Creating a KNMI.nl data loader

Quinten Cabo Radboud University

ABSTRACT

In this report I detail the process of making data from the Dutch meteorological institute more approachable for data scientists that want to make weather prediction models. To this end I created a pipeline that collects the incoming data from KNMI into an sql database. This data is then served through an api to a torch data loader as the final interface for the data scientist.

1 INTRODUCTION

The Dutch Koninklijk Nederlands Meteorologisch Instituut (KNMI) continuously collects data about the weather using sensors throughout the Dutch Kingdom. They provide this data to the world using their notification service [3].

The data in this dataset is not easy to access. You first need to make an account on the knmi.nl website. Then you need to apply to get two different access tokens. If you get accepted, you can use token 1 to subscribe to a mqtt stream where KNMI publishes new data. Items in this stream are json documents that contains a link. You can make a specific request to this link with token 2. In the response you get a temporary download link from which you can actually download the data. You should not use an authentication header on this final link because then it does not actually download.

As you can gather, getting access to the data is not a good experience. The main goal of this project was to make it easier to get access to the weather data from KNMI.

The intended use case of the dataset is data scientists creating weather prediction models. Especially models which are well suited for a time series dataset.

2 QUALITY ASSESSMENT

Once you actually manage to download a file using the process in the introduction it is actually an .nc file. It used to be h5 files but during the course of this project KNMI changed to nc files.

The NC (NetCDF) is a binary file format much like a zip file but without directories. Each file in an nc file is somewhat confusingly called a dataset. Each dataset in an nc file stores multidimensional arrays and scalar values. An .nc is like a zip file without any directories. A tree with only leaves.

Before accessing the quality of the data I collected 85 nc files. Then, using the program in Appendix 1 I established that every nc file had the same datasets inside of it. This was good news because that made the shape of the data uniform across nc files (unlike the h5 files). Each nc file has 103 different database files inside of it.

The quality of the KNMI data itself was actually surprisingly good. Each dataset turned out to have very useful attributes. Namely ‘comment‘, ‘long name‘, ‘unit‘, ‘size‘, ‘shape‘ and ‘datatype‘. A table with these values for all datasets can be found in the appendix. Some of the comments were very detailed. An example of a detailed comment can be found in the Appendix. From these comments I learned that **each KNMI nc file contains the measurements of 103 variables by 69 weather stations at a single moment in time.**

The quality of the data in terms of accuracy seems to be high as it comes from scientific measurement tools.

The dataset contains good information about the units of data. For instance pressure units or temperature in c. The time unit however took seconds since 1950-01-01 00:00:00 which seems like an odd choice until you learn that the KNMI was founded in 1950.

2.1 Quality Issues

There are many missing values. But the fact that we load then as a MaskedArray does not make this a large issue. This is the best way of doing missing values I have ever seen and I will use this myself moving forward when possible.

All vector data seems to be quite usable and in theory very suitable for machine learning models. Of course, some data fields might be more useful than other data to craft the model but this judgement is up to the data scientist.

One thing to mention is that the shape of the vectors is (69,1). That is a list of 69 lists of lists with 1 value. There seems no reason why it is like that instead of just 69 values.

A large issue with the data is the differing station names. Most of the time the station names are the same across nc files, but sometimes they are not. This means that if you would train on the data naively you would get a timeseries where some of the values of a specific moment are actually from different stations in the Netherlands.

Furthermore, this data is only really useful when it is presented as a time series. But, as each file only has the measurements of one moment in time from 69 different stations, the files need to be combined somehow to make it interesting.

I will mainly improve the quality by improving how easy the data will be to load.

3 RESHAPING/WRANGLING

Because I want the data scientist to query the dataset using the data loader I think a relational data model is the best fit. Using a relational database I can easily query the central database for exactly the data that the data scientist requested. A relational database additionally provides an easy way to combine the data from multiple nc files because it can just be added to the table. The database schema only holds a single table called Measurement. Each row in the the table holds the measurements of a single station at a single point in time. I do not need more than one table. You can already filter the columns you want in the select statement. Having more tables will only needlessly increase complexity. The schema of this table can be found in the appendix

3.1 Pipeline

The pipeline is called upon when an item actually comes in from the KNMI stream. The pipeline consists of the following steps:

(1) Download and save the nc file by sending the required requests.

