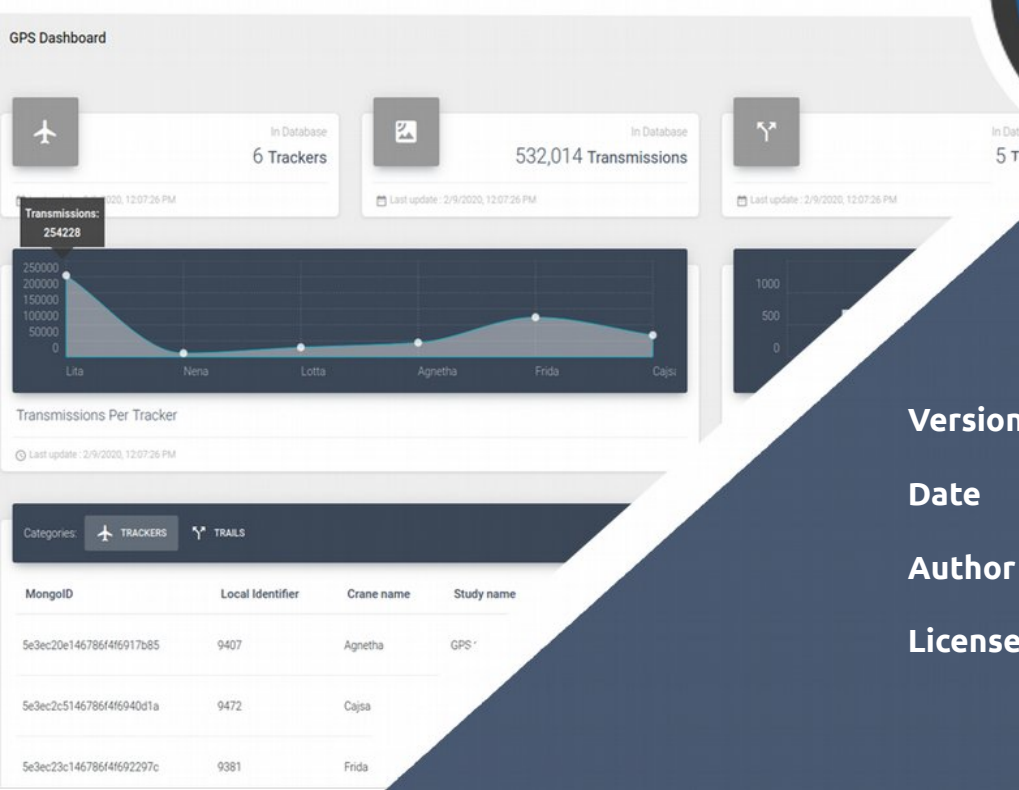
Cookbook

DATA MODELING IN MONGODB
USING MONGOENGINE



Version : 1.0

Date : 01-09-2020

Author : The GeoStack Project

Open Content License

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"

2) **Cookbook: ETL-Process with datasets:**

   In the cookbook: "ETL-Process with datasets", we learned how to analyze the dataset and filter the data that serves the needs of our application.

If you have not read these documents yet, please do so before reading this document.

The purpose of this document is to provide information related to creating MongoDB database models using MongoEngine.

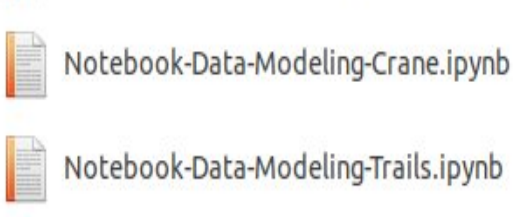
First some basic information related to MongoDB is given. This information will be useful when creating the database model yourself.

In chapter 4 we are going to create a model using the data set and the information we acquired in the the cookbook: "ETL-Process with data sets" . So if you have not read that cookbook yet, you should do so before reading this chapter.

The notebook used during cookbook can be found in the folder called: "Cookbook-Notebook" which located in the same folder as this document. This folder contains everything which is used and created during this cookbook.

You will also find a folder called: "Remaining-Data-Modeling-Notebooks". The notebooks found in this folder contain the data modeling and importing process of the remaining Crane datasets, GPS-Route datasets and the World-Port-Index dataset, which will be imported in the PostgreSQL datastore.

In the illustration below you can find the content of the Notebooks folder.



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 loaded in our databases to achieve our end goal which is creating a fully functional Geographical software stack.

Table of Contents

Table of Illustrations

1 Introduction to NoSQL data stores

In the cookbook: “ETL-Process with data sets” a small introduction to NoSQL datastores was given. During this chapter in depth detail will be given regarding NoSQL datastores.

1.1 Positioning NoSQL to Relational data stores

NoSQL databases are different than relational databases like MySql or PostgreSQL. In relational database you need to create the tables, define schema’s, set the data types of fields etc. before you can actually insert the data. In NoSQL you don’t have to worry about that. You can insert, update and remove data on the fly.

NoSQL data stores are easy to scale and are much faster in some operations that need to be performed on the database. In some cases an relational database should be used. One scenario where a relational database is useful is in a banking system. This is because a relational database follows the ACID properties (Atomicity, Consistency, Isolation and Durability).

Some major limitations you will encounter when using relation databases are mentioned below:

- ➔ In relational databases the structure and schema of the data needs to be defined before processing the data.
- ➔ These days a lot of applications store their data in JSON format. Relational databases don’t provide an easy way to perform operations such as:
  1) **Data Definition operations** - creating database, files, file groups, tables.
  2) **Data Manipulation operations** - Insert, Update, Delete data from objects.
  3) **Data Control operations** - GRANT, REVOKE.
  4) **Transaction Control Operations** – ROLLBACK,COMMIT.
  5) **Database Maintenance Operations** - BACKUP, RESTORE, REBUILD.
- ➔ Relational databases have ACID properties (Atomicity, Consistency, Isolation, Durability) to maintain. Enforcing these takes a lot of computing which slows down response times.

1.2 Relational database models Vs NoSQL

Both RDBMS and NoSQL data stores have their advantages and disadvantages. In illustration 1 you can see a graph containing the ratio between functionality and Scalability and performance.

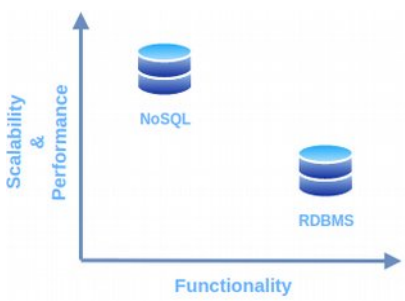


Illustration 1: Scalability VS. Performance

RDBMS: Structured data, which provides more functionality but gives less performance.
NoSQL: Structured or semi structured data, less functionality and high performance.

NoSQL data stores should be used when the following points apply to your situation.

- ✔ When you want to store and retrieve huge amount of data.
- ✔ The relationship between the data you store is not that important.
- ✔ The data is not structured and changing over time.
- ✔ Constraints and Joins support is not required at database level.
- ✔ Data grows continuously and you need to scale the database regulary to handle the data.

2 Introduction to MongoDB

MongoDB is a cross-platform, document-oriented database. It's classified as a NoSQL datastore which is also known as non-relational database. It was created by 2 people who wanted to overcome the limitations of relational databases mentioned in chapter 1.

Using MongoDB as data store has a lot of advantages.
Some of the major advantages of MongoDB are as follows:

Illustration 2: Logo MongoDB

- ➔ High Availability:

  A very useful feature found in MongoDB is the auto replication feature. This means, whenever a failure takes place in the data store, the data replicates itself to the previous data.

- ➔ High Scalability:

  MongoDB makes use of sharding. Sharding is a type of database partitioning that separates very large databases into smaller, faster, more easily managed parts called data shards.

  In illustration 3 you see a visual representation of data operations without sharding.

Illustration 3: Data processing without sharding

In illustration 4 you see a visual representation of data operations with sharding.

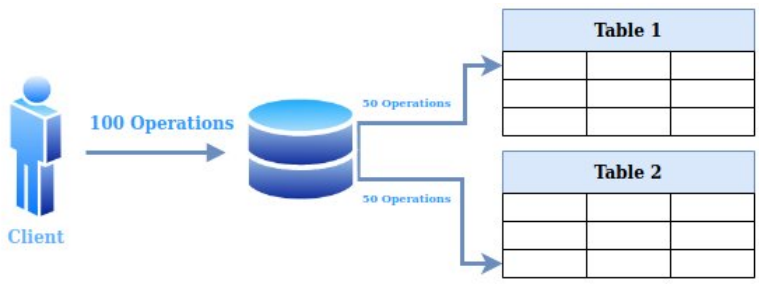


Illustration 4: Data processing with sharding

In illustration 5 a visual representation of remaining features which MongoDB offers.

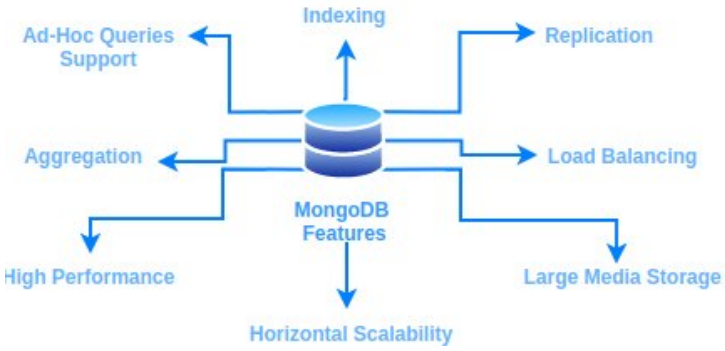
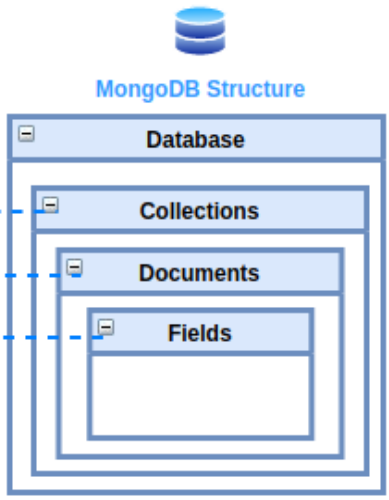


Illustration 5: MongoDB Features

2.1 MongoDB Database structure

The MongoDB database structure is similar to the structure of an relational database. The structure is as follows:

- A MongoDB Database consists of collections……
- A MongoDB Collection consists of documents……..
- A MongoDB Document consists of fields. ……..



