Air Transport Operations Modelling Project

AE4439

Théo van den Berge

Delft University of Technology

Air Transport Operations Modelling Project

by

Théo van den Berge

Student number

Théo van den Berge	4805291

TU Delft

Contents

1 Introduction

During the Sustainable Air Transport master program a lot of attention is paid to data and its importance, but not so much on the analysis and visualization of said data. The Air Transport Modeling project (AE4439) is one of the electives within this program that gives students the opportunity to further explore data analysis and visualization. In this report the work related to this elective will be presented, where an interactive data dashboard was developed to visualize a dataset of full-freighter cargo flights. More specifically, this dataset contains 82 MD-11F flights that depart from Flughafen Frankfurt am Main (FRA) to destinations all across the globe. In some cases the aircraft operates a direct flight, while in other (and most) cases the aircraft operates multi-leg flights before returning to Frankfurt. As such, it must be noted that only the outbound flights from FRA have been included in this dataset.

Moreover, this dataset with synthetically generated flights has been made available by Felix Brandt as part of his PhD work, under the Creative Commons CC BY 4.0 license. For more information the reader is kindly redirected to *Felix Brandt's GitHub page*. The data application can be accessed through the internet using the following URL: `https://theovdberge.pythonanywhere.com/`, keeping in mind that it may take some time for the web page to load. Additionally, the code for the data application is published on *Theo van den Berge's GitHub page*.

In the data application the user is presented with a variety of information regarding these cargo flights. First of all, a map is shown where all the flight legs can be seen, including the stop-over legs. When the user hovers over an airport the name of the airport is displayed. Additionally, a variety of filtering options are present, such as the destination, the flight number, departure time (at Frankfurt) and the load factor. It must be noted that the filter option 'Dangerous goods' has **not** has not been implemented as of yet.

The first two graphs that can be seen are two histograms displaying the number of daily departures at Frankfurt, where the first graph displays the departures per day, and the second graph displays the hourly departures. The second set of graphs relate to the transported volume per destination and the load factor per flight number per day of the week. It is important to note that the load factor here is based on the payload-range diagram of the MD11, meaning that more accurate load factors are presented. Furthermore, the load factors are calculated per leg. The final element of the data application is a table where more detailed, numerical, data can be seen and interpreted by the user, both in terms of flights and individual shipments.

Furthermore, in the first chapter, the reader will receive a brief introduction about the dataset that has been used, before the Python code with its packages and frameworks will be discussed. Thereafter, a concise data analysis will be presented, followed by a conclusion.

Additionally, the second aim of this report is to provide a means of documentation for this project for future reference, in case one wishes to adapt/change the dashboard still.

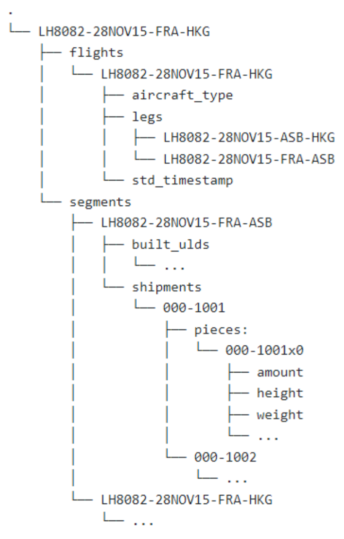
2 Data used

The data that is used for this dashboard has been made available by Felix Brandt as part of his PhD work, under the Creative Commons CC BY 4.0 license. This dataset consists of 82 flights that are operated by Lufthansa Cargo AG in calendar week 48 of the year 2015. Furthermore, all of these flights are being operated by a single aircraft type, namely the MD-11F, which is a Full Freighter aircraft. This means that the aircraft does not carry passengers, but only cargo. As such, the aircraft is built specifically to accept ULD containers. Additionally, this dataset only encompasses flights that are departing from Flughafen Frankfurt am Main (FRA), which may have intermediate stop-overs.

2.1. Data structure

All data files are provided in the YAML-format, which are comparable to Python dictionaries in both the appearance as well as their way of interaction in Python. These files have the following naming convention of
`flightnumer-date-FRA-destination.schedule.yaml`, where it is important to note that only the final destination is indicated in the file name.

