MobilityDB Workshop

COLLABORATORS

	TITLE : MobilityDB Workshop		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Mahmoud SAKR and Esteban ZIMÁNYI	December 21, 2022	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

List of Figures

List of Tables

Abstract

Every module in this workshop illustrates a usage scenario of MobilityDB. The data sets and the tools are described inside each of the modules. Eventually, additional modules will be added to discover more MobilityDB features.

While this workshop illustrates the usage of MobilityDB functions, it does not explain them in detail. If you need help concerning the functions of MobilityDB, please refer to the documentation.

If you have questions, ideas, comments, etc., please contact me on mahmoud.sakr@ulb.ac.be.

Chapter 1

Managing Ship Trajectories (AIS)

AIS stands for Automatic Identification System. It is the location tracking system for sea vessels. This module illustrates how to load big AIS data sets into MobilityDB and do basic exploration.

The idea of this module is inspired from the tutorial of MovingPandas on ship data analysis by Anita Graser.

1.1 Contents

This module covers the following topics:

- Loading large trajectory datasets into MobilityDB
- Create proper indexes to speed up trajectory construction
- Select trajectories by a spatial window
- Join trajectories tables by proximity
- Select certain parts inside individual trajectories
- Manage the temporal speed and azimuth features of ships

1.2 Data

The Danish Maritime Authority publishes about 3 TB of AIS routes in CSV format here. The columns in the CSV are listed in Table 1.1. This module uses the data of one day April 1st 2018. The CSV file size is 1.9 GB, and it contains about 10 M rows.

1.3 Tools

The tools used in this module are as follows:

- MobilityDB, on top of PostgreSQL and PostGIS. Here I use the MobilityDB docker image.
- QGIS

Timestamp	Timestamp from the AIS base station, format: 31/12/2015 23:59:59
Type of mobile	Describes what type of target this message is received from (class A AIS Vessel, Class B AIS vessel, etc)
MMSI	MMSI number of vessel
Latitude	Latitude of message report (e.g. 57,8794)
Longitude	Longitude of message report (e.g. 17,9125)
Navigational status	Navigational status from AIS message if available, e.g.: 'Engaged in fishing', 'Under way using engine', mv.
ROT	Rot of turn from AIS message if available
SOG	Speed over ground from AIS message if available
COG	Course over ground from AIS message if available
Heading	Heading from AIS message if available
IMO	IMO number of the vessel
Callsign	Callsign of the vessel
Name	Name of the vessel
Ship type	Describes the AIS ship type of this vessel
Cargo type	Type of cargo from the AIS message
Width	Width of the vessel
Length	Lenght of the vessel
Type of position fixing device	Type of positional fixing device from the AIS message
Draught	Draugth field from AIS message
Destination	Destination from AIS message
ETA	Estimated Time of Arrival, if available
Data source type	Data source type, e.g. AIS
Size A	Length from GPS to the bow
Size B	Length from GPS to the stern
Size C	Length from GPS to starboard side
Size D	Length from GPS to port side

Table 1.1: AIS columns

1.4 Preparing the Database

Create a new database `DanishAIS`, then use your SQL editor to create the MobilityDB extension as follows:

```
CREATE EXTENSION MobilityDB CASCADE;
```

The `CASCADE` command will additionally create the PostGIS extension.

Now create a table in which the CSV file will be loaded:

```
CREATE TABLE AISInput(
  T timestamp,
  TypeOfMobile varchar(50),
  MMSI integer,
  Latitude float,
  Longitude float,
  navigationalStatus varchar(50),
  ROT float,
  SOG float,
  COG float,
  Heading integer,
  IMO varchar(50),
  Callsign varchar(50),
  Name varchar(100),
  ShipType varchar(50),
  CargoType varchar(100),
  Width float,
  Length float,
  TypeOfPositionFixingDevice varchar(50),
  Draught float,
  Destination varchar(50),
  ETA varchar(50),
  DataSourceType varchar(50),
  SizeA float,
  SizeB float,
  SizeC float,
  SizeD float,
  Geom geometry(Point, 4326)
);
```

1.5 Loading the Data

For importing CSV data into a PostgreSQL database one can use the `COPY` command as follows:

```
COPY AISInput(T, TypeOfMobile, MMSI, Latitude, Longitude, NavigationalStatus,
  ROT, SOG, COG, Heading, IMO, CallSign, Name, ShipType, CargoType, Width, Length,
  TypeOfPositionFixingDevice, Draught, Destination, ETA, DataSourceType,
  SizeA, SizeB, SizeC, SizeD)
FROM '/home/mobilitydb/DanishAIS/aisdk_20180401.csv' DELIMITER  ',' CSV HEADER;
```

This import took about 3 minutes on my machine, which is an average laptop. The CSV file has 10,619,212 rows, all of which were correctly imported. For bigger datasets, one could alternative could use the program pgloader.

We clean up some of the fields in the table and create spatial points with the following command.

```
UPDATE AISInput SET
  NavigationalStatus = CASE NavigationalStatus WHEN 'Unknown value' THEN NULL END,
  IMO = CASE IMO WHEN 'Unknown' THEN NULL END,
  ShipType = CASE ShipType WHEN 'Undefined' THEN NULL END,
  TypeOfPositionFixingDevice = CASE TypeOfPositionFixingDevice
  WHEN 'Undefined' THEN NULL END,
  Geom = ST_SetSRID( ST_MakePoint( Longitude, Latitude ), 4326);
```

This took about 5 minutes on my machine. Let's visualize the spatial points on QGIS.

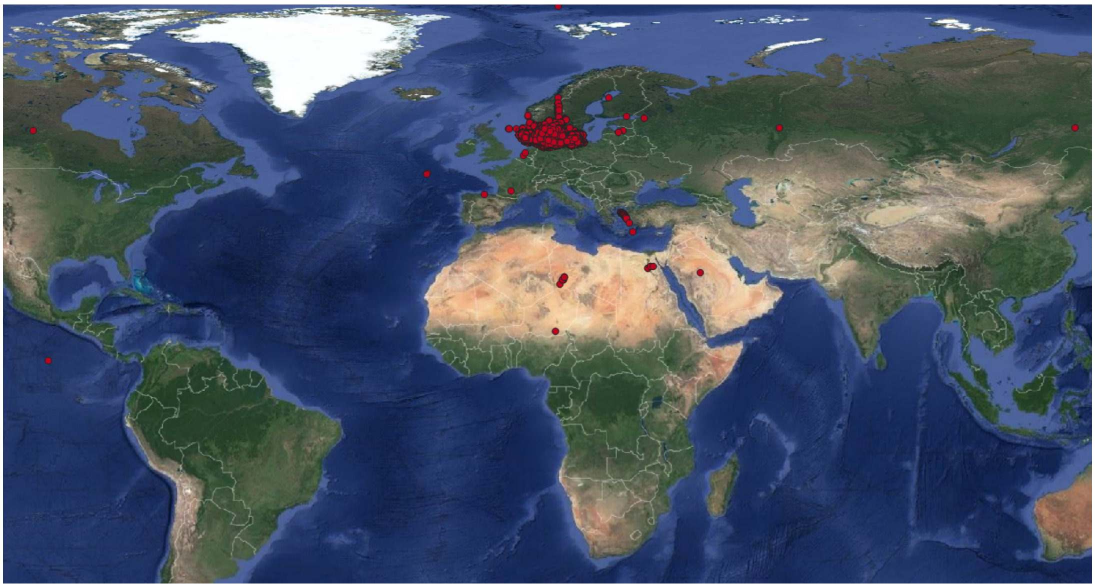


Figure 1.1: Visualizing the input points

Clearly, there are noise points that are far away from Denmark or even outside earth. This module will not discuss a thorough data cleaning. However, we do some basic cleaning in order to be able to construct trajectories:

- Filter out points that are outside the window defined by bounds point(-16.1,40.18) and point(32.88, 84.17). This window is obtained from the specifications of the projection in https://epsg.io/25832.
- Filter out the rows that have the same identifier (MMSI, T)

```
CREATE TABLE AISInputFiltered AS
SELECT DISTINCT ON(MMSI,T) *
FROM AISInput
WHERE Longitude BETWEEN -16.1 and 32.88 AND Latitude BETWEEN 40.18 AND 84.17;
-- Query returned successfully: 10357703 rows affected, 01:14 minutes execution time.
SELECT COUNT(*) FROM AISInputFiltered;
--10357703
```

1.6 Constructing Trajectories

Now we are ready to construct ship trajectories out of their individual observations:

```
CREATE TABLE Ships(MMSI, Trip, SOG, COG) AS
SELECT MMSI,
  tgeompoint_seq(array_agg(tgeompoint_inst( ST_Transform(Geom, 25832), T) ORDER BY T)),
  tfloat_seq(array_agg(tfloat_inst(SOG, T) ORDER BY T) FILTER (WHERE SOG IS NOT NULL)),
  tfloat_seq(array_agg(tfloat_inst(COG, T) ORDER BY T) FILTER (WHERE COG IS NOT NULL))
FROM AISInputFiltered
GROUP BY MMSI;
-- Query returned successfully: 2995 rows affected, 01:16 minutes execution time.
```

This query constructs, per ship, its spatiotemporal trajectory `Trip`, and two temporal attributes `SOG` and `COG`. `Trip` is a temporal geometry point, and both `SOG` and `COG` are temporal floats. MobilityDB builds on the coordinate transformation feature of PostGIS. Here the SRID 25832 (European Terrestrial Reference System 1989) is used, because it is the one advised by Danish Maritime Authority in the download page of this dataset. Now, let's visualize the constructed trajectories in QGIS.

```
ALTER TABLE Ships ADD COLUMN Traj geometry;
UPDATE Ships SET Traj= trajectory(Trip);
-- Query returned successfully: 2995 rows affected, 3.8 secs execution time.
```

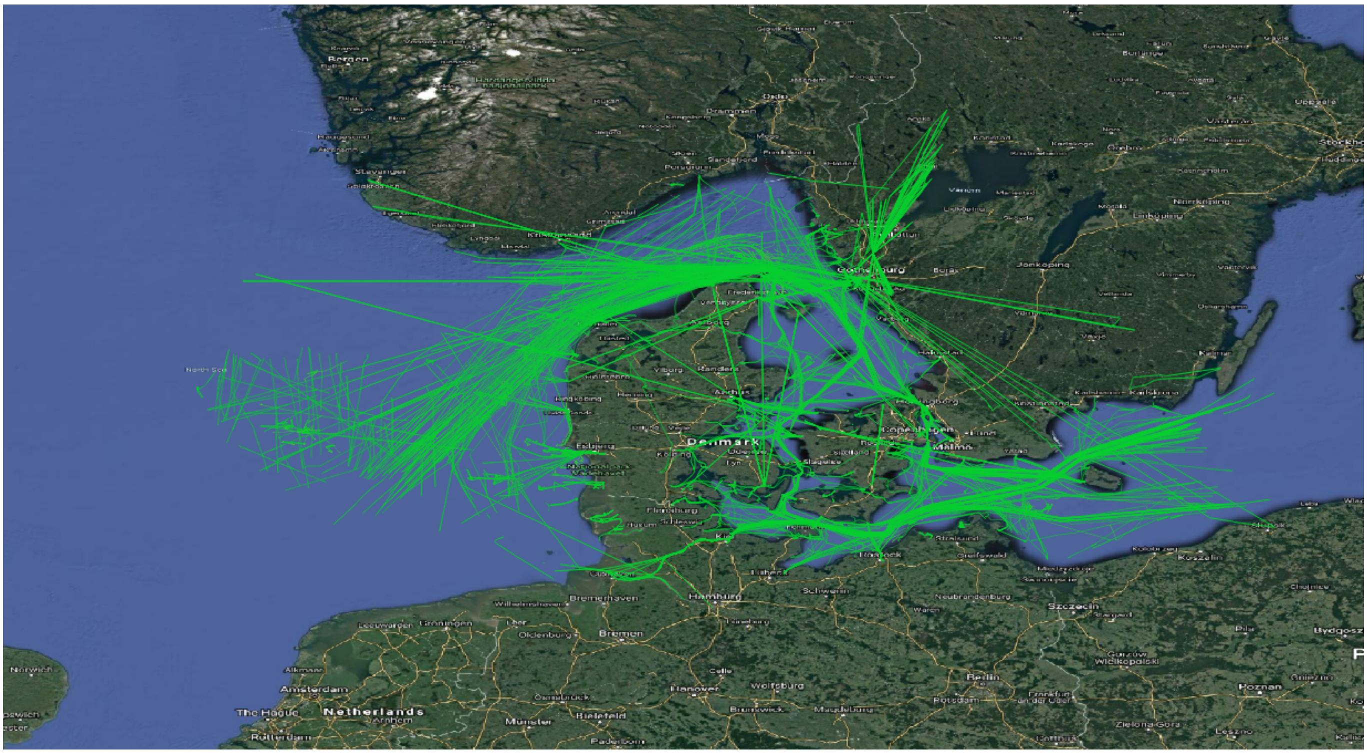


Figure 1.2: Visualizing the ship trajectories

1.7 Basic Data Exploration

The total distance traveled by all ships:

```
SELECT SUM( length( Trip ) ) FROM Ships;
--500433519.121321
```

This query uses the `length` function to compute per trip the sailing distance in meters. We then aggregate over all trips and calculate the sum. Let's have a more detailed look, and generate a histogram of trip lengths:

```
WITH buckets (bucketNo, RangeKM) AS (
  SELECT 1, floatrange '[0, 0]' UNION
  SELECT 2, floatrange '(0, 50)' UNION
  SELECT 3, floatrange '[50, 100)' UNION
  SELECT 4, floatrange '[100, 200)' UNION
  SELECT 5, floatrange '[200, 500)' UNION
  SELECT 6, floatrange '[500, 1500)' UNION
  SELECT 7, floatrange '[1500, 10000)' ),
histogram AS (
  SELECT bucketNo, RangeKM, count(MMSI) as freq
  FROM buckets left outer join Ships on (length(Trip)/1000) <@ RangeKM
  GROUP BY bucketNo, RangeKM
  ORDER BY bucketNo, RangeKM
)
SELECT bucketNo, RangeKM, freq,
  repeat('■', ( freq::float / max(freq) OVER () * 30 )::int ) AS bar
FROM histogram;
--Total query runtime: 5.6 secs

bucketNo,   bucketRange,        freq     bar
1;          "[0,0]";            303;       ■■■■■
2;          "(0,50)";           1693;      ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
3;          "[50,100)";         267;       ■■■■■
4;          "[100,200)";        276;       ■■■■■
5;          "[200,500)";        361;       ■■■■■■
6;          "[500,1500)";       86;        ■■
7;          "[1500,10000)";     6;
```

Surprisingly there are trips with zero length. These are clearly noise that can be deleted. Also there are very many short trips, that are less than 50 km long. On the other hand, there are few long trips that are more than 1,500 km long. Let's visualize these last two cases in Figure 1.3. They look like noise. Normally one should validate more, but to simplify this module, we consider them as noise, and delete them.

```
DELETE FROM Ships
WHERE length(Trip) = 0 OR length(Trip) >= 1500000;
-- Query returned successfully in 7 secs 304 msec.
```

Now the `Ships` table looks like Figure 1.4.

Let's have a look at the speed of the ships. There are two speed values in the data; the speed calculated from the spatiotemporal trajectory `speed(Trip)`, and the `SOG` attribute. Optimally, the two will be the same. A small variance would still be OK, because of sensor errors. Note that both are temporal floats. In the next query, we compare the averages of the two speed values for every ship:

```
SELECT ABS(twavg(SOG) * 1.852 - twavg(speed(Trip))* 3.6 ) SpeedDifference
FROM Ships
ORDER BY SpeedDifference DESC;
--Total query runtime: 8.2 secs
--990 rows retrieved.

SpeedDifference
NULL
NULL
NULL
```

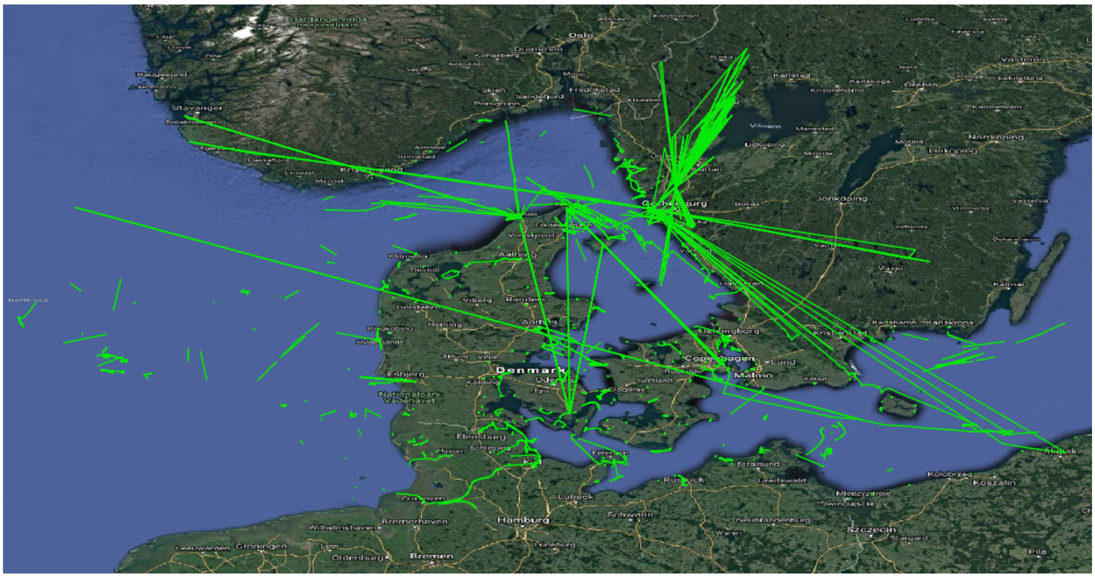


Figure 1.3: Visualizing trips with abnormal lengths

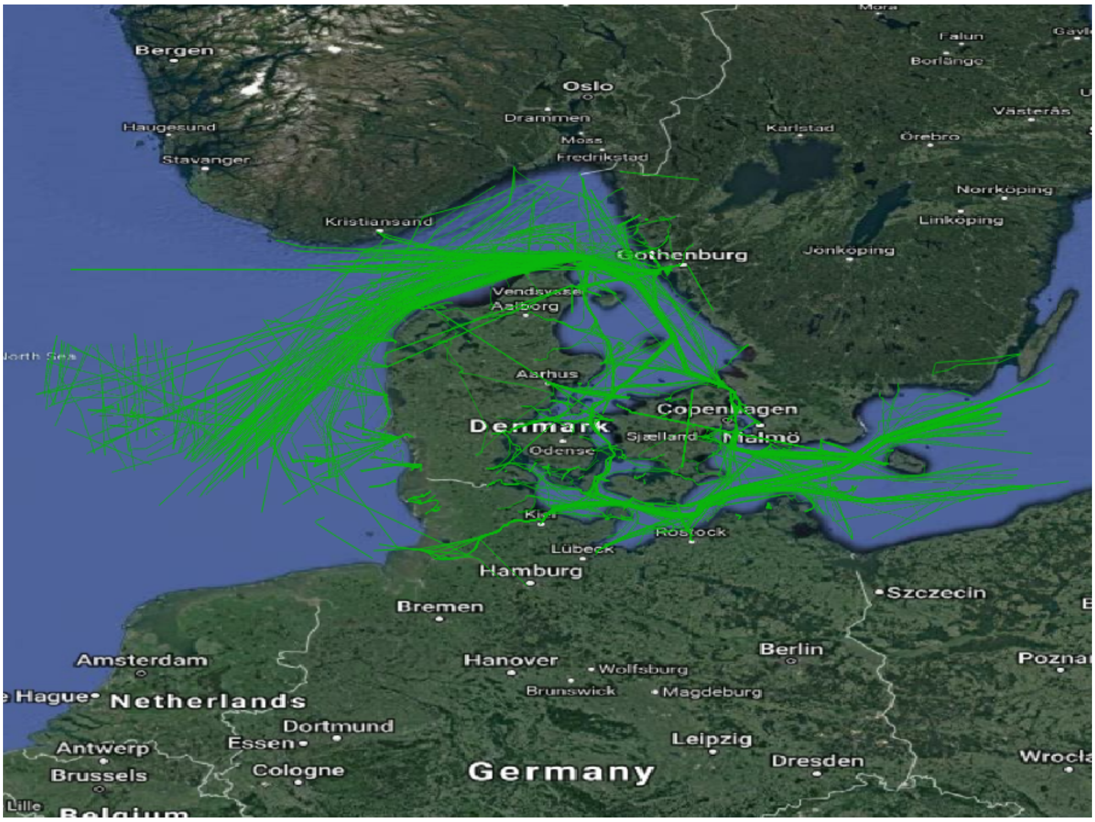


Figure 1.4: Ship trajectories after filtering

```
NULL
NULL
107.861100067879
57.1590253627668
42.4207839833568
39.5819188407125
33.6182789410313
30.9078594633161
26.514042447366
22.1312646226031
20.5389022294181
19.8500569368283
19.4134688682774
18.180139457754
17.4859077178001
17.3155991287105
17.1739822139821
12.9571603234404
12.6195380496344
12.2714437568609
10.9619033557275
10.4164745930929
10.3306155308426
9.46457823214455
...
```

The `twavg` computes a time-weighted average of a temporal float. It basically computes the area under the curve, then divides it by the time duration of the temporal float. By doing so, the speed values that remain for longer durations affect the average more than those that remain for shorter durations. Note that SOG is in knot, and Speed(Trip) is in m/s. The query converts both to km/h.