2.2 Field, document and collection

In MongoDB an attribute of a data row is called a field. A data type is assigned to each field. In the table below, you can find an overview of all data types in MongoDB.

Double	**Array**	**Binary data**
String	**Undefined**	**Object Id**
Object	**Boolean**	**Date**
Null	**Regular Expression**	**JavaScript**
Symbol	**JavaScript with scope**	**Integer**
Min key	**Max key**	**Timestamp**

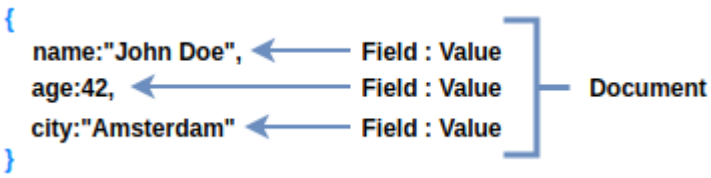


Illustration 6: A MongoDB Document example

In MongoDB a data record is called a document. A MongoDB document format is very similar to a JSON document format. MongoDB stores documents in BSON, which is the binary encoded version of JSON. Basically, the name BSON itself comes from Binary encoded JSON. A visual representation of one MongoDB document can be found in illustration 6 .

Multiple of these documents combined are called a collection. Collections are similar to Tables in RDBS (Relational database model systems).

In illustration 7 you can find a visual representation of a table in a relational database and MongoDB collection, consisting of multiple documents.

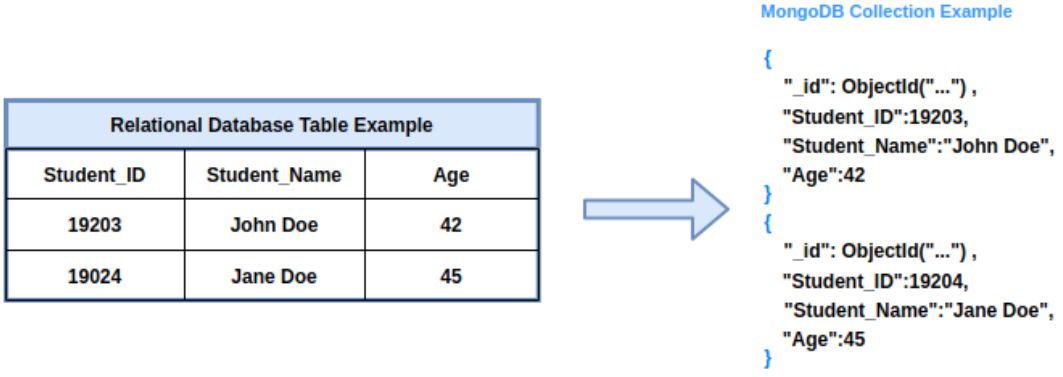


Illustration 7: Relational database table -> MongoDB Collection

3 Data modeling in MongoDB

As mentioned in Chapter 2, MongoDB is a document data store. Each record in a MongoDB collection is a document. Documents are a structure composed of file and value pairs, similar to JSON objects or other mapping data types. To model and index our database, we are going to use a python object data mapper called: “MongoEngine”.

When designing data models, always consider the application usage of the data. The key challenge that comes with data modeling in MongoDB, is balancing the requirements of the application.

The requirements to balance are as follows:

- ➔ the amount of data and the natural structure of the data.
- ➔ the queries and modifications your application will perform on the data, including the frequency of these operations and their performance and isolation requirements.
- ➔ the performance characteristics of MongoDB. An example of this is the maximum size of an document, which is 16MB.

Other aspects that should be carefully thought out are as follows:

- ➔ Design schema according to the need.
- ➔ Objects which are queried together should be contained in one document.
- ➔ Consider the frequent use cases.

As a general rule, if you have a lot of child documents or if the child documents are large, a separate collection might be best. Smaller and/or fewer documents tend to be a natural fit for embedding.

In illustration 8 you can see a visual representation of embedding documents vs referencing documents.

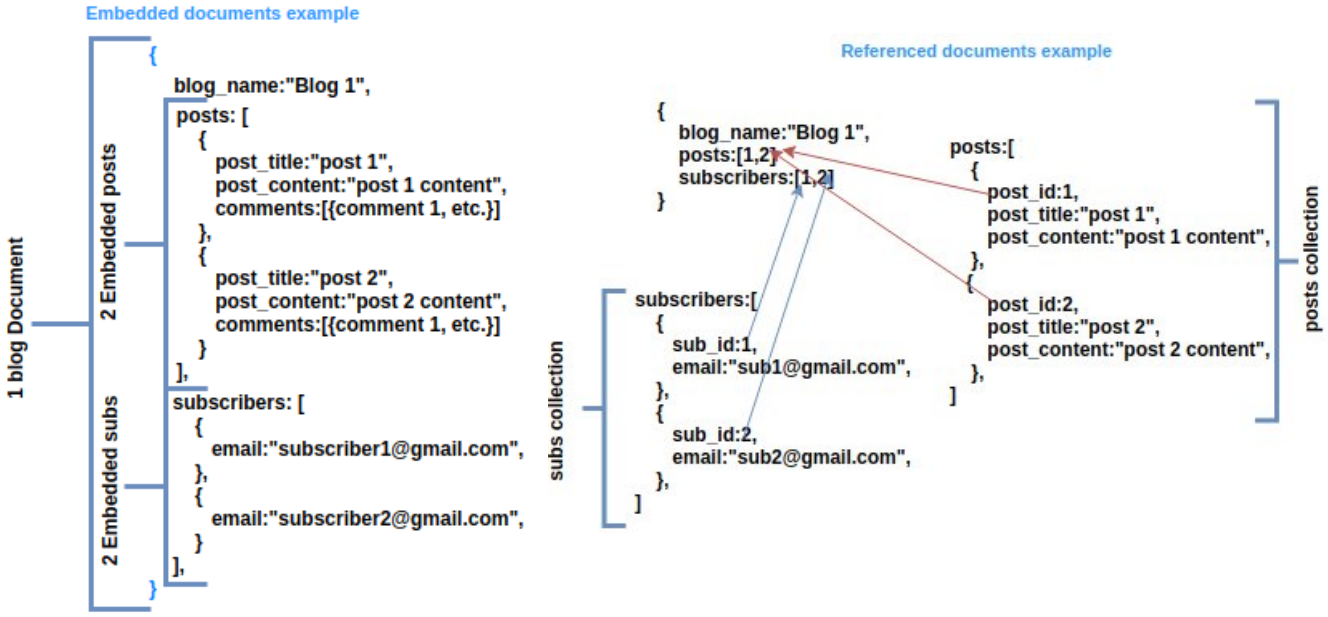


Illustration 8: Embedded VS Referenced Documents

3.1 Referenced Documents example

This type of data modeling is more similar to relational database modeling. References store the relationships between data by including links or references from one document to another. Applications can resolve these references to access the related data. In illustration 9, you can see an example of a referenced document. In this case we have 3 collections in our database:

1. Blogs, which contains the documents representing blogs.
2. Posts, which contains the documents representing the posts.
3. Subscribers, which contains the documents represent the subscribers.

The blog documents contain references to the post(s), belonging to that blog, in the posts collection. It also contains references to the users subscribed to that blog.

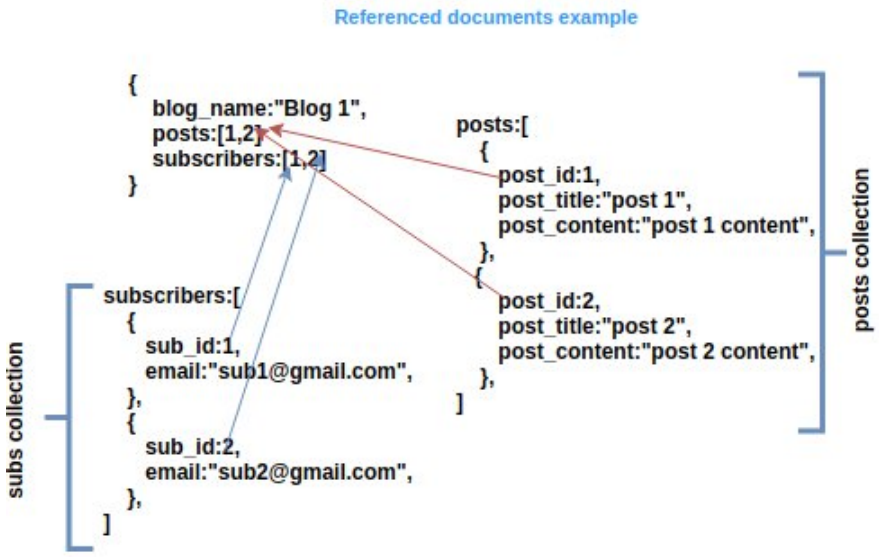


Illustration 9: Referenced documents

A few cases of when you should use referenced documents are as follows:

- ➔ When modeling many to many relationships
- ➔ When you have sub-documents that become bigger overtime
- ➔ When you have large sub-document information (16mb is the limit)
- ➔ Consistency of the data is a priority.
- ➔ You want to ensure your cache is used efficiently.
- ➔ The embedded version would be unwieldy.

Some advantages of a referenced model are as follows:

- ✔ Consistency
- ✔ Queryability: When we query the data set we can be more sure that the results are the canonical versions of the data rather than embedded copies.
- ✔ Better cache usage
- ✔ More efficient hardware usage: embedding data gives us larger documents with multiple copies of the same data but referring helps reduce the disk and RAM our database needs.

3.2 Embedded documents example

With embedding, a document contains multiple sub-documents. In illustration 10, can see an pseudo representation of a document that uses embedding. In this illustration the parent document is a blog with the name: “Blog 1”. The blog contains 2 embedded posts. Each post is called a sub-document.