The core of these YAML-files relies on individual shipments that make up ULDs, where for each shipment various parameters have been recorded, such as the amount, dimensions and weight. It is this part of the YAML-files that allow the dashboard to be created, as the dashboard has been built on a shipment-basis.
To illustrate the example of a typical file structure, a route from Frankfurt (FRA) to Hong Kong (HKG) is used, which can be seen in Figure 2.1. As can be seen, this route consists of two legs as a stop is made in Aşgabat, Turmenistan (ASB). As such, there are two legs stored under '`segments`'. Per segment one can find information about the individual shipments and it is this information that is used to feed the dashboard. The information under '`flights`' can be used to retrieve more information about the legs and the departure time.



```
.
└── LH8082-28NOV15-FRA-HKG
    ├── flights
    │   └── LH8082-28NOV15-FRA-HKG
    │       ├── aircraft_type
    │       ├── legs
    │       │   ├── LH8082-28NOV15-ASB-HKG
    │       │   └── LH8082-28NOV15-FRA-ASB
    │       └── std_timestamp
    └── segments
        ├── LH8082-28NOV15-FRA-ASB
        │   ├── built_ulds
        │   │   └── ...
        │   └── shipments
        │       └── 000-1001
        │           ├── pieces:
        │           │   └── 000-1001x0
        │           │       ├── amount
        │           │       ├── height
        │           │       ├── weight
        │           │       └── ...
        │           └── 000-1002
        │               └── ...
        └── LH8082-28NOV15-FRA-HKG
            └── ...
```

Figure 2.1: An example file structure of the data files used.

3 Code

The program that has been used for the analysis and the visualization of the data has been fully written in Python. In order to ease the coding process for the dashboard, a framework for the dashboard was used. More specifically, the Python package `Dash` was used, which is a low-code framework which enables the user to rapidly build data apps. Combining this framework with Dash's Plotly package, it becomes relatively straightforward to build a data app with interactive charts and figures. This chapter will briefly discuss the Dash Framework and the Plotly implementation, as well as provide an overview of the packages that have been used and their version number.

3.1. Framework - Dash

As mentioned in the introduction of this chapter, Dash was used as a framework for the dashboard. Dash allows the user to fully program a website, or web-based application, using Python, without the need to use HTML or CSS. The framework generates the HTML and CSS code as required based on the `main.py` file. One of the main advantages of this framework is that the user does not need to write the HTML and CSS manually, nor does the user need to worry about dynamic lay-out behaviour (for example, different screen sizes on various devices or resizing windows), as this is all taken care of by the framework. Furthermore, all the interactions with the application can be handled in Python.
There are multiple ways to Rome with regards on how to build the data application, but in this particular case it was chosen to use the 'columns and rows'-approach. This means that the application is based on (figurative) rows and columns, where all elements are on a certain row and have a certain column width. To illustrate this approach, an example has been provided in Figure 3.1.

In Figure 3.1 it can be seen that, for example, the title of the application is positioned on the first row of the page, whereas the map and filter options are on the second row (red). While the title row consists of a single column, the second row consists of two columns (orange), namely one for the map and on for the (grouped) filter options. Additionally, nested columns and rows are possible, as has been done for the filter options. Even though the entirety of the filter options are in a single column, the 'Destination' and 'Flight number' drop-down boxes are on a nested row (green), and each in a separate column. This logic can also be applied to the next row, containing the 'Dangerous goods' drop-down box, the 'Data options' radio buttons and the 'Reset' button. However, in this case, the row has been divided in three columns.
Of course, the same logic can be applied to the rest of the application, but only this first section of the application was discussed for illustrative purposes.

This approach results in a very intuitive method that allows the user to quickly position elements on the page, while ensuring correct positioning and grouping for the dynamic lay-out.

In the actual code a column is represented by the `dbc.Col` command, where the content is specified and the width can be indicated as an integer between 1-12 (as a row is 12 units wide). A row can be called using the `dbc.Row` command, where the content can be specified. Furthermore, it should be

