Building and visualizing a NoSQL time series data store using InfluxDB2, Python and Grafana

Author:	Philipp Moritzer
	5034255
	pmoritzer@stud.hs-bremen.de
Supervisor:	Prof. Dr.-Ing. Uta Bohnebeck
	uta.bohnebeck@hs-bremen.de
	Hochschule Bremen

Submission:	31st July 2022

Table of contents

Introduction

Big Data, IoT, and analytics, and as a result, new methods of storing this fluid data have emerged. Almost all streaming data, particularly IoT data, but also real-time analytics and server and application monitoring, has a time stamp and thus is time series data. This project will go on to demonstrate the differences between traditional databases and NoSQL (time series) databases, why they are used, and how to build a project on top of them. The completed project stores, visualizes, and analyzes data using InfluxDB2, Python, and Grafana using time-stamped CSV data containing bird migration data as a time analytics approach.

Fundamentals

NoSQL basics

NoSQL is a different approach to database design that excludes traditional relational database management systems. While relational database systems attempt to abstract the underlying data structure, NoSQL databases are more data-specific, which should make them more efficient for this specific data. They are intended to implement data structures in a manner that is more closely aligned with the target system, whereas traditional relational databases cannot be structured in this sort of way. The motivation for using NoSQL databases is that they do not require the use of a predefined schema, resulting in a simpler design, horizontal scaling, and greater control over data availability. Because NoSQL distributes its data, it must adhere to the CAP theorem, which states that only two of the three following attributes can be achieved: consistency, availability, and partition tolerance. Depending on the application to implement and the needs of the use cases, different databases prioritize consistency over availability, while others prioritize availability over consistency. [1, 2, 3]

NoSQL vs Relational Databases

NoSQL and relational databases both have the same basic goals: to store and retrieve data, as well as to coordinate changes. The distinction is that NoSQL databases abandon some of the capabilities of relational databases in order to increase scalability. NoSQL databases, in particular, typically have much simpler coordination capabilities than traditional relational systems, sometimes even none at all. NoSQL databases typically remove all or most of the SQL query language, as well as a complex optimizer required for SQL to be useful. The benefit of the tradeoff is that NoSQL databases are often very simple and can handle unstructured data, resulting in higher scalability in the best case. This has the disadvantage of losing overview by storing a large amount of unstructured and unformatted data, and optimization is in the hands of the developr rather than the optimizer when using relational databases. [4]

Time Series Databases

A time series is a grouping of values organized by time. Stock prices, for example, may fluctuate throughout the day as trades are executed, or a weather sensor may record atmospheric temperatures every minute. Any event that is recorded over time, whether on a regular or irregular basis, is considered time series data. A time series database is intended to make time series data retrieval and statistical analysis easier. While time series data like orders, shipments, and logs have long been stored in relational databases, the data sizes and volumes were frequently insignificant. As data grew faster and larger, special-purpose databases became necessary. Time series databases, for example, address the needs of increasing, high-velocity data volumes from IoT, event, and application logs. These are frequently write-intensive workloads. As a result, memory buffering is frequently used to support fast writes and reads in time series databases. Looking at the CAP Theorem, InfluxDB focuses on either CP or AP but tries to find a middle ground between the two. [3, 5]

Why use a time series database

It is necessary to consider the problems that time series databases attempt to solve. IoT devices and real-time analytics generate a large amount of data, and with the increasing amount of data produced with a time stamp, there must be a way to deal with this. Regular time series data, such as measurements taken every 10 seconds, and irregular data, such as API requests, are the two types of time series data. A modern time series database should be capable of handling both types of data. Time series databases are designed to deal with problems that arise with these high volume measurement data, and as a result, they solve three major characteristics with the data they use: exceptionally high volume, natural time order, and the entire set of data being more valuable than individual records. When these issues arise in the development of an application or system, it is recommended that a time series database is used. They concentrate on optimizing frequent writes, merging data, and constructing sums and averages in order to treat the data as a whole and store combined data beyond the retention period. They also provide optimized query languages for handling data based on the use case. [7] Consider using a traditional SQL database to store time series data. In general, it is possible, but problems arise when a certain data threshold is reached. It is common in time series databases to set a retention period so that data from a specific period is stored as a cumulation. Using the same logic in SQL, there will eventually have to be the same number of deletes as inserts, a use case that a traditional database system is not designed to handle well. It is

also possible to shard a traditional database to scale across systems, but this requires more application code to be written. Time series databases and its libraries, such as InfluxDB and its python library, support this out of the box. [8]

InfluxDB basics

According to https://db-engines.com/, InfluxDB is the most popular NoSQL time series data created by InfluxData. Its primary application is handling massive amounts of time-stamped data. Collecting IoT data is a common example because every data set in IoT is time-based. However, it is frequently used for Analytics as well as IoT data, for example, in this project time-stamped data for bird migration will be handled using InfluxDB2. The first version of InfluxDB came out in 2013 and the 1.x version is still commonly used. However, in 2020 InfluxDB was released as a stable version and is the way to go. It features a new query language, which will be explained in the next section, as well as a standalone binary with a graphical web interface to explore data. [9, 10]

Flux Query Language

InfluxDB includes a query language called Flux for querying, analyzing, and acting on data. Flux queries perform various operations, but in general, a query is constructed as follows:

```
from(bucket)
    |> range(start: x, stop: y)
    |> #filteroptions
```

The pipe symbol `|>` marks pipe forward data and therefore every function or expression that follows after that symbol takes the former expression as an input.