The query shows that 26 out of the 990 ship trajectories in the table have a difference of more than 10 km/h or NULL. These trajectories are shown in Figure 1.5. Again they look like noise, so we remove them.

Now we do a similar comparison between the calculated azimuth from the spatiotemporal trajectory, and the attribute COG:

```
SELECT ABS(twavg(COG) - twavg(azimuth(Trip)) * 180.0/pi() ) AzimuthDifference
FROM Ships
ORDER BY AzimuthDifference DESC;
--Total query runtime: 4.0 secs
--964 rows retrieved.

264.838740787458
220.958372832234
180.867071483688
178.774337481463
154.239639388087
139.633953692907
137.347542674865
128.239459879571
121.107566199195
119.843262642657
116.685117326047
116.010477588934
109.830338231363
106.94301191915
106.890186229337
106.55297972109
103.20192549283
102.585009756697
...
```

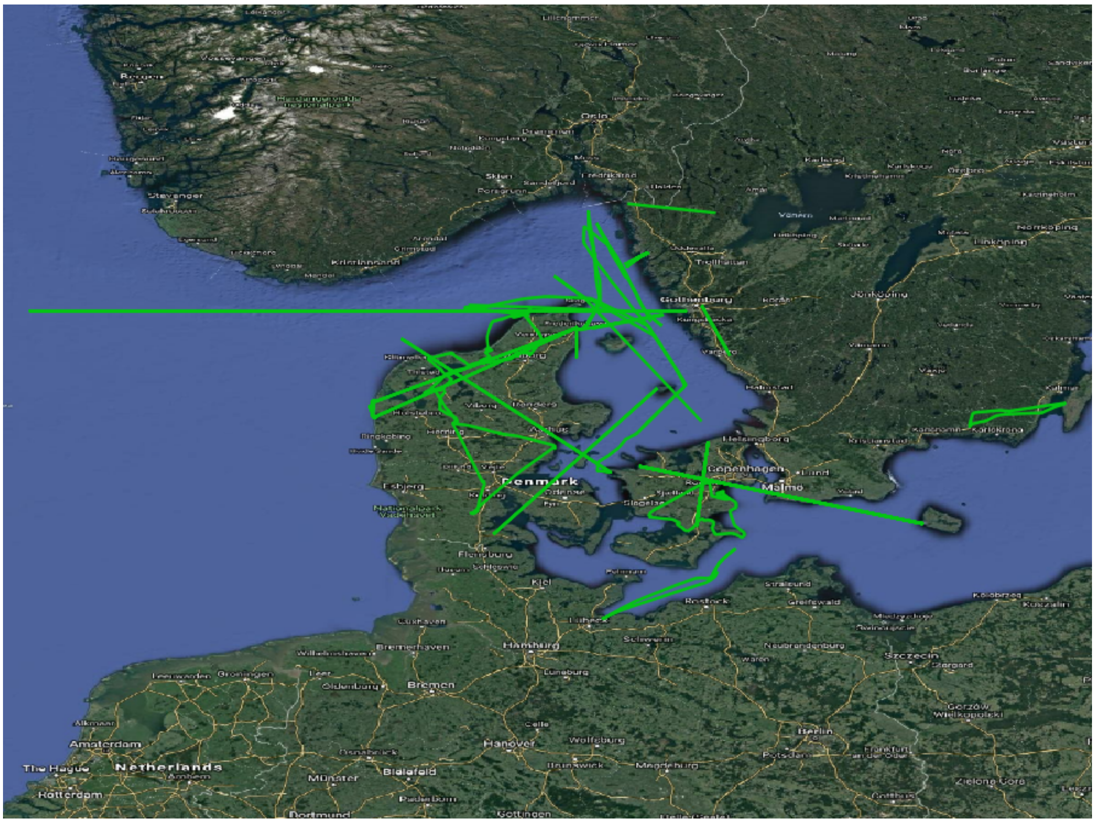


Figure 1.5: Ship trajectories with big difference between `speed(Trip)` and `SOG`

Here we see that the COG is not as accurate as the SOG attribute. More than 100 trajectories have an azimuth difference bigger than 45 degrees. Figure 1.6 visualizes them. Some of them look like noise, but some look fine. For simplicity, we keep them all.

1.8 Analyzing the Trajectories

Now we dive into MobilityDB and explore more of its functions. In Figure 1.7, we notice trajectories that keep going between Rødby and Puttgarden. Most probably, these are the ferries between the two ports. The task is simply to spot which Ships do so, and to count how many one way trips they did in this day. This is expressed in the following query:

```
CREATE INDEX Ships_Trip_Idx ON Ships USING GiST(Trip);

WITH Ports(Rodby, Puttgarden) AS (
  SELECT ST_MakeEnvelope(651135, 6058230, 651422, 6058548, 25832),
    ST_MakeEnvelope(644339, 6042108, 644896, 6042487, 25832)
)
SELECT S.*, Rodby, Puttgarden
FROM Ports P, Ships S
WHERE intersects(S.Trip, P.Rodby) AND intersects(S.Trip, P.Puttgarden)
--Total query runtime: 462 msec
--4 rows retrieved.
```

This query creates two envelope geometries that represent the locations of the two ports, then intersects them with the spatiotemporal trajectories of the ships. The `intersects` function checks whether a temporal point has ever intersects a geometry. To speed up the query, a spatiotemporal GiST index is first built on the `Trip` attribute. The query identified four Ships that

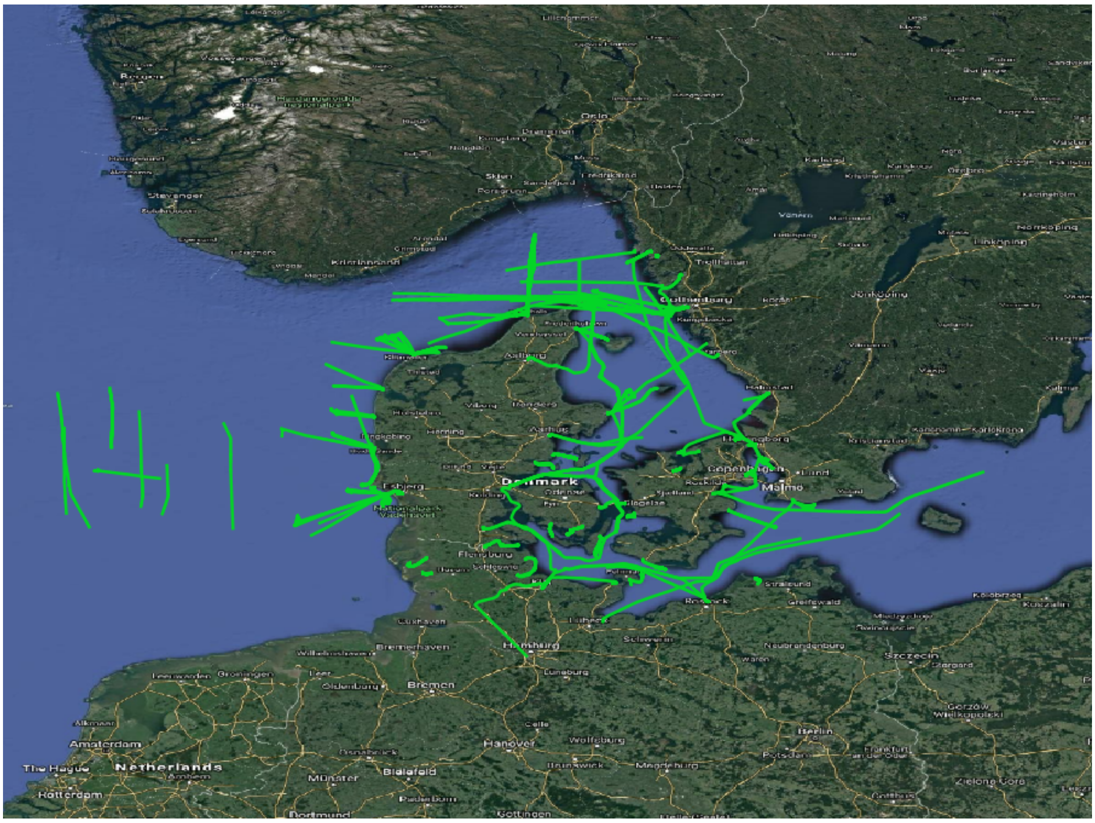


Figure 1.6: Ship trajectories with big difference between `azimuth(Trip)` and `COG`

commuted between the two ports, Figure 1.8. To count how many one way trips each of them did, we extend the previous query as follows:

```
WITH Ports(Rodby, Puttgarden) AS (
  SELECT ST_MakeEnvelope(651135, 6058230, 651422, 6058548, 25832),
    ST_MakeEnvelope(644339, 6042108, 644896, 6042487, 25832)
)
SELECT MMSI, (numSequences(atGeometry(S.Trip, P.Rodby)) +
  numSequences(atGeometry(S.Trip, P.Puttgarden)))/2.0 AS NumTrips
FROM Ports P, Ships S
WHERE intersects(S.Trip, P.Rodby) AND intersects(S.Trip, P.Puttgarden)
--Total query runtime: 1.1 secs

MMSI    NumTrips
219000429;  24.0
211188000;  24.0
211190000;  25.0
219000431;  16.0
```

The function `atGeometry` restricts the temporal point to the parts where it is inside the given geometry. The result is thus a temporal point that consists of multiple pieces (sequences), with temporal gaps in between. The function `numSequences` counts the number of these pieces.

With this high number of ferry trips, one wonders whether there are collision risks with ships that traverse this belt (the green trips in Figure 1.7). To check this, we query whether a pair of ship come very close to one another as follows:

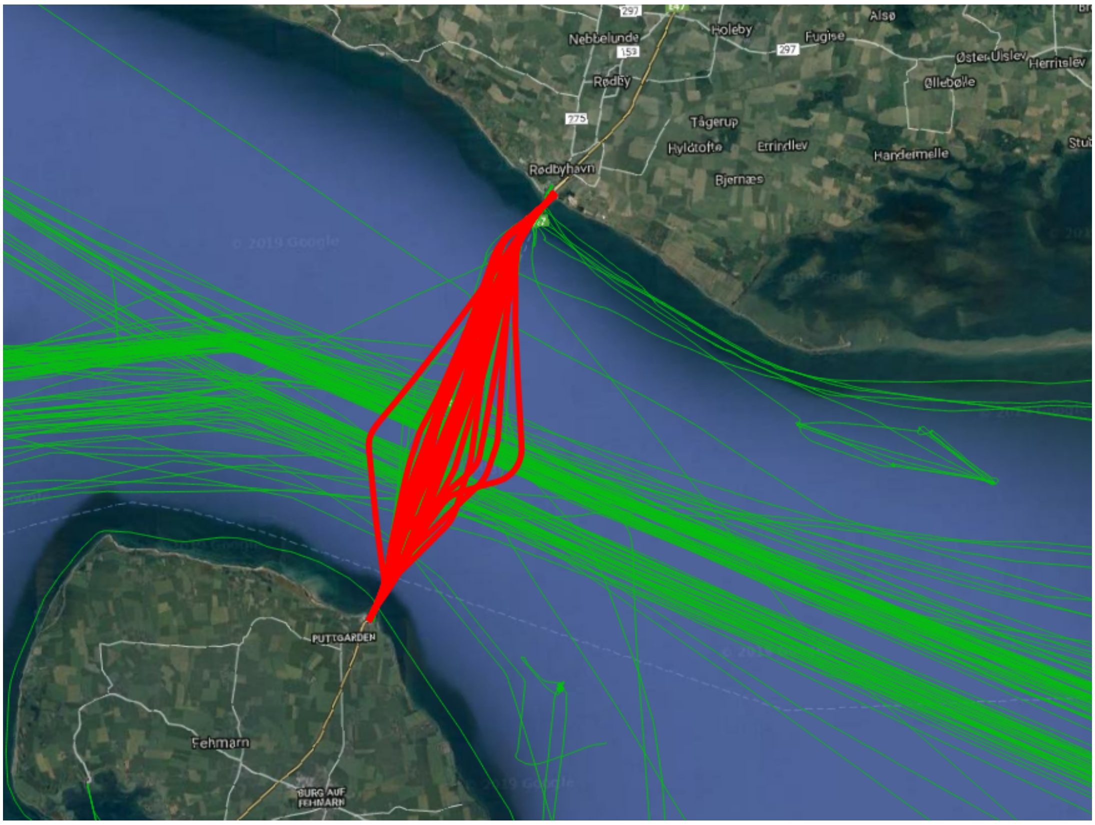


Figure 1.7: A sample ship trajectory between Rødby and Puttgarden

```
WITH B(Belt) AS (
  SELECT ST_MakeEnvelope(640730, 6058230, 654100, 6042487, 25832)
),
BeltShips AS (
  SELECT MMSI, atGeometry(S.Trip, B.Belt) AS Trip,
    trajectory(atGeometry(S.Trip, B.Belt)) AS Traj
  FROM Ships S, B
  WHERE intersects(S.Trip, B.Belt)
)
SELECT S1.MMSI, S2.MMSI, S1.Traj, S2.Traj, shortestLine(S1.trip, S2.trip) Approach
FROM BeltShips S1, BeltShips S2
WHERE S1.MMSI > S2.MMSI AND
dwithin(S1.trip, S2.trip, 300)
--Total query runtime: 28.5 secs
--7 rows retrieved.
```

The query first defines the area of interest as an envelope, the red dashed line in Figure 1.9). It then restricts/crops the trajectories to only this envelope using the `atGeometry` function. The main query then find pairs of different trajectories that ever came within a distance of 300 meters to one another (the `dwithin`). For these trajectories, it computes the spatial line that connects the two instants where the two trajectories were closest to one another (the `shortestLine` function). Figure 1.9 shows the green trajectories that came close to the blue trajectories, and their shortest connecting line in solid red. Most of the approaches occur at the entrance of the Rødby port, which looks normal. But we also see two interesting approaches, that may indicate danger of collision away from the port. They are shown with more zoom in Figure 1.10 and Figure 1.11