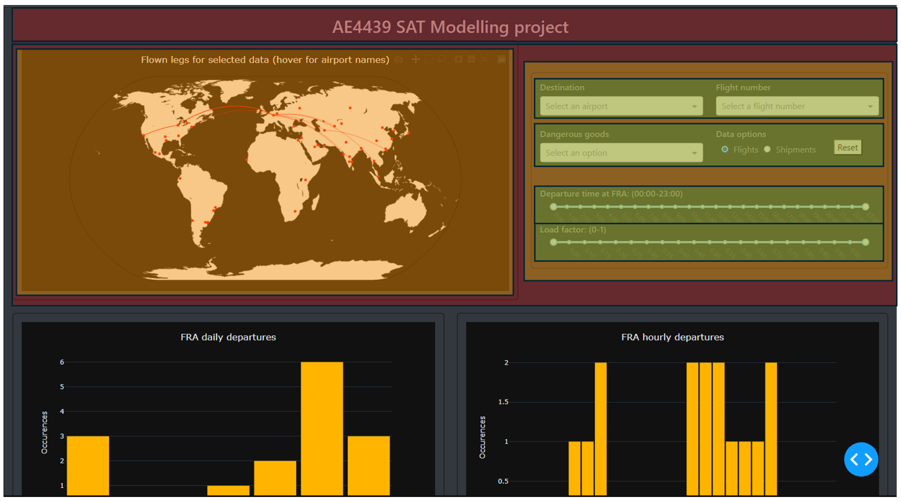


Figure 3.1: An example of the 'columns and rows'-approach using the Dash framework.

noted that the borders of a 'cell' are created using the `dbc.Card` and `dbc.CardBody` commands. Within these cards, the above-mentioned commands for a row and column can be used.

3.1.1. Call-backs

Another important feature of the Dash framework, is the call-back feature. This allows the user to interact with the application on the front-end, and the changes to be processed on the back-end before being displayed again on the front-end. In other words, when the users interacts with one or more filter options, the call-backs ensure that the right filter options are applied and the new graphs are displayed.

Generally, a single callback per figure or graph is used, where the relevant inputs and outputs can be defined. Within Dash a callback can be called using `@app.callback`, after which the I/Os are defined, as well as a function that handles the updating of the graph. This function can for example handle some filtering, before calling the figure generation function, as the output of a callback has to be the figure itself and not raw data for example. An example of a callback can be seen below in Figure 3.2.

```python
# Update button callback
@app.callback(
    Output('flightnr_filter', 'value'),
    Output('dest_filter', 'value'),
    Output('slider-dep', 'value'),
    Output('slider-lf', 'value'),
    Output('dataSelection', 'value'),
    Input('ResetButton', 'n_clicks'),
    prevent_initial_call = True
    )

def update_filters(value1):
    flightfilter, destinations = None, None
    slider1, slider2 = [0,23], [0,1]
    radioItem = 'Shipments'
    return(flightfilter, destinations, slider1, slider2, radioItem)
```

Figure 3.2: An example of a callback function in Dash that resets all the filter options.

In Figure 3.2 it can be seen that a callback function is defined for resetting all the filter options to the original values. First, the outputs are listed - where the IDs of the respective elements (as defined in the code) are being called - after which it is specified that the output should be used for the 'value' of the filter. Thereafter, the input is defined. In this case, it is a single button and its property - the number of times it has been clicked. Of course, in this particular example the number of clicks is not relevant, but rather *if* the button has been clicked.

Then, a `update_filters(value1)` function is defined. The `value1` is simply the variable storing the value received from the input. In an example where one would have more than one input, the input parameters of the function must match the number of listed inputs. Lastly, the function returns the new values for the filters, keeping in mind that every output must be defined in the `return(...)` section of the function. The order in which they are listed, is the order in which the outputs are defined under the `@app.callback` section of the callback.

3.1.2. Web hosting

Dash offers the possibility to host the application locally (through `http://127.0.0.1:8050/`), but this is not very useful in case the user wants the application to be accessible to other users. Therefore, a web hosting service has to be used. For this particular project, it has been chosen to use the hosting service of PythonAnywhere. PythonAnywhere is a free service that enables users to host Python files on the internet and make the application accessible to others through a URL.

The service is relatively straight-forward as all the base files can be uploaded, as well as the main application file. After specifying the desired output, the website can be loaded.

3.2. Plotly

Plotly is a Python package that enables the user to visualize data in an interactive way through a variety of chart types. For example, histograms, pie charts, scatter plots or maps can be used. The power of this package lays in the fact that these graphs come with many built-in features, such as filtering or on-hover information. For example, in Figure 3.3 a chart is shown with the different load factors for various flight numbers throughout the week. When the user only wants to see the load factors on a Monday, the user can simply double-click on 'Monday' in the legend. Similarly, when the user wants to exclude the flights on Monday from the chart, 'Monday' can be clicked a single time.

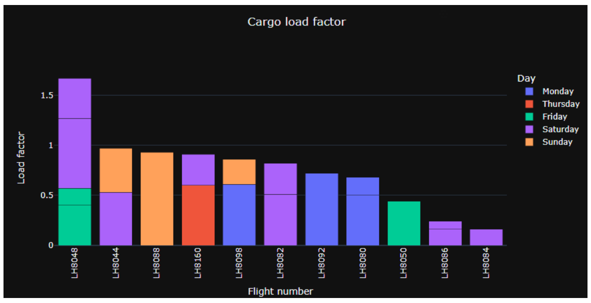


Figure 3.3: An example of a Plotly graph with a legend.