Typically, a bucket, which is a store for time series data in InfluxDB, will be chosen first to serve as a source. A time range is required for a flux query because queries without a time window are resource-intensive; therefore, it is up to the user to specify a common time-window that complements the data required. When only these fields are specified in the query, the result is an output table with a timestamp as the individual identifier key and the measurements. Following that, one can filter the data based on their requirements, such as after a field. There are also many built-in functions that can be used to reduce, sum up, interpret, or map data. This can be used to prepare the data for visualization or other purposes (e.g. classification, machine learning). As an example, consider the following query that filters data in a time range of 12 hours and selects the fields `lat` and `lon` as location data and with the measurement tag `location`:

```
from(bucket: "bucket")
    |> range(start: 2021-01-01T00:00:00Z, stop: 2021-01-01T12:00:00Z)
    |> filter(fn: (r) => r._measurement == "location" and r._field == "lat"
and r.lat == "lat")
```

[11, 12]

InfluxDB vs Timescale vs Prometheus

Prometheus and TimescaleDB serve a similar purpose than InfluxDB by being time series databases. While InfluxDB is the most used time series database according to (https://db-engines.com/de/ranking/time+series+dbms, 10.07.2022, 18:06) the other two databases are very commonly used. The question that arises is why alternatives to InfluxDB exist and how the other candidates differ.

Prometheus:

- Uses its own query language PromQL
- More features for monitoring purposes
- Less support for for real-time analytics or machine learning
- Only milisecond timestamps vs InfluxDB's nanoseconds
- Less resource usage

TimescaleDB:

- Based on PostgreSQL
- Uses SQL as query language
- Relational data model
- Inferior performance to InfluxDB

Applications running in the cloud infrastructure often use the vendor's own database. For AWS it is called Amazon Timestream, for Azure Azure Time Series Insights. They are in these type of compositions because it centralises the management to the vendor infrastructure. [13, 14, 15]

Classifiying Data in InfluxDB using Naive Bayes Classification

Based on the input, the Naive Bayes classification is described as a probabilistic classifier that should be able to predict a probability. It is based on the Bayes Theorem: $$ P(A|B) = {P(A|B)😛(A) \over P(B) } $$

A distribution over a set of classes is calculated given an observation of an input. After that, the classifier can be trained to determine which class has the highest probability. Consider the the following:

$$P(Class : | : Field)$$

Given enough training data the probability can be predicted based on a given field. A simplified example would be that if a bird is in the northern hemisphere in the winter we could predict what kind of species the bird belongs to, e.g.:

$$ P("Northern":"Winter"😐:"Larus:fuscus") $$

In words, if a data point is in the northern stratosphere and it is winter we can predict with a certain probability that the bird is a *Larus fuscus*. This type of classification can be performed using the Flux query language.

[11, 16, 17]

Tutorial

Goals

The purpose of this project is to process a large amount of time series data. The data is stored using a NoSQL approach with InfluxDB. A Grafana Dashboard is constructed to visualize the data, and appropriate visualisation tools are employed. The data will be parsed using Python and processed to InfluxDB using functional programming in combination with the Python client library provided by InfluxDB.

The data - Movebank: Animal tracking

The data processed in this example is

Prerequisites

- Python 3.10 (with pip3) https://www.python.org/downloads/
- Docker (20+, Version 20.10.13 is used for this project) https://docs.docker.com/get-docker/

- docker-compose installed (1.20+, Version 1.29.2 is used for this project) https://docs.docker.com/compose/install/

It is advised to use the most recent versions. The following commands can be used to determine whether the requirements have been met:

```
$ python3 --version
# Output: Python 3.10.*
$ docker -version
# Output: Docker version 20.10.13, build a224086
$ docker-comopse --version
# Output: docker-compose version 1.29.2, build 5becea4c
```

Spinning up the composition

docker-compose

There is no need to install InfluxDB or Grafana locally to quickly spin up the environment. Although it is possible to follow along using locally installed instances, docker-compose makes it easier to spin up the services. If no project root has yet been created, create a new folder for the project and add the following `docker-compose.yml` file:

```
version: "3.9"

services:
  influxdb:
    image: influxdb:2.2.0-alpine
    container_name: influxdb
    ports:
      - "8083:8083"
      - "8086:8086"
      - "8090:8090"
      - "2003:2003"
    volumes:
      - ./data/influxdb:/var/lib/influxdb2
    networks:
      - network1

  grafana:
    image: grafana/grafana:7.5.16
    container_name: grafana
    ports:
      - "3000:3000"
    user: "0"
    links:
      - influxdb
    volumes:
      - ./data/grafana:/var/lib/grafana
    networks:
      - network1

networks:
  network1:
```

