HPS
https://hpc.gwdg.de

Lars Quentin

MPI-based Creation and Benchmarking of a Dynamic Elasticsearch Cluster

1 Introduction

2 Spawner

3 Ingestor

4 Querier

5 Test Evaluation

6 Conclusion

Insights

- Why a custom spawner and new specialized benchmarker is required
- How the following works:
  - distributed cluster spawner
  - distributed ingestion benchmarker
  - distributed query benchmarker
- How to create a new benchmark scenario from scratch

Motivation: Data Lakes

Why are Data Lakes needed

- Research becomes evermore data-driven and compute-intensive
  - More Simulations
  - Data Science, Machine Learning
- HPC becomes more data oriented
- Better data-management tooling needed
- HPC operates on raw data
  $\Rightarrow$ Data Lakes

Motivation: Data Lakes

Why are Data Lakes needed

- Research becomes evermore data-driven and compute-intensive
  - More Simulations
  - Data Science, Machine Learning
- HPC becomes more data oriented
- Better data-management tooling needed
- HPC operates on raw data
  $\Rightarrow$ Data Lakes

Metadata management

- Providing storage is easy
- Managing storage is hard
- Keep data findable, manage data
- Fully indexed
- Fully (fuzzy) searchable
- No-SQL data store / search engine
  - Elasticsearch

Motivation: Elasticsearch and Rally

Elasticsearch for HPC

- Elasticsearch is designed for cloud-use
  - Always running
  - Same host, same IP
  - Only ethernet
- This is not given in HPC:
  - Jobs spawned on demand
  - Every job gets different nodes
  - Changing IPs between runs
  - ETH, IB, Intel OPA
- Thus, a custom **stateful** workflow is required for HPC use!

Motivation: Elasticsearch and Rally

Elasticsearch for HPC

- Elasticsearch is designed for cloud-use
  - Always running
  - Same host, same IP
  - Only ethernet
- This is not given in HPC:
  - Jobs spawned on demand
  - Every job gets different nodes
  - Changing IPs between runs
  - ETH, IB, Intel OPA
- Thus, a custom **stateful** workflow is required for HPC use!

Benchmarking Elasticsearch

- HPC is all about performance
- Elastic's benchmarker: *rally* [1]
  - Used for in-house performance regression testing
  - Written in Python
  - Distributed using *thespian* agent framework
  - After previous unpublished research at GWDG:
    - Doesn't work with over 60 nodes
- Not viable for HPC-scale benchmarking

Contributions

The 4 main contributions of this work:

Contributions

The 4 main contributions of this work:

1. **On-demand Elasticsearch Cluster Spawner**
   - Zero-configuration
   - Dynamic resolution, based on SLURM MPI envionment
   - Arbitrary cluster size
   - Stateful between runs

Contributions

The 4 main contributions of this work:

1. **On-demand Elasticsearch Cluster Spawner**
   - Zero-configuration
   - Dynamic resolution, based on SLURM MPI envionment
   - Arbitrary cluster size
   - Stateful between runs
2. **Ingestion Benchmarker**
   - Distributed, MPI-based
   - Benchmarks easily portable from rally

Contributions

The 4 main contributions of this work:

1. **On-demand Elasticsearch Cluster Spawner**
   - Zero-configuration
   - Dynamic resolution, based on SLURM MPI envionment
   - Arbitrary cluster size
   - Stateful between runs
2. **Ingestion Benchmarker**
   - Distributed, MPI-based
   - Benchmarks easily portable from rally
3. **Query Benchmarker**
   - Distributed, MPI-based
   - Mixed queries for realistic load
   - Custom scenario support using own JSON-based DSL

Contributions

The 4 main contributions of this work:

1. **On-demand Elasticsearch Cluster Spawner**
   - Zero-configuration
   - Dynamic resolution, based on SLURM MPI envionment
   - Arbitrary cluster size
   - Stateful between runs
2. **Ingestion Benchmarker**
   - Distributed, MPI-based
   - Benchmarks easily portable from rally
3. **Query Benchmarker**
   - Distributed, MPI-based
   - Mixed queries for realistic load
   - Custom scenario support using own JSON-based DSL
4. **Example workflow** for canonical dataset