Plotly is a very powerful, and smart, package that relies primarily on a Pandas DataFrame as input. Furthermore, there are a plethora of examples and documentation available online, making it relatively easy to get started with this package.

3.3. Package versions

While working on this project using different machines, as well as different Python versions, package compatibility issues have been a problem. Therefore, a list of the Python and packages version that are required for this application are listed below in Table 3.1. However, this does not mean that the application solely runs on these versions, but these versions have been demonstrated as stable.

Table 3.1: An overview of the packages that have been used and their version numbers.

Package name	Version number
Python	3.9.13
datetime	3.9.13 (follows Python version)
os	3.9.13 (follows Python version)
numpy	1.24.1
pandas	1.5.2
dash	2.17.1
yaml	6.0
plotly	5.9.0

4 Data analysis

In the previous chapters it has been described how the data application has been built, whereas this chapter will focus on using that application to analyze the given data.

Again, the data set consists of 82 full freighter flights that are operated by the MD11 from Flughafen Frankfurt am Main. Of these flights, only the outbound flights have been included in the data set, meaning that the return flights have not been included in the analysis.

First of all, where are these flights being operated to? Examining the world map given in Figure 4.1, it can be seen that the majority of the destinations are in Asia and the Americas, with only a few destinations on the African continent. It is therefore slightly surprising that the most frequented destination is in Russia, with Krasnoyarsk International Airport (KJA) with 8 flights, followed by Shanghai Pudong International Airport (PVG) with 7 flights. The FRA-PVG route is however the most operated route. Despite the fact that KJA receives more flights, the routes the aircraft take from FRA to KJA are different on almost every occasion. Note that in this example intermediate stops are not considered.

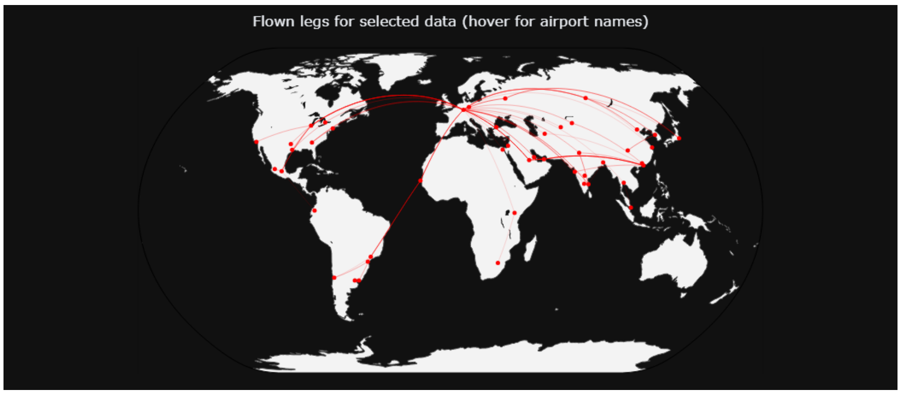


Figure 4.1: An overview of the destinations that are being served by the cargo flights.

However if intermediate stops were to be considered, the most frequented intermediate stop is by far Chicago O'Hare Airport (ORD) with 18 visits, which is also reflected by the amount of cargo getting off-loaded in ORD - over 460 tons. The amount of cargo that is off-loaded, can be seen in Figure 4.2, where it can also be seen that both KJA and PVG complete the top three.

In terms of departure times, not many trends can be found or conclusions can be drawn. The distribution of the number of departures per day of the week is relatively constant, with Saturday being slightly more popular than the other days. This can be seen in Figure 4.3a. Furthermore, with regards to departure times the same can be said, as no clear pattern can be discerned. However, the (late) afternoon is clearly more popular than the morning. The late evening and early morning sees little to no departures,