Volumes are used by both InfluxDB and Grafana. The next step is to make sure that a directory called `data/` is created in the project root directory, which contains two empty subdirectories called `grafana/` and `influxdb/`. These directories will be filled to persist data when using these applications. Later in the tutorial, they will be filled in. This is how the project should be structured now:

```
- <project-root>
  - data/
    - grafana/
    - influxdb/
  docker-compose.yml
```

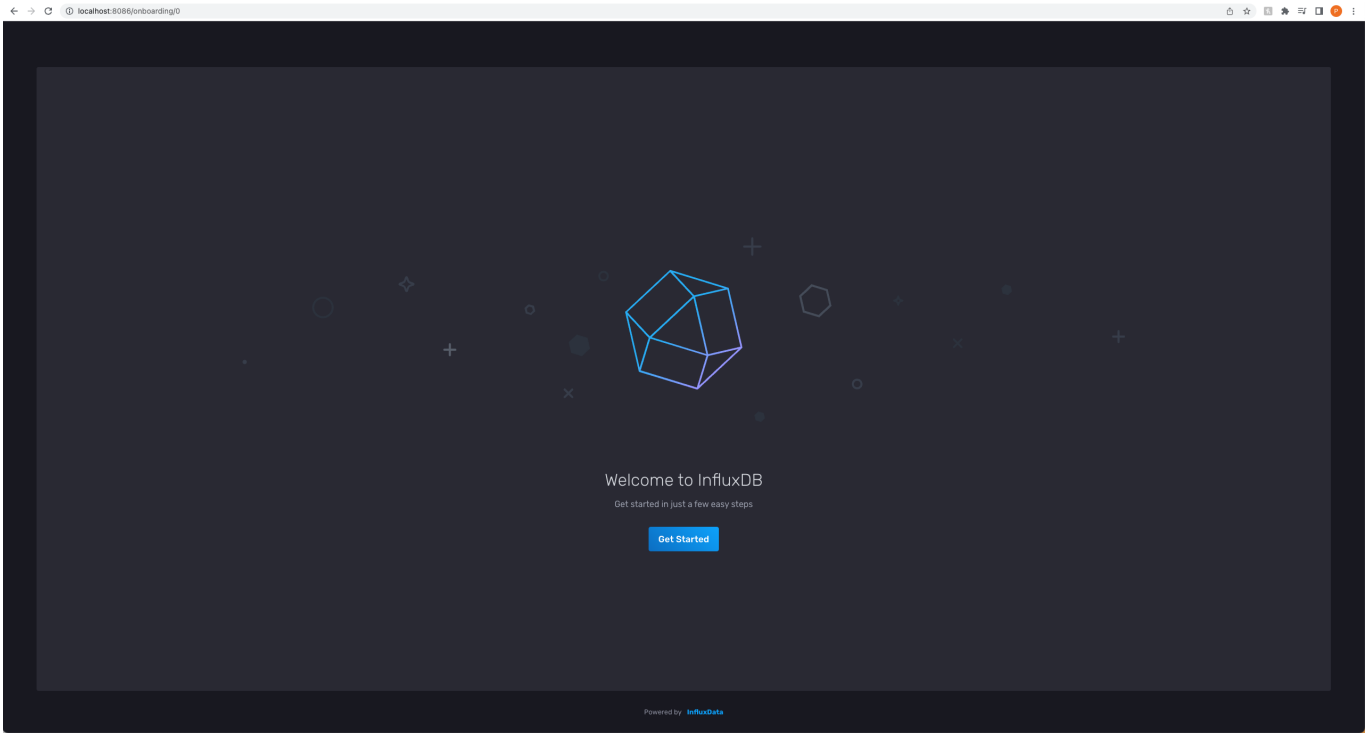
Also a network is added in the `docker-compose.yml` file so our Python application is able to communicate with the services on the same network later on when using Docker. Docker-compose is used to spin up a local instance of InfluxDB and Grafana. To start both services, enter

```
docker-compose up
```