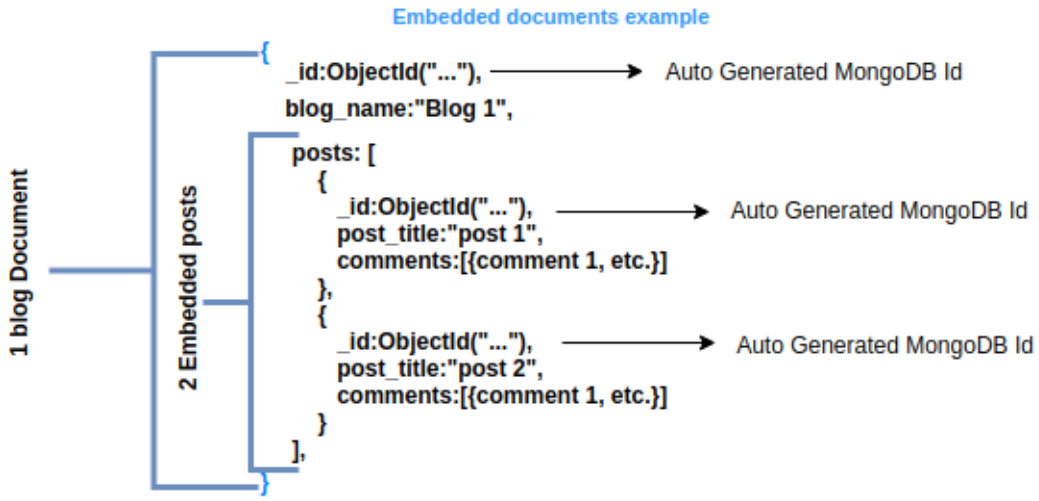


Illustration 10: Embedded Document

Embedded documents capture relationships between data by storing related data in a single document structure. MongoDB documents make it possible to embed document structures in a field or array within a document.

A few cases of when u should use Embedded documents are as follows:

- ➔ 1 to one relationships between entities
- ➔ 1 to many relationships between entities
- ➔ Sub-documents don’t grow overtime
- ➔ Sub-documents do not contain a lot of data
- ➔ Reads greatly outnumber writes.
- ➔ You’re comfortable with the slim risk of inconsistent data across the multiple copies.
- ➔ You’re optimizing for speed of access.

Some advantages of a embedded model are as follows:

- ✔ Speed of access: embedding everything in one document means we need just one database look-up.
- ✔ Potentially greater fault tolerance at read time: in a distributed database our referred documents would live on multiple machines, so by embedding we’re introducing fewer opportunities for something to go wrong and we’re simplifying the application side.

3.3 Embedded VS. Referenced: Crane dataset

Before we start creating the correct datamodel for our Crane (Tracker) dataset an example is discussed on the 2 ways to go about modeling the dataset. We know we that we are going to have either 1 or 2 collections in our database.

When we are going to create an embedded model we will have a collection containing all documents representing a Crane. In each document, which represents a Crane, we have an array called points (Transmissions). This array contains the embedded Transmissions (called points in the illustration below) belonging to the Crane in the Crane Document.

In the example below we will only have 1 collection in our database. This collection will contain the (meta)data related to a Crane and all the transmissions. When creating a model using this method, the collection size will be very big after adding a lot of Cranes and Transmissions.

Illustration 11: Embedded crane data

As you can see in the illustration above, we have 2 different Cranes in our database. Each crane contains an array called:”points”, which contains 2 transmissions belonging to that crane.

If we are going to model our database using a referenced model we will end up with 2 collections in our database. These collections are as follows:

1. A collection containing all documents representing a Crane (Crane name, Crane ID etc.). This can be seen as the collection that contains the **STATIC data (Values that stay the same).**
2. A collection containing all transmissions (called points in illustration 12). This can be seen as the collection that contains all the **DYNAMIC data (Value that change).**

Illustration 12: Referenced crane data

As you can see in illustration 12, we have 2 separate collections. The Bird collection (Crane collection) contains 2 different cranes. This Point collection (Transmission collection) contains 4 different collections. Each Crane has 2 transmissions. The tag-local-identifier fields in the Point documents are a reference to the tag-id in the document to which the points belong.

3.4 Indexing models

Indexes can be seen as separate collections containing only the data on which the index was created.

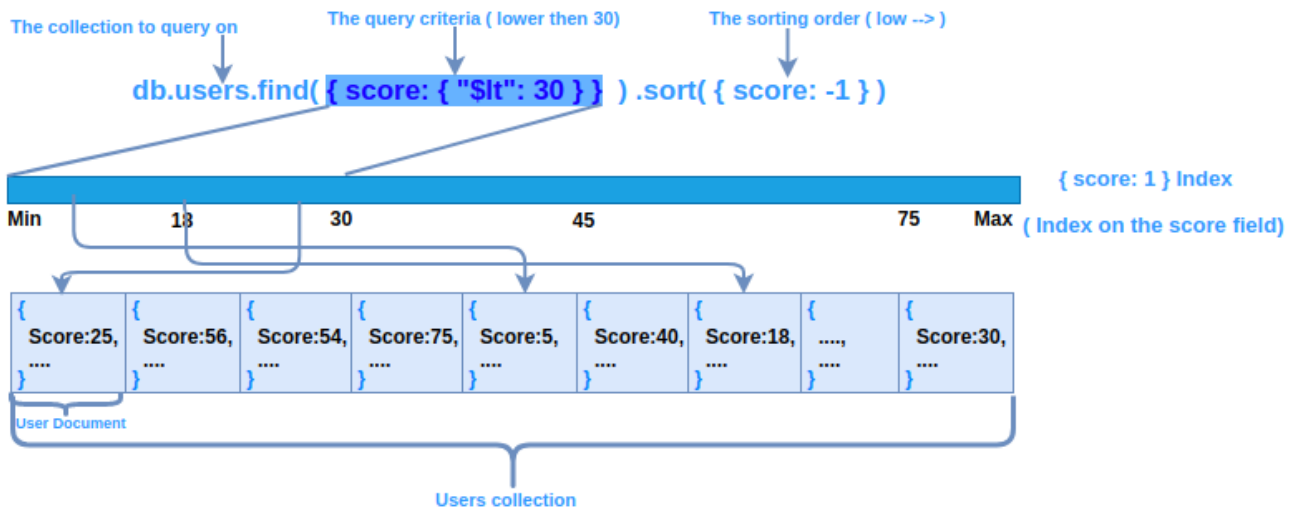
When indexing data you should consider the use-cases of your application. When creating a lot of indexes there is a possibility you slow down your application.

Indexes support the efficient execution of queries in MongoDB. Without indexes, MongoDB must perform a collection scan, which means it has to scan every document in a collection, to select those documents that match the query statement. If an appropriate index exists for a query, MongoDB can use the index to limit the number of documents it must inspect.

As mentioned above, indexes are special data structures that store a small portion of the collection's data set in an easy to traverse form. The index stores the value of a specific field or set of fields, ordered by the value of the field. The ordering of the index entries supports efficient equality matches and range-based query operations. In addition, MongoDB can return sorted results by using the ordering in the index.

Indexes are special data structures that store a small portion of the collection's data set in an easy to traverse form. The index stores the value of a specific field or set of fields, ordered by the value of the field. The ordering of the index entries supports efficient equality matches and range-based query operations. In addition, MongoDB can return sorted results by using the ordering in the index.

The following diagram illustrates a query that selects and orders the matching documents using an index :



In section 4.5 of this document and example is given on how to create indexes yourself. Indexing data will come in very useful when visualizing the data in the GeoStack web applications since it will speed up the data retrieving process significantly.

4 Data modeling: Crane dataset

In this chapter we are going to create a data model for dataset used in the cookbook:"ETL-Process with datasets". We are going to start of by importing and validating the dataset.

After we confirmed that the data is correct, we are going to create the model. When our model is done, we will create and run a function which loads the data according to that model. When all of that is done we are going to create some basic queries before indexing the database. Then we are going to create indexes on the database. Finally we will query the indexed database and compare the results of an unindexed database to an indexed database.

We want to start of by importing the modules required to perform this process. So let's create a new Jupyter Notebook and add the lines shown in illustration 13 in the first cell.

```
[2]: import pandas as pd # Pandas is used to read the JSON dataset.
     from mongoengine import * # MongoEngine is used to model, import and index datasets.
     from datetime import datetime # The Python datetime module is used to convert timestamps.
```

Illustration 13: Import the required modules

4.1 Reading and validating the Data

Now that we have imported the required modules we want to read the dataset, which we are going to model, in a Pandas dataframe. How this is done is shown in illustration 14.

```
[2]: SW_Crane_Frida = pd.read_json('../../../Course-Datasets/JSON/Crane_JSON/Frida_SW.json')
```

Illustration 14: Read the dataset in a DataFrame

To be sure we loaded the correct data we are going to print the first row of the dataframe. How this is done is shown in illustration 15.

```
[4]: SW_Crane_Frida[:1] # Print the first row of the dataframe.
```

[4]:

	event-id	study-name	timestamp	visible	ground-speed
0	1154727247	GPS telemetry of Common Cranes, Sweden	2013-07-21 03:06:32	True	0.0

Illustration 15: Validate the dataset

Now we want to connect to a database using MongoEngine. The code for this is shown in illustration 16. The database we are going to connect to is called : "Crane_Database".

MongoEngine will create this database automatically if it does not exists yet. To connect to a database we use the code shown in illustration 16.

```
[5]: connect('Crane_Database') # Connect to the Crane_Database.

[5]: MongoClient(host=['localhost:27017'], document_class=dict, tz_aware=False, connect=True, read_preference=Primary())
```

Illustration 16: Connect to the database

As you can see in illustration 16, we are now connected to a database called "Crane_Database", which is running on port 27017 which default port of MongoDB.

4.2 Creating the model

Now that we are connected to our database, we can start creating the model. First we want to split the **DYNAMIC (Value that change)** and **STATIC (Values that stay the same)** data in our dataset. We do this because, we only want to add the static data once.

The static data (Tracker data) is as follows:

Column name	Description
Study-name	The name of the study.
individual-taxon-canonical-name	The Latin name of the Crane.
individual-local-identifier	The ID of the tracker.

The dynamic data (Transmission data) is as follows:

Column name in dataset	Description
event-id	The ID of the transmission.
timestamp	The datetime of when the transmission was send..
location-lat	Latitude location of the crane.
location-long	Longitude location of the crane.
height-above-ellipsoid	The height of the crane.
ground-speed	Speed of the Crane.
heading	The direction of the Crane.
visible	Checks whether the tracker is still sending transmissions.
sensor-type	The type of sensor attached to the tracker.
tag-voltage	The amount of voltage running trough the tracker.