Figure 1.8: All ferries between Rødby and Puttgarden

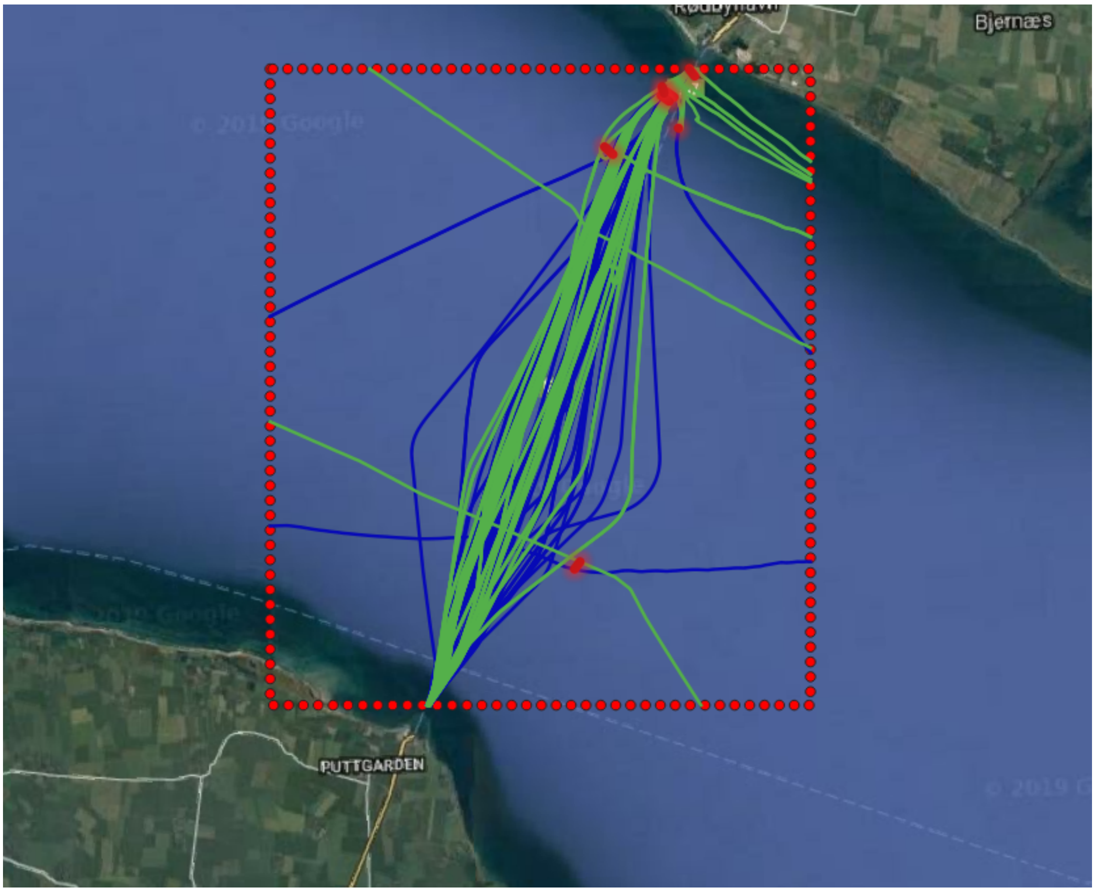


Figure 1.9: Ship that come closer than 300 meters to one another

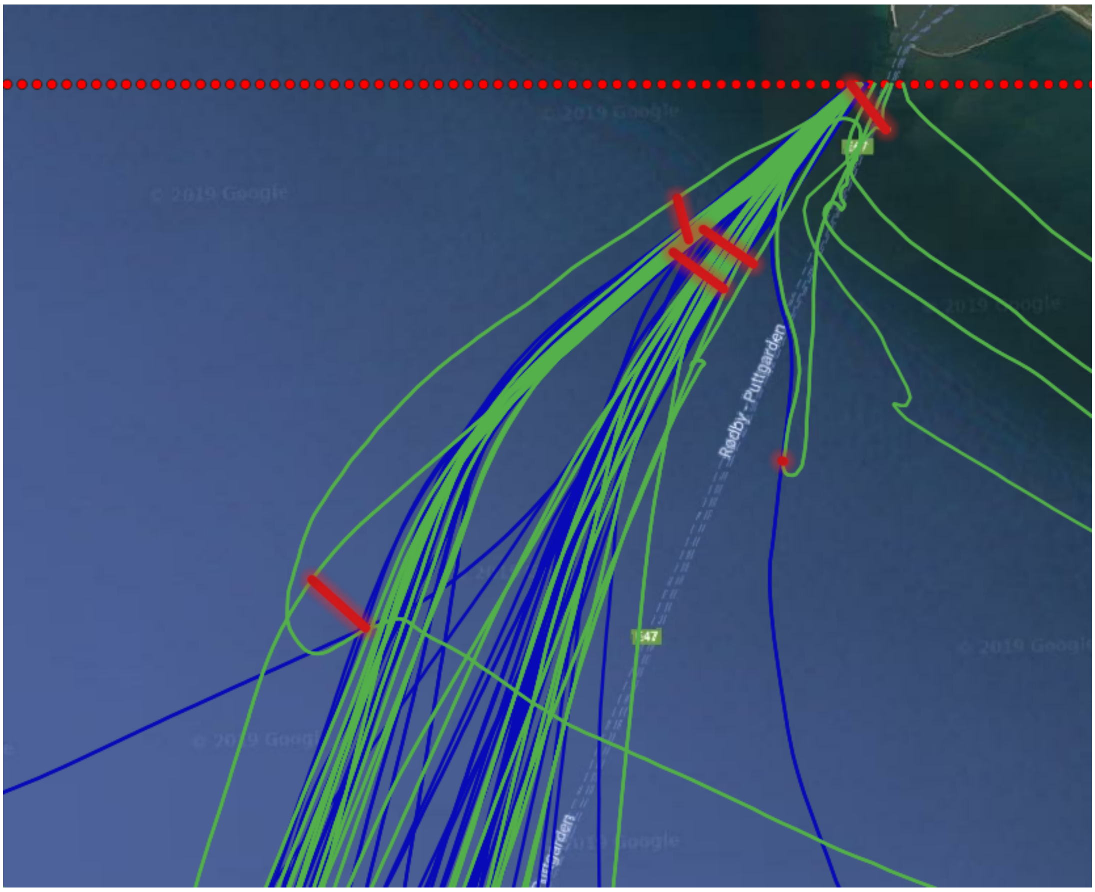


Figure 1.10: A zoom-in on a dangerous approach

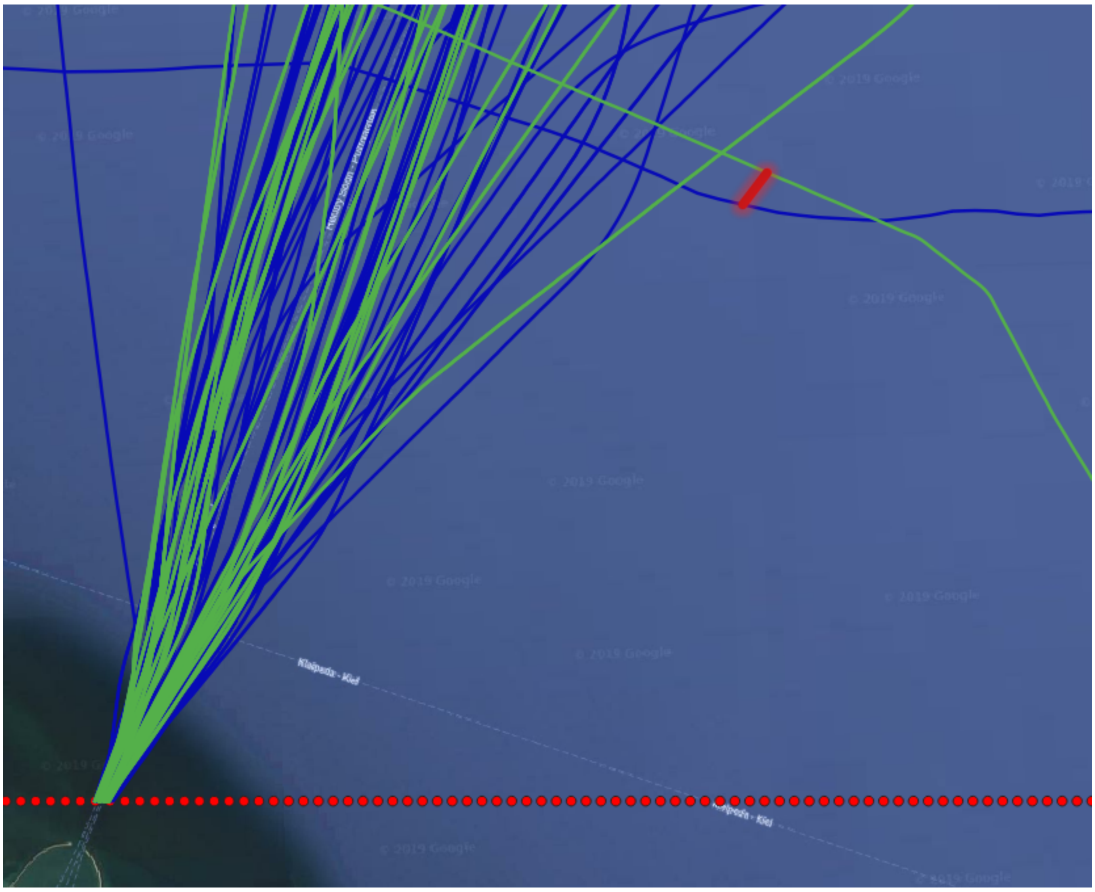


Figure 1.11: Another dangerous approach

Chapter 2

Dashboard and Visualization of Ship Trajectories (AIS)

This module builds on the Managing Ship Trajectories (AIS) module by creating a business intelligence dashboard to visualize and manipulate data. The module shows how to set up a Grafana dashboard with an existing database, create basic visualizations, set properties for different outputs, and use Variables to create dynamic visuals.

2.1 Contents

The module covers the following topics:

- Setting up a Grafana dashboard and connecting to a database
- Visualize a statistic from simple aggregations
- Visualize spatial frequency with a heat-map (not aggregated)
- Visualize frequency in spatial extent with a heat-map (pre-aggregated)
- Visualize spatio-temporal proximate objects
- Create dynamic queries with variables

2.2 Tools

The tools used in this module are as follows:

- MobilityDB, on top of PostgreSQL and PostGIS
- Grafana (version 9.0.7)

2.3 Setting up the Data Source

Data for the workshop is loaded into a MobilityDB database hosted on Azure, with all login information provided in the [Sign-in and Connect to Data Source] section below.

The raw data in CSV format is also available on the MobilityDB-workshop repository.

2.4 Setting up the Visualization Dashboard

We can use Grafana, an open-source technology, to create a business intelligence dashboard. This will allow different users to set up their own queries and visualizations, or easily slice through data in a visual way for non-technical users.

Start by setting up Grafana on your system:

1. macOS

```
brew update
brew install grafana
brew services start grafana
```

2. Debian or Ubuntu

```
# Note These are instructions for Grafana Enterprise Edition (via APT repository),
# which they recommend. It includes all the Open Source features and can also use
# Enterprise features if you have a License.

# Setup Grafana Keys
sudo apt-get install -y apt-transport-https
sudo apt-get install -y software-properties-common wget
wget -q -O - https://packages.grafana.com/gpg.key | sudo apt-key add -

# Add repository for stable releases
echo "deb https://packages.grafana.com/enterprise/deb stable main" &&
  | sudo tee -a /etc/apt/sources.list.d/grafana.list

# Install Grafana
sudo apt-get update
sudo apt-get install grafana-enterprise
```

3. Windows

   Use the Windows installer available at the Grafana website.

2.5 Sign in and Connect to Data Source

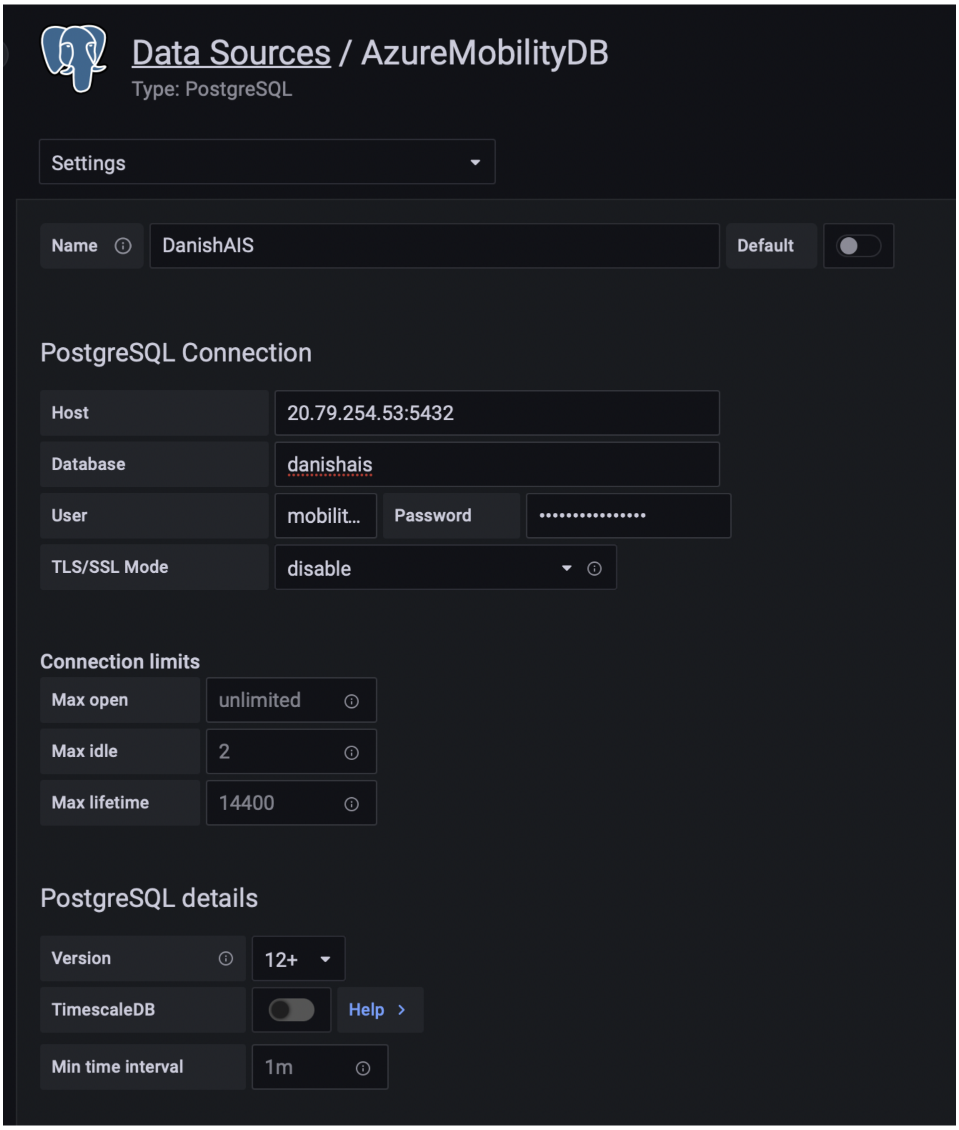
We can now sign in to Grafana by going to http://localhost:3000/. Set up a new account if needed. Additional instructions to login can be found here following the build your first dashboard instructions.

Next, we **add a data source** for Grafana to interact with. In this case, we can follow the Grafana instructions for adding a data source and search for PostgreSQL as the data source.

The workshop is using the following settings to connect to the postgres server on Azure.

- Name: *DanishAIS*
- Host: *20.79.254.53:5432*
- Database: *danishais*
- User: *mobilitydb-guest*
- Password: *mobilitydb@guest*
- TLS/SSL Mode: *disable*
- Version: *12+*

Then press save and test.



Data Sources / AzureMobilityDB
Type: PostgreSQL

Settings

Name: DanishAIS
Default: off

PostgreSQL Connection
Host: 20.79.254.53:5432
Database: danishais
User: mobilit...
Password: ••••••••••••••••
TLS/SSL Mode: disable

Connection limits
Max open: unlimited
Max idle: 2
Max lifetime: 14400

PostgreSQL details
Version: 12+
TimescaleDB: off

Min time interval: 1m

Figure 2.1: Data Source settings

2.6 Creating a Dashboard

With the dashboard configured, and a datasource added, we can now build different panels to visualize data in intuitive ways.

2.6.1 Speed of Individual Ships

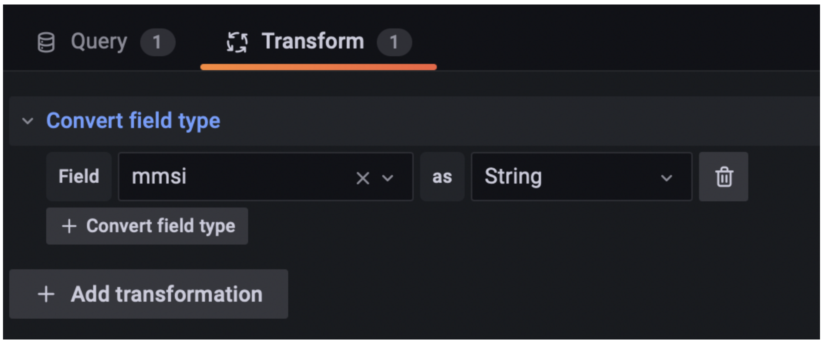
Let's visualize the speed of the ships using the previously built query. Here we will represent it as a statistic with a color gradient.

1. Add a new panel
2. Select *DanishAIS* as the data source
3. In Format as, change "Time series" to "Table" and choose "Edit SQL"

4. Here you can add your SQL queries. Let's replace the existing query with the following SQL script:

```
SELECT mmsi, ABS( twavg(SOG) * 1.852 - twavg( speed(Trip))* 3.6 ) AS SpeedDifference
FROM Ships
ORDER BY SpeedDifference DESC
LIMIT 5;
```

5. We can also quickly do some datatype transformations to help Grafana correctly interpret the incoming data. Next to the Query button, select "Transform", add "Convert field type" and choose *mmsi* as *String*.



Field: mmsi as String

Figure 2.2: Datatype transformations in Grafana

6. We will modify some visualization options in the panel on the right.

   First, choose *stat* as the visualization

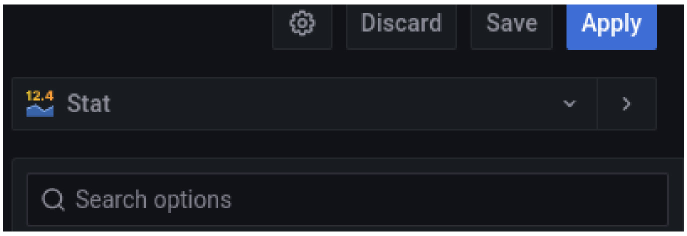
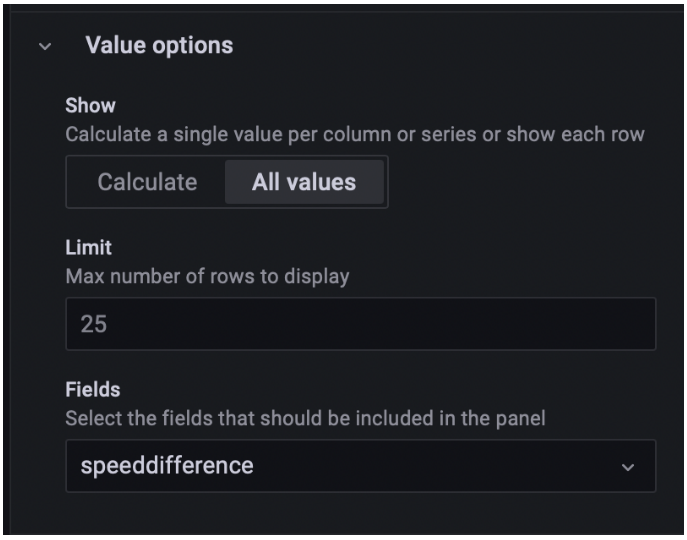


Figure 2.3: Choosing visualization type

Panel Options: Give the panel the title *Incorrect AIS Boat Speed Reporting*

Value Options:

- Show: All values
- Fields: speeddifference



Value options

Show

Calculate a single value per column or series or show each row

Calculate | **All values**

Limit

Max number of rows to display

25

Fields

Select the fields that should be included in the panel

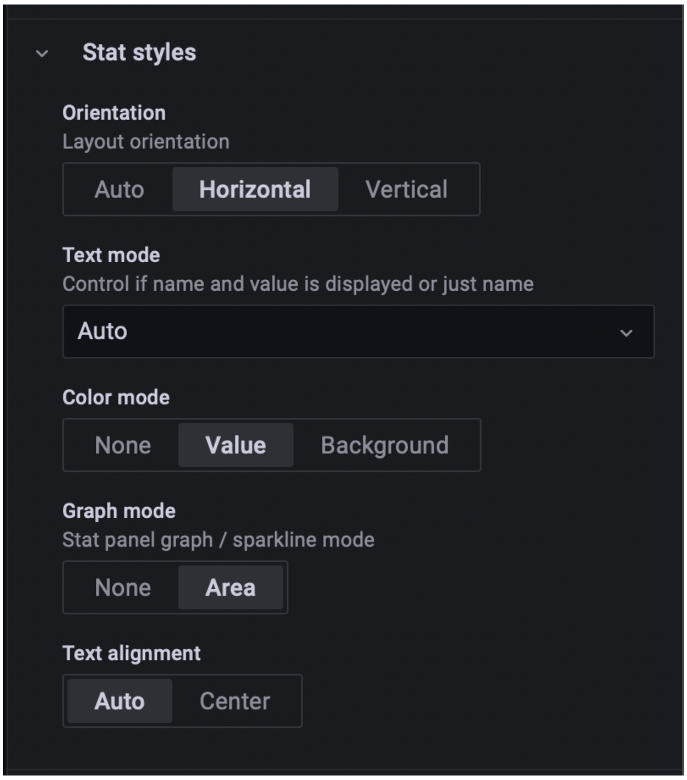
speeddifference

Figure 2.4: Value options dialogue box

Note: we can include a limit here instead of in our SQL query as well.

Stat Styles:

- Orientation: Horizontal



Stat styles

Orientation

Layout orientation

Auto | **Horizontal** | Vertical

Text mode

Control if name and value is displayed or just name

Auto

Color mode

None | **Value** | Background

Graph mode

Stat panel graph / sparkline mode

None | **Area**

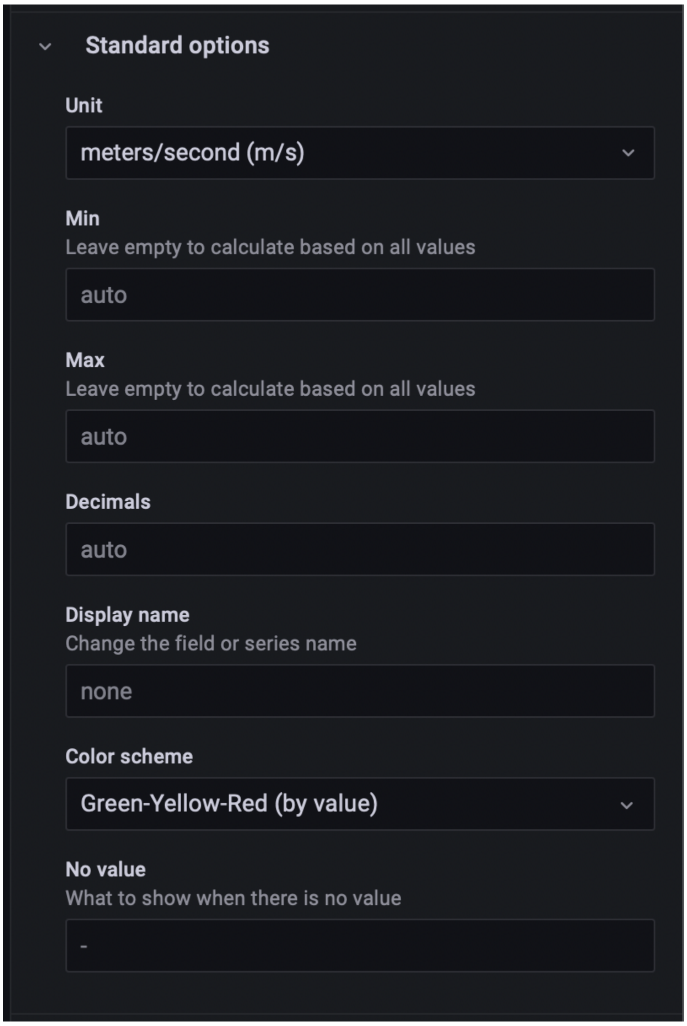
Text alignment

Auto | Center

Figure 2.5: Stat styles dialogue box

Standard Options:

- Unit: Velocity → meter/second (m/s). *Note: you can scroll in the drop-down menu to see all options.*
- Color scheme: Green-Yellow-Red (by value)



Standard options

Unit
meters/second (m/s)

Min
Leave empty to calculate based on all values
auto

Max
Leave empty to calculate based on all values
auto

Decimals
auto

Display name
Change the field or series name
none

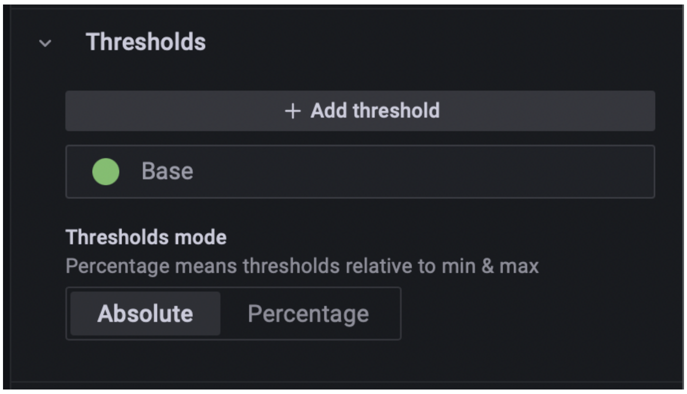
Color scheme
Green-Yellow-Red (by value)

No value
What to show when there is no value
-

Figure 2.6: Standard options dialogue box

Thresholds:

- remove the existing threshold by clicking the little trash can icon on the right. Adding a threshold will force the visualization to color the data a specific color if the threshold is met.



Thresholds

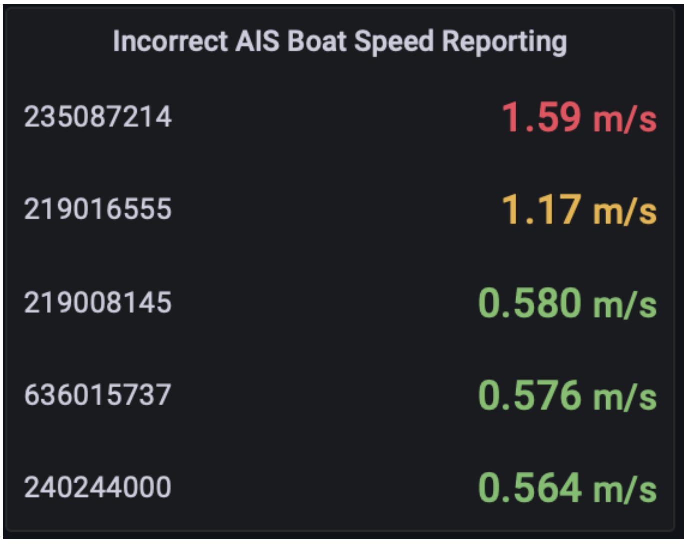
+ Add threshold

● Base

Thresholds mode
Percentage means thresholds relative to min & max
Absolute Percentage

Figure 2.7: Thresholds dialogue box

The final visualization will look like the screenshot below.