Background: Elasticsearch

- Distributed search engine
- Document-based NoSQL-Storage
- Internally based on Apache Lucene
- Provides JSON-based REST interface
- Apache 2.0 fork: Opensearch
- Advantages:
  - Mature ecosystem
  - Very battle-tested
  - A lot of tooling / library support

Background: Benchmarking

- For elasticsearch: All literature uses rally [2] [3] [4]
- Alternatives: Just use a HTTP benchmarker
  - JMeter [5]
  - wrk [6]
  - Grafana k6 [7]
- Most NoSQL comparisons are done by database vendors [8]
  - Bad financial incentives

On-Demand, Dynamic Cluster Spawner

Features

On-Demand, Dynamic Cluster Spawner

Features

- Fully automated, uses MPI envionment provided by SLURM

On-Demand, Dynamic Cluster Spawner

Features

- Fully automated, uses MPI envionment provided by SLURM
- Dynamically fetches the hosts
  - Not required to know them beforehand
  - IPs and hardware can be changed between runs

On-Demand, Dynamic Cluster Spawner

Features

- Fully automated, uses MPI envionment provided by SLURM
- Dynamically fetches the hosts
  - Not required to know them beforehand
  - IPs and hardware can be changed between runs
- Very portable through containerization (Singularity)

On-Demand, Dynamic Cluster Spawner

Features

- Fully automated, uses MPI envionment provided by SLURM
- Dynamically fetches the hosts
  - Not required to know them beforehand
  - IPs and hardware can be changed between runs
- Very portable through containerization (Singularity)
- Stateful: Same cluster can be respawned
  - on **different nodes**
  - **without reingestion**

On-Demand, Dynamic Cluster Spawner

Features

- Fully automated, uses MPI envionment provided by SLURM
- Dynamically fetches the hosts
  - Not required to know them beforehand
  - IPs and hardware can be changed between runs
- Very portable through containerization (Singularity)
- Stateful: Same cluster can be respawned
  - on **different nodes**
  - **without reingestion**
- NIC-agnostic. Tested on:
  - Ethernet
  - Infiniband

On-Demand, Dynamic Cluster Spawner (cont.)

High-Level Workflow:

Prerequisites:

- All hosts are known to each other via the MPI environment
- All nodes have at least one shared mount

On-Demand, Dynamic Cluster Spawner (cont.)

High-Level Workflow:

Prerequisites:

- All hosts are known to each other via the MPI environment
- All nodes have at least one shared mount

Workflow:

1. Each node creates a config

On-Demand, Dynamic Cluster Spawner (cont.)

High-Level Workflow:

Prerequisites:

- All hosts are known to each other via the MPI environment
- All nodes have at least one shared mount

Workflow:

1. Each node creates a config
2. MPI-Gather all hostnames to the root rank

On-Demand, Dynamic Cluster Spawner (cont.)

High-Level Workflow:

Prerequisites:

- All hosts are known to each other via the MPI environment
- All nodes have at least one shared mount

Workflow:

1. Each node creates a config
2. MPI-Gather all hostnames to the root rank
3. The root node updates the configs for all nodes

On-Demand, Dynamic Cluster Spawner (cont.)

High-Level Workflow:

Prerequisites:

- All hosts are known to each other via the MPI environment
- All nodes have at least one shared mount

Workflow:

1. Each node creates a config
2. MPI-Gather all hostnames to the root rank
3. The root node updates the configs for all nodes
4. Each rank stats its singularity container with the config bind-mounted in

On-Demand, Dynamic Cluster Spawner (cont.)

High-Level Workflow:

Prerequisites:

- All hosts are known to each other via the MPI environment
- All nodes have at least one shared mount

Workflow:

1. Each node creates a config
2. MPI-Gather all hostnames to the root rank
3. The root node updates the configs for all nodes
4. Each rank stats its singularity container with the config bind-mounted in

See the accompanying report for a more low-level workflow.

On-Demand, Dynamic Cluster Spawner (cont.)

Example Generated Config

```
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
```

Ingestion Benchmarker

- Two purposes:
  1. Ingest JSON corpus into Elasticsearch cluster for query benchmarks
  2. Measure performance of **write-performance** and throughput
- Features:
  - Distributed, MPI-based
  - I/O optimized through *offset caching*
  - Supports statically typed index definitions
  - Supports Newline Delimited JSON (NDJSON)
    - Thus compatible with rally!
  - Configurable via CLI: bulk size, shards per node

Offset Caching

Problem

- Data has to be parititioned.