(2) Load the nc file
(3) Get the time value (there is only 1 per nc file)
(4) Create a list of rows to insert into the database from the nc file
(5) Create each row by iterating over the column names and for each station:
  (a) Deal with the fact that the vectors are of shape (69,1) instead of just (69,)
  (b) Deal with the masked values and add missing values as None to the row to insert
  (c) Do not touch string as these vectors are actually (69,) so just put them in the row to insert
(6) Insert the list of rows into a relational database

It might look a bit simple now, but I spend a lot of time thinking about this, exploring the data and considering different options to finally get to this simplicity. For instance, I also considered vector databases or making a larger h5 file, but I finally settled on a relational model.

The implementation of the pipeline was done with two python functions. The implementation of these two functions can be found in the appendix and in the code repository.

3.1.1 Pipeline Performance. A single nc file is about 419145 bytes on average. It takes my pipeline on average 1.43 seconds to process and nc file and then on average 2.54 seconds to save the rows into the database. These are just simple benchmarks using python time.time and this is just running on my laptop.

3.1.2 Tools used in the pipeline. I wrote the two wrangle functions in python 3.10 as python is a language that is easy to write with many dependencies. I used poetry [4] to manage the python environment and dependencies. For the database I decided to use duckdb [5]. I wanted to try out this technology after hearing about it. I also used the netCDF4 library to actually load the nc files. I use paho-mqtt to listen to the data stream from KNMI.

4 SERVING

Using the pipeline developed in the previous report the data is now stored in the duckdb database. I think the best way for the data scientist to interact with the data is a data loader. This data loader will deliver tensors with a shape of (n, s, k) where:

- n is based on how many readings there are in the specified date range
- s is the number of valid stations asked for
- k is the number of different keys specified

However, where does the data loader get the data from? To actually get the data, the data loader will download the data from an api that I built around the database. I think a simple api is the best fit here as it allows to abstract the duckdb database and KNMI pipeline into a simple interface for the data loader.

4.1 The API

The api itself (created using bottle [1]) is simple. You can query the '/parquet' endpoint with a get request to export parquet files from the sql database. The get request takes the following four parameters.

- after - an ISO 8601 date string that specifies the beginning of the time range for which you want to retrieve data
- until - an ISO 8601 date string that specifies the end of the time range for which you want to retrieve data
- stations - A string of comma separated station names that specifies from which stations you want to download data
- variables - A string of comma separated variable names that specifies which variables you want to download.

The code that implements the /parquet endpoint can be found in the appendix. I expanded the '/stations' and '/variables' endpoints. These endpoints allow you to GET the available variable and station names programmatically (json lists). Finally, I added the '/ping' endpoint to check availability of the api. The code for these endpoints can be found in the git repository.

4.1.1 Index page of the api. How would people know how to use the api? To solve that issue I created a static html page at the root of the api '"/"'. You can find a picture of this page in the final Appendix. On this page I explained the api, and I provided an example in Python of how to download data from the api. More ways to download the dataset are better then just one so I also added a download form to this page. This form is another way that people can download the parquet files. I think having a form greatly increases the accessibility of the data.

4.1.2 Deploying the API. Since the API is written with bottle, it is WSGI compatible. This led me to think about the possibility of deploying the API as an azure cloud function. However, this would not be the best fit as the cloud function does not have state and thus could not easily access the duckdb database. For this reason the API is best to be hosted on a machine under a desk or something like EC2. Due to the single database file nature of duckdb the API and the pipeline should be hosted on the same machine. Not hosting the pipeline and the API on the same machine would add needless complexity.

4.1.3 Availability Concerns. It is simple and easy to download a parquet with data from the api. Maybe even too easy because someone could easily spam the api. If this is seen as an availability issue then rate limiting measures could be taken. For instance IP based rate limiting could be added. This rate limiting could afford to be quite stringent because I do not expect a benign user of my use case to need to make more than 3-6 downloads a day. I could also foresee an inference time use case where there would be more downloads but in that case the api should just be avoided all together and instead just run the pipeline locally.