Incorrect AIS Boat Speed Reporting

235087214	1.59 m/s
219016555	1.17 m/s
219008145	0.580 m/s
636015737	0.576 m/s
240244000	0.564 m/s

Figure 2.8: Individual ship speed statistics visualization

2.6.2 Routes Used Most Frequently Visualized with a Static Heat Map

We can visualize the routes used by ships with a heat map generated from individual GPS points of the ships. This approach is quite costly, so we will use TABLESAMPLE SYSTEM to specify an approximate percentage of the data to use. If the frequency of locations returned varies in different areas, a heatmap using individual datapoints could be misleading without further data pre-processing. An alternative approach could be to use the postGIS ST_AsGeoJSON to generate shapes in geoJSON format which can be used in Grafana's World Map Panel plugin.

1. Add a panel, select DanishAIS as the data source and Format As Table.
2. Using Edit SQL, add the following SQL code:

```
-- NOTE: TABLESAMPLE SYSTEM(40) returns ~40% of the data.
SELECT
  latitude,
  longitude,
  mmsi
FROM aisinputfiltered TABLESAMPLE SYSTEM (40)
```

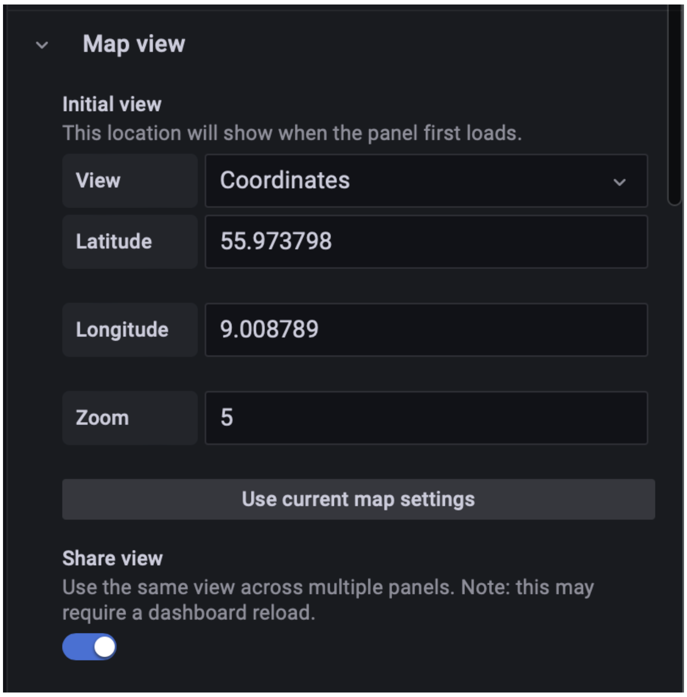
3. Change the visualization type to *Geomap*.
4. On the map, zoom in to fit the data points into the frame and modify the following visualization options:

   Panel Options:

   - Title: Route Usage Frequency

   Map View:

   - Use current map setting (this will use the current zoom and positioning level as default)
   - Share View: enable (this will sync up the movement and zoom across multiple maps on the same dashboard)



Initial view

This location will show when the panel first loads.

View: Coordinates

Latitude: 55.973798

Longitude: 9.008789

Zoom: 5

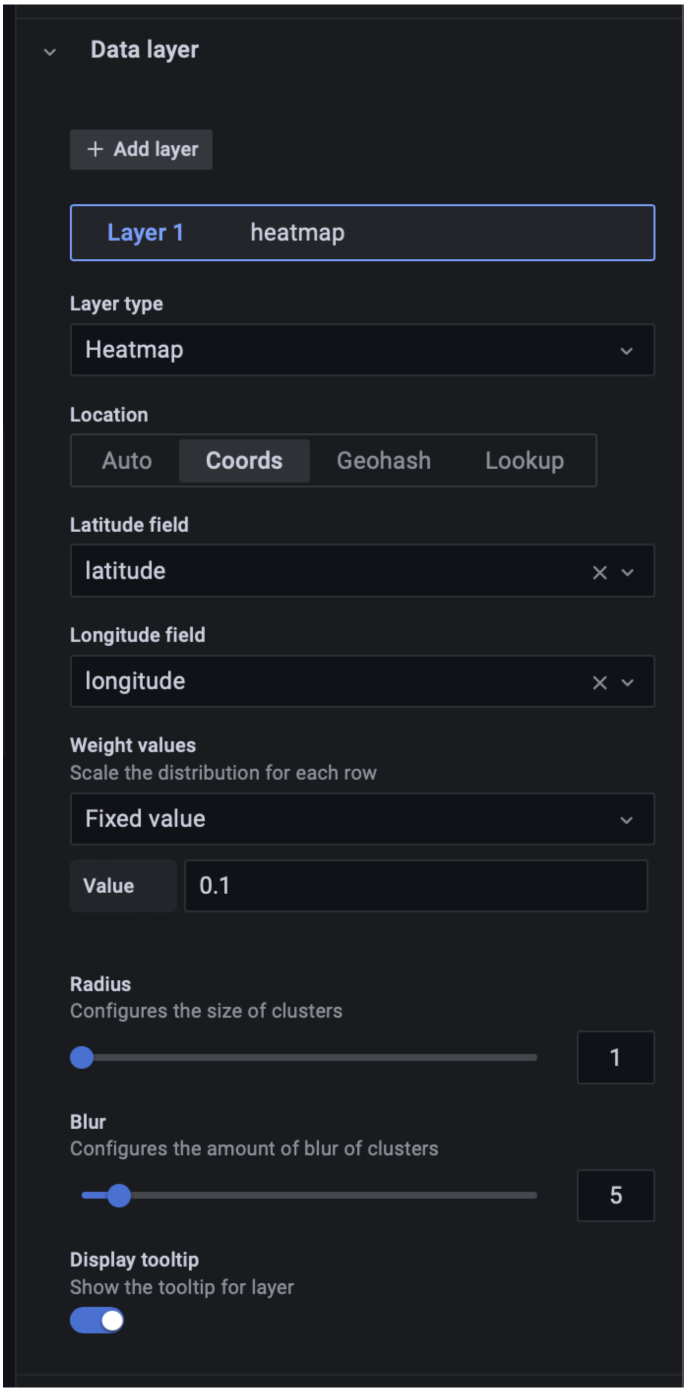
Share view

Use the same view across multiple panels. Note: this may require a dashboard reload.

Figure 2.9: Setting initial view in map view dialogue box

Data Layer:

- Layer type: Heatmap
- Location: Coords
- Latitude field: latitude
- Longitude field: longitude
- Weight values: 0.1
- Radius: 1
- Blur: 5



Data layer

Layer 1 heatmap

Layer type: Heatmap

Location: Auto | **Coords** | Geohash | Lookup

Latitude field: latitude

Longitude field: longitude

Weight values
Scale the distribution for each row
Fixed value

Value: 0.1

Radius
Configures the size of clusters
1

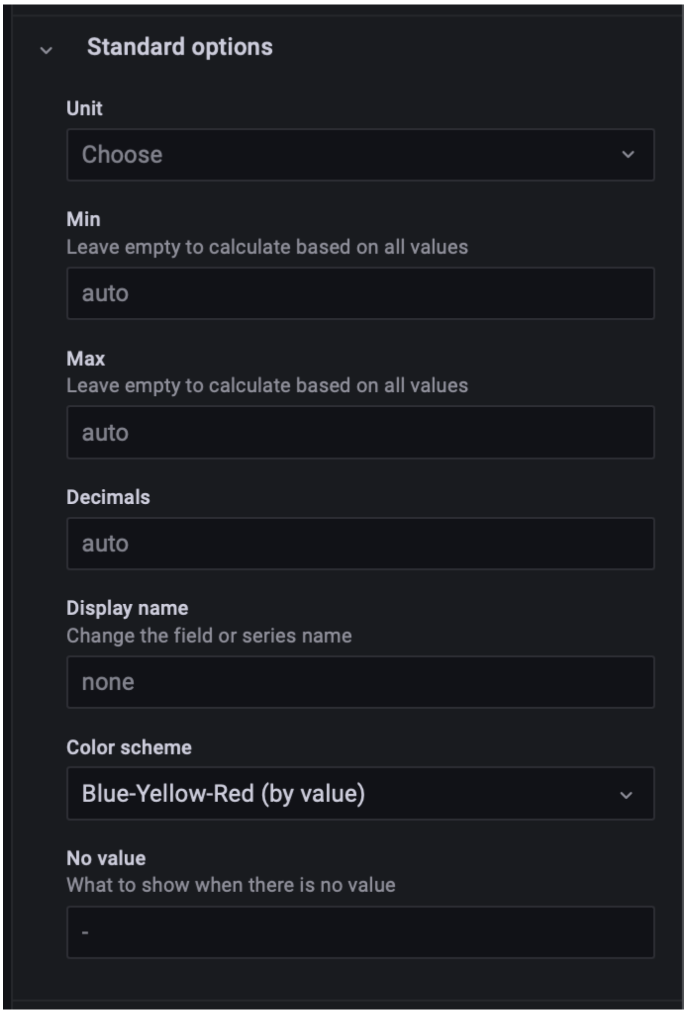
Blur
Configures the amount of blur of clusters
5

Display tooltip
Show the tooltip for layer

Figure 2.10: Setting up heat-map in data layer dialogue box

Standard Options:

- Color scheme: Blue-Yellow-Red (by value).



Standard options

Unit
Choose

Min
Leave empty to calculate based on all values
auto

Max
Leave empty to calculate based on all values
auto

Decimals
auto

Display name
Change the field or series name
none

Color scheme
Blue-Yellow-Red (by value)

No value
What to show when there is no value
-

Figure 2.11: Choosing color scheme in standard options dialogue box

The final visualization will look like the screenshot below.

Note: The number of datapoints rendered can be manipulated by changing the parameter of the TABLESAMPLE SYSTEM() call in the query.

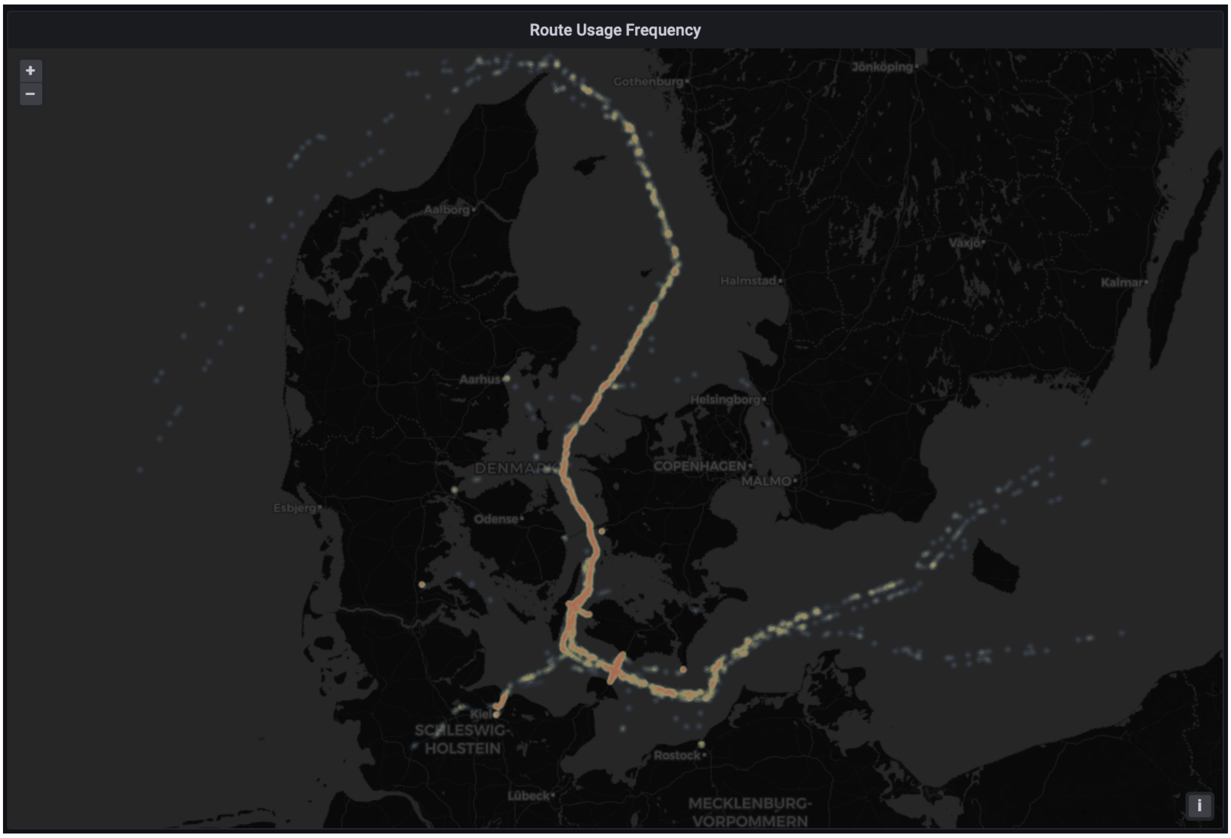


Figure 2.12: Route usage frequency heat-map visualization

2.6.3 Number of Boats Moving Through a Given Area

1. Create a new panel, and set DanishAIS as the Source, Format as: “Table”.
2. Select visualization as: “Geomap”
3. Add this SQL in the “SQL editor” section

```
-- Table with bounding boxes over regions of interest
WITH ports(port_name, port_geom, lat, lng)
       AS (SELECT p.port_name, p.port_geom, lat, lng
           FROM
             -- ST_MakeEnvelope creates geometry against which to check intersection
             (VALUES ('Rodby',
                    ST_MakeEnvelope(651135, 6058230, 651422, 6058548, 25832)::geometry,
                    54.53, 11.06),
                   ('Puttgarden',
                    ST_MakeEnvelope(644339, 6042108, 644896, 6042487, 25832)::geometry,
                    54.64, 11.36)) AS p(port_name, port_geom, lat, lng))

-- p.lat and p.lng will be used to place the port location on the visualization
SELECT P.port_name,
       sum(numSequences(atGeometry(S.Trip, P.port_geom))) AS trips_intersect_with_port,
       p.lat,
       p.lng
FROM ports AS P,
     Ships AS S
```

```
WHERE intersects(S.Trip, P.port_geom)
GROUP BY P.port_name, P.lat, P.lng
```

Note: You will see queries are build using the WITH statement (common table expressions - CTE). This helps to break the query down into parts, and also helps make it easier to understand by others.

4. The options (visualization settings - on the right side of the screen) should be as follows:

   Data Layer

   - Layer type: → “markers”
   - Style Size: → “Fixed” and value: 20
   - Color: → “trips_intersect_with_port” (This will color points on the map based on this value)

   Standard options

   - Min → 88
   - Max → 97
   - Color scheme → “Green-Yellow-Red (by value)”

   Note: At the writing of this tutorial, the Geomap plugin is in beta and has some minor bugs with how colors are rendered based when the “Min” and “Max” values are auto calculated.

In the visualization below we can see port Rodby has a higher number of ships coming and going to it and that’s why it is colored red. This visualization can show relative activity of ships in regions and ports.

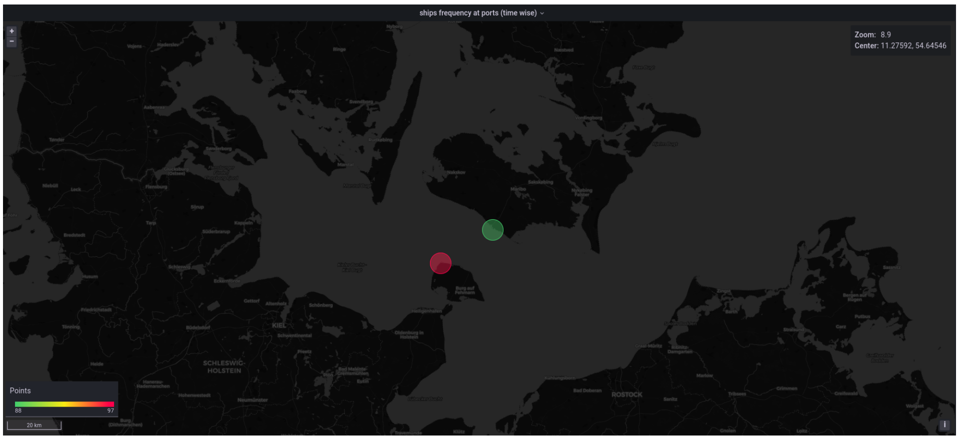


Figure 2.13: Frequency intersecting with geometric envelop visualization

2.6.4 Boats in Proximity in a Given Time Range

Follow the similar steps to add a Geomap panel as before, we include the following SQL script:

```
-- 2 CTEs are help to make these queries user-friendly; TimeShips and TimeClosestShips.
WITH
  -- The TimeShips CTE returns the data for a time period from 1am to 6:30am
  TimeShips AS (
    SELECT
      MMSI,
      atPeriod(S.Trip, period '[2018-01-04 01:00:00, 2018-01-04 06:30:00)' ) AS trip
    FROM
      Ships S
),
```

```
  -- The TimeClosestShips CTE returns the time, location, and closest distance of the boats
  -- that are within 300m of each other. Note the use of dwithin in the WHERE clause
  -- improves performance by limiting the computation to only those ships that were within
  -- 300m.
  TimeClosestShips AS (
    SELECT
      S1.MMSI AS "boat1", S2.MMSI AS "boat_2",
      startValue( atMin(S1.trip <-> S2.trip)) AS closet_distance,
      startTimestamp( atMin(S1.trip <-> S2.trip)) AS time_at_closest_dist,
      S1.trip AS "b1_trip",
      S2.trip AS "b2_trip"
    FROM
      TimeShips S1, TimeShips S2
    WHERE
      S1.MMSI > S2.MMSI AND
      dwithin(S1.Trip, S2.Trip, 300)
)
-- The final SELECT is used to project the time_at_closest_distance onto the sequence of
-- locations to return the lat and long of both ships.
SELECT  t.boat1, t.boat_2, t.closet_distance, t.time_at_closest_dist,
  ST_X( ST_Transform( valueAtTimestamp(b1_trip, time_at_closest_dist), 4326) ) AS b1_lng,
  ST_Y( ST_Transform( valueAtTimestamp(b1_trip, time_at_closest_dist), 4326) ) AS b1_lat,
  ST_X( ST_Transform( valueAtTimestamp(b2_trip, time_at_closest_dist), 4326) ) AS b2_lng,
  ST_Y( ST_Transform( valueAtTimestamp(b2_trip, time_at_closest_dist), 4326) ) AS b2_lat

FROM TimeClosestShips t;
```

To add the points to the map modify the following options:

Panel Options:

- Title: Ships within 300m

Map View:

- Share view: enabled

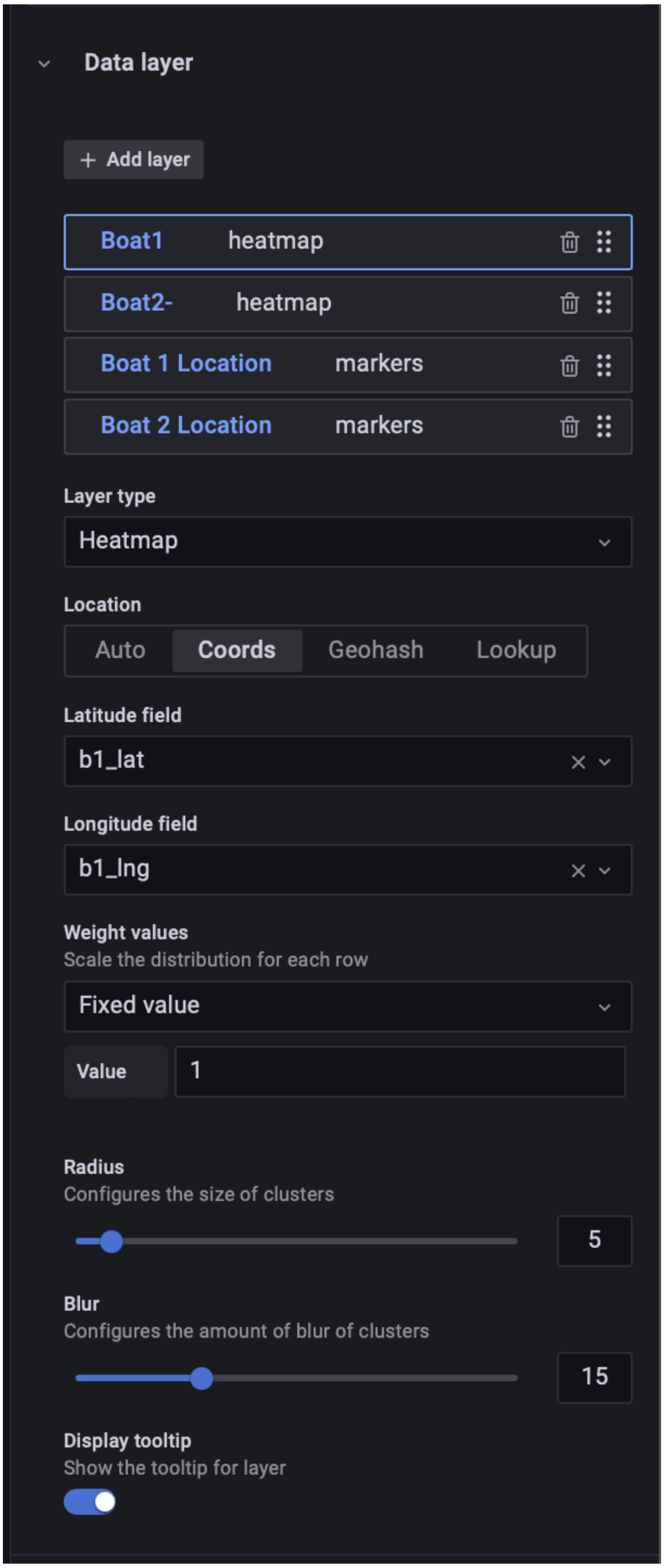
Data Layer:

- Layer 1: rename to Boat1
- Layer type: Heatmap
- Location: Coords
- Latitude field: b1_lat
- Longitude field: b1_lng
- Radius: 5
- Blur: 15

Click on “+ Add layer” to add another heat map layer to the data, this time using b2_lat and b2_long as the coordinates. We can also add a layer to show the precise locations with markers for both ships (using b1_lat, b1_lng, b2_lat and b2_long), setting each marker to a different color. For the Boat 1 and Boat 2 Locations, we use the following options:

Data Layer:

- Value: 1
- Color: select different color for each boat.