We can add the static data to a document called Tracker, because it contains basic and Static information about the study, the tracker and the Drane. We also want to add some data ourselves because some important data is not provided in the dataset. This data is static and will be created during the process of loading the data in MongoDB. The data is as follows:

Data to add	How data is obtained
The date when the study started.	This data is extracted from the timestamp of the first transmission
The date when the study ended.	This data is extracted from the timestamp of the last transmission
The name of the Crane.	We are going to add the name when we import the dataset.
The amount of transmissions belonging to the tracker.	We are going to assign the length of the dataset (amount of rows) to this field.

There are a couple of other things that became clear while analyzing the dataset:

- ✔ Each Crane has only 1 tracker.
- ✔ A tracker has a lot of transmissions, the amount of transmissions will keep growing overtime.
- ✔ Each transmission belongs to one tracker.

Because the amount of transmissions per tracker is going to grow overtime and the relationship between the documents is one to many, we should reference the tracker in the transmission document. If we don't do this, we will end up with a huge list of transmissions in the tracker document. MongoDB will automatically create an _id field on the documents. The value of the _id field in the tracker document will be the value of the reference field in the transmission document.

Because, there are a lot of fields in the transmission document, we should split the fields into smaller documents, which will be embedded in a transmission.

Because, the size of the sub-documents will not change overtime and the relationship between these sub-documents and the transmission document is one to one, we can embed the sub-documents in the transmission document.

The structure of the 2 main documents is as follows:

1) **Tracker document:** Contains data related to the tracker. The data is as follows:

Column name in dataset	Description	Field name	Field type
Study-name	The name of the study	study_name	String
individual-taxon-canonical-name	The Latin name of the Crane	individual_taxon_canonical_name	String
individual-local-identifier	The ID of the tracker	individual_local_identifier	Int

The tracker document will also contain the data we are going to add manually when loading the data in MongoDB. This data is as follows:

Data	Field name	Field type
The date when the study started.	start_date	DateTimeField()
The date when the study ended.	end_date	DateTimeField()
The name of the Crane.	name	StringField()
The amount of transmissions belonging to the tracker.	transmission_Count	IntField()

2) **Transmission document:** Contains data related to the transmission. Because one transmission contains a lot of data, it's best to split this document into smaller sub-documents. Because the sub-documents will not grow overtime, they can be embedded in the transmission document. As mentioned above, we also want to create a reference to the tracker, to which the transmission belongs. The data located in the transmission document is as follows:

Column name in dataset	Description	Field name	Field type
event-id	The ID of the transmission	event_id	IntField()
timestamp	The datetime of when the transmission was send.	timestamp	DateTimeField()

The sub-documents which will be embedded in the transmission document are as follows:

1. **Geometry document**, which contains data related to the geometry. The data is as follows:

Column name in dataset	Description	Field name	Field Type
location-lat, location-long	Location of the crane, We are going to load this as an array.	coordinates	PointField()
height-above-ellipsoid	The height of the crane	alt	IntField()

2. **Speed document**, which contains data related to the speed and direction. The data is as follows:

Column name in dataset	Description	Field name	Field Type
ground-speed	Speed of the Crane	speed	FloatField()
heading	The direction of the Crane	heading	IntField()

3. **TrackerMetadata document**, which contains the data that is send with a transmission but is related to the tracker. The data is as follows:

Column name in dataset	Description	Field name	Field Type
visible	Checks whether the tracker is still sending transmissions.	visible	BooleanField()
sensor-type	The type of sensor attached to the tracker.	sensor_type	StringField()
tag-voltage	The amount of voltage running trough the tracker.	tag_voltage	FloatField()

To embed the sub-documents and reference the tracker document, we need to add 4 more fields to our transmission document. The fields are as follows:

Field to add	Description	Field type
geometry	A reference to the Geometry document.	EmbeddedDocumentField(Geometry)
speed	A reference to the Speed document.	EmbeddedDocumentField(Speed)
metadata	A reference to the TrackerMetadata document.	EmbeddedDocumentField(TrackerMetadata)
tracker	A reference to the tracker, to which the transmission belongs.	ReferenceField(Tracker)

A diagram, created using the information above, is shown in illustration 17.

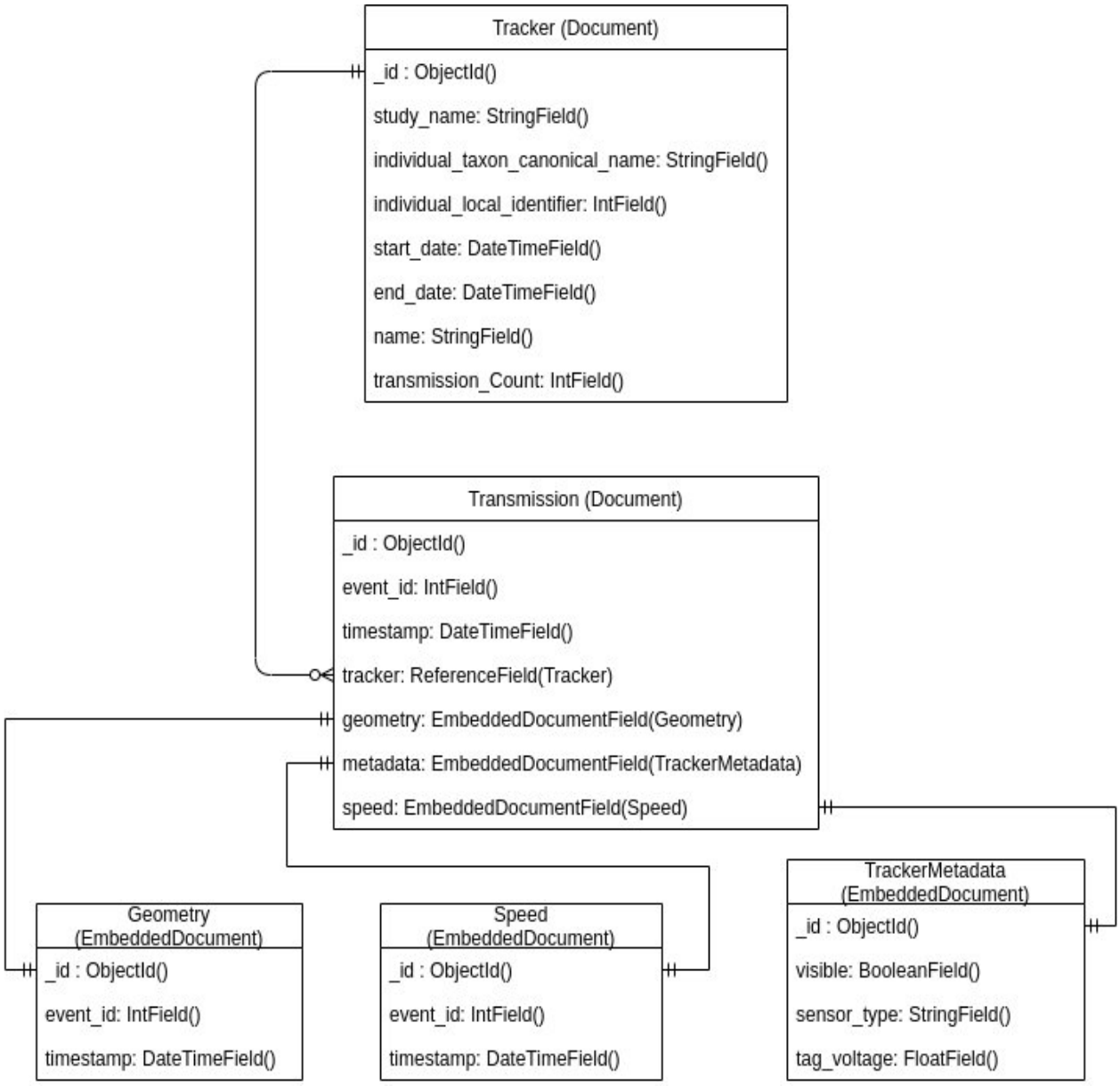


Illustration 17: Database model diagram

Now that we know what our database structure is going to be, we can start coding the model. We are going to use Python with MongoEngine to create that database model. We are going to start of by creating the Tracker Document structure. How this is done is shown in illustration 18 .

```
[6]: # Creating the Tracker document
class Tracker(Document):

    # Name of the study
    study_name = StringField()

    # Name of the bird, in latin.
    individual_taxon_canonical_name = StringField()

    # Id of the Crane.
    individual_local_identifier = IntField()

    #Start date of the study.
    start_date = DateTimeField()

    #End date of the study.
    end_date = DateTimeField()

    #Name of the Crane
    name = StringField()

    #Amount of the transmissions related to the tracker.
    transmission_Count= IntField()
```

Illustration 18: Coding the tracker document

Next we are going to create the sub-documents, which will be embedded in the transmission document. First we create the Geometry document. How this is done is shown in illustration 19.

```
[6]: # Creating the Geometry document
class Geometry(EmbeddedDocument):

    # coordinates of transmission (coord=[1,2])
    coord = PointField()

    # altitude of tansmission
    alt = FloatField()
```

Illustration 19: Coding the geometry document

Now let's create the speed document. How this is done, is shown in illustration 20.

```
[6]: # Creating the Speed document
class Speed(EmbeddedDocument):

    # Speed of the Crane
    ground_speed = FloatField()

    # Heading of the Crane in degrees
    heading = IntField()
```

Illustration 20: Coding the speed document

The last embedded document which are going to create is the TrackerMetadata document. How this is done, is shown in illustration 21.

```
[6]: # Creating the TrackerMetadata document
     class TrackerMetadata(EmbeddedDocument):

         #Is the tracker still visible or not?
         visible = BooleanField()

         # Type of sensor used in tracker.
         sensor_type = StringField()

         # Voltage level of the tracker.
         tag_voltage = FloatField()
```

Illustration 21: Coding the TrackerMetadata document

The last document we are going to create is the Transmission document. In this document we will add the reference to the tracker and embed the sub-documents. How this is done is shown in illustration 22.

```
[6]: # Creating the Transmission document
     class Transmission(Document):

         # Identifier of the transmission
         event_id = IntField()

         # Timestamp of when transmission was send
         timestamp = DateTimeField()

         # Embedded geometry of transmission
         # When loading the data, an instance of a geometry document
         # will be passed as value for the geometry field.
         geometry = EmbeddedDocumentField(Geometry)

         # Embedded speed related data of transmission
         # When loading the data, an instance of a speed document
         # will be passed as value for the speed field.
         speed = EmbeddedDocumentField(Speed)

         # Embedded metadata of transmission
         # When loading the data, an instance of a TrackerMetadata document
         # will be passed as value for the metadata field.
         metadata = EmbeddedDocumentField(TrackerMetadata)

         # Reference to the tracker the transmission belongs to.
         # When loading the data, an instance of a Tracker document
         # will be passed as value for the tracker field.
         tracker = ReferenceField(Tracker)
```

Illustration 22: Coding the transmission document

4.3 Loading the data using the model

Now that our model is created, we can start importing our data set in our MongoDB data store. To be able to do this with multiple data sets we are going to create a function in which we can pass a data frame to which we assigned the data set we want to import.

