HPS
https://lquenti.de/

Lars Quentin

# Evaluation of Time-Series Databases

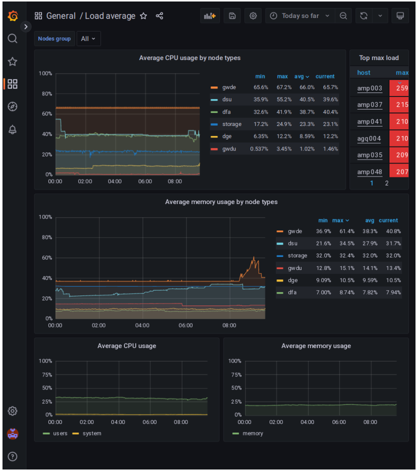
Elasticsearch and InfluxDB

# Table of Contents

# Why do we care about Time Series Metrics Data?

- Usage Overview
- Find Bottlenecks
- Help with Workload Balancing
- Demand Analysis and Forecasting
- Optimize Energy Efficiency

But why do we care about performance?



# Monitoring System Architecture

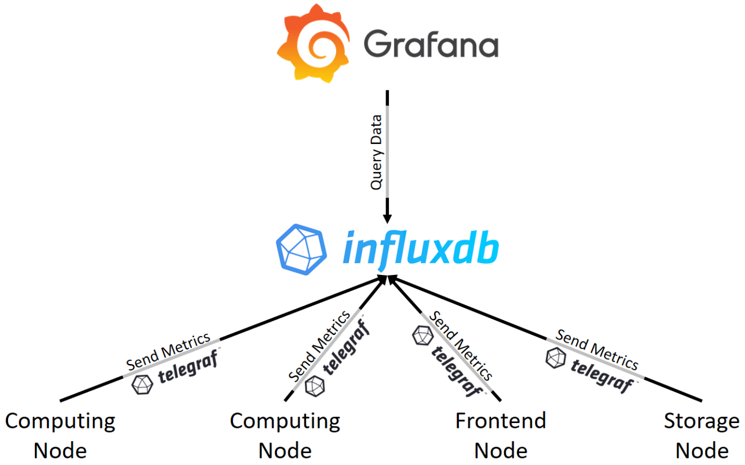



Figure: Monitoring System Architecture

# But actually...

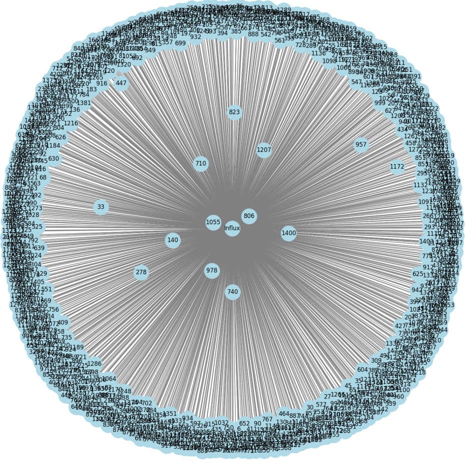


Figure: Emmy's 1422 nodes located in Göttingen

# The Need for Speed: From Lucene to Grafana

- We Evaluate Two Time-Series Databases:
  - Elasticsearch, a distributed search engine.
  - InfluxDB, a time-series database.
- In order to understand why, one has to look at their shared history.

## Lucene

- Java-based Search Engine Library
- Developed in 1999 for Apache Nutch
- Fuzzy Full-Text Search

Figure: Lucene Logo

## Elasticsearch

- Distributed Search Engine
- Based on Lucene
- Developed in 2010
- Used at Wikipedia, Netflix, Stackoverflow, LinkedIn

Figure: Elasticsearch Logo

# Money, Money, Money

- Elasticsearch rose to popularity amongst the DevOps community.
- Thus, it grew beyond the scope of a hobby project and needed funding.
- And a database alone is not enough for business applications.
- Thus, the ELK stack was born.

# ELK(B) stack

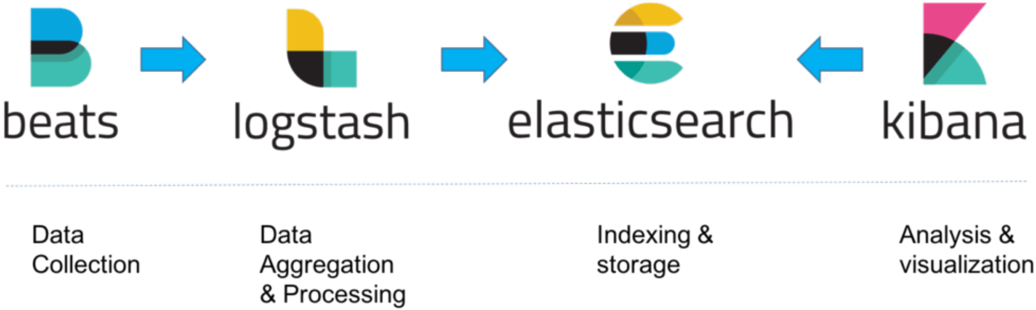
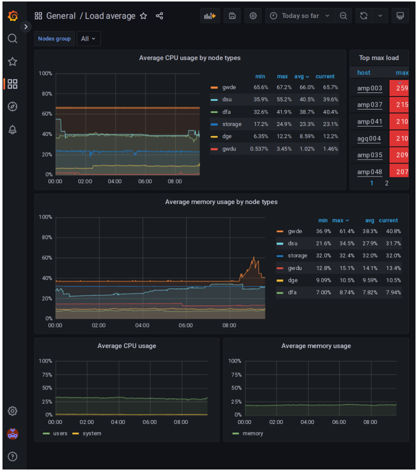