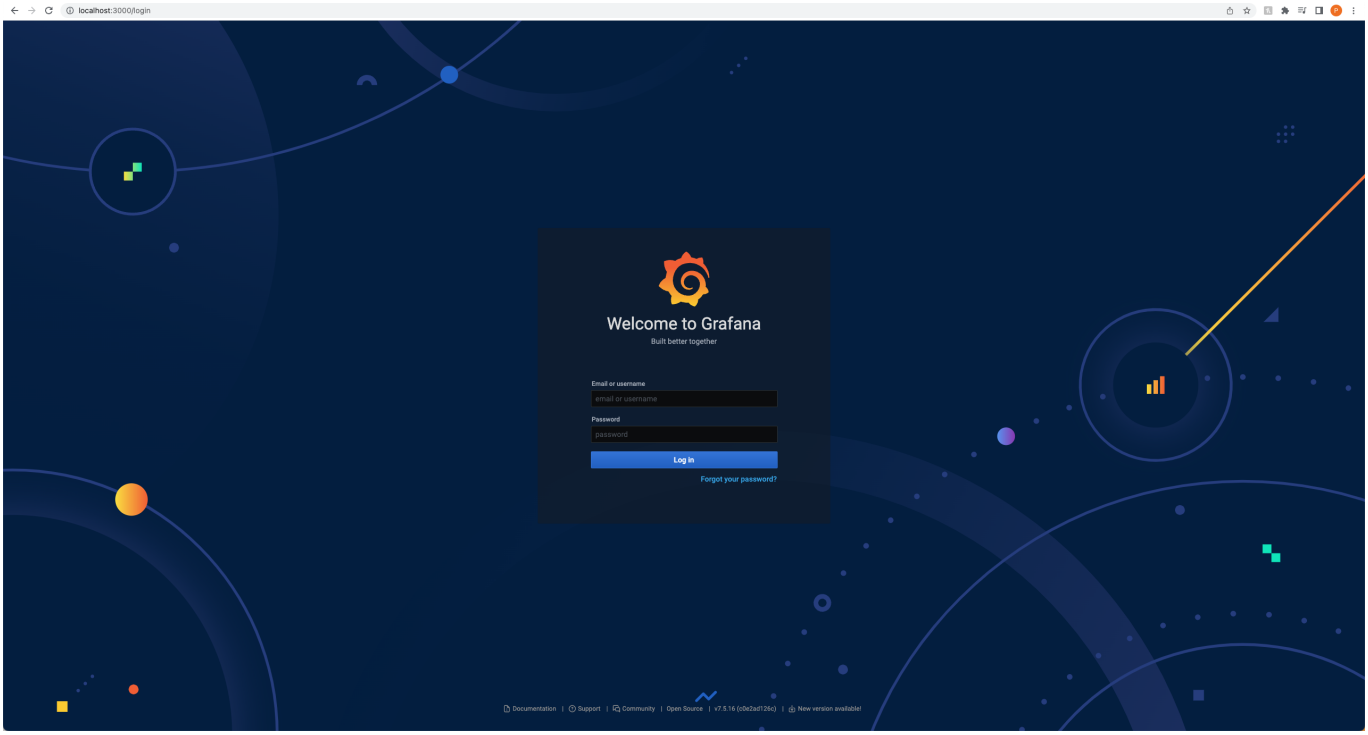
> Note for Unix-Users: If the Grafana-Container fails to start due to permission errors, the permission of the ./data/grafana Folder should be changed to 472 and made sure it is owned by the account using it.

The InfluxDB exposed the Port 8086 for the web interface and should now be reachable locally on `http://localhost:8086/`:

InfluxDB on localhost:8086



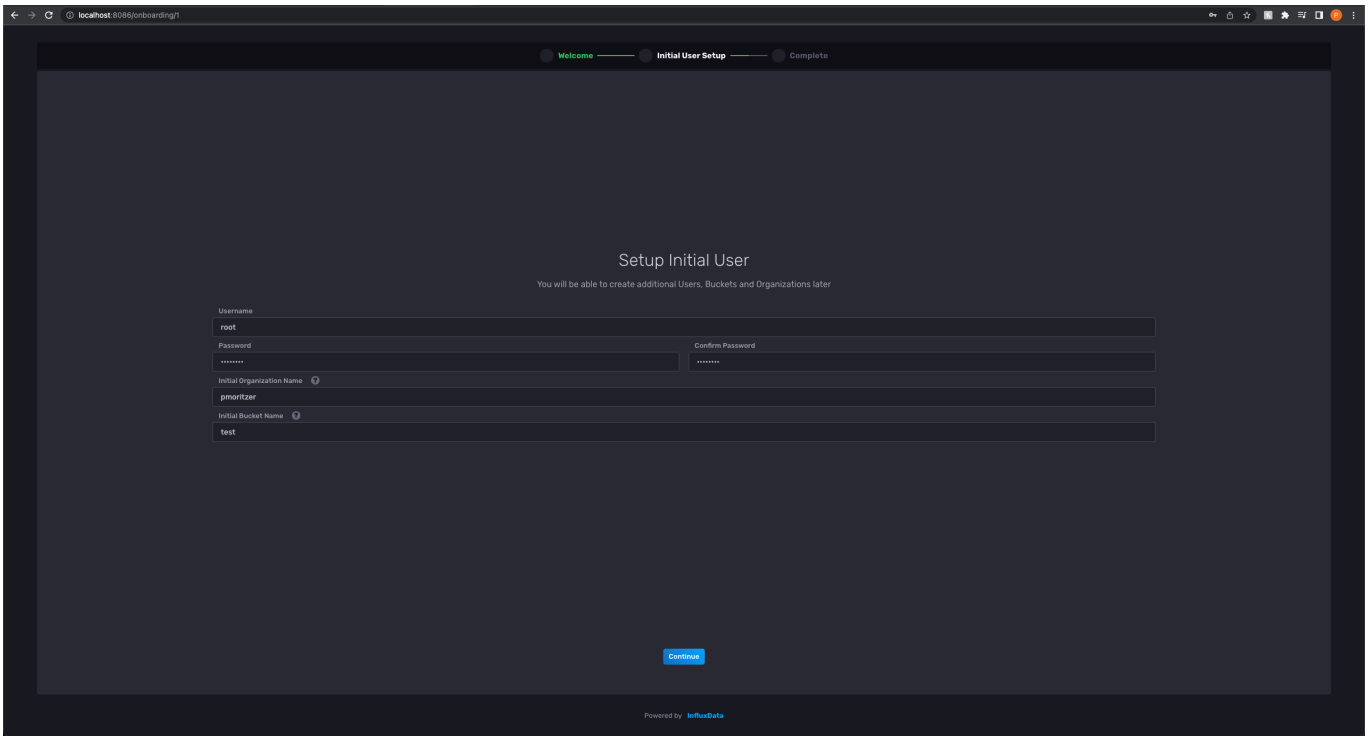
While Grafana uses Port 3000 for its web interface and should be reachable by typint `http://localhost:3000` in a browser:



