12, 2024. DOI: 10.5281/zenodo.10495969. URL: https://zenodo.org/records/10495969 (visited on 01/18/2024).

[12] *elasticsearch_benchmarker_mpi/containers/start_es_cluster.py at main · lquenti/elasticsearch_b* URL: https://github.com/lquenti/elasticsearch_benchmarker_mpi/blob/main/containers/start_es_cluster.py (visited on 05/10/2024).

[13] *elasticsearch_benchmarker_mpi/benchmarker/ingestor.py at main · lquenti/elasticsearch_benchm* URL: https://github.com/lquenti/elasticsearch_benchmarker_mpi/blob/main/benchmarker/ingestor.py (visited on 05/10/2024).

[14] *elasticsearch_benchmarker_mpi/benchmarker/queryer.py at main · lquenti/elasticsearch_benchm* URL: https://github.com/lquenti/elasticsearch_benchmarker_mpi/blob/main/benchmarker/queryer.py (visited on 05/10/2024).

[15] *rally-tracks/nyc_taxis at master · elastic/rally-tracks*. GitHub. URL: https://github.com/elastic/rally-tracks/tree/master/nyc_taxis (visited on 02/15/2024).

[16] *TLC Trip Record Data - TLC*. URL: https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page (visited on 02/15/2024).

A Format Specifications

A.1 Ingest Benchmarker Static Index Specification

```
{
"properties": {
  "surcharge": {
    "scaling_factor": 100,
    "type": "scaled_float"
  },
  "dropoff_datetime": {
    "type": "date",
    "format": "yyyy-MM-dd HH:mm:ss"
  },
  "trip_type": {
    "type": "keyword"
  },
  "mta_tax": {
    "scaling_factor": 100,
    "type": "scaled_float"
  },
  [...]
  "trip_distance": {
    "scaling_factor": 100,
    "type": "scaled_float"
  },
  "pickup_location": {
    "type": "geo_point"
  }
}
}

```

Listing 1: Static index specified for the NYC taxis benchmark (Truncated for readablity)

A.2 Ingest Benchmarker Cached Offsets

```
{
  "number_of_workers":3,
  "offsets":[
    {
      "rank":0,
      "starting_line":0,
      "starting_byte":0,
      "number_of_lines":8333
    },
    {
      "rank":1,
      "starting_line":8333,
      "starting_byte":4157901,
      "number_of_lines":8333
    },
    {
      "rank":2,
      "starting_line":16666,
      "starting_byte":8315734,
      "number_of_lines":null
    }
  ]
}
```

Listing 2: Cached offsets of a injestion run (25k docs, 3 MPI worker, unminified)

A.3 Query Benchmarker Input Format

```
[
  {
    "search_queries": [
      {
        /* everything in here just gets sent to ES */
        "body": {
          /* The raw ES query sent to the server */
        }
      }
    ],
    "warmup_time_secs": 30, /* optional */
    "execution_time_secs": 120, /* optional */
  },
  {
    "search_queries": [
      {
        "body": {
          /* The first of two queries sent iteratively
             (random permutation) */
        }
      },
      {
        "body": {
          /* The second of two queries sent iteratively
             (random permutation) */
        }
      }
    ],
    "warmup_time_secs": 30, /* optional */
    "execution_time_secs": 180, /* optional */
    "sleep_between_requests_secs": 0.25 /* optional */
  }
]

```

Listing 3: Structure of the Query Benchmarker JSON based DSL

A.4 Query Benchmarker Output Format

```
[
  {
    "query_result": [
      {
        "body": {
          /* The inside of the first "query" in the input */
        },
        "responses": [
          [
            /* load generator 1 */
            {"latency": 0.35, "docs": 2500},
            {"latency": 0.32, "docs": 2500},
            {"error_code": 501}
            {"latency": 0.33, "docs": 2500},
          ],
          [
            /* load generator 2 */
            {"latency": 0.35, "docs": 2500},
            {"latency": 0.32, "docs": 2500},
            {"error_code": 501}
            {"latency": 0.33, "docs": 2500},
          ]
        ]
      },
      {
        "body": {
          /* The inside of the second "query" in the input */
        },
        "responses": [
          /* same as above */
        ]
      }
    ]
    /* same as input */
    "warmup_time_secs": 30,
    "execution_time_secs": 120,
    "sleep_between_requests_secs": 0.25
  },
  {
    /* ... */
  }
]
```

