Cookbook

ETL-Process with
Datasets
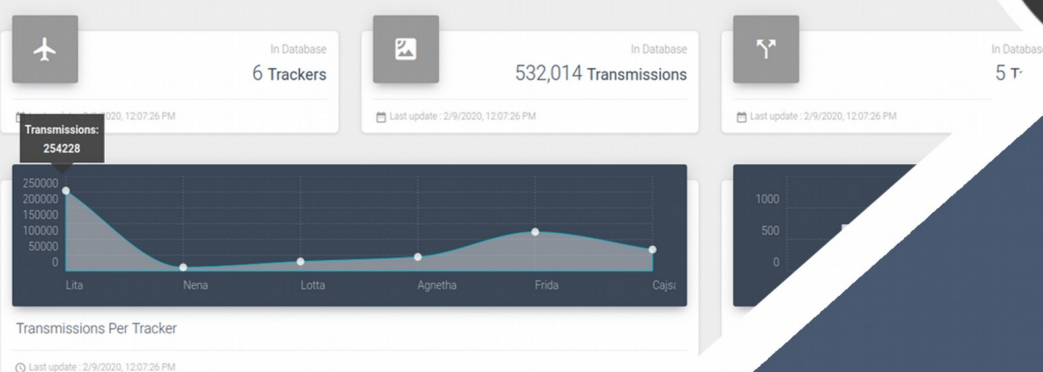
Purpose of this document

This cookbook serves as an extension for the following documents:

1) **Cookbook: Creating the GeoStack Course VM:**

   The datastores and tools which we will be using during this cookbook are set up during the cookbook: "Creating the Geostack Course VM"
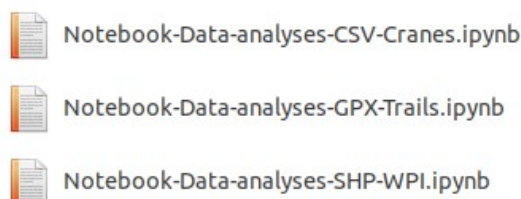
The purpose of this document is to provide information related to extracting, transforming and loading data. First the basics of the ETL-process are discussed. While discussing the basics of the process, examples are given of a variety of file formats and data stores that you can come across when performing the ETL-process yourself. Afterwards a detailed instruction is given on how to perform the process with a real life example of a Crane dataset obtained from the website (the datasource): https://www.movebank.org/.

The loading phase described in this cookbook shows a simple example of how to import data in MongoDB. The cookbook: "Data modeling in MongoDB" describes this phase (loading phase) in more detail.

The notebook which contains all the code created during this cookbook is called: "Notebook-Created-During-Cookbook" and can be found in the folder called: "Cookbook-Notebook" which is located in the same folder as this document. This folder also contains some test data which is created during this cookbook.

In the cookbook: "Creating the GeoStack Course VM" you created an empty notebook in which you should add the code, described in the cookbook, yourself. If you get stuck you can always use the notebook: "Notebook-Created-During-Cookbook" which contains all the code which has to be created.

You will also find a folder called: "Remaining-Analyses-Notebooks". The notebooks found in the folder: "Remaining-Analyses-Notebooks" contains the data analyses of the remaining Crane (Tracker) datasets, the GPS-Route (Trail) datasets and the World-Port-Index dataset. They also show you how to compare and analyze similar datasets with a different data structure (e.g. the Swedish and German Crane tracker datasets). In the illustration below you can find the content of the Remaining Notebooks folder.

Notebook-Data-analyses-CSV-Cranes.ipynb

Notebook-Data-analyses-GPX-Trails.ipynb

Notebook-Data-analyses-SHP-WPI.ipynb

The dataset used in the remaining Jupyter Notebooks can be found in the folder "Course-Datasets", which is located in the root folder of The GeoStack Course. Remember to go through these notebooks after you have read this document! We need to have these datasets analyzed and transformed to achieve our end goal of creating a fully functional Geographical software stack.

A special thanks to the following people and organizations:

➢ **The people and contributors from Movebank.org for providing the datasets used in this document and in the GeoStack course.**

➢ **The people from MongoDB for providing the great software that will be used throughout this document and other documents.**

Table of Contents

Table of figures

1. Introduction to the ETL-Process

Data is one of the most valuable aspects in a company. For data to be valuable there needs to be a way to gather it from data sources, organize the data and store it in the desired data store. For this to happen correctly we use a process called the ETL-Process.

In the image below you can find an visual representation of the ETL-Process.
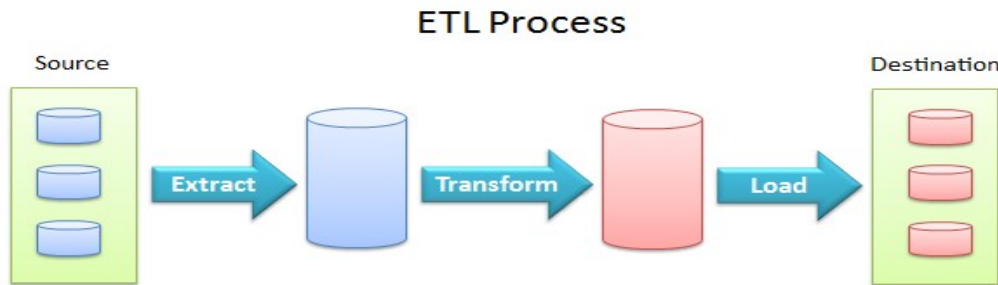
Figure 1: ETL-Process (Yousuf , Applied informatics, 2019)

ETL is an abbreviation for:
- **Extract:** Extract the data from different data sources. This is the first phase of the ETL-Process.
- **Transform:** Transform the data in the desired file format. This is the second phase of the ETL-Process.
- **Load:** Load the data in the desired data store. This is the third phase of the ETL-Process.

1.1 Why use ETL ?

The ETL-Process has, since it was introduced 50 years ago, allowed companies and organizations to have a consolidated view of their data. The process makes sure that companies are able to analyze data that resides in multiple locations and data that is stored in a variety of file formats. It also enables them to streamline the review process which results in better business decisions.

Some other reasons to use ETL are as follows:
- Transactional databases cannot answer complex business questions that can be answered by ETL.
- ETL provides a method of moving the data from various sources into a data warehouse.
- As data sources change, the Data Warehouse will automatically update.
- Well-designed and documented ETL system is almost essential to the success of a Data Warehouse project.
- Allow verification of data transformation, aggregation and calculations rules.
- ETL process allows sample data comparison between the source and the target system.
- ETL process can perform complex transformations and requires the extra area to store the data.
- ETL helps to Migrate data into a Data Warehouse. Convert to the various formats and types to adhere to one consistent system.
- ETL is a predefined process for accessing and manipulating source data into the target database.

1.2 ETL vs. ELT

Below some differences between the ETL-Process and the other well process called ELT are discussed:

➔ In ETL (extract, transform, load) operations, data are extracted from different sources, transformed separately, and loaded into a database and possibly other targets.

➔ In ELT, the extracts are fed into the single staging database that also handles the transformations.

➔ A cited advantage of ELT is the isolation of the load process from the transformation process, since it removes an inherent dependency between these stages.
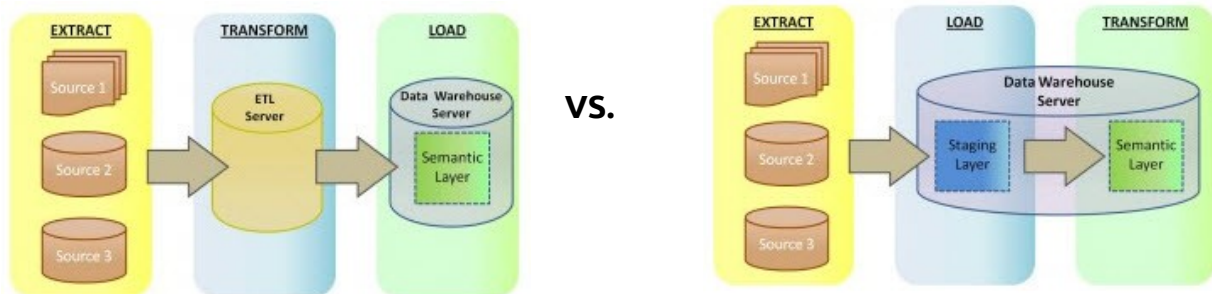
Figure 2: ETL VS. ELT (Frieland, Iri, 2019)

1.3 The phases of the ETL-Process

You can divide the ETL-Process in three phases. Each of the phases will be described below.

1.3.1 Extract

In the first phase we extract the data from an array of data sources. The data that will be extracted during this phase will be identified by someone in the organization. These data sources could include:

- A database from google that contains points of interests on a map.
- A database from the government that contains crime rates per regions in a country.
- An API from an electronics website that contains information about electronic devices.
- A cloud store containing information about research that has been done on the subject.

Figure 3: Different data sources (Talend, 2019)

When extracting data from data-sources it can come in a variety of data formats. A few common data formats include:

- Relational databases

A relation database is a set of formally described tables from which data can be accessed or reassembled in many different ways without having to reorganize the database tables. PostgreSQL and MySQL are examples of relational databases. In figure 4 you can see a visual representation of a relational database.
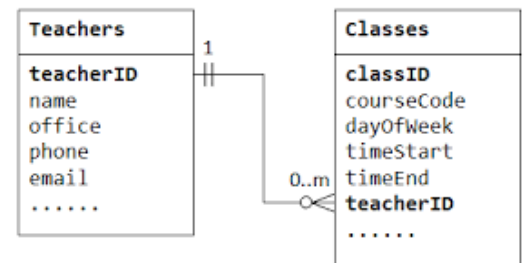
Figure 4: Relational database table

- Non-relational databases

Non-relational database, also referred to as NoSQL databases, are databases that don't follow the relational model provided by traditional databases. MongoDB and ElasticSearch are examples of non relational databases. In figure 5 you can find a visual representation of a non-relational database.

Figure 5: Non-relational document (FullStack academy, 2019)

- XML (Extensible Markup Language)

Extensible Markup Language (XML) is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable. The design goals of XML emphasize simplicity, generality and usability across the internet. A visual representation of a XML document can be found in figure 6.
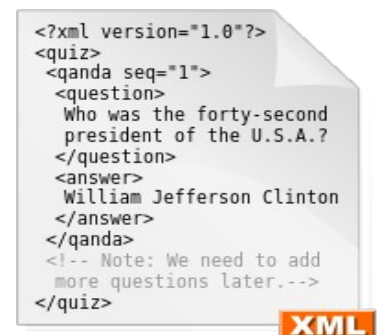
Figure 6: XML (Google, 2019)

- CSV (comma-separated values)

CSV is a simple file format used to store tabular data, such as a spreadsheet or database. Files in the CSV format can be imported to and exported from programs that store data in tables, such as Microsoft Excel or OpenOffice Calc. CSV stands for "comma-separated values". A visual representation of an CSV file can be seen in figure 7.