Boat1	heatmap
Boat2-	heatmap
Boat 1 Location	markers
Boat 2 Location	markers

Layer type: Heatmap

Location: Auto | **Coords** | Geohash | Lookup

Latitude field: b1_lat

Longitude field: b1_lng

Weight values
Scale the distribution for each row: Fixed value

Value: 1

Radius
Configures the size of clusters: 5

Blur
Configures the amount of blur of clusters: 15

Display tooltip
Show the tooltip for layer

Figure 2.14: Multiple layers in data layers dialogue box

The final visualization looks like the below.

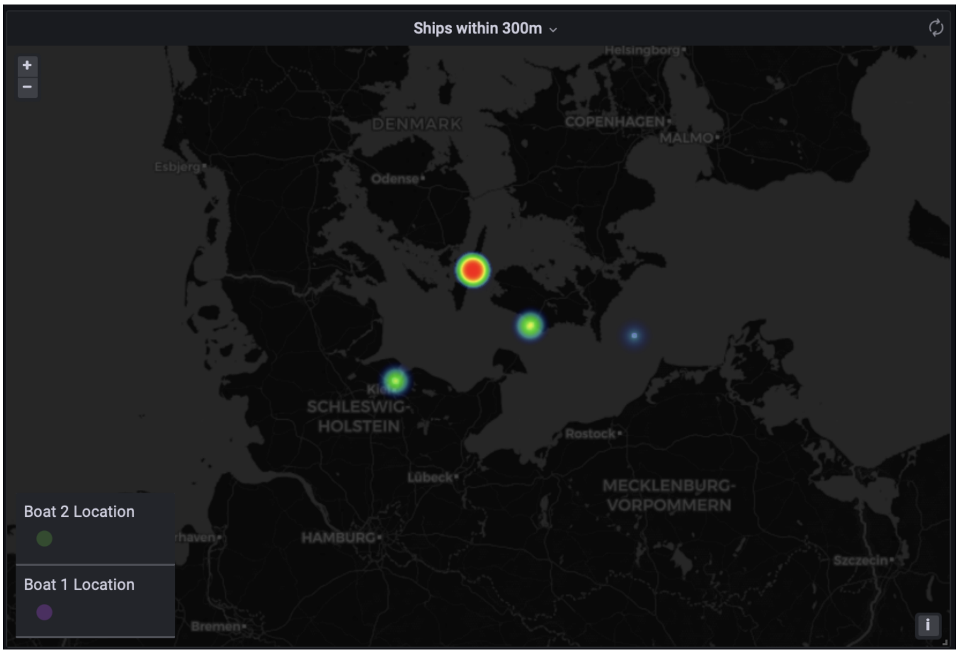


Figure 2.15: Visualization of ships within 300m using heat-map

It's helpful to include the tooltip for layers to allow users to see the data behind the visualization, which helps in interpretation and is a good way for subject-matter-experts to provide concrete feedback. Using the tooltip, we can quickly see that the same ship can be within 300m to multiple other ships in the same time frame (as seen in the screenshot below). This can result in a higher frequency of results in a heat map view than expected. SQL queries should be modified to ensure they are correctly interpreted.

Not surprisingly, we see there are lots of results for proximity within ports. We could avoid including results in ports by excluding all results that occur within envelopes defined by ST_MakeEnvelope, as seen in the previous queries.

Figure 2.16: Multiple results for the same ship at various times while in a port

2.7 Dynamic Dashboards - Creating Variables

We can use variables in Grafana to manipulate time-ranges that are used as inputs to MobilityDB queries. We'll create a drop-down type variable called **"FromTime"** that will be used as an input for the time period within which a query returns results.

1. In the dashboard window, click "Dashboard settings" icon; the gear symbol, on the top-slightly-right of the window.

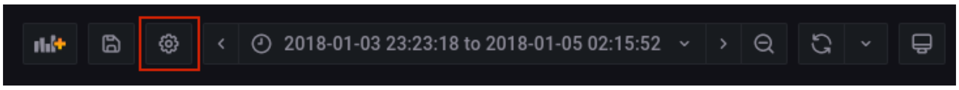


Figure 2.17: Dashboard settings gear box

2. Click on the "Variables" in the next window on the top-left side of the screen.

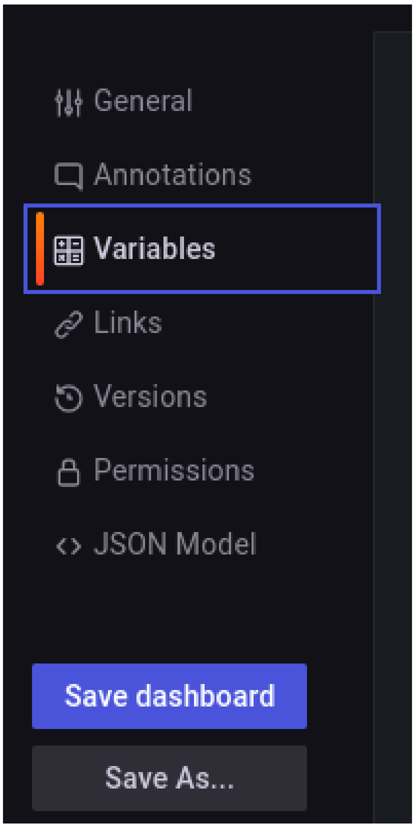
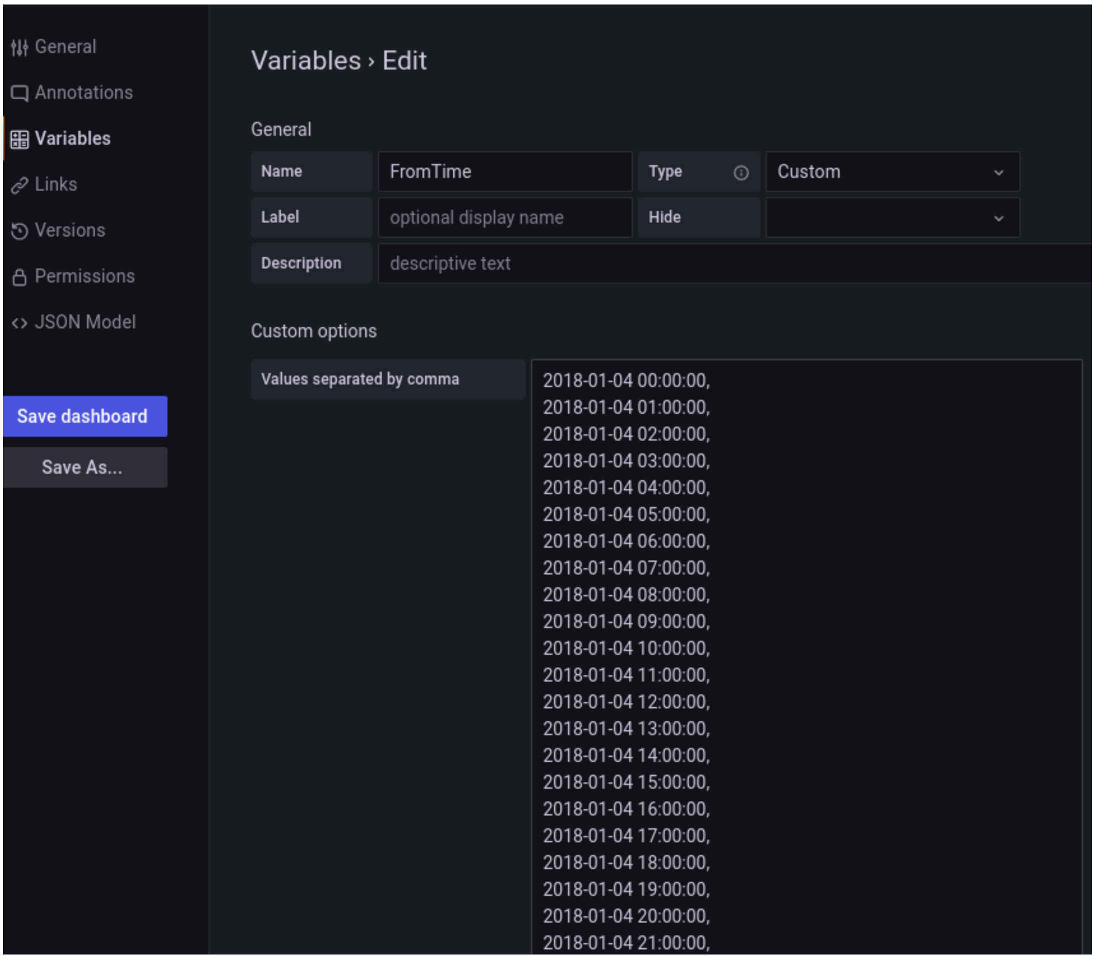


Figure 2.18: Selecting Variables in dashboard settings

3. You'll see a screen that explains the variables in Grafana and also points to the Templates and variables documentation. Click on the "Add variable" button.
4. In "General"
   - Name → FromTime
   - Type → Custom
5. In "Custom options" we will manually add all the time ranges with 1 hour increment. e.g. "2018-01-04 00:00:00, 2018-01-04 01:00:00 … 2018-01-04 23:00:00"

6. You get a screen like below. Towards the bottom there is also a "Preview of values" that shows what the drop-down options will look like for the variable you created. In this case, we are creating the timestamps in the same format that MobilityDB will accept.



Variables › Edit

General

Name: FromTime
Type: Custom
Label: optional display name
Hide:
Description: descriptive text

Custom options

Values separated by comma:

```
2018-01-04 00:00:00,
2018-01-04 01:00:00,
2018-01-04 02:00:00,
2018-01-04 03:00:00,
2018-01-04 04:00:00,
2018-01-04 05:00:00,
2018-01-04 06:00:00,
2018-01-04 07:00:00,
2018-01-04 08:00:00,
2018-01-04 09:00:00,
2018-01-04 10:00:00,
2018-01-04 11:00:00,
2018-01-04 12:00:00,
2018-01-04 13:00:00,
2018-01-04 14:00:00,
2018-01-04 15:00:00,
2018-01-04 16:00:00,
2018-01-04 17:00:00,
2018-01-04 18:00:00,
2018-01-04 19:00:00,
2018-01-04 20:00:00,
2018-01-04 21:00:00,
```

Figure 2.19: Creating user-defined list of custom variables

7. We can create another variable called "ToTime" with values shifted 1 hour. So the starting value would be "2018-01-04 01:00:00" and the final value will be "2018-01-05 00:00:00".

Now we can modify some queries by including the newly created variables which will return results from a specific time window. We have now provided a user with the ability to dynamically modify visualization queries and slice through time.

2.7.1 Dynamic Query: Number of Boats Moving Through a Given Area in a Certain Time Period

In the query code we just need to make slight changes for it to take time values from the variables. In the original query, shown below:

```
SELECT P.port_name,
      sum( numSequences( atGeometry( S.Trip, P.port_geom))) AS trips_intersect_with_port,
      p.lat,
      p.lng
FROM ports AS P, Ships AS S
WHERE intersects(S.Trip, P.port_geom)
GROUP BY P.port_name, P.lat, P.lng
```

We just need to modify the trips_intersect_with_port parameter in the SELECT statement to look like:

```
sum
(numSequences(atGeometry( atPeriod(S.Trip, period '[$FromTime, $ToTime)'), P.port_geom)))
    AS trips_intersect_with_port
```

Essentially we just wrapped “S.Trip” with “atPeriod()” and passed our custom period range. The full query with this modification is below:

```
-- Table with bounding boxes over regions of interest
WITH ports(port_name, port_geom, lat, lng)
       AS (SELECT p.port_name, p.port_geom, lat, lng
           FROM
             (VALUES ('Rodby',
                    ST_MakeEnvelope(651135, 6058230, 651422, 6058548, 25832)::geometry,
                    54.53, 11.06),
                   ('Puttgarden',
                    ST_MakeEnvelope(644339, 6042108, 644896, 6042487, 25832)::geometry,
                    54.64, 11.36)) AS p(port_name, port_geom, lat, lng))

SELECT P.port_name,
       sum(numSequences(atGeometry(atPeriod(S.Trip, period '[$FromTime, $ToTime)'),
                                   P.port_geom))) AS trips_intersect_with_port,
       p.lat,
       p.lng
FROM ports AS P,
     Ships AS S
WHERE intersects(S.Trip, P.port_geom)
GROUP BY P.port_name, P.lat, P.lng
```

We can select the start time, “FromTime” → “2018-01-04 02:00:00” & “ToTime” → “2018-01-04 06:00:00” . As we can see below, the port Rodby has less activity during this period and that’s why it is green now. But overall Rodby has more activity so when we look at the entire days data it is colored red.

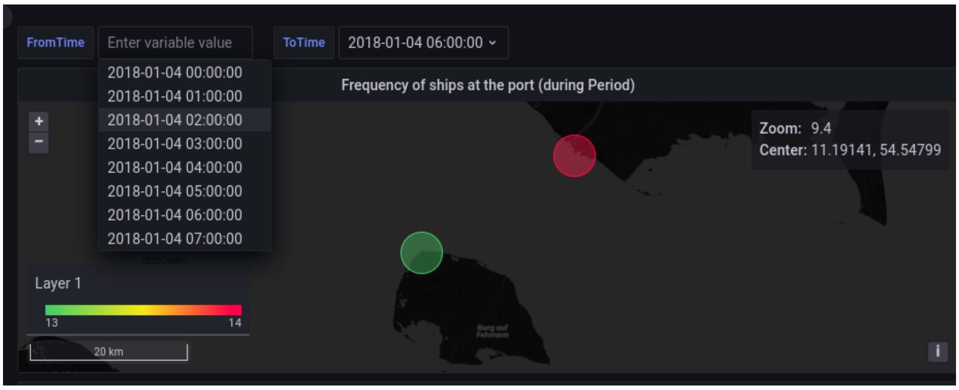

FromTime Enter variable value
ToTime 2018-01-04 06:00:00
Zoom: 9.4
Center: 11.19141, 54.54799

Figure 2.20: Visualization of geometry intersection using dynamic variables

2.7.2 Global Variables

Grafana also has some built-in variables (global variables) that can be used to accomplish the same thing we did with custom variables. We can use the global variables ${__from:date} and ${__to:date} instead of the $FromTime and $ToTime we created. The time range can then be modified with the time range options in the top right of the dashboard.

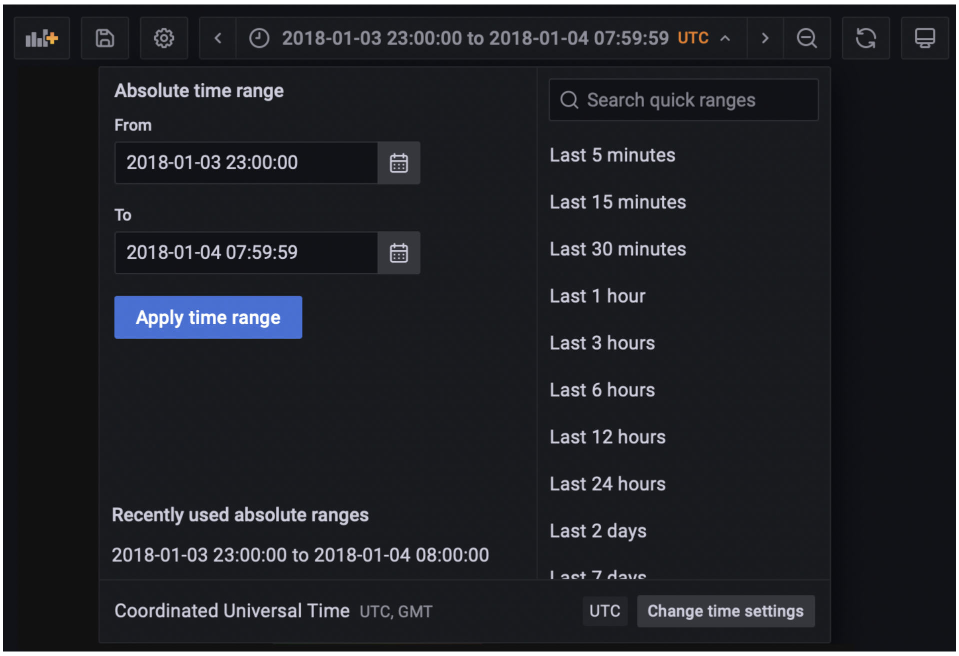
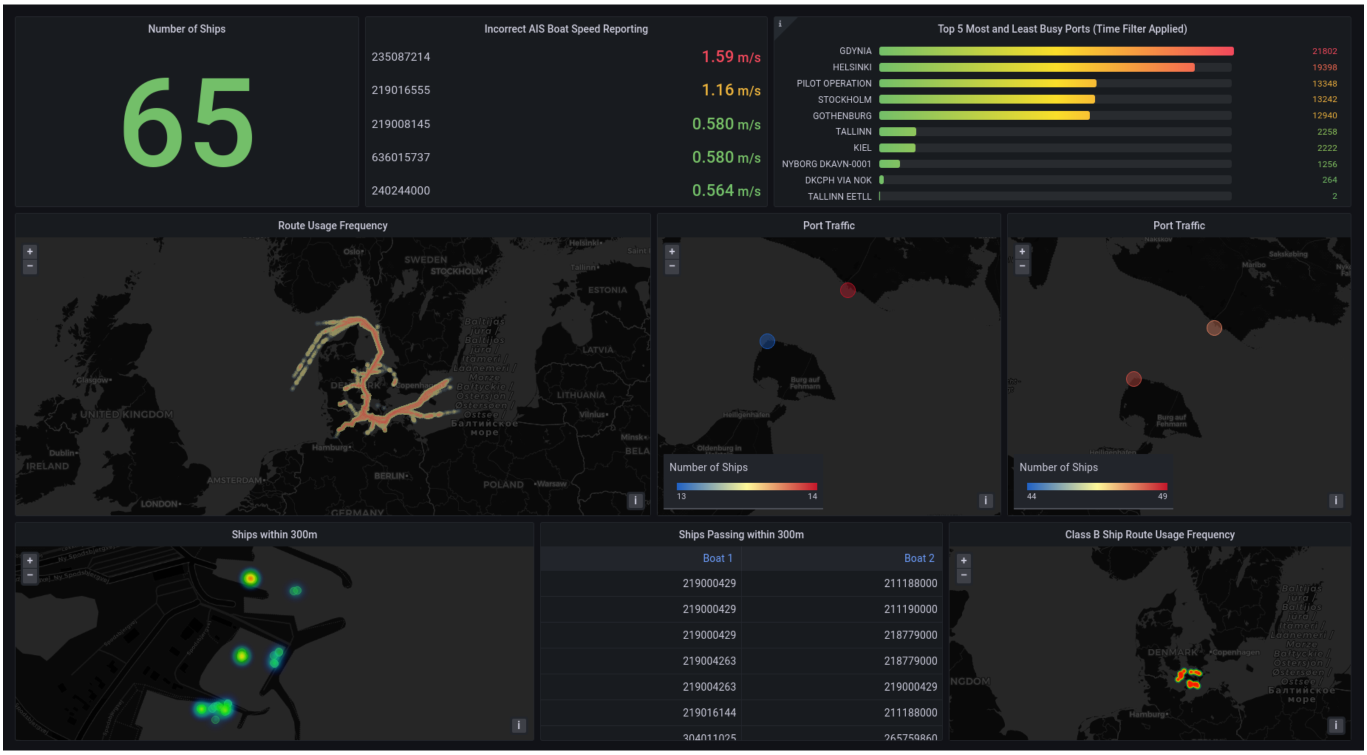


Figure 2.21: Assigning time range using global variables

Note: It is important to be aware of the timezone used for the underlying data relative to the queries for global variables. Time zones can be adjusted at the bottom of the time range selection, "Change time settings". For this example, we change the time zone to UTC to match our dataset.

