Life Time

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 That means I do not have to deal with data deletion requests or data becoming outdated over time.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.

This lifetime heavily influenced the design of my pipeline and the dataloaders. The pipeline can already deal with a rapidly growing database by transforming the nc files into insert sql queries. Herby I offload the task of storing the data and merging it with the old pile data to Duckdb. 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 recognize that at some scale it would be best to leave behind Python and port to a compiled language.

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 this issue from 2021 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.

It would be best if the data scientists will stay within this year's boundaries because then they can download the part of the dataset they want in one go.