Grafana on localhost:3000

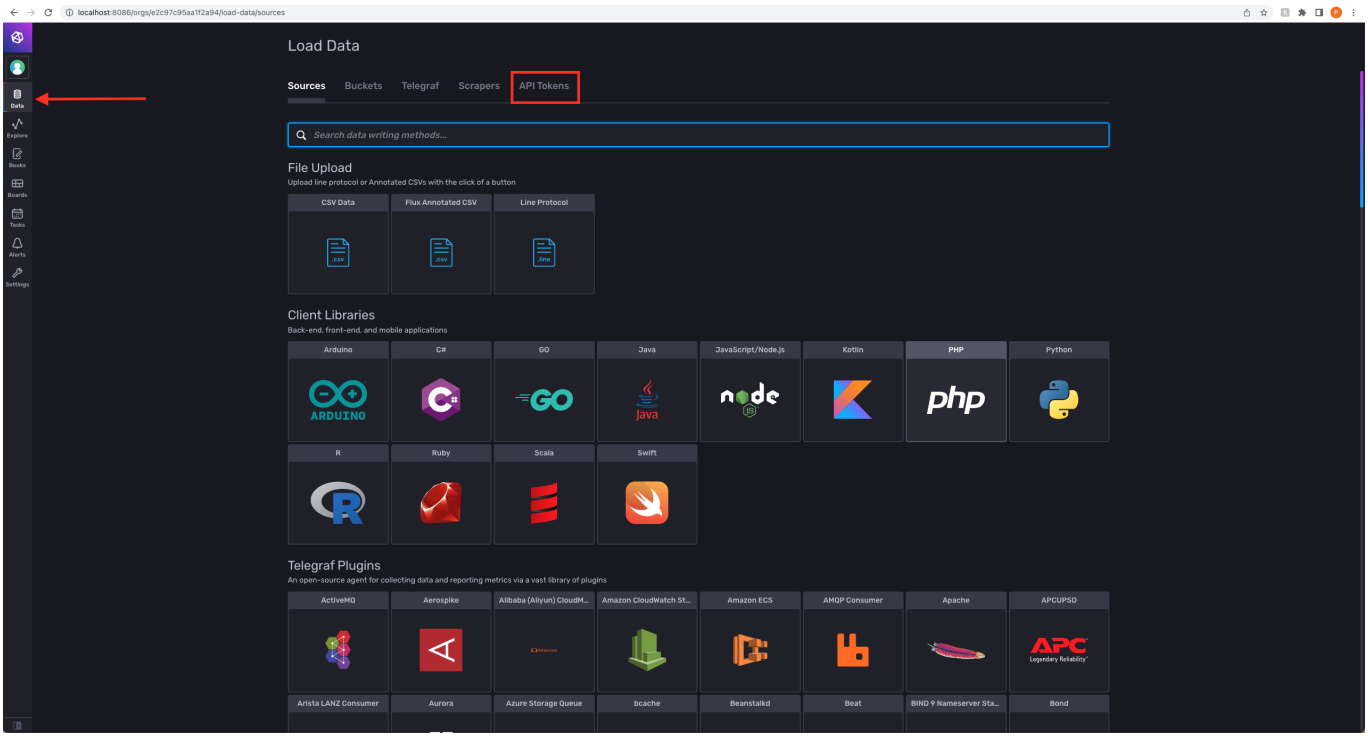
Setting up InfluxDB

To access the InfluxDB web interface, go to `localhost:8086`. Press 'Get Started' to begin the setup procedure. The setup then asks for a login, password, organization name, and an initial bucket name in the next stage. The password for `root` is `password`, and the initial organization name is my tag `pmoritzer`. The bucket name must be set initially, but the bucket is not needed because our data processing unit python application will create a bucket dynamically in the future. The information should be saved because it will be required to log in to the web interface later on and the organization name will serve as an identifier in the client application.