Another measure that could be taken to relieve availability concerns is adding Basic http auth to the api. This would efficiently restrict the use of the api to only the authorised data scientist.

4.2 The Data loaders

With the api now up and running, I could make the data loader. The interface of the data loader looks like this:

```
dataset = RemoteKNMIdataset(train=True, download=True,
                    after="1950-01-01", until="
    2025-12-01",
                    download_url = "http://0.0.0.0:9999
    ",
```

```
                        stations={"06343"}, variables={"rh"
    , "td"})
```

Every input besides the train boolean has sensible default which means that the data scientist could also just type:

```
dataset = RemoteKNMIdataset(train=True)
```

This dataset when given to a data loader would deliver the (n, s, k) tensors. After making the torch [7] dataset I decided to also add another way of loading the data where we load the data into a pandas [6] data frame instead of a torch dataset.

4.2.1 Local data loader. With the previous data loading methods we assumed that the pipeline and api were running on a different machine than the machine of the data scientist. But duckdb is an database that stores all the data in a single file. This approach has the benefit that the complexity of running the pipeline locally is quite manageable. Given that the dataset grows every 10 minutes it could be considered much more efficient to just run the pipeline locally on the machine of the data scientist. Running the pipeline locally would also be a good idea for a model at inference time. A local pipeline on the same machine could just feed the latest data to the model during inference time to generate weather predictions. I show the difference between the local setup in Figure 1 and Figure 2 in the appendix.

To make running the pipeline locally simpler I used poetry a tool that simplifies dependency management and project setup.

As long as the data scientist can get auth tokens from KNMI and puts them into ‘auth.toml‘ they can set up the pipeline in just two commands:

```
poetry install
poetry run listen-nc
```

In a way adding this easy setup with poetry can be considered as serving the pipeline itself to the data scientist.

When running the pipeline locally it is unnecessary and inefficient to involve the api because the database file is just right there. To support this case properly, I created a local version of both the torch dataset class and the dataframe. These local versions load the data straight from the duckdb database file instead of the api. The interface of these versions is the same except that the parameters for downloading data have been exchanged for the single database path parameter.

5 LIFECYCLE

I am dealing with weather data. Weather data is special in that it is updated at every single moment in time but the past data never needs to change. Weather data also does not satisfy the definition of personal data so the GDPR just does not apply. That means I do not have to deal with data deletion requests or data becoming outdated over time. The KNMI dataset regularly gets extended with new data that really never gets stale. It does not get stale as it wil always be useful to train weather models.

This lifetime heavily influenced the design of my pipeline and the data loaders. The pipeline is designed to deal with a growing database by transforming the nc files into insert sql queries. Hereby I offload the task of storing the data and merging it with the old pile data to duckdb.

5.1 Increased number of messages from KNMI

Even if the messages from KNMI would come in every second I think that my pipeline can easily handle it. However, if the messages would come in more often then that I would think about using parallel computing to speed up the pipeline. I would add a queue to the pipeline. That would mean that when a KNMI message comes in it is first put in the queue before it is fully processed by a pool of workers. This should be relatively easy to implement as in the end the pipeline is contained in just two functions and this queue system is built into Python. To my knowledge this setup would scale very well. However, I do recognise that at some scale it would be best to leave behind Python and port to a compiled language.

5.2 The long run - Large dataset

But what would happen if we run this data collection for a couple of years? How would I deal with a KNMI dataset that is a couple gigabytes or terabytes in size. duckdb can handle it fine. I found a github issue from 2021 [2] on the duckdb GitHub that claimed that the max file size of a duckdb database is well within the millions of peta bytes. duckdb itself is not going to be a limiting factor when the database gets large. This means that the local data loading methods should not really struggle.

The downloading itself should also not struggle with large files. This is due to the fact of how I implemented it. The Python code never tries to load the entire dataset at a time into memory, but it will only load and send 32kb at the time. So in theory not much would change when the dataset grows very large.

However, in practice the situation would change. As everyone that has downloaded some large files knows, the larger your file the larger the chance of an error. Having to restart a file download because of an error is a terrible developer experience. So if the dataset becomes really large I would end up changing the api to partition the data. The idea would be that the api would send multiple download links with each downloading a part of the requested data. If the client already downloaded some of the files it could just continue at the latest link where there was an error instead of starting over completely.