Offset Caching

Problem

- Data has to be parititioned.
- This should be done fairly, i.e.
  For N nodes and L lines, rank i gets
  $\left[\frac{i}{N} \cdot L, \frac{i+1}{N} \cdot L\right)$

Offset Caching

Problem

- Data has to be parititioned.
- This should be done fairly, i.e.
  For N nodes and L lines, rank i gets
  $\left[\frac{i}{N} \cdot L, \frac{i+1}{N} \cdot L\right)$
- For the computation the number of lines need to be known. (1st read)

Offset Caching

Problem

- Data has to be parititioned.
- This should be done fairly, i.e.
  For N nodes and L lines, rank i gets
  $\left[\frac{i}{N} \cdot L, \frac{i+1}{N} \cdot L\right)$
- For the computation the number of lines need to be known. (1st read)
- Afterwards, the line has to be found. (2nd read)
  - Can't just seek, since JSON documents have variadic size!

Offset Caching

Problem

- Data has to be parititioned.
- This should be done fairly, i.e.
  For N nodes and L lines, rank i gets
  $\left[\frac{i}{N} \cdot L, \frac{i+1}{N} \cdot L\right)$
- For the computation the number of lines need to be known. (1st read)
- Afterwards, the line has to be found. (2nd read)
  - Can't just seek, since JSON documents have variadic size!
- A lot of I/O, the corpus is 75GB.

Offset Caching

Problem

- Data has to be parititioned.
- This should be done fairly, i.e.
  For N nodes and L lines, rank i gets $\left[\frac{i}{N} \cdot L, \frac{i+1}{N} \cdot L\right)$
- For the computation the number of lines need to be known. (1st read)
- Afterwards, the line has to be found. (2nd read)
  - Can't just seek, since JSON documents have variadic size!
- A lot of I/O, the corpus is 75GB.

Solution

- Just one node computes it, and caches it in a file!

Offset Caching

Problem

- Data has to be parititioned.
- This should be done fairly, i.e.
  For N nodes and L lines, rank i gets
  $\left[\frac{i}{N} \cdot L, \frac{i+1}{N} \cdot L\right)$
- For the computation the number of lines need to be known. (1st read)
- Afterwards, the line has to be found. (2nd read)
  - Can't just seek, since JSON documents have variadic size!
- A lot of I/O, the corpus is 75GB.

Solution

- Just one node computes it, and caches it in a file!
- Steps:
  1. Read 1: Count number of lines.
  2. Compute starting and ending line for each rank.
  3. Read 2: Find the byte offsets for each rank.
  4. Save everything into a `.offsets.json` file.

Offset Caching (cont.)

Example `.offset.json` file for 3 nodes

```
{
  "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 }
  ] }
```

Ingestion Benchmarker (cont.)

Workflow: Setup (root only)

- Create cache offsets if not already existing
  - Requires same number of load generators

Ingestion Benchmarker (cont.)

Workflow: Setup (root only)

- Create cache offsets if not already existing
  - Requires same number of load generators
- Create empty Elasticsearch index with following settings:
  - Strict type mappings (Elasticsearch syntax)
  - One shard per Cluster node (configurable)
  - `requests.cache.enable: false`

Ingestion Benchmarker (cont.)

Workflow: Setup (root only)

- Create cache offsets if not already existing
  - Requires same number of load generators
- Create empty Elasticsearch index with following settings:
  - Strict type mappings (Elasticsearch syntax)
  - One shard per Cluster node (configurable)
  - `requests.cache.enable: false`

Workflow: Benchmark

- Each rank chooses one ES node to send to

Ingestion Benchmarker (cont.)

Workflow: Setup (root only)

- Create cache offsets if not already existing
  - Requires same number of load generators
- Create empty Elasticsearch index with following settings:
  - Strict type mappings (Elasticsearch syntax)
  - One shard per Cluster node (configurable)
  - `requests.cache.enable: false`

Workflow: Benchmark

- Each rank chooses one ES node to send to
- Seek to starting byte based on offsets

Ingestion Benchmarker (cont.)