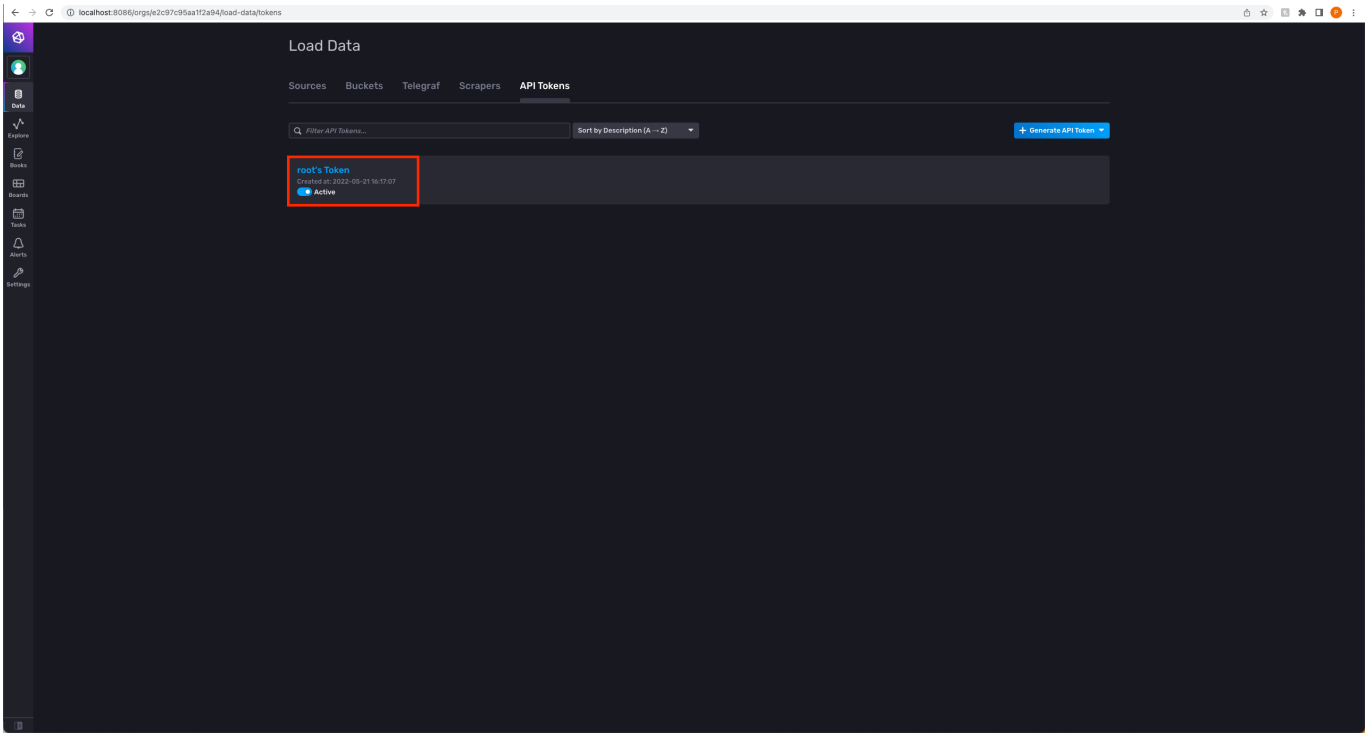


First setup step for InfluxDB

Because we want to inject data manually, simply select "Configure later" in the following step. The final step in setting up InfluxDB is to obtain the necessary API token. To do so, select the Data tab in the left sidebar, followed by the "API Tokens" tab in the tab view on top. A window will appear when you click on root's Token including the API Token. To access the InfluxDB from an external application, the token should be saved somewhere. Because security is not an issue in this proof of concept, we can use the root token; however, in a production environment, separate users with access rights should be set up.



InfluxDB API-Token 1/2



InfluxDB API-Token 2/2

Writing the client application

Setting up the environment

First, environment variables that will be required to connect to the local instance of InfluxDB will be set. Using an environment file, a folder called `<project-root>/env/` will be created within the project root directory. A file called `env.app` will be created that looks as follows:

```
INFLUX_URL=http://influxdb:8086
INFLUX_TOKEN=<root-api-token>
INFLUX_ORG=<org-name>
```

Replace the `INFLUX_TOKEN` property to equal the generated API-Token from the setup and the `INFLUX_ORG` properties with the organization name set. The data processing client application is able to connect to the database by loading these definded properties.