Since the data is a time series I would partition the data over different spans of time values. The dataset grows at a very predictable rate (every 10 minutes with similar size increments) and that means that you can calculate how long it would take for the database to grow by 2gb. Then I would round that amount of time to the nearest year boundary and partition the data there. If a data scientist then requests data from multiple partitions the api would send two download links. Each of these download links would download half of the data. Either I would actually send half of the parquet bytes and the client would just concatenate the bytes or the server could send a multipart zip file.

Ideally the data scientist stays within the year boundaries because then they only have to download one partition.

6 CONCLUSION

In this project I created an easier way to load data provided by the KNMI Notification api. I achieved this by creating a pipeline, an api and torch data loaders. The full code can be found under this link: https://github.com/tintin10q/KNMI.nl-Dataloader

REFERENCES

[1] Bottle Developers. 2024. Bottle Tutorial. https://bottlepy.org/docs/dev/tutorial.html. Accessed: 2024-06-21.

[2] DuckDB Developers. 2024. Issue #1394: Persistent storage does not persist. https://github.com/duckdb/duckdb/issues/1394. Accessed: 2024-06-21.

[3] KNMI Developers. 2024. Notification Service. https://developer.dataplatform.knmi.nl/notification-service. Accessed: 2024-06-21.

[4] Poetry Developers. 2024. Poetry Documentation. https://python-poetry.org/. Accessed: 2024-06-21.

[5] Hannes Mühleisen and Mark Raasveldt. 2024. *duckdb: DBI Package for the DuckDB Database Management System.* https://r.duckdb.org/ R package version 1.0.0.9000, https://github.com/duckdb/duckdb-r.

[6] The pandas development team. 2020. *pandas-dev/pandas: Pandas.* https://doi.org/10.5281/zenodo.3509134

[7] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. In *NIPS-W*.

A APPENDIX

B QUALITY ASSESSMENT

Code to figure out if all nc files had the same datasets inside of it.

```
from collections import defaultdict
import glob, netCDF4

counts = defaultdict(int)
file_count = 0
for db in iter_nc_files():
    for key in db.variables:
        counts[key] += 1
    file_count += 1
print(f"Iterated over {file_count} files. ")
print(f"There are {len(counts)} different keys with all
     the files")
print(f"Each key appearance counts:\n", counts)
```

An example of how detailed some of the comments in an nc file were. This is the zm field:

```
zm Note:
The sensor is not installed at equal heights at all types of
measurement sites:
At 'AWS' sites the device is installed at 1.80m.
At 'AWS/Aerodrome' and 'Mistpost' (note that this includes
site Voorschoten (06215) which is 'AWS/Mistpost') the device
is installed at 2.50m elevation.
Exceptions are Berkhout AWS (06249), De Bilt AWS (06260)
and Twenthe AWS (06290) where the sensor is installed
at 2.50m.
```