Listing 4: Structure of the Query Benchmarker results

B CLI interfaces

B.1 Ingest Benchmarker

```
usage: ingestor.py [-h] --data DATA --index INDEX --hosts HOSTS
                   [--bulksize BULKSIZE] [--indexname INDEXNAME]
                   [--shards SHARDS]

Load ingestor for Elasticsearch.

options:
  -h, --help            show this help message and exit
  --data DATA           Path to the JSON data to be ingested
  --index INDEX         Path to the Metadata Description to be ingested
  --hosts HOSTS         Comma separated hosts. Example: "--hosts
                        host1:9200,10.0.0.2:9200,localhost:9200"
  --bulksize BULKSIZE   Number of documents sent per request (defaults to
                        1000) (Optional)
  --indexname INDEXNAME
                        Name of the elasticsearch index created for the
                        specified data.
  --shards SHARDS       Number of shards, defaults to 1 shard per host

```

Listing 5: CLI interaface of the Ingest Benchmarker

B.2 Query Benchmarker

```
usage: queryer.py [-h] --hosts HOSTS [--indexname INDEXNAME] --output OUTPUT
                  [--warmup WARMUP] [--execution EXECUTION] [--sleep SLEEP]
                  [--testmode] [--allowbad]
                  json_file

Elasticsearch Query Benchmarker.

positional arguments:
  json_file             Path to the JSON file containing the query
                        configurations.

options:
  -h, --help            show this help message and exit
  --hosts HOSTS         Comma separated hosts. Example: "--hosts
                        host1:9200,10.0.0.2:9200,localhost:9200"
  --indexname INDEXNAME
                        Name of the Elasticsearch index to query.
  --output OUTPUT       Filepath for the output results.
  --warmup WARMUP       Default warmup time in seconds (defaults to 60).
  --execution EXECUTION
                        Default execution time in seconds (defaults to 300).
  --sleep SLEEP         How long to wait until the next request (defaults to
                        0.0).
  --testmode            Run the script in test mode. Executes only on root,
                        once per query.
  --allowbad            Allow yellow and red clusters (useful for specific
                        debugging)

```

Listing 6: CLI interaface of the Query Benchmarker

C Example Generated Elasticsearch Config

```yaml
cluster.name: securemetadata
node.name: securemetadata4
node.roles: ["master", "data"]
network.host: 0.0.0.0
cluster.initial_master_nodes: [securemetadata0]
# Expects hostnames to be DNS resolvable
discovery.seed_hosts: [
  "hostname_of_rank_0",
  "hostname_of_rank_1",
  "hostname_of_rank_2"
]
xpack.security.enabled: false
```

Listing 7: Autogenerated example config for Elasticsearch cluster node (reformatted)

D NYC taxis example document

```
{
  "total_amount": 6.3,
  "improvement_surcharge": 0.3,
  "pickup_location": [
    -73.92259216308594,
    40.7545280456543
  ],
  "pickup_datetime": "2015-01-01 00:34:42",
  "trip_type": "1",
  "dropoff_datetime": "2015-01-01 00:38:34",
  "rate_code_id": "1",
  "tolls_amount": 0.0,
  "dropoff_location": [
    -73.91363525390625,
    40.76552200317383
  ],
  "passenger_count": 1,
  "fare_amount": 5.0,
  "extra": 0.5,
  "trip_distance": 0.88,
  "tip_amount": 0.0,
  "store_and_fwd_flag": "N",
  "payment_type": "2",
  "mta_tax": 0.5,
  "vendor_id": "2"
}
```

Listing 8: Example document of the NYC taxis corpus (unminified)