2.8 Final Dashboard

The final dashboard will look like this. Note there are a couple additional query views that were not covered explicitly in the workshop.



Incorrect AIS Boat Speed Reporting

235087214	1.59 m/s
219016555	1.16 m/s
219008145	0.580 m/s
636015737	0.580 m/s
240244000	0.564 m/s

Top 5 Most and Least Busy Ports (Time Filter Applied)

GDYNIA	21802
HELSINKI	19398
PILOT OPERATION	13348
STOCKHOLM	13242
GOTHENBURG	12940
TALLINN	2258
KIEL	2222
NYBORG DKAVN-0001	1256
DKCPH VIA NOK	264
TALLINN EETLL	2

Ships Passing within 300m

Boat 1	Boat 2
219000429	211188000
219000429	211190000
219000429	218779000
219004263	218779000
219004263	219000429
219016144	211188000
304011025	265759860

Figure 2.22: Full Dashboard

Chapter 3

Managing Flight Data and Creating Dashboard with Grafana

3.1 Contents

The module covers the following topics in 3 parts:

Part 1 - Data and Environment Preparation

- Preparing the Database
- Data Cleaning
- Setting up the Dashboard and Connecting to Data Source

Part 2 - Working with Discrete Points

- Visualizing time-series data for a single airplane
- Visualizing discrete geographic points on a map

Part 3 - Working with Continuous Trajectories in MobilityDB

- Creating trajectories for individual flights
- Visualizing statistics from temporal aggregations
- Visualizing statistics from multiple queries returning temporal aggregations
- Returning value changes from temporal data
- Visualizing spatial statistics from nested temporal conditions (intrinsic and dynamic)

3.2 Tools

The tools used in this module are as follows:

- MobilityDB, on top of PostgreSQL and PostGIS
- Grafana (version 9.0.7)

3.3 Part 1 - Data and Environment Preparation

3.3.1 Preparing the Database

Dataset link

Create a new database “opensky”, then use your SQL editor to create the MobilityDB extension as follows:

```
CREATE EXTENSION MobilityDB CASCADE;
```

The CASCADE command will additionally create the PostGIS extension.

Now create a table in which the CSV file will be loaded:

```
CREATE TABLE flights(
    et              bigint,
    icao24          varchar(20),
    lat             float,
    lon             float,
    velocity        float,
    heading         float,
    vertrate        float,
    callsign        varchar(10),
    onground        boolean,
    alert           boolean,
    spi             boolean,
    squawk          integer,
    baroaltitude    numeric(7,2),
    geoaltitude     numeric(7,2),
    lastposupdate   numeric(13,3),
    lastcontact     numeric(13,3)
);
```

Load the data into the database using the following command. Replace the `<path_to_file>` with the actual path of the CSV file. Do this for all files.

```
COPY flights(et, icao24, lat, lon, velocity, heading,
     vertrate, callsign, onground, alert, spi, squawk,
     baroaltitude, geoaltitude, lastposupdate, lastcontact)
FROM '<path_to_file>' DELIMITER  ',' CSV HEADER;
```

All the times in this dataset are in unix timestamp (an integer) with timezone being UTC. So we need to convert them to PostgreSQL timestamp type.

```
ALTER TABLE flights
    ADD COLUMN et_ts timestamp,
    ADD COLUMN lastposupdate_ts timestamp,
    ADD COLUMN lastcontact_ts timestamp;

UPDATE flights
    SET et_ts = to_timestamp(et),
        lastposupdate_ts = to_timestamp(lastposupdate),
        lastcontact_ts = to_timestamp(lastcontact);
```

You can check the size of the database with:

```
SELECT pg_size_pretty( pg_total_relation_size('flights') );
```

3.3.2 Data Cleaning

Delete all icao24 that have all NULL latitudes

```
-- icao24_with_null_lat is used to provide a list of rows that will be deleted
WITH icao24_with_null_lat AS (
    SELECT icao24, COUNT(lat)
    FROM flights
    GROUP BY icao24
    HAVING COUNT(lat) = 0
      )
DELETE
FROM flights
WHERE icao24 IN
-- this SELECT statement is needed for the IN statement to compare against a list
(SELECT icao24 FROM icao24_with_null_lat);
```

Note: This data cleaning is not comprehensive. It was just to highlight that before creating trajectories, it may be very important to have a look at the data and do some cleaning as that will directly impact the quality of mobilityDB trajectories being created. If there as NULLs in mobilityDB trajectories, some operation on it can give error.

3.3.3 Setting up the Dashboard and Connecting to Data Source

Data for the workshop is loaded into a MobilityDB database hosted on Azure, with all login information provided in the Sign-in and Connect to Data Source section below.

The workshop is using the following settings in Grafana to connect to the postgres server on Azure. More detailed instruction to set up Grafana can be found in section 2.3 to 2.5 of the Dashboard and Visualization of Ship Trajectories (AIS) workshop.

- Name: OpenSkyLOCAL
- Host: *20.79.254.53:5432*
- Database: opensky
- User: *mobilitydb-guest*
- Password: *mobilitydb@guest*
- TLS/SSL Mode: *disable*
- Version: *12+*

The data used for this workshop provided by The OpenSky Network. This is data from a 24hr period from June 1, 2020 (dataset link). The raw data is originally provided in separate CSV documents for each hour of the day.

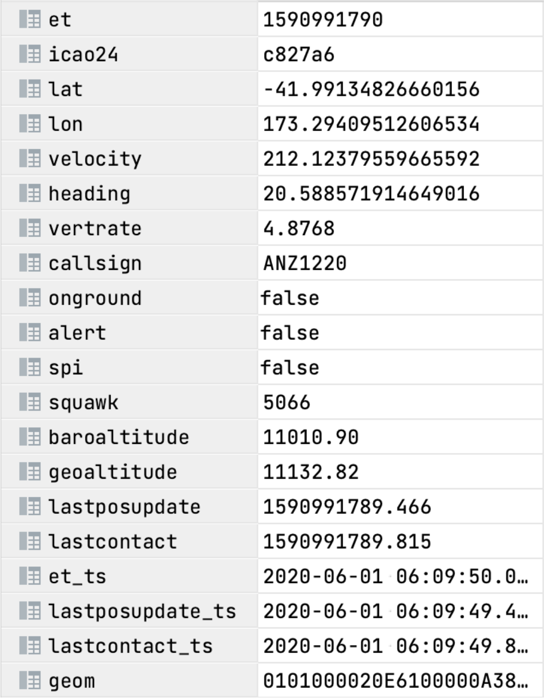
Open a new browser and go to http://localhost:3000/ to work in your instance of Grafana. With a new dashboard we can start creating the panels below.

3.4 Part 2 - Working with Discrete Points

3.4.1 Visualizing 24hr Flight Pattern of Single Airplane

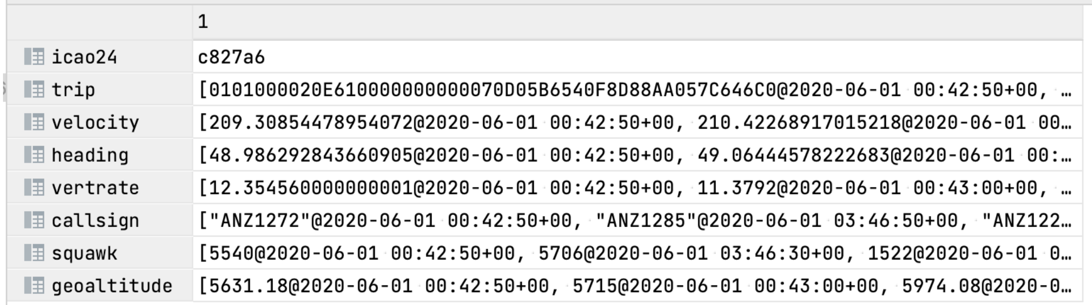
We will start by looking at a single airplane. Grafana proves to be a good way to quickly visualize our dataset and can be useful to support pre-processing and cleaning. If using a connection to the Azure database, required tables are already created.

A full description of each parameter is included in the OpenSky original dataset readme. The table structure in the Azure dataset after loading and transformations looks like the following:



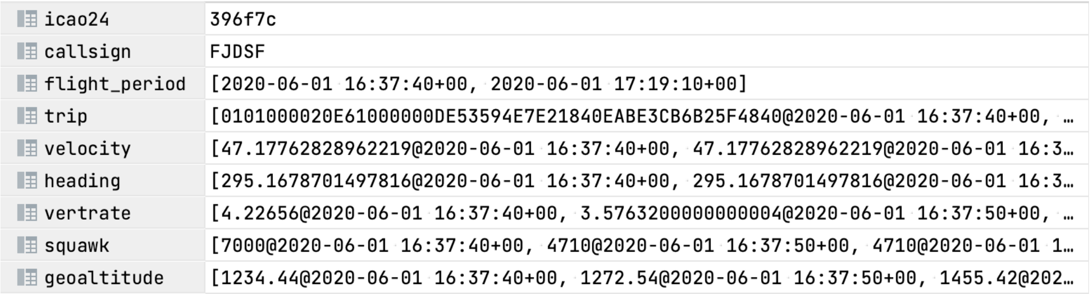
et	1590991790
icao24	c827a6
lat	-41.99134826660156
lon	173.29409512606534
velocity	212.12379559665592
heading	20.588571914649016
vertrate	4.8768
callsign	ANZ1220
onground	false
alert	false
spi	false
squawk	5066
baroaltitude	11010.90
geoaltitude	11132.82
lastposupdate	1590991789.466
lastcontact	1590991789.815
et_ts	2020-06-01 06:09:50.0…
lastposupdate_ts	2020-06-01 06:09:49.4…
lastcontact_ts	2020-06-01 06:09:49.8…
geom	0101000020E6100000A38…

Figure 3.1: First row of table “single_airframe”, with 24hrs of flight information for airplane “c827a6”



	1
icao24	c827a6
trip	[0101000020E610000000000070D05B6540F8D88AA057C646C0@2020-06-01 00:42:50+00, …
velocity	[209.30854478954072@2020-06-01 00:42:50+00, 210.42268917015218@2020-06-01 00…
heading	[48.986292843660905@2020-06-01 00:42:50+00, 49.06444578222683@2020-06-01 00:…
vertrate	[12.354560000000001@2020-06-01 00:42:50+00, 11.3792@2020-06-01 00:43:00+00, …
callsign	["ANZ1272"@2020-06-01 00:42:50+00, "ANZ1285"@2020-06-01 03:46:50+00, "ANZ122…
squawk	[5540@2020-06-01 00:42:50+00, 5706@2020-06-01 03:46:30+00, 1522@2020-06-01 0…
geoaltitude	[5631.18@2020-06-01 00:42:50+00, 5715@2020-06-01 00:43:00+00, 5974.08@2020-0…

Figure 3.2: Full table “single_airframe_traj” for airplane “c827a6” with data in mobilityDB trajectories format



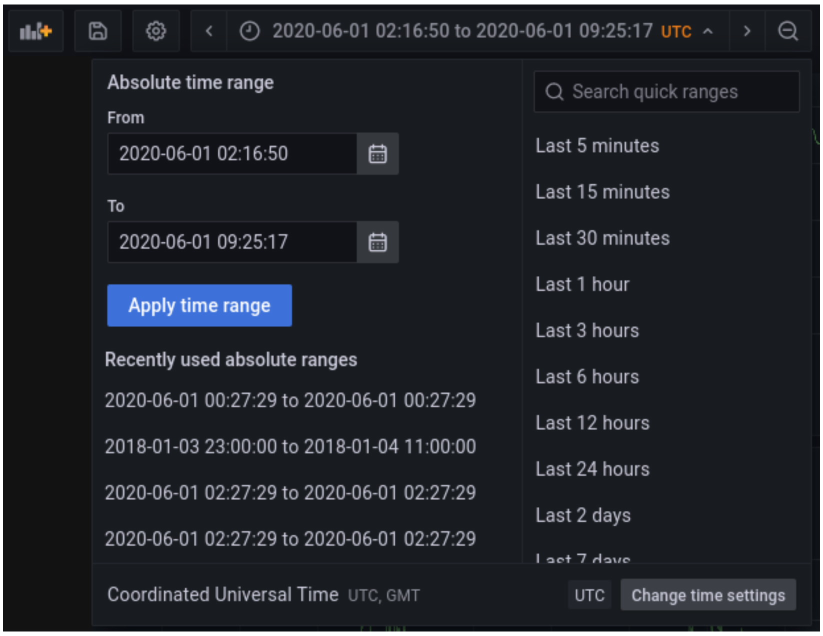
icao24	396f7c
callsign	FJDSF
flight_period	[2020-06-01 16:37:40+00, 2020-06-01 17:19:10+00]
trip	[0101000020E61000000DE53594E7E21840EABE3CB6B25F4840@2020-06-01 16:37:40+00, …
velocity	[47.17762828962219@2020-06-01 16:37:40+00, 47.17762828962219@2020-06-01 16:3…
heading	[295.1678701497816@2020-06-01 16:37:40+00, 295.1678701497816@2020-06-01 16:3…
vertrate	[4.22656@2020-06-01 16:37:40+00, 3.5763200000000004@2020-06-01 16:37:50+00, …
squawk	[7000@2020-06-01 16:37:40+00, 4710@2020-06-01 16:37:50+00, 4710@2020-06-01 1…
geoaltitude	[1234.44@2020-06-01 16:37:40+00, 1272.54@2020-06-01 16:37:50+00, 1455.42@202…

Figure 3.3: First row of table “flight_traj_sample”, which includes 200 flight trajectories.

3.4.1.1 Change Timezone in Grafana

Make Sure you are visualizing the data in the correct timezone. The data we had was in UTC. To change the timezone,

1. Click on the time-range panel on the top-right of the window.



2020-06-01 02:16:50 to 2020-06-01 09:25:17 UTC

Absolute time range

From: 2020-06-01 02:16:50

To: 2020-06-01 09:25:17

Recently used absolute ranges

2020-06-01 00:27:29 to 2020-06-01 00:27:29

2018-01-03 23:00:00 to 2018-01-04 11:00:00

2020-06-01 02:27:29 to 2020-06-01 02:27:29

2020-06-01 02:27:29 to 2020-06-01 02:27:29

Coordinated Universal Time UTC, GMT

Figure 3.4: Grafana time range panel

2. In the pop-up window, on the bottom there is “Change time settings”. Click that to set the desired timezone.

3.4.1.2 Visualize the Coordinates of a Single Airplane

Let’s visualize the latitude and longitude coordinates of an airplane’s journey throughout the day. For this one we will not color the geo-markers, but it is possible to color them using some criterion.

1. Add a new panel
2. Select “OpenSkyLOCAL” as the data source
3. In Format as, change “Time series” to “Table” and choose “Edit SQL”

4. Here you can add your SQL queries. Let's replace the existing query with the following SQL script:

```
-- icao24 is the unique identifier for each airframe (airplane)
SELECT et_ts, icao24, lat, lon
-- TABLESAMPLE SYSTEM (n) returns only n% of the data from the table.
FROM flights TABLESAMPLE SYSTEM (5)
WHERE icao24 IN ('738286') AND $__timeFilter(et_ts)
```

5. Change the visualization type to "*Geomap"*.
6. The options (visualization settings - on the right side of the screen) should be as follows:

   Panel Options

   - Title →GPS location over time

   Map View

   - Initial view: For this one zoom in on the visualization on the panel as you see fit and then click "use current map settings " button.

   Data Layer

   - Layer type: → "markers"
   - Style size → Fixed Value: 2
   - Color → Green

In this visualization we can see that the airplane is visiting different countries and almost completing a loop. This indicates that there are more than 1 trips (flights) completed by this single airplane. The coordinates are sparse because we are sampling the results using "TABLESAMPLE SYSTEM (5)" in our query. This is done to speed up the visualization.

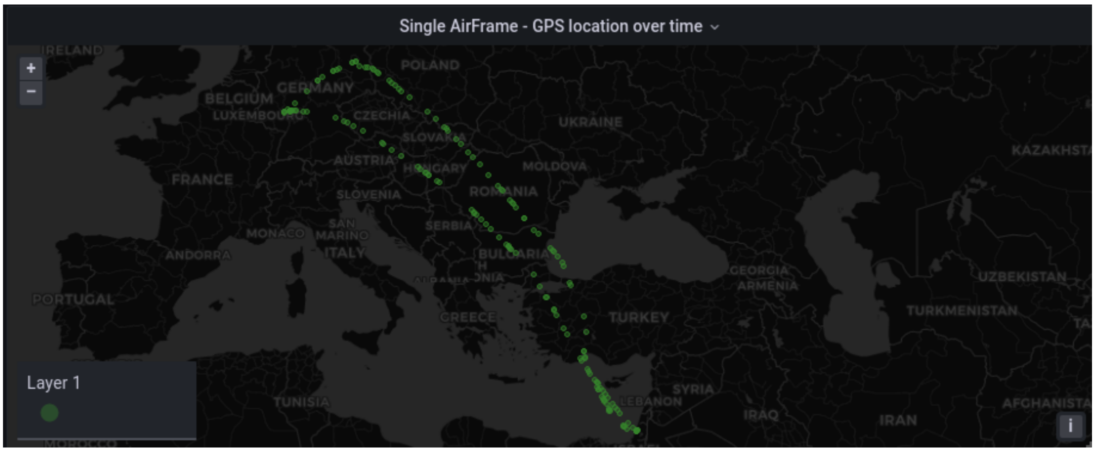


Figure 3.5: Single airframe geopoints vs time

3.4.2 Time-series Graphs for a Single Airplane

3.4.2.1 Velocity vs Time

Following the similar steps to add a Geomap panel as before, we include the following SQL script. Note $__timeFilter() is a Grafana global variable. This global variable will inject time constraint SQL-conditions from Grafana's time range panel.

1. In Format as, use "Time series"

```
SELECT
  et_ts AS "time",
  velocity
FROM flights
WHERE icao24 = 'c827a6' AND $__timeFilter(et_ts)
```

1. Change the visualization type to “Time Series”.
2. The options (visualization settings - on the right side of the screen) should be as follows:

   Panel Options

   - Title → Single AirFrame - Velocity vs Time

In the visualization we can see clearly that on this day, this airframe took 3 flights. That is why its speed curve has 3 humps. The zero speed towards the end of each hump is a clear indicator that plane stopped, thus it must have completed its flight.

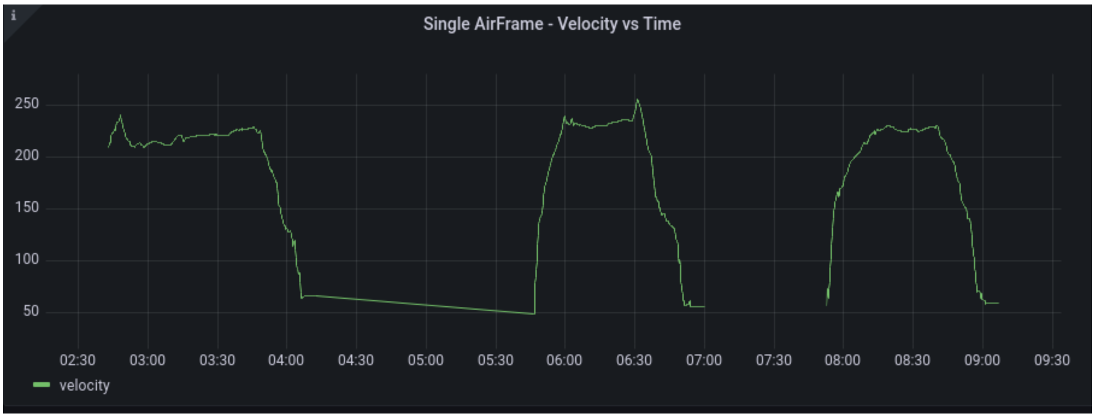


Figure 3.6: Single airframe velocity vs time

3.4.2.2 Altitude vs Time

Follow the similar steps to add a Geomap panel as before, we include the following SQL script.

1. In Format as, we have “Time series”

```
SELECT
  et_ts AS "time",
  baroaltitude, geoaltitude
FROM flights
WHERE icao24 = 'c827a6' AND $__timeFilter(et_ts)
```

1. Change the visualization type to “Time Series”.
2. The options (visualization settings - on the right side of the screen) should be as follows:

   Panel Options

   - Title → Single AirFrame - Altitude vs Time

In the visualization we can again see that on this day, the airframe took 3 flights, as altitude reaches zero between each flight. There is some noise in the data, which appear as spikes. This would be almost impossible to spot in a tabular format, but on a line graph these data anomalies can be easily identified.