Setting up the application

Create a file called `main.py`in the project's root directory that will serve as an entrypoint for the client application. First we will do a simple output to check whether the application works.

```
print ("Hello World")
```

main.py

```
$ python3 main.py
# Output: Hello World
```

For the dependencies a `requirements.txt`file is created in the project's root directory with following content to install the python client library for InfluxDB using pip. To make sure they are locally available, run the follwing command:

```
influxdb-client == 1.29.0
```

```
$ pip3 install -r requirements.txt
```

Next, we're going to dockerize the application. To do so, a Dockerfile will be created. Docker ensures that the application can run with the dependencies defined regardless of the system's environment.

```
# syntax=docker/dockerfile:1
FROM python:3.10.4-slim-bullseye

WORKDIR /app

COPY requirements.txt requirements.txt

RUN pip3 install -r requirements.txt
```

```
COPY . .

CMD ["python3","-u","./main.py"]
```

Create a `run.sh` script (or `.bat` for Windows Users) that executes following commands in a row:

```
$ docker build --no-cache -t influxdb-sample .
$ docker network create project_network1
$ docker run --network=project_network1 --env-file ./env/env.app influxdb-
sample
```

This script builds our application into a Docker Image, creates a network and starts the container while making sure it can communicate with the services on the same network, `network1`.

```
$ ./run.sh
# Output: Hello World
```

Every time we want to start the application locally we can run the `run.sh`script.

Implementing functionality

First a function for connecting to InfluxDB will be implemented. create a folder `app/` including a file `connect_to_influx.py`. The content of the file will be as follows:

```
import time
from influxdb_client import InfluxDBClient

def connect_to_influxdb(url, token, org, retries=10, tried=0) ->
InfluxDBClient:
    print("Connecting to InfluxDB on " + url)
    client = InfluxDBClient(url=url, token=token, org=org, debug=True)
    health = client.health()

    if health.status == "pass":
        print("Connected to InfluxDB on " + url + "/")
        return client
    else:
        if tried < retries:
            print("Connection to {} refused, retrying {}
times.".format(url, (retries-tried)))
            tried += 1
            time.sleep(10)
            return connect_to_influxdb(url, token, org, retries, tried)
        else:
            raise ConnectionError("Connection to influxdb failed.")
```

The file contains a function that returns an InfluxDBClient connection object. The method invokes the constructor with the required information (URL, API-Token, organisation). If the connection is successful and healthy, this custom wrapper method returns the

InfluxDBClient object. If it is not, it tries to connect to the given InfluxDB instance as many times as specified. If it fails, the application will exit with a connection error.

The application logic for writing our sample data to InfluxDB must now be implemented. First, we'll import the database connection function we just wrote, as well as any other imports required for this logic. Among these are the InfluxDB client library, csv for handling CSV data, and the `rx` functional programming library.

```
from app.connect_to_influx import connect_to_influxdb

from influxdb_client import Point, WriteOptions

from csv import DictReader
from collections import OrderedDict
from decimal import Decimal
import os

# rx for functional programming
import rx # functional programming library
from rx import operators
```

The parameters for connecting to the database will be loaded using environment variables in the following step. It must be ensured that the application is started using Docker with env-files. The script will fail if these environment variables are not specified.

```
url = os.environ['INFLUX_URL'] or 'http://localhost:8086'
token = os.environ['INFLUX_TOKEN'] or "<token>"
org = os.environ['INFLUX_ORG'] or 'pmoritzer'
```

Following that, a function will be defined to parse each row of the CSV containing the bird migration data. We will need the following fields for this analysis:

- timestamp
- event-id
- lon
- lat
- manually-marked-outlier
- individual-taxon-canonical-name
- tag-local-identifier
- individual-local-identifier

The remaining columns have little value for data analysis, so they are not considered in this project. The InfluxDB interface's Point-Object is returned by the following function. To create the dynamic data structure, we can use the provided Builder-Pattern. Each Point will later be stored as a dataset in an InfluxDB bucket.

```
def parse_row(row: OrderedDict):

    return Point("migration-point").tag("type", "migration-
value").measurement("migration") \
        .field("event-id", row['event-id']) \
```

```
        .field("lon", Decimal(row['location-long'])) \
        .field("lat", Decimal(row['location-lat'])) \
        .field("manually-marked-outlier", row['manually-marked-outlier']) \
        .field("individual-taxon-canonical-name", row['individual-taxon-
canonical-name']) \
        .field("tag-local-identifier", row['tag-local-identifier']) \
        .field("individual-local-identifier", row['individual-local-
identifier']) \
        .time(row['timestamp'])
```

The function above is then called for each row of the csv file using functional programming. It must be ensured that the.csv dataset mentioned earlier is located in the project's root directory and is named `migration_original.csv`.

```
data = rx \
    .from_iterable(DictReader(open('migration_original.csv', 'r'))) \
    .pipe(operators.map(lambda row: parse_row(row)))
```

The command above maps each row from the source data set to an Ordered Dictionary entry and parses it to a. Afterwards we are left with a dictionary that is readable by the InfluxDB data client.