Workflow: Setup (root only)

- Create cache offsets if not already existing
  - Requires same number of load generators
- Create empty Elasticsearch index with following settings:
  - Strict type mappings (Elasticsearch syntax)
  - One shard per Cluster node (configurable)
  - `requests.cache.enable: false`

Workflow: Benchmark

- Each rank chooses one ES node to send to
- Seek to starting byte based on offsets
- Send the requests blockingly, as fast as possible

Ingestion Benchmarker (cont.)

Workflow: Setup (root only)

- Create cache offsets if not already existing
  - Requires same number of load generators
- Create empty Elasticsearch index with following settings:
  - Strict type mappings (Elasticsearch syntax)
  - One shard per Cluster node (configurable)
  - `requests.cache.enable: false`

Workflow: Benchmark

- Each rank chooses one ES node to send to
- Seek to starting byte based on offsets
- Send the requests blockingly, as fast as possible
- Track response time *directly* after

Ingestion Benchmarker (cont.)

Workflow: Setup (root only)

- Create cache offsets if not already existing
  - Requires same number of load generators
- Create empty Elasticsearch index with following settings:
  - Strict type mappings (Elasticsearch syntax)
  - One shard per Cluster node (configurable)
  - `requests.cache.enable: false`

Workflow: Benchmark

- Each rank chooses one ES node to send to
- Seek to starting byte based on offsets
- Send the requests blockingly, as fast as possible
- Track response time *directly* after
- Wait at barrier

Ingestion Benchmarker (cont.)

Workflow: Setup (root only)

- Create cache offsets if not already existing
  - Requires same number of load generators
- Create empty Elasticsearch index with following settings:
  - Strict type mappings (Elasticsearch syntax)
  - One shard per Cluster node (configurable)
  - `requests.cache.enable: false`

Workflow: Benchmark

- Each rank chooses one ES node to send to
- Seek to starting byte based on offsets
- Send the requests blockingly, as fast as possible
- Track response time *directly* after
- Wait at barrier
- MPI Gather all data at root, dump into JSON file

Query Benchmarker

- Measures query-/**read**-performance against previously ingested data.

Query Benchmarker

- Measures query-/**read**-performance against previously ingested data.
- Works through scenarios in a fork-join model.
  - Supports mixing queries in same scenario

Query Benchmarker

- Measures query-/**read**-performance against previously ingested data.
- Works through scenarios in a fork-join model.
  - Supports mixing queries in same scenario
- Features:
  - Distributed, MPI-based

Query Benchmarker

- Measures query-/**read**-performance against previously ingested data.
- Works through scenarios in a fork-join model.
  - Supports mixing queries in same scenario
- Features:
  - Distributed, MPI-based
  - Fully configurable by JSON-DSL; no hard-coded scenarios
    - No need to edit the source code
    - Embeds Elasticsearch syntax internally $\Rightarrow$ accessible for ES-users
    - Simplification of Rally syntax $\Rightarrow$ easy to port

Query Benchmarker

- Measures query-/**read**-performance against previously ingested data.
- Works through scenarios in a fork-join model.
  - Supports mixing queries in same scenario
- Features:
  - Distributed, MPI-based
  - Fully configurable by JSON-DSL; no hard-coded scenarios
    - No need to edit the source code
    - Embeds Elasticsearch syntax internally $\Rightarrow$ accessible for ES-users
    - Simplification of Rally syntax $\Rightarrow$ easy to port
  - Bypasses the cache

Query Benchmarker

- Measures query-/**read**-performance against previously ingested data.
- Works through scenarios in a fork-join model.
  - Supports mixing queries in same scenario
- Features:
  - Distributed, MPI-based
  - Fully configurable by JSON-DSL; no hard-coded scenarios
    - No need to edit the source code
    - Embeds Elasticsearch syntax internally $\Rightarrow$ accessible for ES-users
    - Simplification of Rally syntax $\Rightarrow$ easy to port
  - Bypasses the cache
  - Parses the responses for more data; not only based on HTTP response code