C RESHAPING WRANGLING

The schema for the Measurement sql table.

```
CREATE TABLE Measurement (
    station      TEXT NOT NULL,
    time         BIGINT NOT NULL CHECK ( time > 0 ),
    wsi          TEXT,
    stationname  TEXT,
    lat         DOUBLE,
    lon         DOUBLE,
    height      DOUBLE,
    D1H         DOUBLE,
    dd          DOUBLE,
    dn          DOUBLE,
    dr          DOUBLE,
    dsd         DOUBLE,
    dx          DOUBLE,
    ff          DOUBLE,
    ffs         DOUBLE,
    fsd         DOUBLE,
    fx          DOUBLE,
    fxs         DOUBLE,
    gff         DOUBLE,
    gffs        DOUBLE,
    h           DOUBLE,
    h1          DOUBLE,
    h2          DOUBLE,
    h3          DOUBLE,
    hc          DOUBLE,
    hc1         DOUBLE,
    hc2         DOUBLE,
    hc3         DOUBLE,
    n           DOUBLE,
    n1          DOUBLE,
    n2          DOUBLE,
    n3          DOUBLE,
    nc          DOUBLE,
    nc1         DOUBLE,
    nc2         DOUBLE,
    nc3         DOUBLE,
    p0          DOUBLE,
    pp          DOUBLE,
    pg          DOUBLE,
    pr          DOUBLE,
    ps          DOUBLE,
    pwc         DOUBLE,
    Q1H         DOUBLE,
    Q24H        DOUBLE,
    qg          DOUBLE,
    qgn         DOUBLE,
    qgx         DOUBLE,
    qnh         DOUBLE,
    R12H        DOUBLE,
    R1H         DOUBLE,
    R24H        DOUBLE,
    R6H         DOUBLE,
    rg          DOUBLE,
    rh          DOUBLE,
    rh10        DOUBLE,
    Sav1H       DOUBLE,
    Sax1H       DOUBLE,
    Sax3H       DOUBLE,
    Sax6H       DOUBLE,
    sq          DOUBLE,
    ss          DOUBLE,
    Sx1H        DOUBLE,
    Sx3H        DOUBLE,
    Sx6H        DOUBLE,
    t10         DOUBLE,
    ta          DOUBLE,
    tb          DOUBLE,
    tb1         DOUBLE,
    Tb1n6       DOUBLE,
    Tb1x6       DOUBLE,
    tb2         DOUBLE,
    Tb2n6       DOUBLE,
    Tb2x6       DOUBLE,
    tb3         DOUBLE,
    tb4         DOUBLE,
    tb5         DOUBLE,
    td          DOUBLE,
    td10        DOUBLE,
    tg          DOUBLE,
    tgn         DOUBLE,
    Tgn12       DOUBLE,
    Tgn14       DOUBLE,
    Tgn6        DOUBLE,
```

```
    tn          DOUBLE,
    Tn12        DOUBLE,
    Tn14        DOUBLE,
    Tn6         DOUBLE,
    tsd         DOUBLE,
    tx          DOUBLE,
    Tx12        DOUBLE,
    Tx24        DOUBLE,
    Tx6         DOUBLE,
    vv          DOUBLE,
    W10         DOUBLE,
    "W10-10"    DOUBLE,
    ww          DOUBLE,
    "ww-10"     DOUBLE,
    zm          DOUBLE,
    PRIMARY KEY (time, station)
    );
```

This function executes the first 5 steps of the pipeline

```
from pathlib import Path
import netCDF4 as nc
import numpy as np
from typing import List

nc_keys_to_save = ( "station", "time", "wsi", "stationname", "lat", "lon", "height", "D1H", "dd", "dn", "dr", "dsd", "dx", "ff", "ffs", "fsd", "fx", "fxs", "gff", "gffs", "h", "h1", "h2", "h3", "hc", "hc1", "hc2", "hc3", "n", "n1", "n2", "n3", "nc", "nc1", "nc2", "nc3", "p0", "pp", "pg", "pr", "ps", "pwc", "Q1H", "Q24H", "qg", "qgn", "qgx", "qnh", "R12H", "R1H", "R24H", "R6H", "rg", "rh", "rh10", "Sav1H", "Sax1H", "Sax3H", "Sax6H", "sq", "ss", "Sx1H", "Sx3H", "Sx6H", "t10", "ta", "tb", "tb1", "Tb1n6", "Tb1x6", "tb2", "Tb2n6", "Tb2x6", "tb3", "tb4", "tb5", "td", "td10", "tg", "tgn", "Tgn12", "Tgn14", "Tgn6", "tn", "Tn12", "Tn14", "Tn6", "tsd", "tx", "Tx12", "Tx24", "Tx6", "vv", "W10", "W10-10", "ww", "ww-10", "zm")

Measurements = List[List[str | int | float]]

def process_nc_file(filename: str | Path) -> Measurements:
    """ Process a single NetCDF file and return a list of lists with values in the order of nc_keys_to_save. """

    db = nc.Dataset(filename) # Step 2
    try:
        time = db.variables['time'][:].item()  # step 3

        to_insert = []  # List of data to insert into the database in the end.
        # For each station iterate each key in the nc file
        for index, station in enumerate(db.variables['station']):
            to_insert.append([])  # Step 4
            for key in nc_keys_to_save: # Step 5

                if key == "time":
                    to_insert[index].append(time)
                    continue

                station_reading = db.variables[key][index]

                is_string = isinstance(station_reading, str)
                if is_string:
                    to_insert[index].append(station_reading)
                    continue

                # Else check for a masked array
                is_masked = False
                if isinstance(station_reading, np.ma.core.MaskedArray) and station_reading.mask:
                    is_masked = station_reading.mask[0]

                if is_masked:
                    to_insert[index].append(None)
                    continue

                value = station_reading.item()  # Otherwise just get the item
                to_insert[index].append(value)

    finally:
        db.close()

    return to_insert
```