The function we are going to create is called: “load_data”. In the function we want to pass a dataframe, the name of the crane and the country the crane belongs to. How this is done is shown in illustration 23.

```
[9]: # Here we create the function which we will use to load the datasets.
     # The function takes the following input parameters:
     # 1) A Dataframe to load
     # 2) The name of the Crane
     # 3) The country of the Crane. This will come in usefull when loading
     #    different types of Cranes (German or Swedish) since, depending on
     #    the country of origin, the elavation column name differs.
     def load_data(df,name,country):
```

Illustration 23: Create function

First we want to create the metadata of our tracker. This is the data not provided by our data set. The metadata we want to create is shown in the table below:

Data to add	How data is obtained
The date when the study started.	This data is extracted from the timestamp of the first transmission
The date when the study ended.	This data is extracted from the timestamp of the last transmission
The name of the Crane.	We are going to add the name when we import the dataset.
The amount of transmissions belonging to the tracker.	We are going to assign the length of the dataset (amount of rows) to this field

To obtain the date of the first transmission, we need to extract the value of the timestamp column from the first row in our dataframe. We are going to use the .at() function provided by pandas to select a column at a certain index. In this case it’s the column ”timestamp” at index 0. The code for this is shown in the illustration below.

```
# Here we create the start date.
start_Date = df.at[0,'timestamp']
```

Illustration 24: Get start date

Now we want to do the same for the end date. To obtain this information, we are going to extract the value of the timestamp column from the last row of our dataframe. We are going to do the same as above to obtain this data. But in this case we need the index of the last row in the dataframe.

To get the index of the last row, we are going to use the .shape() method provided by pandas. This method returns the length of the dataframe. We need to substract 1 from the length of the dataframe, otherwise our index will be out of bounds. How this is done is shown in illustration 25.

```
# Here we create the end date.
end_Date = df.at[df.shape[0]-1,'timestamp']
```

Illustration 25: Get end date

Now we want to obtain the total transmission count belonging to the tracker. To obtain this information, we are simply going to assign the length of the dataframe to a variable called transmission_Count. How this is done is shown in illustration 26.

```
# Here we create the amount of transmissions belonging to the tracker.
transmission_Count = df.shape[0]
```

Illustration 26: Get transmission count

Now that we have created our own metadata we want to create the full tracker document using data from the dataset and the data we created ourselves.
Creating the tracker document is only done once per dataset. We do this because the tracker document only contains static data that will not change.

To be able to reference the tracker in our transmission documents, we need to assign it to a variable called: “tracker”. This variable will be passed to the transmission document later on.

The data, obtained from our dataset, that belongs to the tracker document, is shown in the table below:

Column name in dataset	Description
Study-name	The name of the study
individual-taxon-canonical-name	The Latin name of the Crane
individual-local-identifier	The ID of the tracker

Since this data is the same in every row from our dataset, we can obtain it by using the same method as obtaining the start date.

The code to create the tracker document is shown in illustration 27 .

```
#Create a new tracker, this is only done once
tracker = Tracker(study_name = df.at[0,'study-name'],
                  individual_taxon_canonical_name = df.at[0,'individual-taxon-canonical-name'],
                  individual_local_identifier = df.at[0,'individual-local-identifier'],
                  start_date = start_Date,
                  end_date = end_Date,
                  name = name,
                  transmission_Count = transmission_Count).save()
```

Illustration 27: Create the tracker document

As you can see in the illustration above, we added a .save() method. This method is an inbuilt method in MongoEngine which saves the tracker document to the database.

Now we want to add the transmissions belonging to the tracker. We could use the .save( ) method to immediately save a transmission document after it’s creation. The problem with doing it this way, is that it takes a long time to save all the transmissions.

MongoDB has a feature called bulk insert. This feature inserts all the created transmission documents at once and speeds up the process of importing data immensely. To be able to use the bulk insert feature we need a list containing all our transmissions.
So lets create an empty list called: “transmissions”. How this is done is shown in the illustration below.

```
# Create an empty list of transmissions to which will append the new transmissions
# after they have been created. This list will be passed to the mongodb bulk insert feature.
transmissions = []
```

Illustration 28: Create transmission list

Next we want to receive a notification when our load_data() function starts appending the transmissions to the list we created above. The code for this is shown in illustration 29.

```
# Print when list appending process starts.
print('Start appending transmissions to list from: ' + str(name) )
```

Illustration 29: Create notification

Next we want to create a transmission document for each row in the dataframe. We also want to create the sub-documents which will be embedded in the transmission document. To create a transmission document for each row, we need to create a for loop which loops trough our dataframe. To loop through a dataframe we use the .iterrows() function.

The iterrows function is a generator that iterates over the rows of the dataframe and returns the index of each row, in addition to an object containing the row itself. To obtain required data from the row object, in our for loop, we can use: row['column name']. How to create the for loop, is shown in illustration 30.

```
# For each row in the dataframe the following code is executed.
for index,row in df.iterrows():
```

Illustration 30: Loop through dataframe

First we want to create the sub-documents which will be embedded in our transmission document. We have 3 sub documents: A geometry document, a speed document and a metadata document. These documents will be embedded in the transmission document.

Let's start with the geometry document. The geometry document has 2 fields:

- ➔ coord, which is an array of the latitude an longitude coordinates
- ➔ alt , which is the altitude.

We are going to assign the geometry document to a variable called geometry. This variable will be passed when creating the transmission document. The code required to create the geometry document is shown in illustration. 31

```
# Create geometry document for Swedish sets in which we pass the required values.
# NOTE: To use Geometry queries you have to insert te longitude value first.
geometry = Geometry(coord = [row['location-long'],row['location-lat']],
                    alt = row['height-above-ellipsoid'])
```

Illustration 31: Create geometry document

Now let's create the trackermetadata document. How this is done, is shown in illustration 32.

```
# Create the metadata document in which we pass the required values.
metadata = TrackerMetadata(visible = row['visible'],
                           sensor_type = row['sensor-type'],
                           tag_voltage = row['tag-voltage'])
```

Illustration 32: Create metadata document

Now we have our trackermetadata document, we can create the last sub-document. This is the speed document. How this is done is shown in illustration 33.

```
# Create the speed document in which we pass the required values.
speed = Speed(ground_speed = row['ground-speed'])
```

Illustration 33: Create the speed document

Now that we have our sub-documents, we can start with creating the transmission document. As mentioned above, we want to append all the transmission documents to the empty transmission list we created earlier. To append items to a list, we use the method .append(). We are also going to pass the sub-documents we created earlier in the transmission document.

The code for the creation and the appending of the transmission documents is shown in illustration 34.

```
# Create transmission document and append them to the transmissions list.
transmissions.append(Transmission(event_id = row['event-id'],
                                  timestamp = row['timestamp'],
                                  geometry = geometry,speed = speed,
                                  metadata = metadata,tracker = tracker))
```

Illustration 34: Create transmission document

Now we want the function to notify use when it's done appending the transmission documents to the list and when it's starting to import the list using the bulk insert feature. How this is done is shown in illustration 35.

```
# Print when list appending is done.
print('Bulk inserting: '+ str(transmission_Count) + ' transmissions from: ' + str(name) )
```

Illustration 35: Start bulk insert

Now we want to save the transmission list to our database. To load data in the database we are going to use the line: {collection name}.objects.insert({list name}, load_bulk = True)

Some explanation concerning the code in illustration 36 is as follows:

- ➔ **Collection name** is the name of the collection you want to save the data to (Transmission in our case).
- ➔ **List name** is the name of the list of data you want to insert. (transmissions in our case).
- ➔ The **load_bulk** parameter indicates whether you want to use the bulk insert feature. (True in our case)

```
# Bulk insert the populated transmissions list
Transmission.objects.insert(transmissions,load_bulk=True)
```

Illustration 36: Bulk insert list

We also want the function to notify use when it's done inserting the data. How this is done is shown in illustration 37.

```
# Print if insert is succesfull
print("Done inserting "+ str(len(df.index)) + " transmissions")
```

Illustration 37: Notify when done

Now that we have created our function, we want to actually import our data in the database. To load the data we are going to call the load_data() function and pass the dataframe, name of the crane and the country it belongs to as parameters in the function. How this is done is shown in illustration 38.

```
[*]: load_data(SW_Crane_Frida,"Frida","sw") # Load the Crane Dataframe using the load_data function.
```

Illustration 38: Load Data

The output of this function should the same as shown in illustration 39.

```
Start appending transmissions to list from: Frida
Bulk inserting: 123805 transmissions from: Frida
Done inserting 123805 transmissions
```

Illustration 39: Output load data function

4.4 Querying the data (Pre-Indexing)

Now that we have imported the data we are going to query the data. The reason we are going to do this before indexing, is to test whether the data has been imported correctly and to compare the speed of our non indexed database to the speed of our indexed database.

For simplicity, we are going to use MongoEngine to query the data. In our application we will use Pymongo to query the data since this is a lot faster. To create a query using MongoEngine, we use the following syntax: "{collection name}.objects({statements to query on}).only({fields to return}).to_json()"

Some explanation concerning the code is as follows:

- ➔ **collection name** is the name of the collection you to perform the query on.
- ➔ **Statements to query on** are the statements on which the query will search / filter.
- ➔ **Fields to return,** are the fields you want to display the data from.
- ➔ The **to_json()** method is used to convert the object(s) returned by MongoEngine to valid JSON

Let's start of with a simple query which returns all transmissions belonging to the crane : Frida.