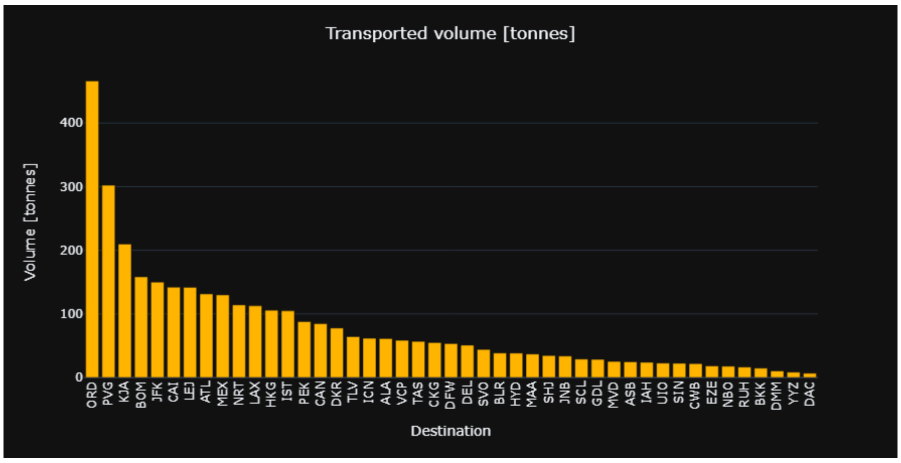
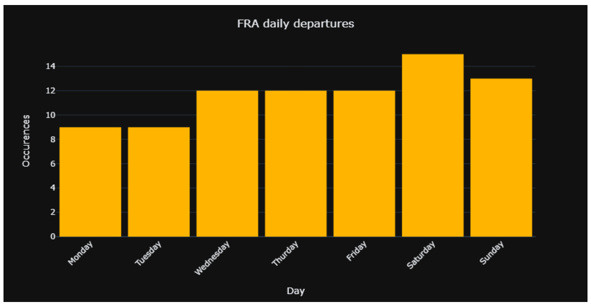
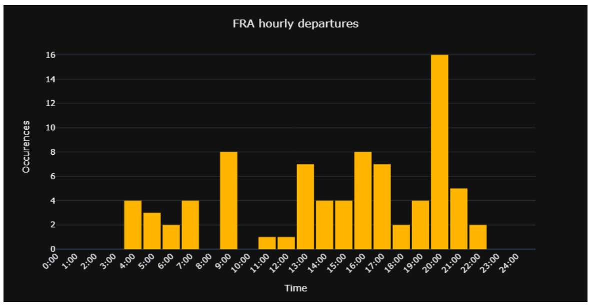


Figure 4.2: The transported volume in tonnes per airport.

which can also be caused by operational restrictions such as curfews at airports. The distribution by the hour can be seen in Figure 4.3. Additionally, most of the flight numbers are being operated at least twice a week, with a lot of routes also being operated up to three times a week. However, there is only a single route that sees daily departures throughout the entire week, namely the direct FRA-PVG route.



(a) The daily departures at Frankfurt Airport.



(b) The hourly departures at Frankfurt Airport.

Figure 4.3: The daily and hourly departures at Frankfurt Airport.

5 Conclusion

In this report the dataset, coding approach, and data analysis was presented for a web-based data application, which can be found here: `https://theovdberge.pythonanywhere.com/`. The data used in this application is provided as part of Felix Brandt's PhD work, namely a dataset consisting of 82 full freighter flights that depart from Frankfurt Airport.

The dataset provides files per individual flights in the YAML-format, which is used as the core of the application. By extracting the relevant data into a dictionary of flights, the data was transformed for easier handling. This also allowed for data filtering, as only the relevant information was retained.

The application itself was written in Python using Dash as framework and Dash's Plotly for the interactive graphs and figures. The application provides the user with a multitude of graphs, showcasing the operated routes, transported volume of cargo (in tonnes), as well as (leg) load factors and departure frequencies. Through the use of call-backs, the user is able to interact with the graphs and apply multiple filtering options, such as flight number and destination. Moreover, the application is hosted using the (free) services from PythonAnywhere, allowing the application to be accessed from anywhere by means of the internet.

Lastly, a concise data analysis was performed on the dataset using the application. This has shown that flights are operated to almost all continents, with a strong focus on Asia and the Americas. Here, Chicacgo O'Hare Airport (ORD) is one of the most frequented airports in terms of total stops (both as final destination and intermediate stop), whereas Krasnoyarsk International Airport (KJA) is the most frequented airport in terms of final destination.
In terms of departure times, not many trends can be found or conclusions can be drawn, as the distribution over the week is relatively consistent. In terms of departure hours, the (late) afternoon is more popular than the early morning and late evening, where little to no flights are being operated.

Overall this application offers a plethora of filtering options to visualize and analyse as the user wishes to do so. The dashboard can easily be extended and further customized with more filtering options or graphs. Therefore, the code has been made public on *GitHub*.