Figure 7: CSV file format (Google, 2019)

- JSON (JavaScript Object Notation)

JSON is an open-standard file format that uses human-readable text to transmit data objects consisting of attribute–value pairs and array data types (or any other serializable value). JSON is a language-independent data format. A visual representation of a JSON document cant be seen in figure 8.

Figure 8: JSON file format (Full Stack Academy)

Data extraction is mostly done in one of the three ways described below:

1) **Update notification**

   The system notifies you when a record has been changed. This is referred to as the easiest method of data extraction.

2) **Incremental extraction**

   When a system cannot provide notifications for updates to the data set, It most likely provides an extract of the records that have been added to the data-store.

3) **Full-extraction**

   If a system is unable to identify when data has been changed, the only option is to re-download all the data. This method of data extraction is only recommended for small amounts of data.

1.3.2 Transform

When the extraction of the data is completed we start with the next phase of the ETL-Process, which is the transformation of the data. The extracted data is most of the time raw data that is not usable in its original form. In case the data does not require any transformation, we call it pass through data. If the data needs transformation, the transformation phase comes into play. The main goal of this phase is to convert the data in a desired file format, that is suited for your system.

Problems that can occur in an extracted data set are called Data Integrity problems. A few examples of these types of problems are as follows:

1. Different spelling of the same person like Jon, John, etc.
2. There are multiple ways to denote company name like Google, Google Inc.
3. Use of different names like Cleaveland, Cleveland.
4. There may be a case that different account numbers are generated by various applications for the same customer.
5. In some files required data remains blank.

A visual representation of some problems can be found in figure 9.
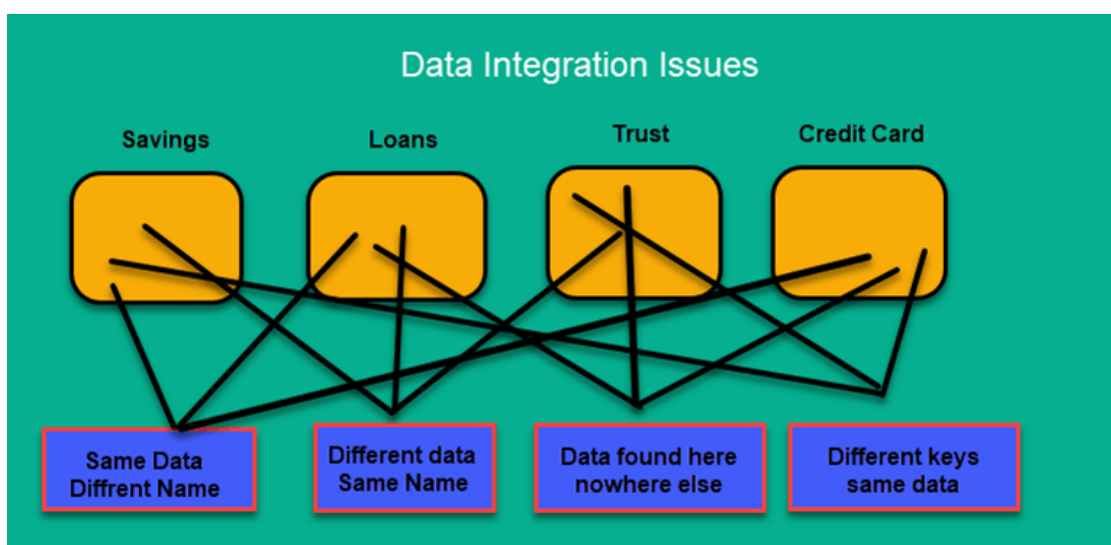
Figure 9: Data integration issues (Guru99, 2019)

The transformation phase can be divided in two types of transformations. During this cookbook we are only going to perform of the points mentioned below. The other points will performed in the cookbook: "Data modeling in MongoDB using MongoEngine". The two transformation types are as follows:

1. **Basic transformations**
These types of transformations include:
➔ **Cleaning**
Cleaning the data is done by, for example, mapping NULL values to 0 or Gender Male to "M" and Gender Female to "F" etc.
➔ **Format revision**
Character set conversion, unit of measurement conversion, date/time conversion, etc.
➔ **Restructuring**
Establishing key relationships across tables.
➔ **Deduplication**
Identifying and removing duplicate records.

2. **Advanced transformations**
These types of transformations include:
➔ **Derivation**
Applying business rules to your data that derive new calculated values from existing data for example, creating a revenue metric that subtracts taxes.
➔ **Filtering**
Selecting only certain rows and/or columns.
➔ **Joining**
Linking data from multiple sources – for example, adding ad spend data across multiple platforms, such as Google Adwords and Facebook Ads.
➔ **Splitting**
Splitting a single column into multiple columns.
➔ **Data validation**
Simple or complex data validation for example, if the first three columns in a row are empty then reject the row from processing.
➔ **Summarization**
Values are summarized to obtain total figures which are calculated and stored at multiple levels as business metrics – for example, adding up all purchases a customer has made to build a customer lifetime value (CLV) metric.
➔ **Aggregation**
Data elements are aggregated from multiple data sources and databases.
➔ **Integration**
Give each unique data element one standard name with one standard definition. Data integration reconciles different data names and values for the same data element.

1.3.3 Load

After we transformed the data in the desired data format we want to load it into our data store. There are a variety of data stores you can choose from. These types of data stores can be broadly classified into:

- **Relational databases**

  Examples of relational data stores include Oracle, MySQL and PostgreSQL. Relational databases store data in tables that each represents a different entity. These types of data stores follow ACID (Atomic,Consistent,Isolation and Durability). ACID is a rule set that allows transactions and rollback and follows a normalized model preventing data duplicates.

Figure 10: Logo Oracle

Figure 12: Logo MySQL

Figure 11: Logo PostgreSQL

Below you can find an example of a relational database model. This example model uses 2 entities:

1. An employee which has the attributes employee_id, first_name, last_name and address. This can be seen in figure 13.

2. A Payment which has the attributes payment_id, amount and date. This can be seen in figure 14.
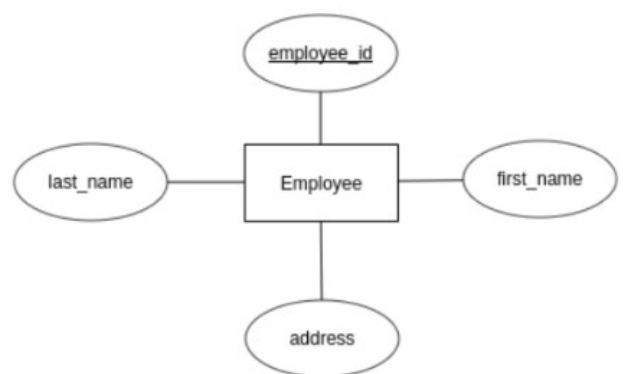
Figure 13: Employee entity diagram (Google, 2019)
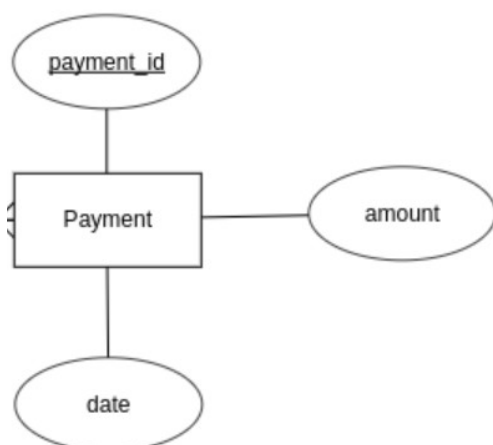
Figure 14: Payment entity diagram (Google 2019)

If an employee has multiple payments, the relation would look like this:
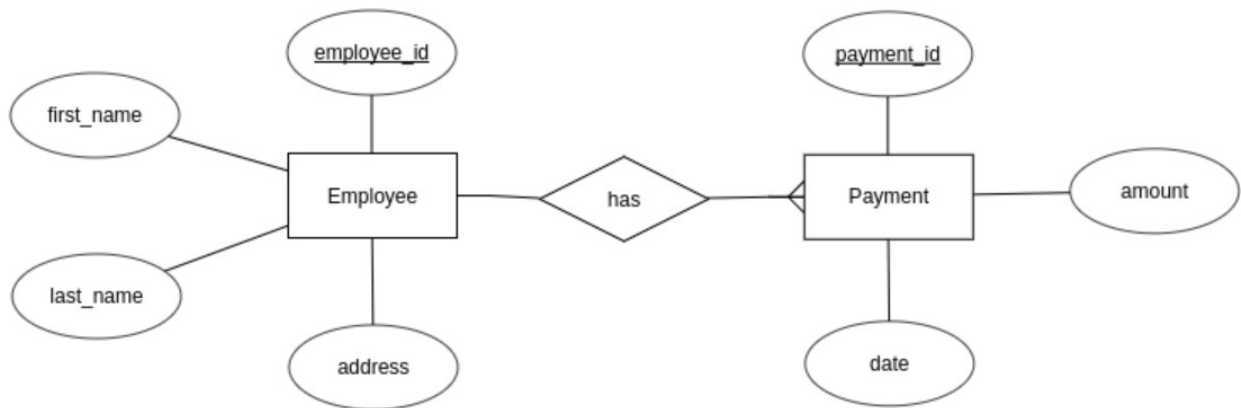
Figure 15: Employee - Payment relation diagram (Google, 2019)

and the table in the database would then look like this:

employee_id	first_name	last_name	address
1	John	Doe	New York
2	Benjamin	Button	Chicago
3	Mycroft	Holmes	London

FOREIGN KEY

payment_id	employee_id	amount	date
1	1	50,000	01/12/2017
2	1	20,000	01/13/2017
3	2	75,000	01/14/2017
4	3	40,000	01/15/2017
5	3	20,000	01/17/2017
6	3	25,000	01/18/2017

Figure 16: Employee - payment database tables (Google, 2019)

The payment has a foreign key (employee_id) which is a reference to the primary key (employee_id) in the employee table.

These types of data stores are highly used in applications where data integrity is important, for example a banking application.

- **Key-value stores**

  A key-value store is a database which uses an array of keys where each key is associated with only one value in a collection. It is quite similar to a dictionary or a map data structure.

  An example of a key-value store is Redis, it is an in memory data structure key-value store. It provides an option to persist data onto the disk.