Figure: ELK Stack

# Grafana

- System Monitoring Solution
- Forked from Kibana
- Specialized for time-series data
- Supports multiple data sources
  - No Elasticsearch vendor lock-in
  - Allows for more specialized database technologies



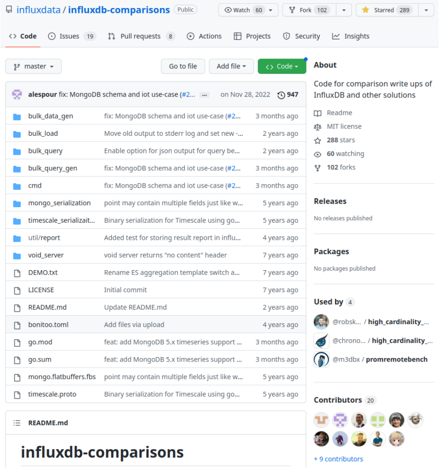
# InfluxDB

- Time-Series Database
- Built for technology applications
- Highly specialized for time-series data
- Also used at the GWDG as a data source for Grafana

Figure: InfluxDB Logo

# Looking in the Rear-View-Mirror: Related Work

- Only a single exhaustive performance comparison of Elasticsearch and InfluxDB.
- Conducted by InfluxData, the company behind InfluxDB.
- Publically available on GitHub.
- In this section, we will deep dive into their methodology and findings.



# Overview

- Measured across 3 vectors
  1. Data ingest performance
  2. On-disk storage requirements
  3. Mean query response time
- Split into 5 disconnected steps
  1. Data Generation
  2. Data Loading
  3. Query Generation
  4. Query Execution
  5. Query Validation

# Influx Comparisons

## 1. Data Generation

- Random and Deterministic (pinned PRNG seed)
- Shared generation logic
- Generated beforehand
- Modelled realistically
  - DevOps related metrics, same structure as telegraf
    - `cpu`, `diskio`, `kernel`, `mem`, `redis`...
  - clamped random walk
    - important for optimizations such as delta compression

# Influx Comparisons

## 2. Data Loading

- KISS
- Batched into bulk queries (default 5000 documents)
- parallelized (default 5 workers)
- sent as fast as possible

# Influx Comparisons

## 2. Data Loading

- KISS
- Batched into bulk queries (default 5000 documents)
- parallelized (default 5 workers)
- sent as fast as possible

## 3. Query Generation

- Random and Deterministic (pinned PRNG seed)
- Shared generation logic
- Generated beforehand

# Influx Comparisons

## 4. Query Execution

- KISS
- Sends parallelized range queries

# Influx Comparisons

## 4. Query Execution

- KISS
- Sends parallelized range queries

## 5. Query Validation

- Done via manual verification
- Ensuring that both aggregation results are approximately the same

# Influx Comparisons

## According to the White paper

- InfluxDB outperformed Elasticsearch by **3.8x** when it came to data ingestion
- InfluxDB outperformed Elasticsearch by up to **7.7x** when measuring query performance
- InfluxDB outperformed Elasticsearch by delivering **9x** better compression

# Influx Comparisons

## According to the White paper

- InfluxDB outperformed Elasticsearch by **3.8x** when it came to data ingestion
- InfluxDB outperformed Elasticsearch by up to **7.7x** when measuring query performance
- InfluxDB outperformed Elasticsearch by delivering **9x** better compression

## Problems

- Bad incentive structure
- Done with Influx version 1
- Not oriented for HPC workloads and topologies
- Data was ingested in bulk

# Under the Hood: Our Methodology

- Extending InfluxData paper's methodology
- Everything not mentioned is the same.
- Adapted to our use case
  - This is a huge feat; Emmy is big
  - We run the recommended production configuration
  - We mainly focus on write, not query read
    - Since this is the bulk of the work
- Split into distinct phases as well.
  - KISS! KISS! KISS!

# 1. Data Generation

- Also random and deterministic, pinned PRNG seeds
- Only generating hardware / kernel measurements, no application metrics
- We use clamped 1D perlin noise
- One file per ingest worker!
  - Less error prone!
  - KISS!

# 2. Data Ingestion

- We don’t use bulk ingestion
  - instead, data of one node per request
- Sending as fast as possible
- Flushing at the end
- Faster is better

# 3. Check Index Compression

- We do not trust their analytics
- Multi-Step process
  1. Get size of data directory
  2. Fill the data
  3. Flush and Compress
     - **InfluxDB**: Tree Compaction
     - **Elasticsearch**: Force Merge API
  4. After that, we measure again
- Smaller $\Delta$ is better

# 4. Design Queries

## Methodology

1. Get a real world Grafana dashboard
2. Extract the queries through the networking tab
3. Port them to the Query Languages
4. Make them parameterized

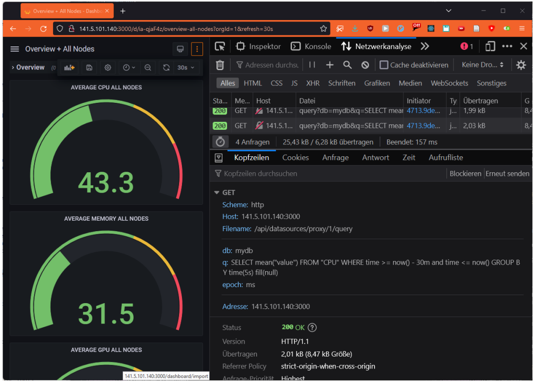



Figure: Extracting through Networking Tab

# 5. Benchmark Queries

- We test querying while ingesting data
- Linear step increment of index size
  - correllate the response time
- Faster is better

# The Podium: Results and Conclusion

Stay tuned ;-)