To be able to do this we first need the ObjectID of the tracker belonging to that crane. To obtain this data, we will run the query shown in illustration 40.

```
In [9]: #Query to find ID of crane Frida
        Tracker.objects(name = 'Frida').only('name','id').to_json()

Out[9]: '[{"_id": {"$oid": "5de04102b54094744cf72be1"}, "name": "Frida"}]'
```

Illustration 40: Obtain ID query

Some explanation concerning the code is as follows:

- ➔ collection we want to query on is the Tracker collection.
- ➔ The statement we want to query on is: Where name is equal to Frida.
- ➔ The fields we want to return are the name and the object id.
- ➔ The to_json() method is used to convert the object returned by MongoEngine to valid JSON

Now that we know the id of our tracker we can create a query which displays all transmissions belonging to that tracker. How this is done is shown in illustration 41.

```
[21]: #Qurey to return first 10 items related to Crane:"Frida"
      Transmission.objects(tracker = '5e8f0e750b0a0aab0461a0e0')[:10].to_json()

[21]: '[{"_id": {"$oid": "5e8f0eb50b0a0aab0461a0e1"}, "event_id": 1154727247, "timest
      eed": 0.0}, "metadata": {"visible": true, "sensor_type": "gps", "tag_voltage":
      te": 1374378694000}, "geometry": {"coord": {"type": "Point", "coordinates": [13
      e": 4100.0}, "tracker": {"$oid": "5e8f0e750b0a0aab0461a0e0"}}, {"_id": {"$oid":
```

Illustration 41: Obtain transmissions query

Some explanation concerning the code is as follows:

- ➔ collection we want to query on is the Transmission collection.
- ➔ The statement we want to query on is: Where name is tracker is equal to the tracker ID we obtained form our previous query.
- ➔ The [:10] represents the amount of records we want to return. In this case 10.
- ➔ The to_json() method is used to convert the object returned by MongoEngine to valid JSON

Now that we know that we have successfully imported our data, we can start speeding up the database. To see how long it took for a query to execute, we can add the .explain() method to the query. How this is done, is shown in illustration 42.

```
#Query to check executing speed of transmissions related to Frida
Transmission.objects(tracker='5de04102b54094744cf72be1').explain()
```

Illustration 42: Explain executing stats query

If we execute the code we will be greeted with a long list of information. This information is shown in illustration 43.

Illustration 43: Query explain results

Some useful information in the illustration above is encircled in red. Some explanation concerning the output is as follows:

- ➔ The query returned 123805 records.
- ➔ The query used a collection scan.
- ➔ It took 384 milliseconds to return 123805 records

In the next chapter we are going to index the database and compare the execution statistics.

4.5 Indexing the database

To properly index the data in the database, we need to clarify what data we will be queried regularly in our application. You can add indexes to the database whenever you want.

The indexes we need on our database are as follows:

- ➔ **2D Sphere index**
  This index will be used to query the coordinates of the crane. This index was automatically created by MongoEngine when assigning the PointField() type to the coord field in our geometry document
- ➔ **2D index**
  We need this index when using the MongoDB $box geometry query.
- ➔ **timestamp index**
  We need an index on the timestamp field because, we will be querying on date time groups often in our application.
- ➔ **tracker index (in the transmission collection)**
  We need this index because we will often query on the tracker field in the transmission document to find transmissions belonging to a certain tracker.

there are 2 ways to create indexes.

1. **Create an index when modeling the data.**
   to create an index while creating the data model, we have to add a meta field to the document we want to create an index on. For example: If we want to create an index on the altitude field in the geometry document, we add the following meta field to our geometry document:

   meta = {
   'collection': 'altitude',
   'indexes': [{'fields': ['alt']}]
   }

   In illustration 44, is shown how to add an index while creating the database model.

```
# Creating the Geometry document with an index on the altitude field
class Geometry(EmbeddedDocument):

    # coordinates of transmission coord=[1,2]
    # PointField automaticly adds 2dspehere index
    # Need to add 2d index manually
    coord = PointField()

    # altitude of tansmission
    alt = FloatField()

    meta = {
        'collection': 'altitude',
        'indexes': [
          {'fields': ['alt']}
        ]
    }
```

Illustration 44: Creating an index while modeling

2. **Create indexes after creating the model**
   We can also create the indexes after we created the datamodel. We are going to use this way to create indexes below. For example: if we want to create an index on the altitude field after creating the data model we would run the following command: Transmission.create_index(("geometry.alt"))

Now that we know what indexes we need we can start creating the indexes. As mentioned above, the 2DSphere index on our coord field was already created when loading our data by assigning the type PointField() to our coord field in our Geometry document. This index is used for all the geometry queries provided by MongoDB, except for the geometry queries using box. In illustration 45 you can see the code that we used to create the 2DSphere index.

```
# coordinates of transmission (coord=[1,2])
coord = PointField()
```

Illustration 45: 2DSphere index

Let's create the 2D index on the coordinates field. This index is needed to be able to query data between a predefined box. How this is done is shown in illustration 46 .

```
In [14]: # Create an 2D index on the coordinates field in the transmission collection
         Transmission.create_index([("geometry.coord.coordinates","2d")])
Out[14]: u'geometry.coord.coordinates_2d'
```

Illustration 46: Create an 2D index on the coordinates fields

Next, we want to create an index on the tracker field in the transmission documents. How this is done is shown in illustration 47.

```
In [12]: # Create an index on the tracker field in the transmission collection
         Transmission.create_index(("tracker"))
Out[12]: u'tracker_1'
```

Illustration 47: Create an index on the tracker fields

Finally, we want to create and index on the timestamp fields in our transmission documents. How this is done is shown in illustration 48.

```
In [83]: # Create an index on the timestamp field in the transmission collection
         Transmission.create_index(("timestamp"))
Out[83]: u'timestamp_1'
```

Illustration 48: Create an index on the timestamp fields

4.6 Querying the data (Post-Indexing)

Now lets perform the same query as in chapter 4.4, but this time with an indexed database. So let's create the query again. How this is done is shown in illustration 49.

```
#Query to check executing speed after indexing
Transmission.objects(tracker='5de04102b54094744cf72be1').explain()
```

Illustration 49: Query after indexing

A part of the output of the code above is shown in illustration 50.

```
{u'executionStats': {u'allPlansExecution': [],
  u'executionStages': {u'advanced': 123805,
   u'alreadyHasObj': 0,
   u'docsExamined': 123805,
   u'executionTimeMillisEstimate': 190,
   u'inputStage': {u'advanced': 123805,
    u'direction': u'forward',
    u'dupsDropped': 0,
    u'executionTimeMillisEstimate': 50,
    u'indexBounds': {u'tracker': [u'[ObjectId('5de04102b54094744cf72be1')
1')]"]},
    u'indexName': u'tracker_1',
    u'indexVersion': 2,
    u'invalidates': 0,
    u'isEOF': 1,
    u'isMultiKey': False,
    u'isPartial': False,
    u'isSparse': False,
    u'isUnique': False,
    u'keyPattern': {u'tracker': 1},
    u'keysExamined': 123805,
    u'multiKeyPaths': {u'tracker': []},
    u'nReturned': 123805,
    u'needTime': 0,
    u'needYield': 0,
    u'restoreState': 967,
    u'saveState': 967,
    u'seeks': 1,
    u'seenInvalidated': 0,
    u'stage': u'IXSCAN',
    u'works': 123806},
   u'invalidates': 0,
   u'isEOF': 1,
   u'nReturned': 123805,
   u'needTime': 0,
   u'needYield': 0,
   u'restoreState': 967,
   u'saveState': 967,
   u'stage': u'FETCH',
   u'works': 123806},
  u'executionSuccess': True,
```

Illustration 50: execution statistics after indexing

Some useful information in the illustration above is encircled in red:

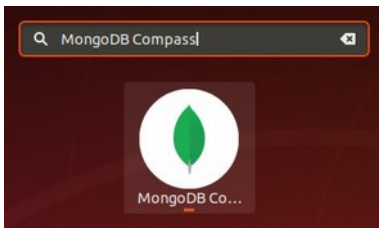
- ➔ The query returned 123805 records.
- ➔ The query used a index scan.
- ➔ It took 50 milliseconds to return 123805 records

As you can see in the execution statistics above, it took a smaller amount of time to execute the query on an indexed database in comparison to a non indexed database.

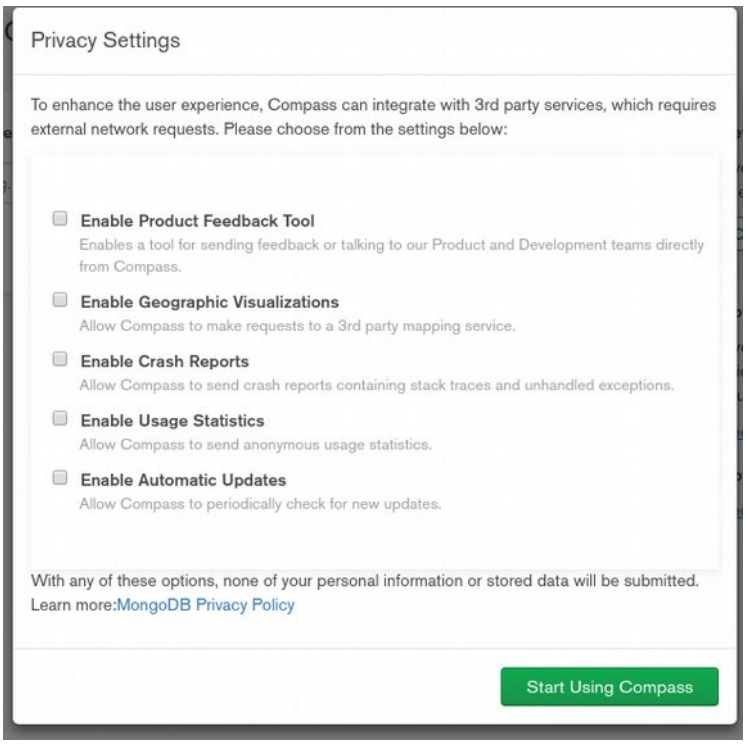
5 Validating the data using MongoCompass

To validate whether we correctly modeled, imported and indexed the data, we are going to use a tool called MongoCompass which is short for “MongoDB Compass”. In the cookbook: “Creating the Geostack Course VM” is described how to install MongoCompass.