  In figure 17 you can find a visual representation of a key which has a value.
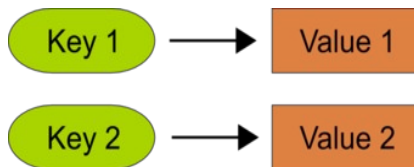
Figure 17: Key-value store

Figure 18: Logo Redis

- **Document stores (Non-relational data stores)**

  Document stores store data in the JSON file format. It stores the mapping of key to values where values can be primitive types, arrays or complex documents. Document stores are a combination of relational database models found in relational databases and key-value stores providing the benefit of both.

  It is possible to create database models in non-relational data stores. This can be done in 2 ways. In figure 19 you can see an example of an embedded database model. In figure 20 you can see an example of a referenced database model.

Figure 19: Embedded document (FullStack Academy,2019)
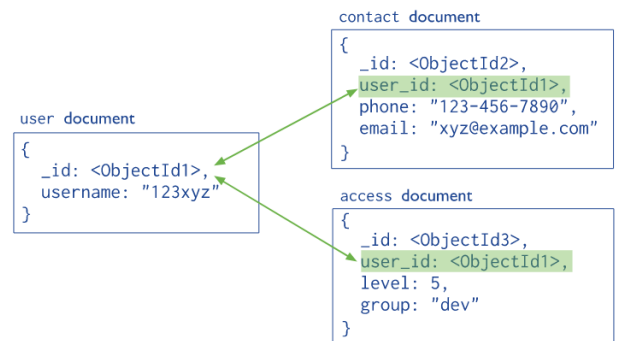
Figure 20: Referenced document (FullStack Academy 2019)

  With the referenced database model you can see the user_id is a reference to other documents. With the embedded database model the document contains sub-document that represent other entities. Examples of document stores are MongoDB and Elasticsearch.

Figure 21: Logo Elasticsearch

Figure 22: Logo MongoDB

2. ETL-Process example: Swedish Crane dataset

This chapter describes a real life example of the ETL-Process. For this example we will be using a data set obtained by an organization called Movebank.

Figure 23: Banner Movebank

The Movebank Data Repository contains published data sets of animal movement data in the Movebank format. This is distinct from the main Movebank tracking database, in which users control access and are responsible for their data quality, and where most data are stored. To be published in the Movebank Data Repository, a data set in Movebank undergoes an official review process and, when accepted, is granted a unique identifier (DOI) and license and is made publicly available.

2.1 Introduction to the data set

The data set we are going to use in this example was obtained by attaching a transmitter to the legs of a Swedish crane. In figure 24 you can see an image of the crane. In figure 25 you can see an image of the transmitter attached to his leg.

Figure 24: Crane (Google, 2019)

Figure 25: Tracker attached to leg (Google, 2019)

Attaching the transmitters was done in a place called Grimsås in Sweden. In figure 27 you can see where Grimsås is located on the map of Sweden and in figure 26 on the map of Europe.
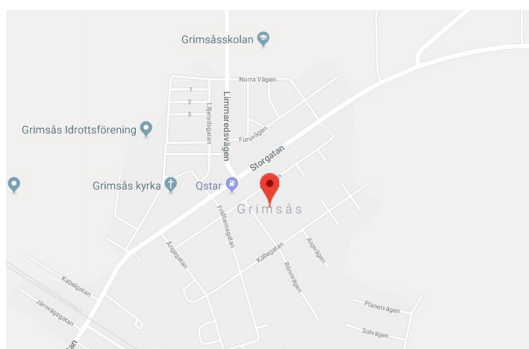
Figure 27: Grimsas zoomed (Google Maps, 2019)
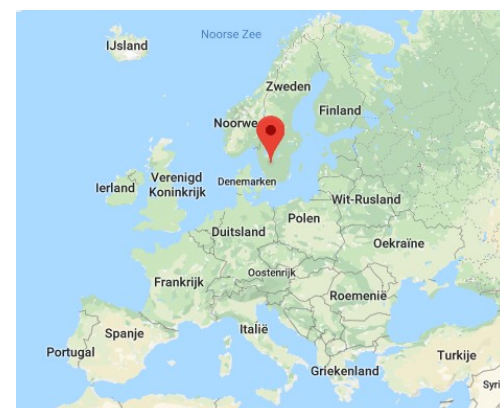
Figure 26: Grimsas on map of Europe (Google Maps, 2019)

Some information related to the dataset is as follows:

➔ The data set contains 123,805 transmissions of this crane over the period of 21-07-2013 to 13-02-2016.
➔ The tracker contains a GPS sensor and a few other environmental sensors, of which the sensor data every 15 minutes (4x / hour) if an SMS message is sent via a GSM channel sent. All that sensor data can be found in the logbook.
➔ The most famous location is North-East France.

In figure 28 you can see one data row from the data set. This datarow shows the column names and their values.

event-id	visible	timestamp	location-long	location-lat	argos:altitude	gps:fix-type	gps:hdop	gps:vdop	ground-speed	heading	height-above-ellipsoid	tag-tech-spec
1154727244	True	2013-07-21 04:22:57	13.577142	57.504177	NaN	3	1.6	2.0	0.0	NaN	194	NaN

tag-voltage	sensor-type	individual-taxon-canonical-name	tag-local-identifier	individual-local-identifier	study-name
4110	gps	Grus grus	9381	9381	GPS telemetry of Common Cranes, Sweden

Figure 28: Data record crane data set

The crane data set can be compared to a data set obtained from a drone. Those transmission also contain data about the location, height, and speed.

In figure 30 below you can find a visual representation of what the ETL-Process will look like in the context of this data set.

Figure 29: ETL-Process (Yousuf, Applied Informatics, 2019)
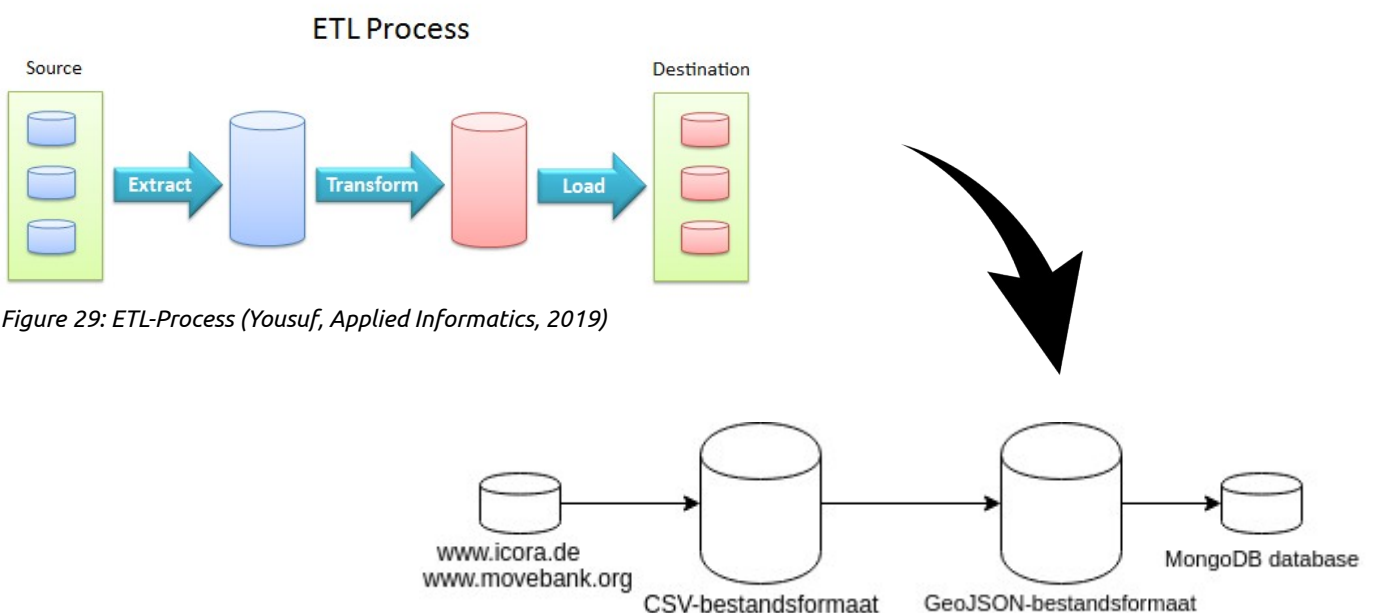
Figure 30: ETL-Process in context of data set

15

2.2 Extracting the data set

The data set will be extracted from the Movebank data repository. The way we are going to do this is by navigating to the Movebank website and downloading the complete data set. This procedure is described in the steps below.

NOTE: If your are following the Complete GeoStack Course you don't have to download this dataset again. You already did this in section 3.4.1 of the cookbook: "Creating the GeoStack Course VM".

1) Navigate to the Movebank downloading page which can be found on the following URL: https://www.movebank.org/panel_embedded_movebank_webapp
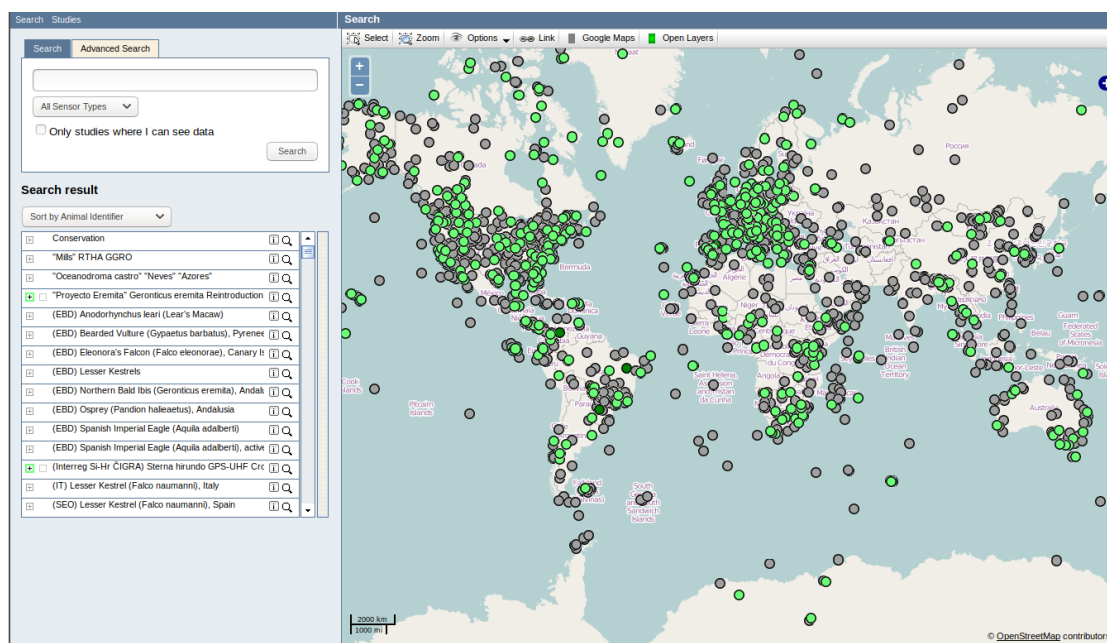
Figure 31: Download Page Movebank