Query Benchmarker

- Measures query-/**read**-performance against previously ingested data.
- Works through scenarios in a fork-join model.
  - Supports mixing queries in same scenario
- Features:
  - Distributed, MPI-based
  - Fully configurable by JSON-DSL; no hard-coded scenarios
    - No need to edit the source code
    - Embeds Elasticsearch syntax internally $\Rightarrow$ accessible for ES-users
    - Simplification of Rally syntax $\Rightarrow$ easy to port
  - Bypasses the cache
  - Parses the responses for more data; not only based on HTTP response code
  - Test mode for easier debugging

Input Format for Query Benchmarker (part 1)

```
[
  {
    "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 */
  },
  ...
```

Input Format for Query Benchmarker (part 2)

```
{
  "search_queries": [
    {
      "body": {
        /* The first of 2 queries sent iteratively (random order) */
      }
    },
    {
      "body": {
        /* The second of 2 queries sent iteratively (random order) */
      }
    }
  ],
  "warmup_time_secs": 30, /* optional */
  "execution_time_secs": 180, /* optional */
  "sleep_between_requests_secs": 0.25 /* optional */
}]
```

Query Benchmarker (cont.)

High-Level Workflow

For each disjunct fork-join benchmark step:

Query Benchmarker (cont.)

High-Level Workflow

For each disjunct fork-join benchmark step:

- Wait for Elasticsearch cluster health to be green

Query Benchmarker (cont.)

High-Level Workflow

For each disjunct fork-join benchmark step:

- Wait for Elasticsearch cluster health to be green
- Partition the hosts onto the load generators

Query Benchmarker (cont.)

High-Level Workflow

For each disjunct fork-join benchmark step:

- Wait for Elasticsearch cluster health to be green
- Partition the hosts onto the load generators
- If warmup time is set: Send queries, discard result (fill OS caches)

Query Benchmarker (cont.)

High-Level Workflow

For each disjunct fork-join benchmark step:

- Wait for Elasticsearch cluster health to be green
- Partition the hosts onto the load generators
- If warmup time is set: Send queries, discard result (fill OS caches)
- After that, until execution time for current step is reached:
  - Select next query in current step

Query Benchmarker (cont.)

High-Level Workflow

For each disjunct fork-join benchmark step:

- Wait for Elasticsearch cluster health to be green
- Partition the hosts onto the load generators
- If warmup time is set: Send queries, discard result (fill OS caches)
- After that, until execution time for current step is reached:
  - Select next query in current step
  - Track before, send query, track after

Query Benchmarker (cont.)

High-Level Workflow

For each disjunct fork-join benchmark step:

- Wait for Elasticsearch cluster health to be green
- Partition the hosts onto the load generators
- If warmup time is set: Send queries, discard result (fill OS caches)
- After that, until execution time for current step is reached:
  - Select next query in current step
  - Track before, send query, track after
  - Parse ES response, save `(latency, docs count)`

Query Benchmarker (cont.)

High-Level Workflow

For each disjunct fork-join benchmark step:

- Wait for Elasticsearch cluster health to be green
- Partition the hosts onto the load generators
- If warmup time is set: Send queries, discard result (fill OS caches)
- After that, until execution time for current step is reached:
  - Select next query in current step
  - Track before, send query, track after
  - Parse ES response, save `(latency, docs count)`
  - Sleep if configured

Query Benchmarker (cont.)

High-Level Workflow

For each disjunct fork-join benchmark step:

- Wait for Elasticsearch cluster health to be green
- Partition the hosts onto the load generators
- If warmup time is set: Send queries, discard result (fill OS caches)
- After that, until execution time for current step is reached:
  - Select next query in current step
  - Track before, send query, track after
  - Parse ES response, save `(latency, docs count)`
  - Sleep if configured
- Wait at MPI barrier for next step

Query Benchmarker (cont.)

High-Level Workflow

For each disjunct fork-join benchmark step:

- Wait for Elasticsearch cluster health to be green
- Partition the hosts onto the load generators
- If warmup time is set: Send queries, discard result (fill OS caches)
- After that, until execution time for current step is reached:
  - Select next query in current step
  - Track before, send query, track after
  - Parse ES response, save `(latency, docs count)`
  - Sleep if configured
- Wait at MPI barrier for next step

See the accompanying report for a more low-level workflow.