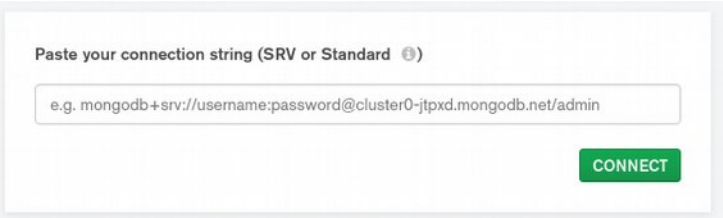
First we need to open MongoCompass which is done by searching for the application: “MongoDB Compass” and clicking on the shortcut as shown in the illustration below:



If this is the first time that you open MongoCompass you will be greeted with a screen as shown in the illustration below. You should un-check all the checkboxes since we don’t want to send any data to external providers. After un-checking all the boxes you should click on “Start Using Compass”.



Next click on the green Connect button as shown in the illustration below. We don’t need to add a connection string since the default connection string is the one which we are going to use.



When connected we are greeted with a screen similar to the one shown in illustration 51. On the left of the screen (encircled in red in the illustration below) you see a database called: “Crane_Database”. Click on it.

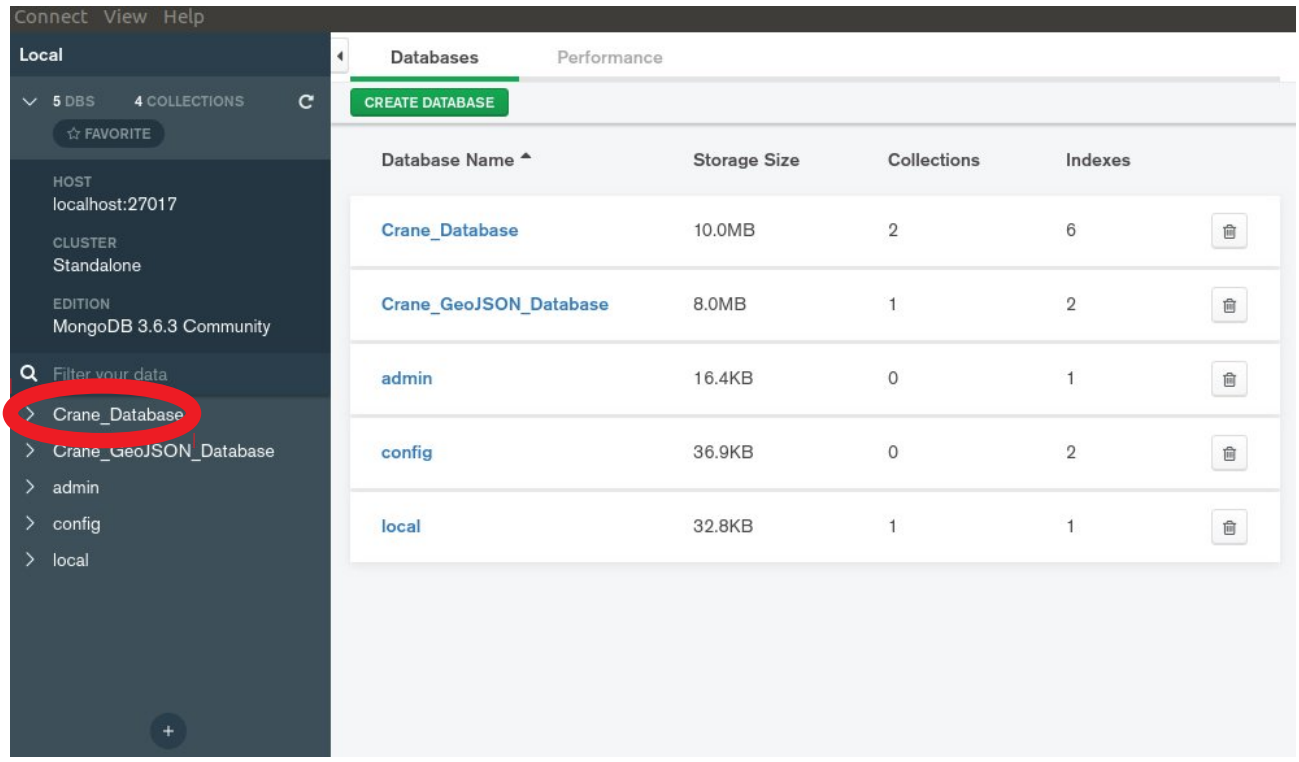


Illustration 51: MongoDB Landing Page

A screen containing information related to the database will pop up. The information displayed on you screen differs from the information displayed in your MongoCompass instance since the database, used in this screenshot, contained more Crane datasets. Information related to the illustration is shown below illustration 52.

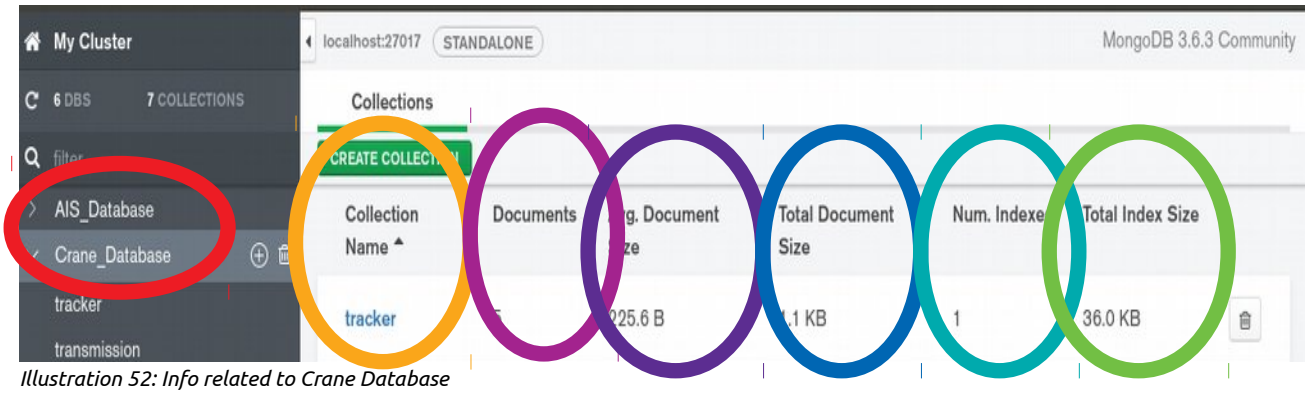


Illustration 52: Info related to Crane Database

Some information concerning illustration 52 is as follows:

- ➔ The red circle shows the database we selected and the collections in that database.
- ➔ The orange circle only shows the collections in the database.
- ➔ The light purple circle shows the amount of documents in the database.
- ➔ The dark purple circle shows the average size per document.
- ➔ The dark blue circle shows the total size of all documents combined.
- ➔ The light blue circle shows the amount of indexes on the collections.
- ➔ The green circle shows the total size of indexes combined.

When you click on the tracker collection encircled in orange you will be greeted with a screen similar to the one shown in illustration 53.

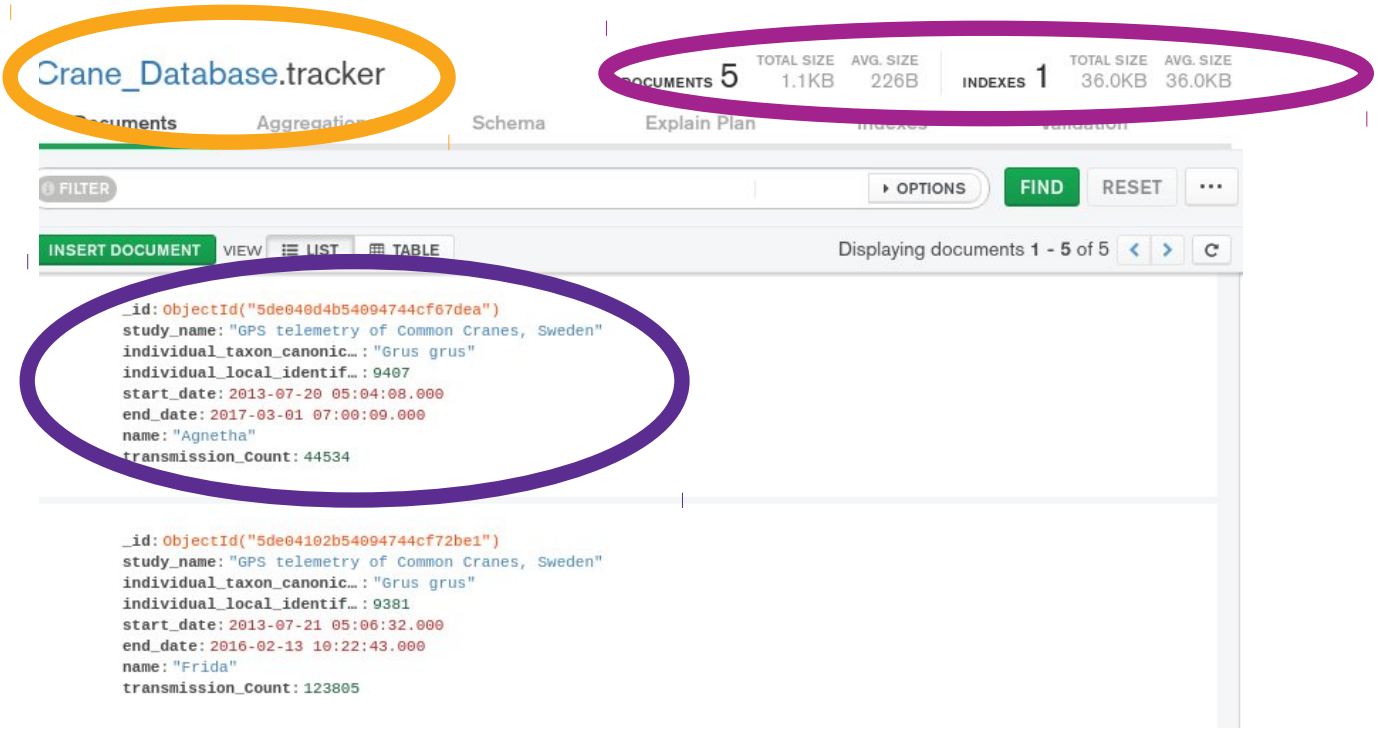


Illustration 53: Tracker document screen

Some information concerning illustration 53 is as follows:

- ➔ The orange circle shows the database and the collection you are currently viewing.
- ➔ The light purple circle shows information related to the collection you are currently viewing.
- ➔ The dark purple circle shows 1 document in the tracker collection.