2) In the search field enter: "grus grus" and then press Search as shown in the illustration:
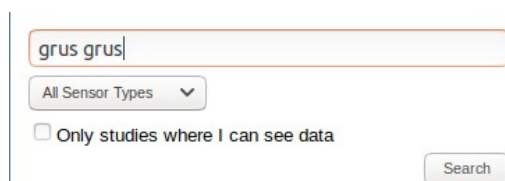
Figure 32: Search field

3) Scroll down till you find the entry : GPS telemetry of Common Cranes, Sweden (n=19):
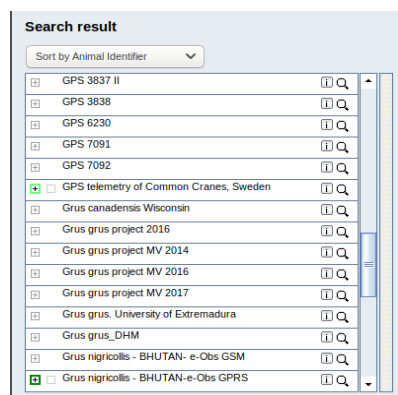
Figure 33: List of entries

16

4) Click on the plus sign to expand the search results.

Figure 34: Extended list of entries

5) Select the box with the ID-number : 9381

Figure 35: Select desired box

6) Click on the information icon, this will show a pop-up

Figure 36: Select information icon

7) Click on download search result.

Download search result

Figure 37: Download button

8) Click on I agree.

Figure 38: Download terms

9) Click on download data.

Figure 39: Download tracking data

That's it! Now we have the study data in the file format CSV. In the next chapter is described how you should explore the downloaded Crane Tracker dataset.

```
event-id,visible,timestamp,location-long,location-lat,argos:altitude,gps:fix-type,gps:hdop,gps:vdop,ground-speed,heading,height-above-ellipsoid,tag-tech-spec,tag-voltage,sensor-type,individual-taxon-canonical-name,tag-local-identifier,individual-local-identifier,study-name
1154727247,true,2013-07-21 03:06:32.000,13.583908,57.503796,,3,2.0,3.0,0.0,,193.0,,4110.0,"gps","Grus grus","9381","9381","GPS telemetry of Common Cranes, Sweden"
1154727246,true,2013-07-21 03:51:34.000,13.578312,57.504863,,3,1.5,2.5,0.5144,,194.0,,4100.0,"gps","Grus grus","9381","9381","GPS telemetry of Common Cranes, Sweden"
1154727245,true,2013-07-21 04:07:09.000,13.578205,57.50415,,3,1.1,1.5,0.0,,199.0,,4110.0,"gps","Grus grus","9381","9381","GPS telemetry of Common Cranes, Sweden"
1154727244,true,2013-07-21 04:22:57.000,13.577142,57.504177,,3,1.6,2.0,0.0,,194.0,,4110.0,"gps","Grus grus","9381","9381","GPS telemetry of Common Cranes, Sweden"
1154727243,true,2013-07-21 04:38:39.000,13.576754,57.504238,,3,1.1,1.6,0.0,,192.0,,4110.0,"gps","Grus grus","9381","9381","GPS telemetry of Common Cranes, Sweden"
1154727242,true,2013-07-21 04:54:27.000,13.574988,57.505005,,3,1.6,1.6,0.5144,,196.0,,4110.0,"gps","Grus grus","9381","9381","GPS telemetry of Common Cranes, Sweden"
1154727241,true,2013-07-21 05:10:09.000,13.573163,57.505985,,3,2.4,1.6,0.0,,195.0,,4110.0,"gps","Grus grus","9381","9381","GPS telemetry of Common Cranes, Sweden"
1154727240,true,2013-07-21 05:25:56.000,13.573152,57.506939,,3,2.0,1.8,0.0,,203.0,,4110.0,"gps","Grus grus","9381","9381","GPS telemetry of Common Cranes, Sweden"
1154727239,true,2013-07-21 05:41:40.000,13.573357,57.507603,,3,1.1,1.0,0.0,,199.0,,4110.0,"gps","Grus grus","9381","9381","GPS telemetry of Common Cranes, Sweden"
1154727238,true,2013-07-21 05:57:33.000,13.5755,57.507931,,3,1.4,2.3,0.0,,194.0,,4110.0,"gps","Grus grus","9381","9381","GPS telemetry of Common Cranes, Sweden"
1154727237,true,2013-07-21 06:13:28.000,13.576068,57.508114,,3,2.8,3.2,0.5144,,197.0,,4110.0,"gps","Grus grus","9381","9381","GPS telemetry of Common Cranes, Sweden"
1154727236,true,2013-07-21 06:29:08.000,13.576232,57.508171,,3,1.8,1.9,0.0,,199.0,,4110.0,"gps","Grus grus","9381","9381","GPS telemetry of Common Cranes, Sweden"
1154727235,true,2013-07-21 06:44:58.000,13.575552,57.508965,,3,0.9,1.1,0.0,,197.0,,4110.0,"gps","Grus grus","9381","9381","GPS telemetry of Common Cranes, Sweden"
1154727234,true,2013-07-21 07:02:04.000,13.574509,57.510025,,3,4.6,7.2,0.5144,,180.0,,4110.0,"gps","Grus grus","9381","9381","GPS telemetry of Common Cranes, Sweden"
1154727233,true,2013-07-21 07:16:34.000,13.575097,57.510265,,3,2.6,3.6,0.0,,187.0,,4110.0,"gps","Grus grus","9381","9381","GPS telemetry of Common Cranes, Sweden"
1154727232,true,2013-07-21 07:31:27.000,13.577202,57.509205,,3,2.9,4.0,0.0,,197.0,,4110.0,"gps","Grus grus","9381","9381","GPS telemetry of Common Cranes, Sweden"
1154727231,true,2013-07-21 07:47:10.000,13.578967,57.50959,,3,1.2,1.9,0.0,,206.0,,4110.0,"gps","Grus grus","9381","9381","GPS telemetry of Common Cranes, Sweden"
```

Figure 40: Data in CSV file format

2.3 Exploring the data set

Now that we have extracted the data from our data source, we want to start exploring the data. While exploring, we want to gather as much information about the data set as possible. This information includes:

- ✔ How much data rows are in the data set?
- ✔ How much columns are in the data set?
- ✔ All column names and data types in the data set.
- ✔ A description of each column.
- ✔ A simple visualization of the data set.

This information will be used to identify inconsistencies , important data and to eventually create a data model. These inconsistencies will be transformed in the next chapter.

For the data analyses we are going to use Jupyter Lab with a Python library called Pandas and for the initial visualization of the data we are going to use Cartopy and Matplotlib. You should have enter the code described below in the notebook which you created in the cookbook: "Creating the GeoStack Course VM "

To be able to run the code mentioned below, we first have to import the Pandas library in Jupyter notebook. The code for this is shown in figure 41.

```
[1]: import pandas as pd
```

Figure 41: Importing Pandas

Now that we have imported the required library, we want to read the data set in the notebook. The name of the data set is "20181003_Dataset_SV_TrackerID_9381_ColorCode_RRW-BuGBk_Crane_Frida.csv" and can be found in the same folder as the notebook. Reading data in a Jupyter notebook can be done using the code shown in figure 42. In the figure we import the dataset located in the folder: "~/GeoStack-Course/Course-Datasets/" which was downloaded in section 3.4.1 of the cookbook: "Creating The GeoStack Course VM".

```
SW_Crane_Frida = pd.read_csv(
    '../../../Course-Datasets/CSV/20181003_Dataset_SV_TrackerID_9381_ColorCode_RRW-BuGBk_Crane_Frida.csv')
```

Figure 42: Loading the dataset

To make sure we imported the correct data set we want to show the first row of the data set. How this is done is shown in figure 43.

```
[3]: SW_Crane_Frida[:1]
```

	event-id	visible	timestamp	location-long	location-lat	argos:altitude
0	1154727247	True	2013-07-21 03:06:32.000	13.583908	57.503796	NaN

Figure 43: Print first row of data set

Now that we have imported the data set, we want to start the information gathering process by answering the questions which we formulated above:

1. How much data rows are in the data set? The output is the amount of rows in the data set.

```
[4]: len(SW_Crane_Frida) # Print the amount of datarows in the dataset.

[4]: 123805
```

Figure 44: Print length data set (amount of rows).

2. How much columns are in the data set? The output is the amount of columns in the data set.

```
[5]: len(SW_Crane_Frida.columns) # Print the amount of columns in the dataset

[5]: 19
```

Figure 45: Print the amount of columns in the data set.

3. What are the column names and their data types?

The first column in the output contains the column names. The second column in the out contains the data type per column.

```
[7]: SW_Crane_Frida.dtypes # Print the column names and datatype.

[7]: event-id                           int64
     visible                             bool
     timestamp                         object
     location-long                    float64
     location-lat                     float64
     argos:altitude                   float64
     gps:fix-type                       int64
     gps:hdop                         float64
     gps:vdop                         float64
     ground-speed                     float64
     heading                          float64
     height-above-ellipsoid           float64
     tag-tech-spec                    float64
     tag-voltage                      float64
     sensor-type                       object
     individual-taxon-canonical-name   object
     tag-local-identifier               int64
     individual-local-identifier        int64
     study-name                        object
     dtype: object
```

Figure 47: Print column names and data types

Column	Type
event-id	int
visible	bool
timestamp	datetime
location-long	float
location-lat	float
argos:altitude	float
gps:fix-type	int
gps:hdop	float
gps:vdop	float
ground-speed	float
heading	float
height-above-ellipsoid	float
tag-tech-spec	float
tag-voltage	float
sensor-type	string
individual-taxon-canonical-name	string
tag-local-identifier	int
individual-local-identifier	int
study-name	string

Figure 46: Markdown table

Lets create a better looking table using the above output. The result of this table is shown in figure 47.

To create a table in Jupyter notebook you have to create markdown cell by using the dropdown box at the top of Jupyter lab window. How this is done is shown in figure 48 .