> Example adapted from: https://github.com/influxdata/influxdb-client-python/blob/master/examples/import_data_set.py (visited: June 4th, 22:00)

The code below establishes a connection to InfluxDB, creates a new bucket using the Bucket API, and writes data to InfluxBD using the Write API, all of which are provided by the Python library.

```
with connect_to_influxdb(url, token, org) as client:
    bucket_name = 'bird-migration'
    bucket_api = client.buckets_api()
    old_bucket = bucket_api.find_bucket_by_name(bucket_name=bucket_name)
    if old_bucket:
        try:
            bucket_api.delete_bucket(old_bucket)
        except:
            exit()
    bucket = bucket_api.create_bucket(bucket_name=bucket_name, org=org)

    with client.write_api(write_options=WriteOptions(batch_size=50000,
flush_interval=10000)) as write_api:
        write_api.write(bucket="bird-migration", record=data)

    query = 'from(bucket:"bird-migration")' \
            ' |> range(start: 0, stop: now())'
    result = client.query_api().query(query=query)
    print()
    print("=== results ===")
    print()
```

The results will be validated by committing the following flux query using the Python code:

```
from(bucket:"bird-migration") |> range(start: 0, stop: now())
```

Using the InfluxDB web client, run the same flux query to see if the data is in our database.

Setting up Grafana

python3 -m pip install influxdb

Python

Grafana

Run sample project

Summary

Summary

Repository & Live-Demo

Sources

- [1] Brad Dayley. Sams Teach Yourself NoSQL with MongoDB in 24 Hours, Video Enhanced Edition. O'REILLY. 2014.
- [2] Kasun Idrasiri, Sriskandarajah Suhothayan. Design Patterns for Cloud Native Applications. O'REILLY. 2021.
- [3] CloudLab. NoSQL - CAP Theorem. Author unknown. Date unknown. URL: https://cloudxlab.com/assessment/displayslide/345/nosql-cap-theorem#:~:text=NoSQL%20can%20not%20provide%20consistency,Consistency%2C%20Availability%20and%20Partition%20Tolerance. (visited: 10.07.2022, 20:15)
- [4] Ted Dunning, Ellen Friedman. Time Series Databases: New Ways to Store and Access Data. O'REILLY. 2014.
- [5] Joe Reis, Matt Housley. Fundamentals of Data Engineering: Plan and Build Robust Data Systems. O'REILLYs. 2022.
- [6] Paul Dix. Why Build a Time Series Data Platform?. db-engines. 2017. https://db-engines.com/en/blog_post/71 (visited: 10.07.2022, 20:15)
- [7] Kovid Rathee. The case for using timeseries databases. 2021. URL: https://towardsdatascience.com/the-case-for-using-timeseries-databases-c060a8afe727 (visited: 10.07.2022, 20:15)
- [8] db-engines. InfluxDB System Properties. 2022. URL: https://db-engines.com/en/system/InfluxDB (visited: 10.07.2022, 20:15)
- [9] influxdata. influxdata - Documentation. 2022. URL: https://docs.influxdata.com/ (visited: 10.07.2022, 20:15)
- [10] influxdata. Get started with Flux. 2022. URL: https://docs.influxdata.com/influxdb/cloud/query-data/get-started/ (visited: 10.07.2022, 20:15)
- [11] Rohan Sreerama. A Deep Dive into Machine Learning in Flux: Naive Bayes Classification. 2020. URL: https://www.influxdata.com/blog/deep-dive-into-machine-learning-in-flux-naive-bayes-classification/ (visited: 10.07.2022, 20:15)
- [12] Igor Bobriakov. Prometheus vs InfluxDB. 2020. URL: https://www.metricfire.com/blog/prometheus-vs-influxdb/ (visited: 10.07.2022, 20:15)
- [13] db-engines. System Properties Comparison InfluxDB vs. Prometheus vs. TimescaleDB. 2022, URL: https://db-engines.com/en/system/InfluxDB%3BPrometheus%3BTimescaleDB (visited: 10.07.2022, 20:15)

- [14] United Manufacturing Hub. Why we chose timescaleDB over InfluxDB. 2022, URL: https://docs.umh.app/docs/concepts/timescaledb-vs-influxdb/
- [15] Team Magic. Building a Naive Bayes classifier using Flux. 2020. URL: https://github.com/RohanSreerama5/Naive-Bayes-Classifier-Flux/blob/master/Naive%20Bayes.pdf (visited: 10.07.2022, 20:15)
- [16] Rohan Sreerama. Naive-Bayes-Classifier-Flux. 2020. URL: https://github.com/RohanSreerama5/Naive-Bayes-Classifier-Flux (visited: 10.07.2022)