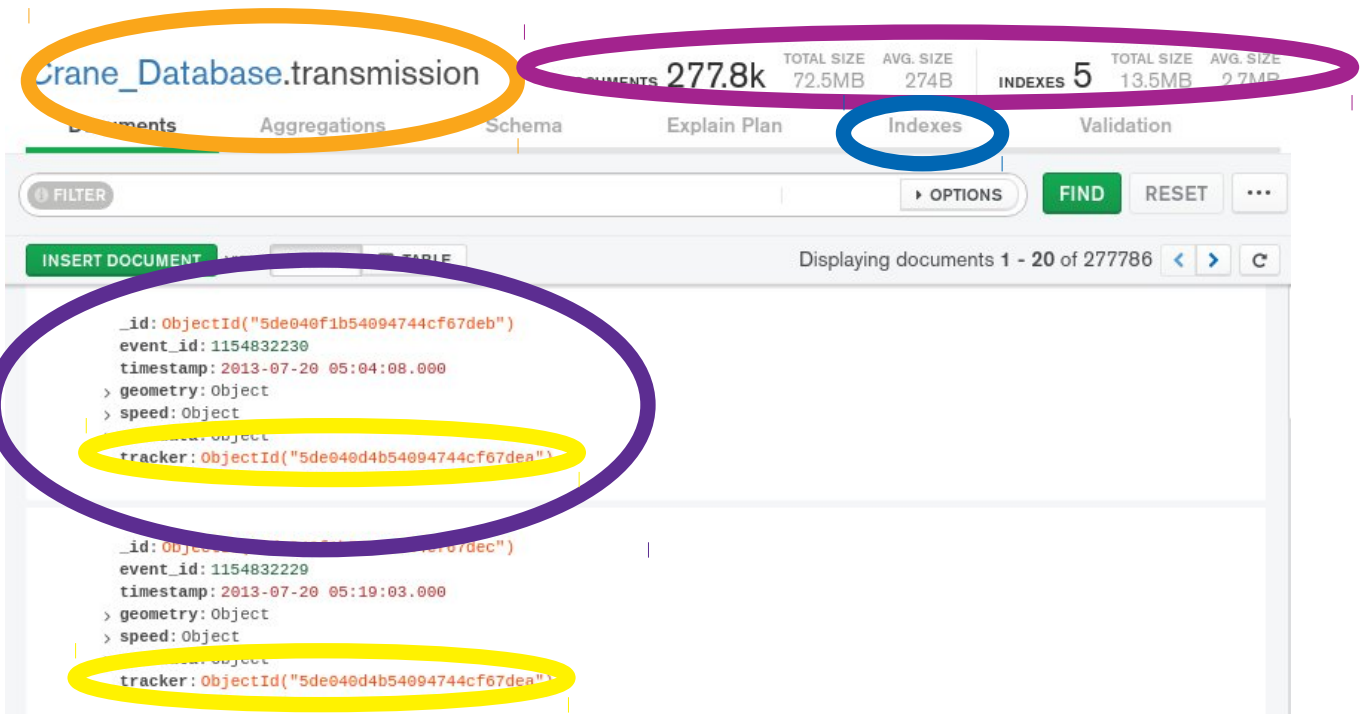


Illustration 54: Transmission document screen

Now we know that our tracker is imported correctly we are going to validate the transmission collection. When you click on the transmission collection, you will be greeted with a screen similar to the one show in illustration 54.

Some information concerning illustration 54 is as follows:

- ➔ The orange circle shows the database and the collection you are currently viewing.
- ➔ The light purple circle shows information related to the collection you are currently viewing.
- ➔ The dark purple circle shows 1 document in the transmission collection.
- ➔ The yellow circles show the reference fields containing the object Id's of the tracker to which the transmission belong.

When you click on the button, encircled in blue, you will be greeted with a list containing all the indexes created on the collection we are currently viewing. This is shown in illustration 55.

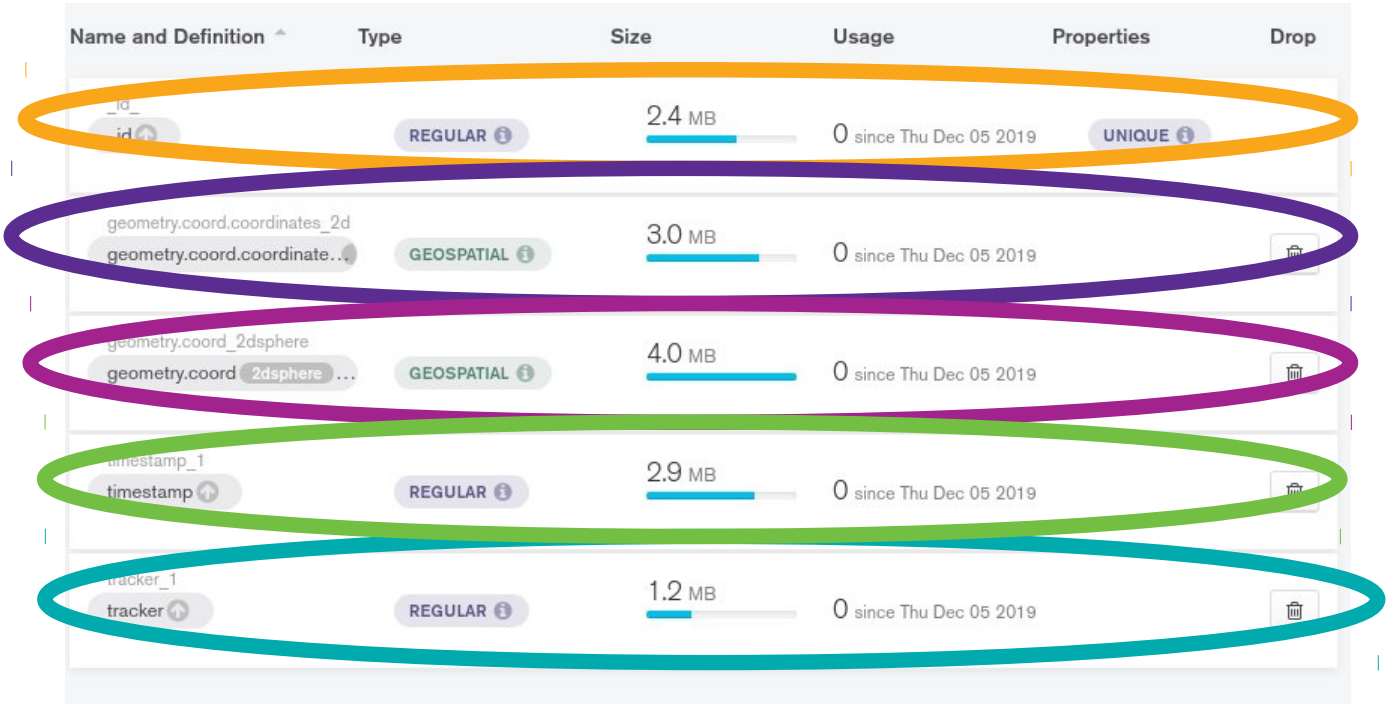


Illustration 55: Index list

Some information concerning illustration 55 is as follows:

- ➔ The orange circle shows the index on the id field. This index was automatically created by MongoEngine when importing the data.
- ➔ The dark purple circle shows the 2D index we created in chapter 4.5.
- ➔ The light purple circle shows the 2DSphere index which was created when we assigned the type PointField() to our coord field when modeling the geometry document.
- ➔ The the green circle shows the index created on the timestamp field in chapter 4.5,
- ➔ The light blue circle shows the index we created on the reference field in our transmission document.

As mentioned above, the database used in this illustration contains more crane datasets, that's the reason some values will differ from your values.

Your database should contain the following collections:

- ✔ tracker
- ✔ transmission

Your tracker collection should contain 1 index called _id

Your transmission collection should contain the following indexes :

- ✔ _id
- ✔ 2d
- ✔ 2dsphere
- ✔ timestamp
- ✔ tracker

If your database contains everything mention above, you can conclude we have modeled, imported and indexed the data correctly. As mentioned in the introduction of this document; There is also a folder located in the same folder as this document called: “Remaining-Data-Modeling-Notebooks”. You should go through all these notebooks and run all the cells to import the remaining Crane Tracker datasets, GPS Route (Trail) datasets and the World Port Index dataset. We need these datasets modeled, imported and indexed to reach the end goal of our Geographical software stack.

Everything done in those notebooks is similar to what we did during this cookbook!

6 Bibliography

1) MongoDB. (z.d.). *Introduction to MongoDB — MongoDB Manual*. Visited 22 October 2019, at https://docs.mongodb.com/manual/introduction/'

2) Singh , C. (2017, 19 september). *Introduction to NoSQL Databases*. Visited 22 October 2019, at https://beginnersbook.com/2017/09/introduction-to-nosql/

3) Singh, C. (2017, 17 september). *Introduction to MongoDB*. Visited 22 October 2019, at https://beginnersbook.com/2017/09/introduction-to-mongodb/

4) Radix. (2019, 10 mei). *What is Sharding? | Radix DLT - Decentralized Ledger Technology*. Visited 22 October 2019, at https://www.radixdlt.com/post/what-is-sharding/

5) Gour, R. (2018, 7 november). *MongoDB Data Modeling With Document Structure*. Visited 27 October 2019, at https://dzone.com/articles/mongodb-data-modeling-with-document-structure

6) MongoDB. (z.d.). *Schema Validation — MongoDB Manual*. Visited 27 October 2019, at https://docs.mongodb.com/manual/core/schema-validation/

7) Fullstack Academy. (2016, 23 januari). *YouTube* [YouTube]. Visited 5 December 2019, at https://www.youtube.com/watch?v=4rhKKFbbYT4

8) Khot, V. (2019, 4 januari). *Saving ML and DL models in MongoDB using python*. Visited 5 December 2019, at https://medium.com/up-engineering/saving-ml-and-dl-models-in-mongodb-using-python-f0bbbae256f0

9) Real Python. (2019, 8 juli). *Introduction to MongoDB and Python*. Visited 5 December 2019, at https://realpython.com/introduction-to-mongodb-and-python/#defining-a-document

10) MongoDB. (z.d.-a). *MongoDB Documentation*. Visited 1 November 2019, at https://docs.mongodb.com