When a cell is a markdown cell you can create a table by writing the same lines as shown in figure 49, but then you replace the column names with the actual column names and the column types with the actual column types. When you run the cell, the output will return a good looking table, as shown in figure 47 .
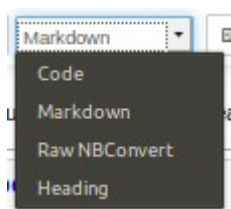
Figure 48: markdown

```
|Column|Type|
|--|--|
|columnname1| columntype 1|
|columnname2| columntype 2|
|columnname3| columntype 3|
```

Figure 49: Input cell

20

4. A description of each column.

In a lot of cases a PDF file is included when downloading a data set. In this PDF you will find a description of each column in the data set. For us that's not the case, so we are going to simply use Google. Most of the columns will become clear when searching the column name in Google, if that's not the case you could always contact the owners of the data set. Once you have found the meaning of the columns in the dataset you should add them to the table which you created in the previous step. The final table can be found in figure 50.

Column	Type	Desc.
event-id	int	The ID of the transmission (Unique)
visible	bool	Is the tracker still visible or not?
timestamp	datetime	The DateTime of transmission
location-long	float	The Longitude location
location-lat	float	The Latitude location
argos:altitude	float	Argos type sensor for height of the transmission
gps:fix-type	int	Type of GPS Tracker
gps:hdop	float	Horizontal dilution of precision
gps:vdop	float	Vertical dilution of precision
ground-speed	float	Speed of the bird while on the ground.
heading	float	Direction the bird is faceing (In degrees)
height-above-ellipsoid	float	Height of the bird HAE -> MSL
tag-tech-spec	float	
tag-voltage	float	Voltage level of the tracker.
sensor-type	string	Type of sensor used in tracker.
individual-taxon-canonical-name	string	Name of the bird, in latin.
tag-local-identifier	int	ID of the tracker.
individual-local-identifier	int	ID of the bird
study-name	string	Name of the study

Figure 50: Table with descriptions

5. Creating a simple visualization of the dataset.

To visualize the data set, we are going to use Cartopy and Matplotlib. An initial visualization of a dataset will give you a good impression of how the dataset is going to look in an application.

To use Cartopy and Mathplotlib in our notebook need to import the required libraries and modules. How this is done is shown in the figure below.

```
[10]:  import cartopy # The Python package for Geospatial data processing.
       import cartopy.crs as ccrs # The module for specifying the map projection
       import cartopy.feature as cfeature # The module for adding features to the map.
       import matplotlib.pyplot as plt # The Python package for creating plots.
```

Figure 51: Import visualization libraries and modules

Now that we have imported the required modules, we want to create the Cartopy map and add the required features to it. How this is done is shown in the illustration below. The code is explained using inline comments.

```python
'''
Below we create a new plot using MathPlotLib.
We pass the a size of the figure as parameter.
'''
plt.figure(figsize = (20, 12))

'''
Below we create a new Catopy map.

We pass the projection of the Cartopy map as parameter.
The projection we are going to use is called: "PlatteCarree". The crs stands for: "Coordinate Reference system".
The type of CRS used in the Cartopy map defines the way the map will be shown. PlatteCarre uses
equirectangular projection (North Latitude and East Longitude).
We assign the instance of the plot to a variable called: "cartopyMapCranes".
'''
cartopyMapCranes = plt.axes(projection=ccrs.PlateCarree())

'''
Below we add the coastal lines to the cartopy map. We pass the resolution: "10m" as parameter. This value defines
the maximum deviation the coastal line can have. The higher the value, the higher the devation of the correct
location of the lines.
'''
cartopyMapCranes.coastlines(resolution='10m')

'''
Below we add the landsurface to the Cartopy map.We give the landsurface (face) the color white.
We give the edges of the landsurface (edge) te color black.
'''
cartopyMapCranes.add_feature(cartopy.feature.LAND.with_scale('10m'), edgecolor='black', facecolor = "white")
'''
Below we add the lakes to the cartopy map.
We give the edges of the lake the color black.
'''
cartopyMapCranes.add_feature(cfeature.LAKES.with_scale('10m'), edgecolor = 'black')

'''
Below we add the sea surface to the Cartopy Map
'''
cartopyMapCranes.add_feature(cfeature.OCEAN)

'''
Below we add the rivers and the borders to the Cartopy Map
'''
cartopyMapCranes.add_feature(cfeature.RIVERS.with_scale('10m'))
cartopyMapCranes.add_feature(cfeature.BORDERS.with_scale('10m'))
```

Figure 52: Create Cartopy map

When we run this cell we will get an map similar to the one shown in the illustration below.
Note: The first time this cell is run it can take a bit longer since the base map has to be downloaded. This process takes about 2 minutes depending on you network speed!
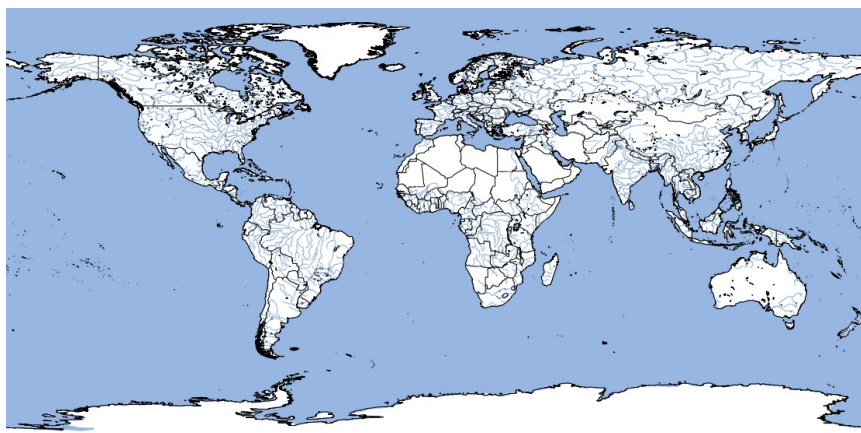
Now that we have our base map we can plot the data set by using a scatter plot. To plot a scatter plot we need the latitude and longitude coordinates of all the data rows in the dataset.

Because of the information gathering process we performed earlier we now know the names of the required columns which are location-long and location-lat.

To plot the datapoints on the Cartopy map we are going to use the syntax:

{CartopyMap name}.scatter({Dataframe name}['{longitude column name}'], {Dataframe name}['{latitude column name}'],color={The color of the data points}, s = {The size of the datapoints}

Where **CartopyMap name** is the name of the Cartopy Map which is cartopyMapCranes in this case and **Dataframe name** is the name of the dataframe which is SW_Crane_Frida in this case.

So let's plot the Crane on the CartopyMap. This is done by adding the following line in the cell which we created above. We add this line below the line which adds the borders to the Cartopy Map.

```
cartopyMapCranes.scatter(SW_Crane_Frida['location-long'], SW_Crane_Frida['location-lat'], color="blue", s = 1)
```

Figure 53: Add data set

Now when we run the cell again we should end up with the map as shown in figure 54.
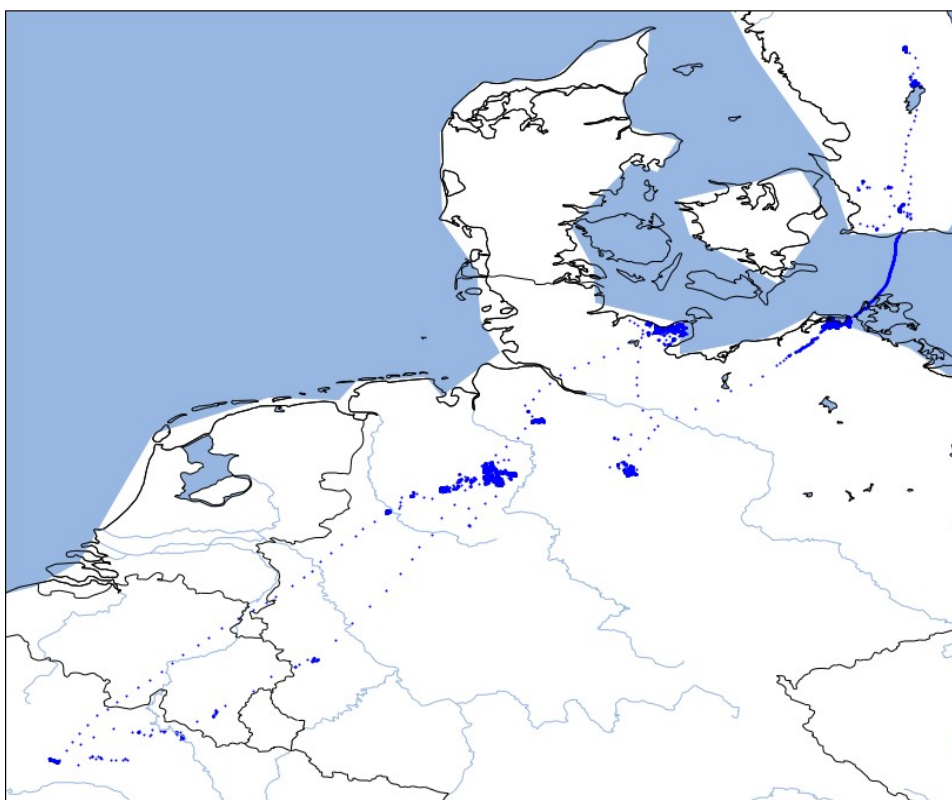
Figure 54: Swedish Crane plot

Now that we have obtained some useful information concerning the dataset we should start the data filtering process. We are going to filter the data which we will not be using in the GeoStack.

To know what data is important for our project we first need to define the scope of our project.

The end goal of the project is to make sure we can visual data by:

➔ time : We need the time data if we want to create a timeline in which we can select a specific time span
➔ coordinates: We need the coordinates to visualize the location from which transmission was send.
➔ Speed : We need to know the speed of the bird if we want to visualize an animation of the flight path of the bird.
➔ Heading (in degrees): we need to know the heading of the bird to visualize which way the bird is flying to.
➔ Height: We need the height of the bird to create a 3D visualization of the bird.
➔ Identifier: We need the identifier of the bird to visualize information on the specific bird.
➔ Type of bird and sensor: We need a way to identify that the bird we are tracking is a Crane and the type of sensors are in the tracker.
➔ Tracker metadata: We need to be able to check whether the tracker is still sending transmissions.

Going back to the table which we created earlier, we can see that the data set contains more data than we actually need. The data needed to achieve the end goals for our data visualization are as follows:

✔ To identify the data record we need the "event-id" column.
✔ For the time the transmission was send we need the "timestamp" column.
✔ For the speed we need the "ground-speed" column.
✔ For the heading we need the column: "heading".
✔ For the coordinates we need the columns "location-long" and "location-lat".
✔ For the height we need the column : "height-above-ellipsoid"'.
✔ For the identifier we need the column: individual-local-identifier.
✔ To identify whether the bird we are tracking is a crane we need the column: individual-taxon-canonical-name.
✔ To identify the type of sensor attached to the crane we need the column: "sensor-type".
✔ Some columns that represent metadata related to the tracker are: "visible" (to check if the tracker is still sending transmissions), "tag-voltage" (which contains the amount of voltage that is running trough the tracker. This will be 0 if tracker has died).