This function executes the final step of the pipeline.

```
def insert_nc_filerows(result: Measurements): # Step 6
    con = get_connection()
    con.executemany(insert_statement, result)
```

D SERVING

Code for the /parquet endpoint. The other endpoints can be found in the Github repository.

```
@route("/parquet")
def download():
    match request.query:
        case {"after": str(after), "until": str(until), "stations": str(stations_string), "variables": str(variables_string)}:
            stations = set(station_list := stations_string.strip().replace(" ", "").replace("\n", "").replace("\r", "").split(","))

            if invalid := stations - ALL_KNMI_STATIONS:
                return HTTPResponse(f'"Could not download data because of {len(invalid)} invalid station name{"s" if len(invalid) != 1 else ""} specified"', status=400)

            variables: Set[str] = set(variables_string.strip().replace(" ", "").replace("\n", "").replace("\r", "").split(","))
            if invalid := variables - ALL_KNMI_VARIABLES:  # If anything remains it's not valid and reject!
                return HTTPResponse(f'"Could not download data because of {len(invalid)} invalid variable name{"s" if len(invalid) != 1 else ""} specified"', status=400)

            try:  # Until
                after_parsable = after.replace("Z", "+00:00")
                parsed_after = datetime.datetime.fromisoformat(after_parsable)
                if parsed_after < knmi_epoch:
```

```
                    return HTTPResponse(f'"after has to be more then {knmi_epoch}"', status=400)
            except ValueError:
                return HTTPResponse(f'"after is not a valid iso 8601 date"', status=400)

            after_total_seconds = (parsed_after - knmi_epoch).total_seconds()
            after_seconds = int(after_total_seconds)

            try:  # After
                until_parsable = until.replace("Z", "+00:00")
                parsed_until = datetime.datetime.fromisoformat(until_parsable)
                if parsed_until < knmi_epoch:
                    return HTTPResponse(f'"until has to be more then {knmi_epoch}', status=400)
            except ValueError:
                return HTTPResponse(f'"until is not a valid iso 8601 date"', status=400)


            if parsed_until < parsed_after:
                return HTTPResponse(f'"after has to before until"', status=400)

            until_total_seconds = (parsed_until - knmi_epoch).total_seconds()
            until_seconds = int(until_total_seconds)

            # Fetch the data from the database

            station_questionmarks = ', '.join(['?'] * len(stations))
            requested_variables = ",".join(variables)  # should be safe because we checked the set with an allow list

            outputfile = tempfile.NamedTemporaryFile("r+b")
            outputfilename = outputfile.name

            query = f"select {requested_variables} from Measurement where station in ({station_questionmarks}) and time > ? and time < ? order by time"
            connection = db_connection.get_readonly_connection()

            try:
                connection.sql(query, params=station_list + [after_seconds, until_seconds]).to_parquet(file_name=outputfilename)
            finally:
                connection.close()

            response.set_header('Content-type', 'application/vnd.apache.parquet')
            response.set_header('Content-Disposition', f'attachment; filename="KNMI_after_{parsed_after}_until_{parsed_until}_with_{len(variables)}_variables_{len(stations)}_stations.parquet"');

            def output():
                try:
                    with open(outputfilename, "rb") as f:
                        while chunk := f.read(32768):
                            yield chunk
                finally:
                    outputfile.close()

            return output()
        case _:
            return HTTPResponse('''"Missing query params. You need:
- after :: iso date string;
- until :: iso date string;
- stations :: comma seperated station names ; see /stations for valid station names
- variables :: comma seperated variable names ; see /variables for valid variable names "''', status=400)
```