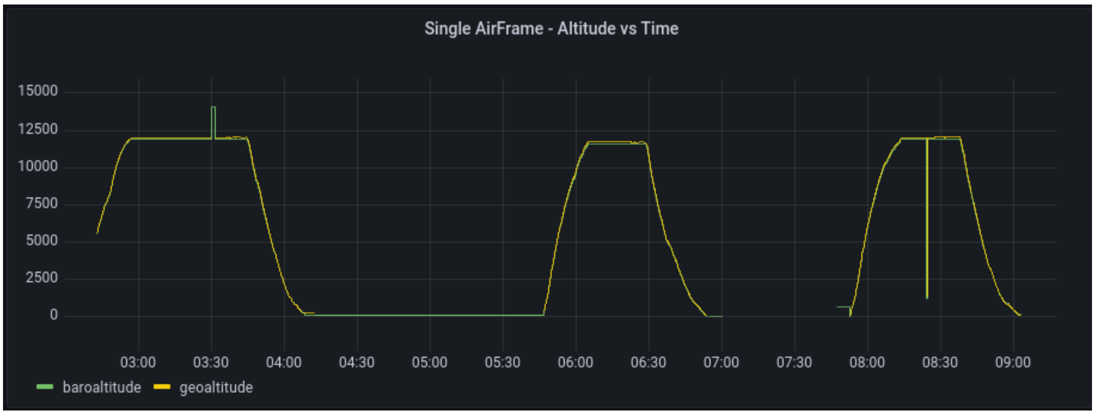


Figure 3.7: Single airframe altitude vs time

3.4.2.3 Vertical-Rate vs Time

Follow the similar steps to add a Geomap panel as before, we include the following SQL script.

1. In Format as, we have "Time series"

```
SELECT
  et_ts AS "time",
  vertrate
FROM flights
WHERE icao24 = 'c827a6' AND $__timeFilter(et_ts)
```

1. Change the visualization type to "Time Series".
2. The options (visualization settings - on the right side of the screen) should be as follows:

   Panel Options

   - Title → Single AirFrame - Verticle-Rate vs Time

The positive values here represents the ascent of the plane. While at cruising altitude, the plane has almost zero vertical-rate and during decent this value becomes negative. So a sequence of positive values, then zero values followed by negative values would represent a single flight.

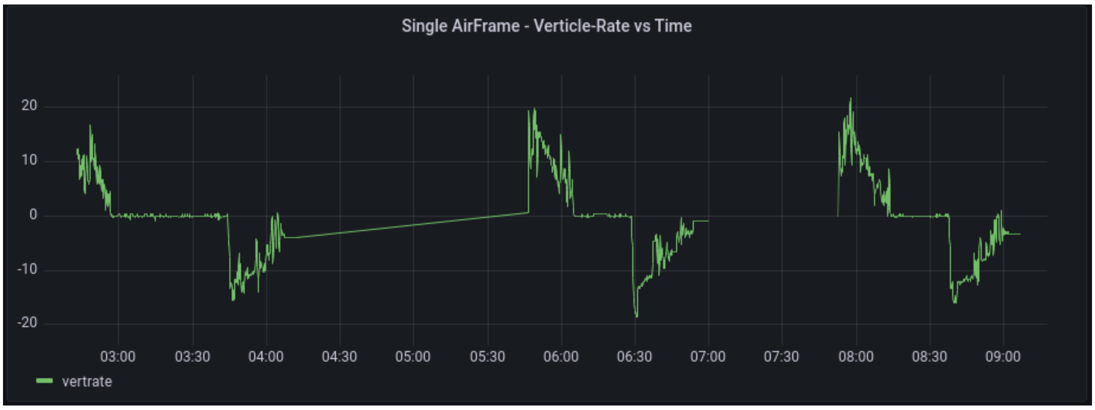


Figure 3.8: Single airframe vertrate vs time

3.4.2.4 Callsign vs Time

The callsign is a unique identifier used for a specific flight path. For example, ANZ1220 is the callsign of the Air New Zealand flight 1220 from Queenstown to Auckland in New Zealand. It is possible for single airplane to make the same flight more than once in a 24hr period if it goes back and forth. This information will be used in later queries to partition an airplanes data into multiple flights.

We can find the time at which the callsign of an airplane changes with the following steps.

1. In Format as, we have “Table”

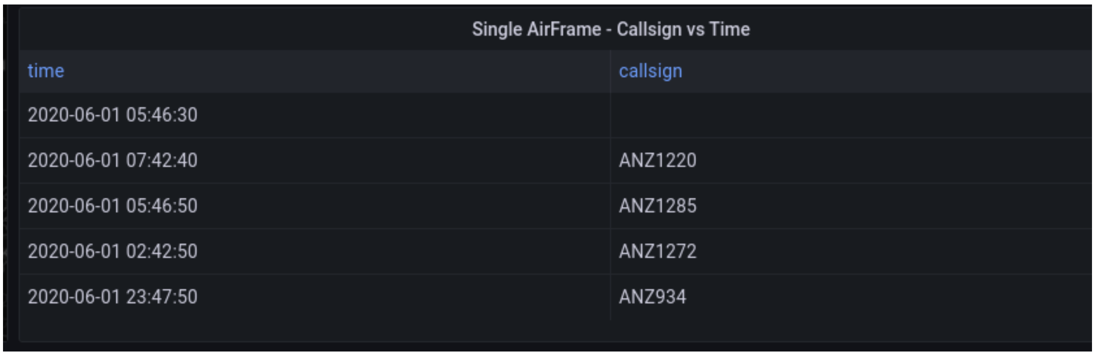
```
SELECT
  min(et_ts) AS "time", callsign
FROM flights
WHERE icao24 = 'c827a6'
GROUP BY callsign
```

1. Change the visualization type to “Table”.
2. The options (visualization settings - on the right side of the screen) should be as follows:

   Panel Options

   - Title → Single AirFrame - Callsign vs Time

In the visualization we can see that this airplane completed 3 flights and started the 4th one towards the very end of the day. We also see there is some NULL data in the callsign column which is why the first timestamp doesn’t have a corresponding callsign.


Single AirFrame - Callsign vs Time

time	callsign
2020-06-01 05:46:30	
2020-06-01 07:42:40	ANZ1220
2020-06-01 05:46:50	ANZ1285
2020-06-01 02:42:50	ANZ1272
2020-06-01 23:47:50	ANZ934

Figure 3.9: Single airframe callsign vs time

3.5 Part 3 - Working with Continuous Trajectories in MobilityDB

For the following queries, we will make use of trajectories for aggregation and creating effective splits in our data based on parameters that change in time.

3.5.1 Creating MobilityDB Trajectories

This step is completed once, only on the ingestion of data. It is shown below to provide an understanding of how to do it. With temporal datatypes and mobilityDB functionality, we can see the queries are very intuitive to create.

We first create a geometry point. This treats each latitude and longitude as a point in space. 4326 is the SRID.

```
ALTER TABLE flights
    ADD COLUMN geom geometry(Point, 4326);

UPDATE flights SET
  geom = ST_SetSRID( ST_MakePoint( lon, lat ), 4326);
```

3.5.1.1 AirFrame Trajectories

Now we are ready to construct airframe or airplane trajectories out of their individual observations. Each “icao24” in our dataset represents a single airplane.

We can create a composite index on icao24 (unique to each plane) and et_ts (timestamps of observations) to help improve the performance of trajectory generation.

```
CREATE INDEX icao24_time_index
    ON flights (icao24, et_ts);
```

We create trajectories for a single airframe because:

- this query serves as a simple example of how to use mobilityDB to create trajectories
- these kind of trajectories can be very important for plane manufacturer, as they are interested in the airplane’s analysis.
- we are creating the building blocks for future queries. Each row would represent a single flight, where flight is identified by icao24 & callsign.

```
CREATE TABLE airframe_traj(icao24, trip, velocity, heading, vertrate, callsign, squawk,
                           geoaltitude) AS
SELECT icao24,
       tgeompoint_seq(array_agg(tgeompoint_inst(geom, et_ts) ORDER BY et_ts)
                    FILTER (WHERE geom IS NOT NULL)),
       tfloat_seq(array_agg(tfloat_inst(velocity, et_ts) ORDER BY et_ts)
                FILTER (WHERE velocity IS NOT NULL)),
       tfloat_seq(array_agg(tfloat_inst(heading, et_ts) ORDER BY et_ts)
                FILTER (WHERE heading IS NOT NULL)),
       tfloat_seq(array_agg(tfloat_inst(vertrate, et_ts) ORDER BY et_ts)
                FILTER (WHERE vertrate IS NOT NULL)),
       ttext_seq(array_agg(ttext_inst(callsign, et_ts) ORDER BY et_ts)
               FILTER (WHERE callsign IS NOT NULL)),
       tint_seq(array_agg(tint_inst(squawk, et_ts) ORDER BY et_ts)
              FILTER (WHERE squawk IS NOT NULL)),
       tfloat_seq(array_agg(tfloat_inst(geoaltitude, et_ts) ORDER BY et_ts)
```

```
                  FILTER (WHERE geoaltitude IS NOT NULL))
FROM flights
GROUP BY icao24;
```

Here we create a new table for all the trajectories. We select all the attributes of interest that change over time. We can follow the transformation from the inner call to the outer call:

- tgeompoint_inst: combines each geometry point(lat, long) with the timestamp where that point existed
- array_agg: aggregates all the instants together into a single array for each item in the group by. In this case, it will create an array for each icao24
- tgeompoint_seq: constructs the array as a sequence which can be manipulated with mobilityDB functionality. The same approach is used for each trajectory, with the function used changing depending on the datatype.

3.5.1.2 Flight Trajectories

Right now we have, in a single row, an airframe's (where an airframe is a single physical airplane) entire day's trip information. We would like to segment that information per flight (an airframe flying under a specific callsign). This query segments the airframe trajectories (in temporal columns) based on the time period of the callsign. Below we explain the query and the reason behind segmenting the data this way.

```
-- Each row from airframe will create a new row in flight_traj depending on when the
-- callsign changes, regardless of whether a plane repeats the same flight multiple
-- times in any period

-- Airplane123 (airframe_traj) |-------------------------|
-- Flightpath1 (flight_traj)   |-----|
-- Flightpath2 (flight_traj)         |--------|
-- Flightpath1 (flight_traj)                  |-------|
-- Flightpath3 (flight_traj)                          |--|
```

```
CREATE TABLE flight_traj(icao24, callsign, flight_period, trip, velocity, heading,
                         vertrate, squawk, geoaltitude)
AS
    -- callsign sequence unpacked into rows to split all other temporal sequences.
WITH airframe_traj_with_unpacked_callsign AS
         (SELECT icao24,
                 trip,
                 velocity,
                 heading,
                 vertrate,
                 squawk,
                 geoaltitude,
                 startValue(unnest(segments(callsign))) AS start_value_callsign,
                 unnest(segments(callsign))::period     AS callsign_segment_period
          FROM airframe_traj)

SELECT icao24                                         AS icao24,
       start_value_callsign                           AS callsign,
       callsign_segment_period                        AS flight_period,
       atPeriod(trip, callsign_segment_period)        AS trip,
       atPeriod(velocity, callsign_segment_period)    AS velocity,
       atPeriod(heading, callsign_segment_period)     AS heading,
       atPeriod(vertrate, callsign_segment_period)    AS vertrate,
       atPeriod(squawk, callsign_segment_period)      AS squawk,
       atPeriod(geoaltitude, callsign_segment_period) AS geoaltitude
FROM airframe_traj_with_unpacked_callsign;
```

Note: We could have tried to create the above (table ”flight_traj”) per flight trajectories by simply including “callsign” in the GROUP BY statement in the query used to create the previous airframe_traj table (`GROUP BY icao24, callsign;`).

The **problem** with this solution: This approach would put the trajectory data of 2 distinct flights where that airplane and flight number are the same in a single row, which is not correct.

MobilityDB functions helped us avoid the use of several hardcoded conditions that depend on user knowledge of the data. This approach is very generic and can be applied anytime we want to split a trajectory by the inflection points in time of some other trajectory.

3.5.2 Aggregating Flight Statistics

We can now use our trajectories to pull flight specific statistics very easily.

3.5.2.1 Average Velocity of Each Flight

1. In Format as, we have “Table”

```
-- Average flight speeds during flight
SELECT callsign,twavg(velocity) AS average_velocity
FROM flight_traj
WHERE twavg(velocity)IS NOT NULL -- drop rows without velocity data
AND twavg(velocity) < 1500 -- removes erroneous data
ORDER BY twavg(velocity) desc;
```

2. Change the visualization type to “Bar gauge”.

3. The options (visualization settings - on the right side of the screen) should be as follows

   Panel Options

   - Title → Average Flight Speed

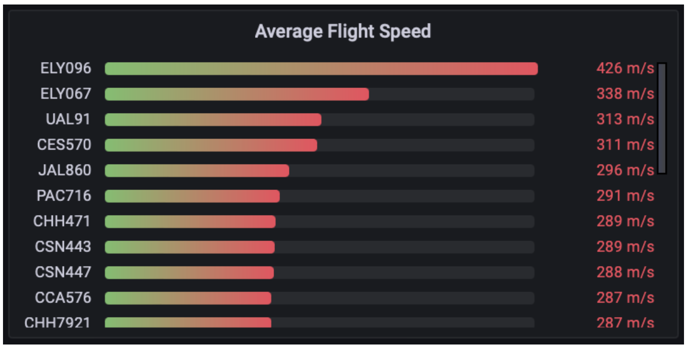
   Bar gauge

   - Orientation → Horizontal

   Standard Options

   - Unit → meters/second (m/s)
   - Min → 200

   The settings we adjust improve the visualization by cutting the bar graph values of 0-200, improving the resolution at higher ranges to see differences.



Average Flight Speed

ELY096	426 m/s
ELY067	338 m/s
UAL91	313 m/s
CES570	311 m/s
JAL860	296 m/s
PAC716	291 m/s
CHH471	289 m/s
CSN443	289 m/s
CSN447	288 m/s
CCA576	287 m/s
CHH7921	287 m/s

Figure 3.10: Average flight speed visualization

3.5.2.2 Number of Private and Commercial Flights

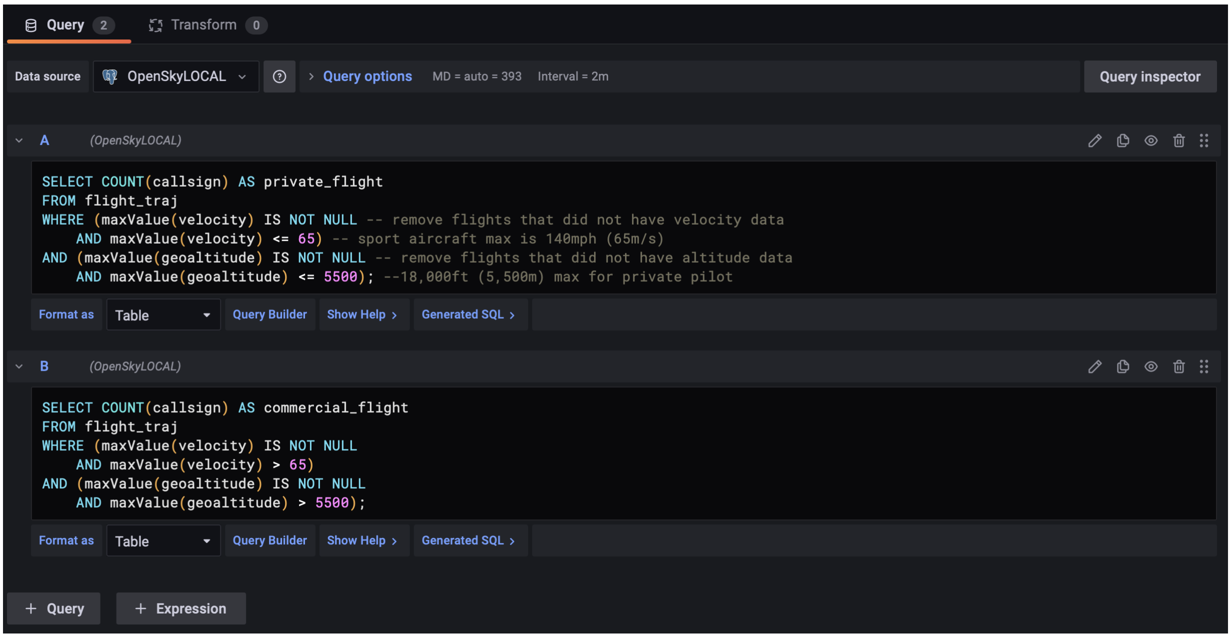
We can easily combine results from multiple queries in the same visualization in Grafana, simplifying the queries themselves. Here we apply some domain knowledge of sport pilot aircraft license limits for altitude and speed to provide an estimated count of each.

1. In Format as, we have “Table”

```sql
-- Flights completed by private pilots (estimate)
SELECT COUNT(callsign) AS private_flight
FROM flight_traj
WHERE (maxValue(velocity) IS NOT NULL -- remove flights without velocity
    AND maxValue(velocity) <= 65) -- sport aircraft max is 140mph (65m/s)
AND (maxValue(geoaltitude) IS NOT NULL -- remove flights without altitude
    AND maxValue(geoaltitude) <= 5500); --18,000ft (5,500m) max for private pilot

-- Count of commercial flights (estimate)
SELECT COUNT(callsign) AS commercial_flight
FROM flight_traj
WHERE (maxValue(velocity) IS NOT NULL
    AND maxValue(velocity) > 65)
AND (maxValue(geoaltitude) IS NOT NULL
    AND maxValue(geoaltitude) > 5500);
```

   In Grafana, when we are in the query editor we can click on “+ Query” at the bottom to add multiple queries that provide different results.



Data source: OpenSkyLOCAL

MD = auto = 393 Interval = 2m

A (OpenSkyLOCAL)

```sql
SELECT COUNT(callsign) AS private_flight
FROM flight_traj
WHERE (maxValue(velocity) IS NOT NULL -- remove flights that did not have velocity data
    AND maxValue(velocity) <= 65) -- sport aircraft max is 140mph (65m/s)
AND (maxValue(geoaltitude) IS NOT NULL -- remove flights that did not have altitude data
    AND maxValue(geoaltitude) <= 5500); --18,000ft (5,500m) max for private pilot
```

Format as: Table

B (OpenSkyLOCAL)

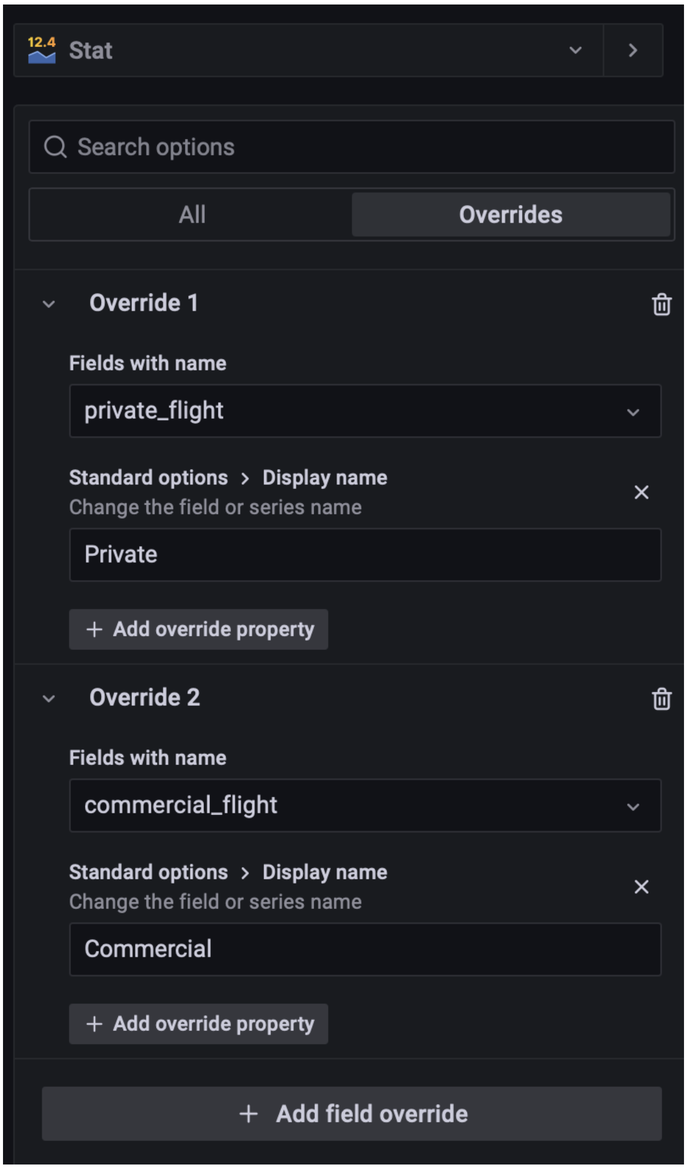
```sql
SELECT COUNT(callsign) AS commercial_flight
FROM flight_traj
WHERE (maxValue(velocity) IS NOT NULL
    AND maxValue(velocity) > 65)
AND (maxValue(geoaltitude) IS NOT NULL
    AND maxValue(geoaltitude) > 5500);
```

Format as: Table

Figure 3.11: Multiple queries providing results for a single visualization

2. Change the visualization type to “Stat”.

   To label the data for each result separately, choose “Overrides” at the top of the options panel on the right. Here you can override global panel settings for specific attributes as shown below.