That's it! We now have identified the data required to fulfill our needs. The next chapter describes what you need to do to filter and transform the identified data.

2.3.2 Filtering the important data

Now that we have identified the relevant data for our application we can start the filtering process. To select and filter the relevant columns we need to create a list containing the column names which we want to filter. We assign this list to a variable called: "columns_to_filter". How this is one is shown in figure 55.

```
[18]:  # Define the columns which we want to filter from the dataset.
       columns_to_filter = ['event-id', 'study-name',
                            'timestamp','visible',
                            'ground-speed','heading',
                            'location-long','location-lat',
                            'height-above-ellipsoid',
                            'individual-taxon-canonical-name',
                            'sensor-type','tag-voltage',
                            'individual-local-identifier']
```

Figure 55: Filter relevant data

Now we are going to create a new data frame to which we are going to assign the filtered data. We are going to call this dataframe "Filtered_SW_Crane_Frida". The we need to specify the old dataframe and pass the list of columns which we want to filter. How this is done is shown in the figure below.

```
[19]:  Filtered_SW_Crane_Frida = SW_Crane_Frida[columns_to_filter]
```

Now that we have created a new data frame, we want to confirm whether we filtered the data correctly.

To show the first row of the filtered data frame we use the code shown in figure 56.

```
[20]:  Filtered_SW_Crane_Frida[:1] # Show first row of new dataframe.
```

	event-id	study-name	timestamp	visible
0	1154727247	GPS telemetry of Common Cranes, Sweden	2013-07-21 03:06:32.000	True

Figure 56: First row filtered data

Now we want to export the filtered data frame to a file format which can be easily used in the world of GIS (Geographic information systems) and thus our GeoStack. This is the file format JSON (JavaScript Object Notation).

In the next chapter two techniques are described which you can use to transform datasets to the file format JSON.

25

2.3.4 Transforming the file format to JSON

The file formats which are mainly used in the world of GIS are JSON and GeoJSON. In the following chapters is shown how to convert datasets to these file formats using 2 techniques.

Below a description is given on how to transform a dataset with the file format CSV to JSON. As mentioned before; Transforming a dataset to JSON can be done using 2 methods.
The first method is using the built in JSON module in Python and the second method is using the JSON module in Pandas.

Below both methods are shown and the differences are explained (if there are any). These methods are shown using some test data which we are going to created ourselves. In this way it's easier to understand.

So let's create our test data. How this is done is shown in figure 57.

```
[21]: crane_test_data = {
          'event-id':432,
          'lat': 1342.43,
          'lon': 33.1
      }
```

Figure 57: Test Data

Now let's perform both methods on the test data which we created above.

Method 1 : Transforming and reading JSON using the built-in JSON module in Python

To use the built-in JSON module in Python we first need to import this module. This is done using the following code:

```
[24]: import json #Import the Python JSON Module
```

Now let's put the module to use by writing the test data to a JSON file called: "json_module_python.json". How this is done is shown in the figure below.

```
[25]: # Write the test data to a file called: "json_module_python.json"
      # We pass the test data as first parameter and the newly created
      # json file as second parameter.
      with open ('json_module_python.json', 'w') as json_file:
          json.dump(crane_test_data, json_file)
```

Figure 58: Write to file

Now let's open the newly created file which is located in the same folder as the notebook. Te content of this file is shown in the illustration below.

```
{"event-id": 432, "lat": 1342.43, "lon": 33.1}
```

Figure 59: Contents of the JSON file

Now let's read the JSON file which we created above using the built-in JSON Python module. How this is done is shown in figure 60.

```
[27]: # Read the test data that was writen to a file in the previous step.
      # We assign the name of the JSON file to a variable called json_file
      # after which we pass the file as parameter in the json.load() function.
      # We assign the loaded data to a variable called: "test_dataset".
      with open('json_module_python.json') as json_file:
          test_dataset = json.load(json_file)
```

Figure 60: Load the data

Now let's print the data which was loaded by using the code above. How this is done, is shown in figure 61.

```
[28]: test_dataset
```

```
[28]: {'event-id': 432, 'lat': 1342.43, 'lon': 33.1}
```

Figure 61: Print the loaded data

The output is **NOT in a valid JSON format.** This is because the strings (e.g event-id) should be wrapped in double quotes. To transform the output to a valid JSON format we have to use the syntax json.dumps() as shown in figure 62.

```
[29]: json.dumps(test_dataset) # Print the loaded test data in a correct JSON format.
```

```
[29]: '{"event-id": 432, "lat": 1342.43, "lon": 33.1}'
```

Figure 62: Valid JSON

To validate whether the JSON output is valid we can navigate to the website : https://jsonformatter.curiousconcept.com/ and paste the record in the text box and click the process button. Don't forget to remove the ' at the beginning and the end of the record. The output should be similar to the illustration below.
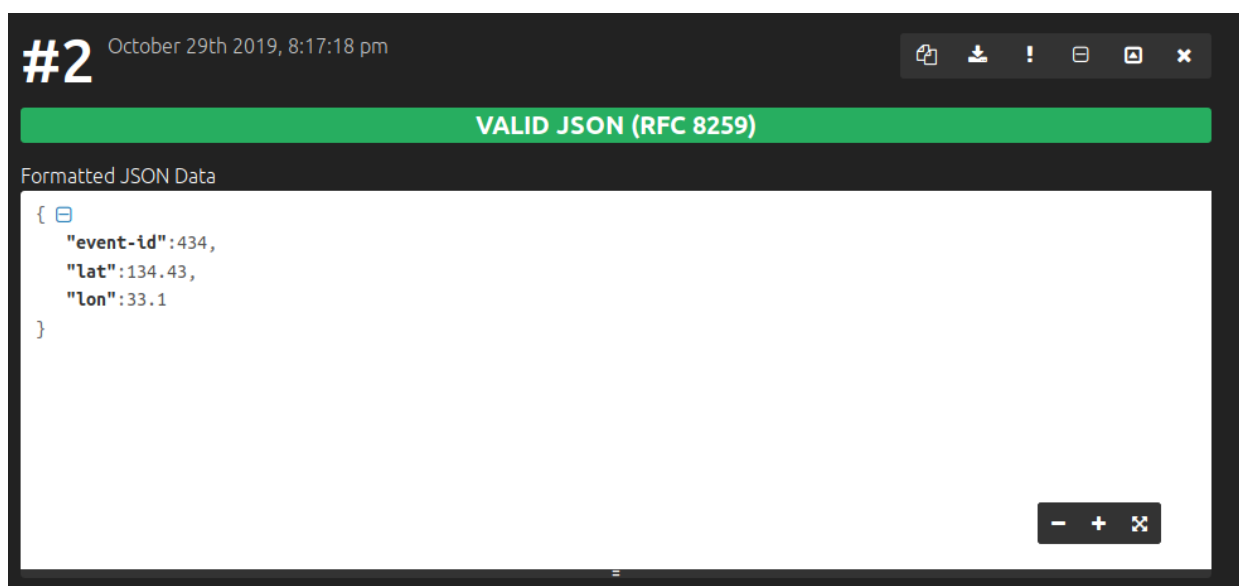
Method 2: Transforming and reading using the Python Package "Pandas"

Transforming a file format to JSON using the Python package: "Pandas" is pretty straight forward.

First we need to create a new dataframe using the test data which we created earlier. The we need to transform and write the dataframe to a JSON file. After this is done we can read the written JSON file using the syntax: "pd.read_json({JSON file name})".

How this is done is shown in figure 63 .

```
[33]: # Create a new dataframe using the created test data.
      pandas_dataframe = pd.DataFrame([crane_test_data])

      # Create a new JSON file and write the dataframe to this file.
      # We do this by using the Pandas function: "to_json()" in which
      # we pass the name of the JSON file as first parameter and the
      # orientation in which the data has to be written as second parameter.
      pandas_dataframe.to_json('json_module_pandas.json',orient = 'records')

      # Read the newly created JSON file and assign it to a dataframe.
      # We use the syntax: "to_json()" to print the data as JSON and
      # not as an dataframe.
      pd.read_json('json_module_pandas.json').to_json()

[33]: '{"event-id":{"0":432},"lat":{"0":1342.43},"lon":{"0":33.1}}'
```

Figure 63: Reading and writing JSON using pandas

As you can see output of the to_json() function is also valid JSON.

Again, just to be sure we can navigate to the website : https://jsonformatter.curiousconcept.com/ and paste the record in the text box and click the process button. The output is shown in figure 64.
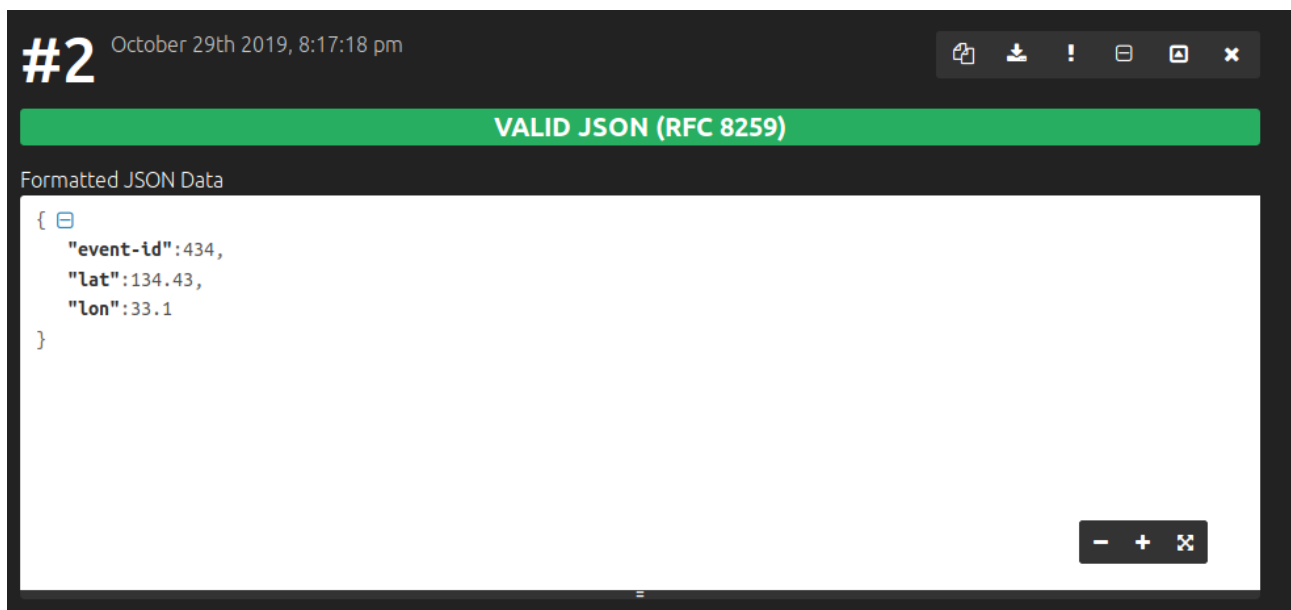

Figure 64: Validate JSON

Now that we know both methods that can be used to transform data to the file format JSON we want to export our Crane dataset to this file format.