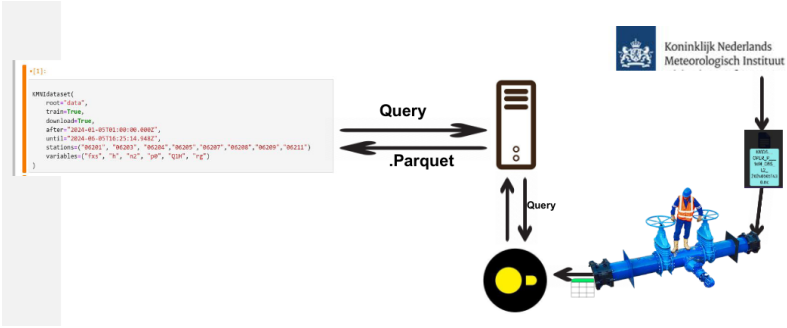

Figure 1: Remote data setup

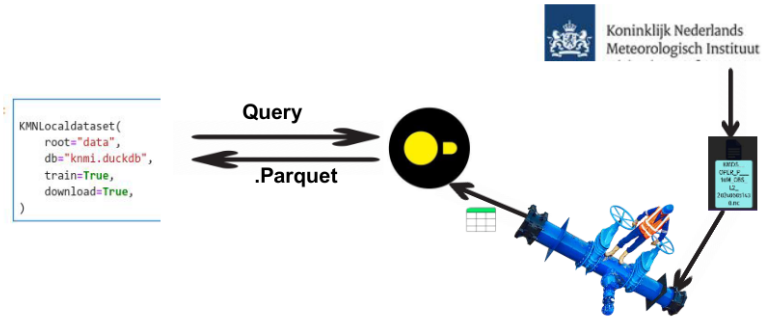

Figure 2: Local data setup

The next image shows the root page of the api.

* 2*

KNMI.nl Data Server

This is a server from which you can download data from knmi.nl the Dutch metrological institute.
New data is collected from their Notification service every 10 minutes or so.

How do I get the data?

You can either download the data using the api or using the form.
There is also a pytorch data loader available.

Download Form

After (ISO 8601 date string): 20 - 06 - 2023

Until (ISO 8601 date string): 20 - 06 - 2024

Stations (comma separated):

```
06205, 06229, 06240, 06238, 06320, 06375, 06275,
06316, 06286, 06257, 78871, 06235, 06312, 06209,
06319, 78873, 06215, 06269, 78990, 06283, 06310,
06267, 06214, 06225, 06321, 06237, 06273, 06323,
```

Variables (comma separated):

```
tb, R12H, tn, tx, qnh, fx, p0, dr, Tn6, Q24H,
Tb2x6, lon, Sax1H, tb3, Q1H, ss, qgx, qgn, dn, rh,
pr, vv, tsd, hc3, Tb1x6, tg, hc, h3, Tb2n6, dd,
h1, station, lat, dx, Sav1H, Sx1H, rg, time, fsd,
```

Download KNMI data

Python API Example

```python
#!/usr/bin/env python
# Download all the data of the last week
import requests, datetime

api_host = "http://0.0.0.0:9999"
save_as = "weather.parquet"

now = datetime.datetime.utcnow()
until = now.isoformat()

one_year_ago = now - datetime.timedelta(days=356)
after = one_year_ago.isoformat()

stations = requests.get(f"{api_host}/stations").json()
variables = requests.get(f"{api_host}/variables").json()

stations = ",".join(stations)
variables = ",".join(variables)

parquet = requests.get(f"{api_host}/parquet", stream=True,
                       params={"until": until, "after": "stations": stations, "variables": variables})

with open(save_as, "wb") as f:  # Save the file
    for chunk in parquet.iter_content(chunk_size=1024 * 32):
        f.write(chunk)
```

Available Variables

Name	Long name	Size	Shape	Units	Type
station	Station id	69	(69,)	None	string
time	time of measurement	1	(1,)	seconds since 1950-01-01 00:00:00	float64
wsi	Station wsi	69	(69,)	None	string