How to Create and Run a Benchmark

1. Choose a dataset or create a synthetic one
   - format as NDJSON

How to Create and Run a Benchmark

1. Choose a dataset or create a synthetic one
   - format as NDJSON
2. Define the Elasticsearch type mappings for each attribute

How to Create and Run a Benchmark

1. Choose a dataset or create a synthetic one
   - format as NDJSON
2. Define the Elasticsearch type mappings for each attribute
3. Design the query document
   - Basically just embedding the Elasticsearch API queries into more JSON
   - Note: They can thus be easily tested using cURL/Postman/Insomnia/...

How to Create and Run a Benchmark

1. Choose a dataset or create a synthetic one
   - format as NDJSON
2. Define the Elasticsearch type mappings for each attribute
3. Design the query document
   - Basically just embedding the Elasticsearch API queries into more JSON
   - Note: They can thus be easily tested using cURL/Postman/Insomnia/...
4. Spawn up the cluster using SLURMs MPI environment

How to Create and Run a Benchmark

1. Choose a dataset or create a synthetic one
   - format as NDJSON
2. Define the Elasticsearch type mappings for each attribute
3. Design the query document
   - Basically just embedding the Elasticsearch API queries into more JSON
   - Note: They can thus be easily tested using cURL/Postman/Insomnia/...
4. Spawn up the cluster using SLURMs MPI environment
5. Run the distributed ingestor to ingest the NDJSON corpus

How to Create and Run a Benchmark

1. Choose a dataset or create a synthetic one
   - format as NDJSON
2. Define the Elasticsearch type mappings for each attribute
3. Design the query document
   - Basically just embedding the Elasticsearch API queries into more JSON
   - Note: They can thus be easily tested using cURL/Postman/Insomnia/...
4. Spawn up the cluster using SLURMs MPI environment
5. Run the distributed ingestor to ingest the NDJSON corpus
6. Run the distributed query benchmarker using the query document

How to Create and Run a Benchmark

1. Choose a dataset or create a synthetic one
   - format as NDJSON
2. Define the Elasticsearch type mappings for each attribute
3. Design the query document
   - Basically just embedding the Elasticsearch API queries into more JSON
   - Note: They can thus be easily tested using cURL/Postman/Insomnia/...
4. Spawn up the cluster using SLURMs MPI environment
5. Run the distributed ingestor to ingest the NDJSON corpus
6. Run the distributed query benchmarker using the query document
7. Analyze the output JSON using a language of your choice
   Python example can be found in the Git repo.

Benchmark

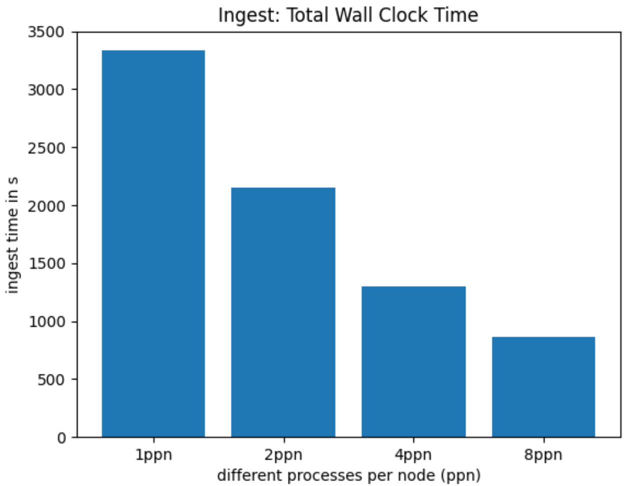
Dataset: NYC Taxis [9]

- All yellow taxi rides in NYC in 2015
- Published by NYC Taxi and Limousine Commision [10]
- 165 million documents, over 75GB
- Also used by Rally (Elastic)
- Most used for scaling testing
- Big documents, but mostly numeric data.