Transforming our Crane dataset to the file format JSON is done using the code shown in figure 65. We write the dataframe to a JSON file called: "Frida_SW.json".

```
[19]: # Transforming the Crane Dataset using the function: "to_json()" from Pandas
      Filtered_SW_Crane_Frida.to_json('../../../Course-Datasets/JSON/Crane_JSON/Frida_SW.json', orient = 'records')
```

Figure 65: Convert to JSON using Pandas

Now let's read the newly created JSON file using both methods. How this is done is shown in figure 66.

```
[ ]: # Read the Transformed dataset using the built-in Python function.
     with open('../../../Course-Datasets/JSON/Crane_JSON/Frida_SW.json') as transformed_Crane_Dataset:
         crane_data = json.load(transformed_Crane_Dataset)

     # Print the contents of the read JSON file (NOTE: This prints just a small portion of the
     # JSON to speed up the dumping process)
     json.dumps(crane_data)[:1000]
```

```
[21]: # Read the Transformed dataset using Pandas
      crane_dataframe = pd.read_json('../../../Course-Datasets/JSON/Crane_JSON/Frida_SW.json')

      # Print one row from dataframe
      crane_dataframe[:1]
```

[21]:	event-id	study-name	timestamp	visible	ground-speed	heading	location-long	loca
0	1154727247	GPS telemetry of Common Cranes, Sweden	2013-07-21 03:06:32	True	0.0	NaN	13.583908	57.

Figure 66: Reading the transformed JSON dataset

Again, just to be sure the JSON is valid we can navigate to the website: https://jsonformatter.curiousconcept.com/ and paste a record, returned the code above, in the text box and click the process button. The output is shown in figure 67.

Figure 67: Valid JSON

Because there were no error's, we can conclude that there is no difference between using the Pandas JSON module and the built-in JSON module from Python when converting file formats to JSON.

2.3.4 Transforming the file format to GeoJSON

Now that we have converted the data to JSON, we also want to convert is to GeoJSON. Although we are not going to use the file format GeoJSON during the GeoStack Course, the file format comes in hand for encoding a variety of geographic data structures.

Some of the Geometry types that GeoJSON supports are shown in the illustration below.

Geometry primitives

Type	Examples

Point	`{`
	`    "type": "Point",`
	`    "coordinates": [30, 10]`
	`}`

LineString	`{`
	`    "type": "LineString",`
	`    "coordinates": [`
	`        [30, 10], [10, 30], [40, 40]`
	`    ]`
	`}`

Polygon	`{`
	`    "type": "Polygon",`
	`    "coordinates": [`
	`        [[30, 10], [40, 40], [20, 40], [10, 20], [30, 10]]`
	`    ]`
	`}`

	`{`
	`    "type": "Polygon",`
	`    "coordinates": [`
	`        [[35, 10], [45, 45], [15, 40], [10, 20], [35, 10]],`
	`        [[20, 30], [35, 35], [30, 20], [20, 30]]`
	`    ]`
	`}`

For more information related to geometry types supported by GeoJSON can be found by clicking on the following URL: https://en.wikipedia.org/wiki/GeoJSON

Geometric objects with additional properties are Feature objects. Sets of features are contained by FeatureCollection objects. So let's start of by creating a list of properties (data columns) which we also want to add to the GeoJSON file. These properties **DO NOT** include the longitude and latitude columns.

We do this by adding the following code to our Jupyter Notebook:

```
[41]: # Defining the properties (Extra data columns) which we want to add to the
      # GeoJSON file.
      properties = ['event-id', 'study-name',
                              'timestamp','visible',
                              'ground-speed','heading',
                              'height-above-ellipsoid',
                              'individual-taxon-canonical-name',
                              'sensor-type','tag-voltage',
                              'individual-local-identifier']
```

Now let's create a function which can be used to convert a Pandas dataframe to GeoJSON. For this we use the code explained in figure 68.

```
[52]: # Here we create a function which transforms a dataframe to GeoJSON
      # The function takes the following input parameters:
      # 1) The dataframe to convert
      # 2) The columns which have to be added to the GeoJSON file as propperties
      #    these columns do not include the latitude and longitude columns.
      # 3) The latitude column name.
      # 4) The longitude column name.

      def dataframe_to_geojson(df,properties,lat,lon):

          # Create and assign a feature Collection object to a variable called: "geojson"
          geojson = {'type':'FeatureCollection', 'features':[]}

          # Itterate (loop) through all the rows in the dataframe
          # The code in this "for loop" is executed on each of the rows in the dataframe.
          for index, row in df.iterrows():
              # Assign a feature object to a variable called "feature".
              feature = {'type':'Feature',
                        'properties':{},
                        'geometry':{'type':'Point',
                                    'coordinates':[]}}

              # Assign the values of the longitude and latitude columns to the geometry object.
              # We use the column names, which were passed as parameter on the function call,
              # as names of the longitude and altitude columns.
              feature['geometry']['coordinates'] = [row[lon],row[lat]]

              # Here we create a for loop which loops through each of the columns assigned to the
              # properties list.
              for column in properties:

                  # Here we assign each column in the list of properties to a new object
                  # in the feature object.
                  feature['properties'][column]=row[column]

              # Here we add all the newly created features to the features object in
              # the GeoJSON object.
              geojson['features'].append(feature)

          # Here we return the created GeoJSON data.
          return geojson
```

Figure 68: Convert dataframe to GeoJSON

Now let's put the function which we created above to use. We do this by using the code shown in figure 69. We pass the Filtered Crane dataframe, the list of properties (extra columns), the latitude column name and the longitude column name as parameters in the function call.

```
[54]:  # Convert the Dataframe to GeoJSON.
       Crane_GeoJSON = dataframe_to_geojson(Filtered_SW_Crane_Frida,properties,'location-lat','location-lat')
```

Figure 69: Run conversion function and assign to variable

Now we want to write the GeoJSON to a file. How this is done is shown in figure 70.

```
[25]:  # Write the GeoJSON to a file called: "Frida_SW_GeoJSON".
       # We use the index=2 parameter to make the GeoJSON easy to read.
       # This is also known as human readable.
       with open('../../../Course-Datasets/JSON/Crane_GeoJSON/Frida_SW_GeoJSON.json', 'w') as geojson_file:
           json.dump(Crane_GeoJSON, geojson_file, indent=2)
```

Figure 70: Write result to file

When we run the code shown in figure 70, a new file is created. If we open the file we will be greeted with our data in the file format GeoJSON, as shown in figure 71.

```
{
  "type": "Feature",
  "properties": {
    "event-id": 1154727247,
    "study-name": "GPS telemetry of Common Cranes, Sweden",
    "timestamp": "2013-07-21 03:06:32.000",
    "visible": true,
    "ground-speed": 0.0,
    "heading": NaN,
    "height-above-ellipsoid": 193.0,
    "individual-taxon-canonical-name": "Grus grus",
    "sensor-type": "gps",
    "tag-voltage": 4110.0,
    "individual-local-identifier": 9381
  },
  "geometry": {
    "type": "Point",
    "coordinates": [
      57.503795999999994,
      57.503795999999994
    ]
  }
},
```

Figure 71: Output GeoJSON

That's it! We have now converted the filtered CSV file to GeoJSON format. To validate if the exported GeoJSON is in correct JSON format you can navigate to:
https://jsonformatter.curiousconcept.com/
And then paste a record of the GeoJSON in the text box and press the process button. The output should look something like the output in figure 72.
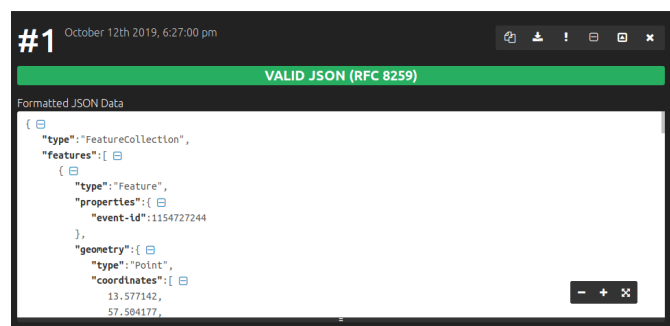

Figure 72: JSON validation

32

2.4 Loading the GeoJSON data

As mentioned in the introduction of this document; During the GeoStack Course we will not be using the file format GeoJSON, we will be using JSON instead.

Although we are not going to use the file format GeoJSON we are still going to load the GeoJSON data in a MongoDB datastore. By doing this you will get the basic knowledge on how to load data in a MongoDB datastore. This knowledge will be usefull when reading the cookbook: "Data modelling in MongoDB".

So now that we have filtered the CSV dataset and converted it to GeoJSON, we want to import it in our MongoDB data store. This is also the final phase of the ETL-Process which is the loading phase. In this chapter we will import the filtered and transformed data set in a MongoDB database called: "Crane_GeoJSON_Database".

The import process will not be done using a MongoDB data model. During the loading process, described in the next chapter, we are going to create an index on the geometry column in the MongoDB datastore. Don't worry if you are not familiar with creating indexes on a datastore. This will be explained in the cookbook: "Data modeling in MongoDB using MongoEngine".

The data loading process described below uses a Python package called: "MongoClient". This is an alternative to MongoEngine which we will be using in the cookbook: "Data modeling in MongoDB using MongoEngine".

We will not be using a data model in the example below. Creating and using data models is also described in the cookbook: "Data modeling in MongoDB using MongoEngine". But more on this later!

2.4.1 The loading process

To load the data in MongoDB we are going to use a MongoDB feature called "bulk import". This feature speeds up the import process significantly by inserting all the data records at once instead of importing the records one by one. Don't worry if some of the things explained below are not clear since it will become more clear in the next cookbook.