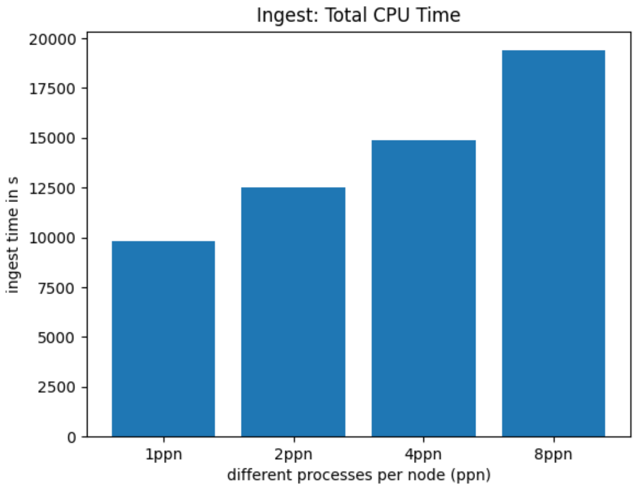
Setup

- 3 nodes on Emmy
- Ethernet
- Ubuntu 22.04 dockerhub image in Singularity
- Elasticsearch 8.11.0 with OpenJDK 21.0.1
- Python 3.9
- OpenMPI 4.1

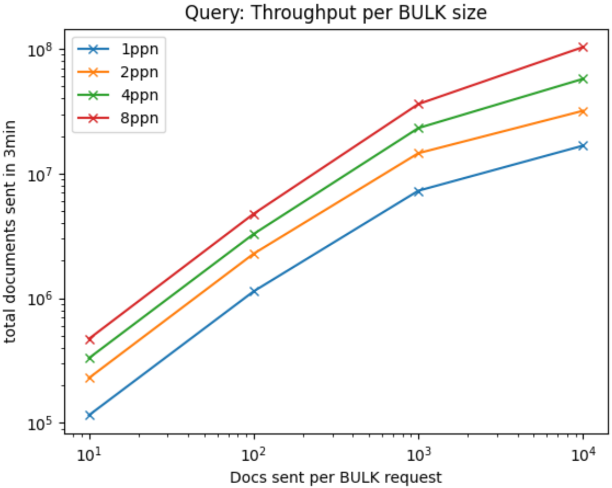
Results


Ingest: Total Wall Clock Time
ingest time in s
3500
3000
2500
2000
1500
1000
500
0
1ppn
2ppn
4ppn
8ppn
different processes per node (ppn)

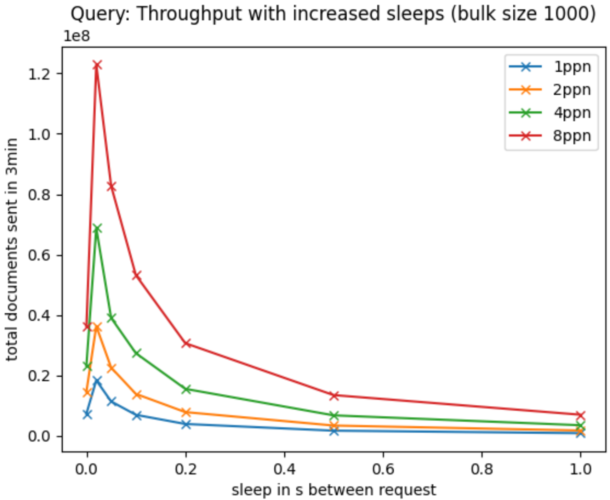
Results


Ingest: Total CPU Time
ingest time in s
20000
17500
15000
12500
10000
7500
5000
2500
0
1ppn
2ppn
4ppn
8ppn
different processes per node (ppn)

Results


Query: Throughput per BULK size
1ppn
2ppn
4ppn
8ppn
10^8
10^7
10^6
10^5
total documents sent in 3min
10^1
10^2
10^3
10^4
Docs sent per BULK request

Results


Query: Throughput with increased sleeps (bulk size 1000)
1e8
total documents sent in 3min
sleep in s between request
1ppn
2ppn
4ppn
8ppn
0.0
0.2
0.4
0.6
0.8
1.0
1.2

Challenges/Open Problems

- Limited response size, hard limit by Elasticsearch’s architecture
- Not possible to map load generator to cluster node according to optimal network topology
- Load generators and clusters cant share the same node
- Elasticsearch requires a custom kernel setting

Summary

- Project was a success, fully implemented both workflow and benchmarker
- Zero configuration needed once the benchmark was initially designed
- Fully integrated into SLURM
- Contributions:
  1. On-demand Elasticsearch Cluster Spawner
  2. Ingestion Benchmarker
  3. Query Benchmarker
  4. Example workflow for canonical dataset

References I

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

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).

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).

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).

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

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).

References II

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

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

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).

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