First we need to import the required modules in our Jupyter notebook. How this is done is shown in figure 73 .

```
[57]: from datetime import datetime # Import the Python datetime module used to convert timestamps.
      from pymongo import MongoClient # Import MongoClient to connect to a MongoDB database.
      from pymongo import GEOSPHERE # The GEOSPHERE module is used to create an index on the geometry column.
```

Figure 73: Import modules

Now let's read the the GeoJSON file using the built-in JSON Python module. How this is done is shown in figure 74.

```
[27]: # Load the GeoJSON file using the built-in JSON Python module.
      # After the file is loaded and read we assing it to a variable called:"geojson".
      # We will use this variable in the GeoJSON import function which we will create below.
      with open('../../../Course-Datasets/JSON/Crane_GeoJSON/Frida_SW_GeoJSON.json', 'r') as file:
          geojson = json.loads(file.read())
```

Figure 74: Load the transformed GeoJSON file

Now we need to create a function which will load the GeoJSON file in a MongoDB datastore. How this is done is shown in figure 75 .

```
[95]:  # Here we create the function which is used to load the GeoJSON data.
       # The function takes the following parameters as input:
       # 1) The dataframe to import
       # 2) The name of the database in which we are going to import the data.
       #     MongoDB will create a new database if the specified database does not exist.
       # 3) The collection in which we want to insert the data.
       # 4) The server on which the database is running.
       # 5) The port on which the server is running.
       def load_geojson(df,db,db_collection,db_server,db_port):

           # Here we connect to the MongoDB database using the input parameters.
           # Again,mongoDB will create a new database if the specified database does not exist.
           client = MongoClient('mongodb://'+db_server+':'+db_port)

           # Here we assign the input parameters related to the database connection to corresponding variables.
           # These variables will be used throughout this function.
           database = client[db]
           collection = database[db_collection]

           # Here we create an index on the geometry field in the database.
           # More information related to creating indexes on MongoDB databases
           # will be given in the cookbook: "Data modeling in MongoDB using MongoEngine".
           collection.create_index([("geometry",GEOSPHERE)])

           # Here we initialize the bulk insert feature of MongoDB
           # This is one of the techniques to do this.
           # The other technique is described in the cookbook: Data modeling in MongoDB using MongoEngine".
           bulk = collection.initialize_unordered_bulk_op()

           # Here we create a for loop which loops through each of the features in the GeoJSON file
           # which is passed as parameter in the function. We add each GeoJSON feature to the MongoDB
           # bulk insert feature instance which was created above.
           for feature in geojson['features']:
               bulk.insert(feature)

           # Here we call the function:".execute()" on the MongoDB bulk feature to which we appended
           # all the Features (Transmissions in this case) in our GeoJSON dataset.
           result = bulk.execute()

           # This line makes sure that we are notified when the bulk insert feature was completed.
           print('succesfully inserted the GeoJSON dataset')
```

Figure 75: Create function for loading data

Now let's put the function, which we created above, to use. How this is done is shown in figure 77. We pass the dataframe which we want to import, the database name, the database collection, the server on which the database is running and the port on which the database is running as parameters.

```
[86]:  load_geojson(Crane_GeoJSON,'Crane_GeoJSON_Database','Transmissions','localhost','27017','mongodb://localhost:27017')
```

Figure 76: Run function

The output of the function is shown in figure 78.

```
succesfully inserted the GeoJSON dataset
```

Figure 77: Output function

That's it. Now the GeoJSON data is imported in a MongoDB database. In the next chapter we are going to validate if the data is stored correctly. This will be done in the next section.

2.4.2 Validation using Mongo CLI

Now that we have imported the GeoJSON dataset we want to make sure that the data is correctly stored in the MongoDB database. For this we open a terminal and enter "mongodb". This will bring you to the MongoDB CLI (Command line interface) as shown in figure 78.

```
2019-10-14T12:58:00.123+0200 I  CONTROL  [initandlisten]
2019-10-14T12:58:00.123+0200 I  CONTROL  [initandlisten] ** WARNING: Access cont
rol is not enabled for the database.
2019-10-14T12:58:00.123+0200 I  CONTROL  [initandlisten] **          Read and wr
ite access to data and configuration is unrestricted.
2019-10-14T12:58:00.123+0200 I  CONTROL  [initandlisten]
---
Enable MongoDB's free cloud-based monitoring service, which will then receive an
d display
metrics about your deployment (disk utilization, CPU, operation statistics, etc)
.

The monitoring data will be available on a MongoDB website with a unique URL acc
essible to you
and anyone you share the URL with. MongoDB may use this information to make prod
uct
improvements and to suggest MongoDB products and deployment options to you.

To enable free monitoring, run the following command: db.enableFreeMonitoring()
To permanently disable this reminder, run the following command: db.disableFreeM
onitoring()
---

>
```
Figure 78: MongoDB CLI

Now we want to list all the databases in our MongoDB instance. This is done by entering the command "show dbs" as shown in figure 79.

```
> show dbs
GeoJSON_Database   0.008GB
admin              0.000GB
config             0.000GB
local              0.000GB
```
Figure 79: Show databases

Now we want to select the database that we created while loading the data. To do this enter use the command "dbs Crane_GeoJSON_Database" since this is the name of the database which we created while loading the data.

```
> use GeoJSON_Database
switched to db GeoJSON_Database
```
Figure 80: Use db command

Now we want to show the collections in the databast. This is done by entering the command "show collections".

```
> show collections
Transmissions
```
Figure 81: Show collections

As you can see it displayed the collection:"transmissions" which is correct. This is correct since we created this collection in the in the import script. To show the data in the collection we type the command: " db.Transmissions.find()". The output should similar to the output shown in figure 83.

Figure 82: Output Find Transmissions

As you can see in figure 82 the GeoJSON data has been successfully imported in a MongoDB datastore.

2.4.3 Validation using Mongo Compass

For extra convenience we are also going to validate, whether we loaded the data correctly, using a tool called "Mongo compass" which is a GUI version of the MongoDB Comand Line Interface.

As shown in figure 83 the import script created a Database called "Crane_GeoJSON_Database" and a collection called "Transmissions".
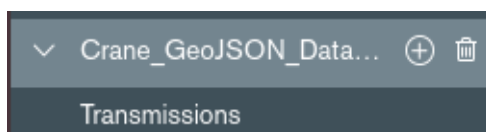

Figure 83: Database and Collection

When you click on the transmissions collection, you will be greeted with a list of MongoDB documents. An example of one of these documents is shown in figure 84.

Figure 84: Document in MongoDB

Some statistics of our database are shown in figure 85.

DOCUMENTS **123.8k**   TOTAL SIZE **31.6MB**   AVG. SIZE **268B**   INDEXES **2**

Figure 85: Database statistics

The following can be concluded from the numbers shown in figure 85:

➢ The Database contains around 1238000 documents, which is correct since our data set contained 123,805 transmissions

➢ The total size of the database is 31.6MB

➢ The average size per document is 268 Bytes, which is very small in comparison to the maximum document size of 16 MegaBytes.

➢ We have 2 Indexes on the database:

    ➢ Index on the _id object. This index is automatically created by MongoDB.

    ➢ Index on the geometry object which was created when loading our data set and using the GEOSPHERE module which was imported.

As mentioned before; some things related to MongoDB may not be clear at this point. Don't worry about this since the next cookbook will explain everything related to the MongoDB datastore, modeling datasets and indexing datasets.

As mentioned before; You will find a folder called: "Remaining-Analyses-Notebooks" and a folder called: "Data" in the same folder as this document. The notebooks found in the folder: "Remaining-Analyses-Notebooks" contains the data analyses of the remaining Crane (Tracker) datasets, GPS-Route (Trail) datasets and the World-Port-Index dataset. They also show you how to compare and analyze data sets with a different data structure.

The datasets used in the remaining Jupyter Notebooks can be found in the folder "~/GeoStack-Course/Course-Datasets".

Remember to go through these notebooks by running all the cells now! We need to have these datasets analyzed and transformed to achieve our end goal of creating a fully functional Geographical software stack.

After going through the remaining data analyses notebooks you should go back to the cookbook: "Creating the GeoStack Course VM"

3. Bibliography

1. *Computer Hope's free computer help*. (z.d.). Geraadpleegd op 02 oktober 2019, van https://www.computerhope.com/

2. *ETL Extract — ETL Database*. (z.d.). Geraadpleegd op 2 oktober 2019, van https://www.stitchdata.com/etldatabase/etl-extract/

3. *ETL Load— ETL Database*. (z.d.). Geraadpleegd op 2 oktober 2019, van https://www.stitchdata.com/etldatabase/etl-load/

4. *ETL Transform — ETL Database*. (z.d.). Geraadpleegd op 2 oktober 2019, van https://www.stitchdata.com/etldatabase/etl-transform/

5. *ETL (Extract, Transform, and Load) Process*. (2019, 30 september). Geraadpleegd op 4 oktober 2019, van https://www.guru99.com/etl-extract-load-process.html

6. Mendis, W. S. (2018, 19 juni). *From RDBMS to Key-Value Store: Data Modeling Techniques*. Geraadpleegd op 4 oktober 2019, van https://medium.com/@wishmithasmendis/from-rdbms-to-key-value-store-data-modeling-techniques-a2874906bc46

7. Shrinivaasan, S. (2015, 24 april). *What are the different types of data stores? How is each different from the others with different use cases? - Quora*. Geraadpleegd op 4 oktober 2019, van https://www.quora.com/What-are-the-different-types-of-data-stores-How-is-each-different-from-the-others-with-different-use-cases/

8. Watts, S. (2017, 7 december). *What is ETL (Extract, Transform, Load)? ETL Explained*. Geraadpleegd op 4 oktober 2019, van https://www.bmc.com/blogs/what-is-etl-extract-transform-load-etl-explained/

9. Movebank. (z.d.). *Movebank*. Geraadpleegd op 4 oktober 2019, van https://www.movebank.org

10. Yousuf, F. (2015, 24 augustus). *ETL (Extract, Transform, and Load) Process & Concept – Applied Informatics*. Geraadpleegd op 2 oktober 2019, van http://blog.appliedinformaticsinc.com/etl-extract-transform-and-load-process-concept/

11. Frieland, D. (2018, 5 oktober). *ETL vs ELT: We Posit, You Judge - IRI*. Geraadpleegd op 2 oktober 2019, van https://www.iri.com/blog/data-transformation2/etl-vs-elt-we-posit-you-judge/

12. FullStack Academy. (2016, 23 januari). *YouTube* [YouTube]. Geraadpleegd op 2 oktober 2019, van https://www.youtube.com/watch?v=4rhKKFbbYT4

13. MongoDB. (z.d.). *Operational Factors and Data Models — MongoDB Manual*. Geraadpleegd op 3 oktober 2019, van https://docs.mongodb.com/manual/core/data-model-operations/