Override 1

Fields with name: private_flight

Standard options > Display name
Change the field or series name

Private

Override 2

Fields with name: commercial_flight

Standard options > Display name
Change the field or series name

Commercial

Figure 3.12: Override options for panel with multiple queries

The final statistics visualization will look like this:

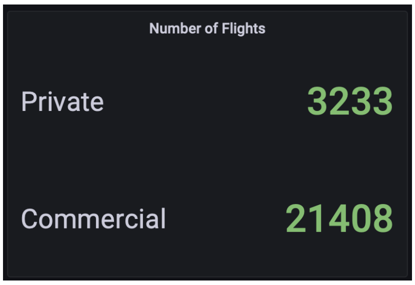


Figure 3.13: Statistic visualization of number of flights by license type

3.5.3 Flights Taking-off in Some Interval of Time (User-Defined)

Note: This query makes used of a sample set of data that has 200 flights to return results. “flight_traj_sample” is just a sampled version of “flight_traj”. As of the writing of this workshop, Grafana does not support display of vectors, and so individual latitude and longitude points are used as a proxy.

In order to make the query use Grafana global time range panel replace the hard-coded timestamps with the ‘[${__from:date}, ${__to:date})’.

```
WITH
-- The flight_traj_time_slice CTE is clipping all the temporal columns
-- to the user specified time-range.
flight_traj_time_slice (icao24, callsign, time_slice_trip, time_slice_geoaltitude,
                        time_slice_vertrate) AS
    (SELECT icao24,
            callsign,
            atPeriod(trip, period '[2020-06-01 03:00:00, 2020-06-01 20:30:00)'),
            atPeriod(geoaltitude, period '[2020-06-01 03:00:00, 2020-06-01 20:30:00)'),
            atPeriod(vertrate,
                     period '[2020-06-01 03:00:00, 2020-06-01 20:30:00)')
     FROM flight_traj_sample TABLESAMPLE SYSTEM (20)),

-- There are 3 things happening in the flight_traj_time_slice_ascent CTE:
-- 1. atRange: Clips the temporal data to create ranges where the vertrate
--  was between '[1, 20]'. This vertrate means an aircraft was ascending.
-- 2. sequenceN: Selects the first sequence from the generated sequences.
--  This first sequence is takeoff and eliminates mid-flight ascents.
-- 3. atPeriod: Returns the period of the first sequence.
flight_traj_time_slice_ascent(icao24, callsign, ascending_trip, ascending_geoaltitude,
                              ascending_vertrate) AS
    (SELECT icao24,
            callsign,
            atPeriod(time_slice_trip,
                     period(sequenceN(
                         atRange(time_slice_vertrate, floatrange '[1,20]'), 1))),
            atPeriod(time_slice_geoaltitude,
                     period(sequenceN(atRange(time_slice_vertrate, floatrange '[1,20]'),
                                      1))),
            atPeriod(time_slice_vertrate,
                     period(sequenceN(atRange(time_slice_vertrate, floatrange '[1,20]'),
                                      1)))
     FROM flight_traj_time_slice),

-- The final_output CTE uses unnest to unpack the temporal data into rows for
-- visualization in Grafana. Each row will contain a latitude, longitude and the altitude
-- and vertrate at those locations.
final_output AS
    (SELECT icao24,
            callsign,
            getValue(unnest(instants(ascending_geoaltitude))) AS geoaltitude,
            getValue(unnest(instants(ascending_vertrate)))    AS vertrate,
            ST_X(getValue(unnest(instants(ascending_trip))))  AS lon,
            ST_Y(getValue(unnest(instants(ascending_trip))))  AS lat
     FROM flight_traj_time_slice_ascent)

SELECT *
FROM final_output
WHERE vertrate IS NOT NULL
  AND geoaltitude IS NOT NULL;
```

Tips for **QGIS** visualization: QGIS uses geometry points for visualization, so for that in the third CTE you can use trajectory function on ascending_trip and unnest the result.

We will modify make the follow adjustments for the visualization.

1. Change the visualization type to “Geomap”.
2. The options (visualization settings - on the right side of the screen) should be as follows:

   Panel Options

   - Title → Flight Ascent in Time Window

   Data Layer:

   - Layer type: Markers
   - Location: Coords
   - Latitude field: lat
   - Longitude field: lon
   - Styles
     - Size: geoaltitude
     - Min: 1
     - Max: 5
     - Color: vertrate
     - Fill opacity: 0.5

   Standard Options:

   - Unit: meters/second (m/s)
   - Color scheme: Green-Yellow-Red (by value)
3. We will also add a manual override (top right of panel options, beside "All") to limit the minimum value of vertebrate. This will make all values below the minimum the same color, making larger values more obvious. This can be used to quickly pinpoint locations where a large rate of ascent existed.

   Overrides

   - Min: 5
   - Max: 20

   - Add field override > Fields with name > vertrate

Here is a zoomed in version of how each individual flight ascent will look, as well as a view of multiple flights at the same time. The marker size is increasing with altitude, and the color is showing more aggressive vertical ascent rates. We can see towards the end of the visualized ascent period, there is a short increased vertical ascent rate.

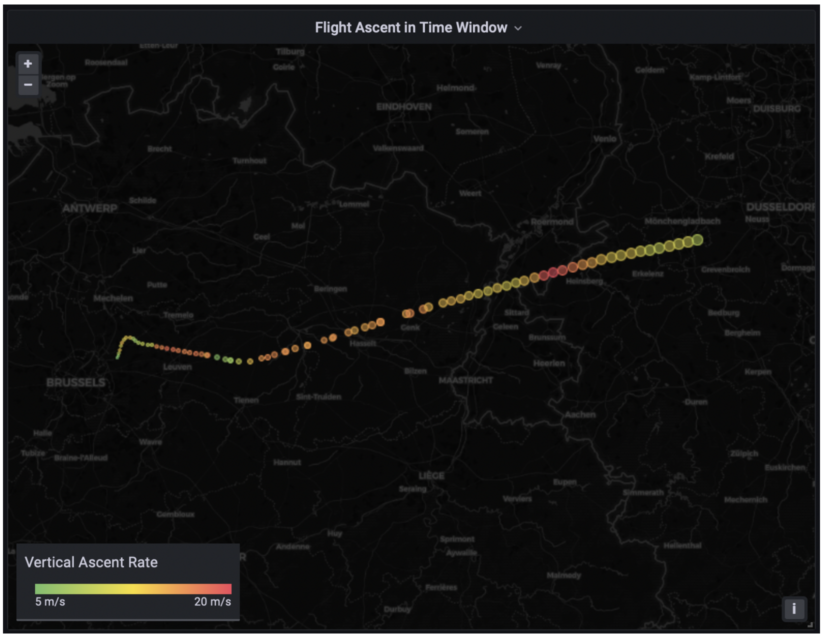


Figure 3.14: Zoomed in view of flight ascent

The final visualization will look like the below.

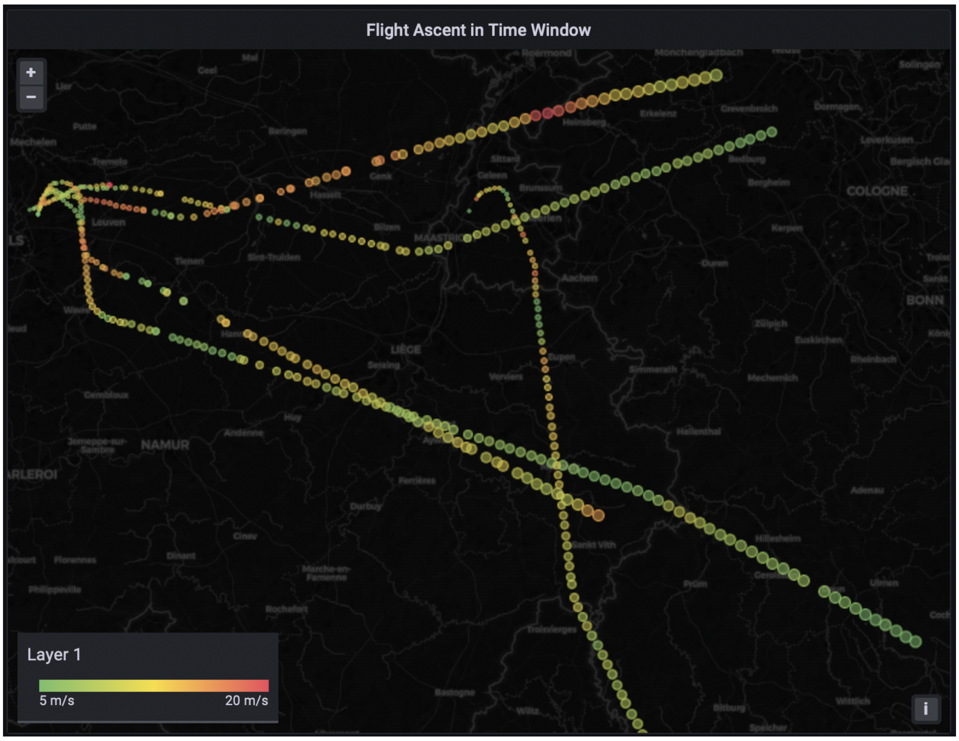


Figure 3.15: Final visualization with multiple flight ascents

3.6 Complete Flight Data Business Intelligence Dashboard

The dashboard, with all the visualizations at the same time, will look like the screenshot below. Here we can continue to extend the dashboards functionality by adding more dynamic variables to have the individual flight data on the left generated with a user query or selection based on the overview take-off information on the right. This is what really empowers decision makers and subject-matter experts (SMEs) to quickly move through data and hone-in on important aspects that may have otherwise been over-looked.

callsign vs time

time	callsign
2020-06-01 03:46:30	
2020-06-01 03:46:50	ANZ1285
2020-06-01 05:42:40	ANZ1220
2020-06-01 00:42:50	ANZ1272
2020-06-01 21:47:50	ANZ934

Average Flight Speed

ELY096	426 m/s
ELY067	338 m/s
UAL91	313 m/s
CES570	311 m/s
JAL860	296 m/s
PAC716	291 m/s
CHH471	289 m/s
CSN443	289 m/s
CSN447	288 m/s
CCA576	287 m/s
CHH7921	287 m/s

Number of Flights

Private	3233
Commercial	21408

Figure 3.16: Flight data business intelligence dashboard

Chapter 4

Managing GTFS Data

The General Transit Feed Specification (GTFS) defines a common format for public transportation schedules and associated geographic information. GTFS-realtime is used to specify real-time transit data. Many transportation agencies around the world publish their data in GTFS and GTFS-realtime format and make them publicly available. A well-known repository containing such data is OpenMobilityData.

In this chapter, we illustrate how to load GTFS data in MobilityDB. For this, we first need to import the GTFS data into PostgreSQL and then transform this data so that it can be loaded into MobilityDB. The data used in this tutorial is obtained from STIB-MIVB, the Brussels public transportation company and is available as a ZIP file. You must be aware that GTFS data is typically of big size. In order to reduce the size of the dataset, this file only contains schedules for one week and five transportation lines, whereas typical GTFS data published by STIB-MIVB contains schedules for one month and 99 transportation lines. In the reduced dataset used in this tutorial the final table containing the GTFS data in MobilityDB format has almost 10,000 trips and its size is 241 MB. Furtheremore, we need several temporary tables to transform GTFS format into MobilityDB and these tables are also big, the largest one has almost 6 million rows and its size is 621 MB.

Several tools can be used to import GTFS data into PostgreSQL. For example, one publicly available in Github can be found here. These tools load GTFS data into PostgreSQL tables, allowing one to perform multiple imports of data provided by the same agency covering different time frames, perform various complex tasks including data validation, and take into account variations of the format provided by different agencies, updates of route information among multiple imports, etc. For the purpose of this tutorial we do a simple import and transformation using only SQL. This is enough for loading the data set we are using but a much more robust solution should be used in an operational environment, if only for coping with the considerable size of typical GTFS data, which would require parallelization of this task.

4.1 Loading GTFS Data in PostgreSQL

The ZIP file with the data for this tutorial contains a set of CSV files (with extension `.txt`) as follows:

- `agency.txt` contains the description of the transportation agencies provinding the services (a single one in our case).
- `calendar.txt` contains service patterns that operate recurrently such as, for example, every weekday.
- `calendar_dates.txt` define exceptions to the default service patterns defined in `calendar.txt`. There are two types of exceptions: 1 means that the service has been added for the specified date, and 2 means that the service has been removed for the specified date.
- `route_types.txt` contains transportation types used on routes, such as bus, metro, tramway, etc.
- `routes.txt` contains transit routes. A route is a group of trips that are displayed to riders as a single service.
- `shapes.txt` contains the vehicle travel paths, which are used to generate the corresponding geometry.
- `stop_times.txt` contains times at which a vehicle arrives at and departs from stops for each trip.

- `translations.txt` contains the translation of the route information in French and Dutch. This file is not used in this tutorial.
- `trips.txt` contains trips for each route. A trip is a sequence of two or more stops that occur during a specific time period.

We decompress the file with the data into a directory. This can be done using the command.

```
unzip gtfs_data.zip
```

We suppose in the following that the directory used is as follows `/home/gtfs_tutorial/`.

We create the tables to be loaded with the data in the CSV files as follows.

```
CREATE TABLE agency (
  agency_id text DEFAULT '',
  agency_name text DEFAULT NULL,
  agency_url text DEFAULT NULL,
  agency_timezone text DEFAULT NULL,
  agency_lang text DEFAULT NULL,
  agency_phone text DEFAULT NULL,
  CONSTRAINT agency_pkey PRIMARY KEY (agency_id)
);

CREATE TABLE calendar (
  service_id text,
  monday int NOT NULL,
  tuesday int NOT NULL,
  wednesday int NOT NULL,
  thursday int NOT NULL,
  friday int NOT NULL,
  saturday int NOT NULL,
  sunday int NOT NULL,
  start_date date NOT NULL,
  end_date date NOT NULL,
  CONSTRAINT calendar_pkey PRIMARY KEY (service_id)
);
CREATE INDEX calendar_service_id ON calendar (service_id);

CREATE TABLE exception_types (
  exception_type int PRIMARY KEY,
  description text
);

CREATE TABLE calendar_dates (
  service_id text,
  date date NOT NULL,
  exception_type int REFERENCES exception_types(exception_type)
);
CREATE INDEX calendar_dates_dateidx ON calendar_dates (date);

CREATE TABLE route_types (
  route_type int PRIMARY KEY,
  description text
);

CREATE TABLE routes (
  route_id text,
  route_short_name text DEFAULT '',
  route_long_name text DEFAULT '',
```

```
  route_desc text DEFAULT '',
  route_type int REFERENCES route_types(route_type),
  route_url text,
  route_color text,
  route_text_color text,
  CONSTRAINT routes_pkey PRIMARY KEY (route_id)
);

CREATE TABLE shapes (
  shape_id text NOT NULL,
  shape_pt_lat double precision NOT NULL,
  shape_pt_lon double precision NOT NULL,
  shape_pt_sequence int NOT NULL
);
CREATE INDEX shapes_shape_key ON shapes (shape_id);

-- Create a table to store the shape geometries
CREATE TABLE shape_geoms (
  shape_id text NOT NULL,
  shape_geom geometry('LINESTRING', 4326),
  CONSTRAINT shape_geom_pkey PRIMARY KEY (shape_id)
);
CREATE INDEX shape_geoms_key ON shapes (shape_id);

CREATE TABLE location_types (
  location_type int PRIMARY KEY,
  description text
);

CREATE TABLE stops (
  stop_id text,
  stop_code text,
  stop_name text DEFAULT NULL,
  stop_desc text DEFAULT NULL,
  stop_lat double precision,
  stop_lon double precision,
  zone_id text,
  stop_url text,
  location_type integer  REFERENCES location_types(location_type),
  parent_station integer,
  stop_geom geometry('POINT', 4326),
  platform_code text DEFAULT NULL,
  CONSTRAINT stops_pkey PRIMARY KEY (stop_id)
);

CREATE TABLE pickup_dropoff_types (
  type_id int PRIMARY KEY,
  description text
);

CREATE TABLE stop_times (
  trip_id text NOT NULL,
  -- Check that casting to time interval works.
  arrival_time interval CHECK (arrival_time::interval = arrival_time::interval),
  departure_time interval CHECK (departure_time::interval = departure_time::interval),
  stop_id text,
  stop_sequence int NOT NULL,
  pickup_type int REFERENCES pickup_dropoff_types(type_id),
  drop_off_type int REFERENCES pickup_dropoff_types(type_id),
  CONSTRAINT stop_times_pkey PRIMARY KEY (trip_id, stop_sequence)
);
CREATE INDEX stop_times_key ON stop_times (trip_id, stop_id);
```

```
CREATE INDEX arr_time_index ON stop_times (arrival_time);
CREATE INDEX dep_time_index ON stop_times (departure_time);

CREATE TABLE trips (
  route_id text NOT NULL,
  service_id text NOT NULL,
  trip_id text NOT NULL,
  trip_headsign text,
  direction_id int,
  block_id text,
  shape_id text,
  CONSTRAINT trips_pkey PRIMARY KEY (trip_id)
);
CREATE INDEX trips_trip_id ON trips (trip_id);

INSERT INTO exception_types (exception_type, description) VALUES
(1, 'service has been added'),
(2, 'service has been removed');

INSERT INTO location_types(location_type, description) VALUES
(0,'stop'),
(1,'station'),
(2,'station entrance');

INSERT INTO pickup_dropoff_types (type_id, description) VALUES
(0,'Regularly Scheduled'),
(1,'Not available'),
(2,'Phone arrangement only'),
(3,'Driver arrangement only');
```

We created one table for each CSV file. In addition, we created a table `shape_geoms` in order to assemble all segments composing a route into a single geometry and auxiliary tables `exception_types`, `location_types`, and `pickup_dropoff_types` containing acceptable values for some columns in the CSV files.

We can load the CSV files into the corresponding tables as follows.

```
COPY calendar(service_id,monday,tuesday,wednesday,thursday,friday,saturday,sunday,
start_date,end_date) FROM '/home/gtfs_tutorial/calendar.txt' DELIMITER ',' CSV HEADER;
COPY calendar_dates(service_id,date,exception_type)
FROM '/home/gtfs_tutorial/calendar_dates.txt' DELIMITER ',' CSV HEADER;
COPY stop_times(trip_id,arrival_time,departure_time,stop_id,stop_sequence,
pickup_type,drop_off_type) FROM '/home/gtfs_tutorial/stop_times.txt' DELIMITER ','
CSV HEADER;
COPY trips(route_id,service_id,trip_id,trip_headsign,direction_id,block_id,shape_id)
FROM '/home/gtfs_tutorial/trips.txt' DELIMITER ',' CSV HEADER;
COPY agency(agency_id,agency_name,agency_url,agency_timezone,agency_lang,agency_phone)
FROM '/home/gtfs_tutorial/agency.txt' DELIMITER ',' CSV HEADER;
COPY route_types(route_type,description)
FROM '/home/gtfs_tutorial/route_types.txt' DELIMITER ',' CSV HEADER;
COPY routes(route_id,route_short_name,route_long_name,route_desc,route_type,route_url,
route_color,route_text_color) FROM '/home/gtfs_tutorial/routes.txt' DELIMITER ','
CSV HEADER;
COPY shapes(shape_id,shape_pt_lat,shape_pt_lon,shape_pt_sequence)
FROM '/home/gtfs_tutorial/shapes.txt' DELIMITER ',' CSV HEADER;
COPY stops(stop_id,stop_code,stop_name,stop_desc,stop_lat,stop_lon,zone_id,stop_url,
location_type,parent_station) FROM '/home/gtfs_tutorial/stops.txt' DELIMITER ','
CSV HEADER;
```

Finally, we create the geometries for routes and stops as follows.

```
INSERT INTO shape_geoms
SELECT shape_id, ST_MakeLine(array_agg(
  ST_SetSRID(ST_MakePoint(shape_pt_lon, shape_pt_lat),4326) ORDER BY shape_pt_sequence))
FROM shapes
GROUP BY shape_id;

UPDATE stops
SET stop_geom = ST_SetSRID(ST_MakePoint(stop_lon, stop_lat),4326);
```

The visualization of the routes and stops in QGIS is given in Figure 4.1. In the figure, red lines correspond to the trajectories of vehicles, while orange points correspond to the location of stops.

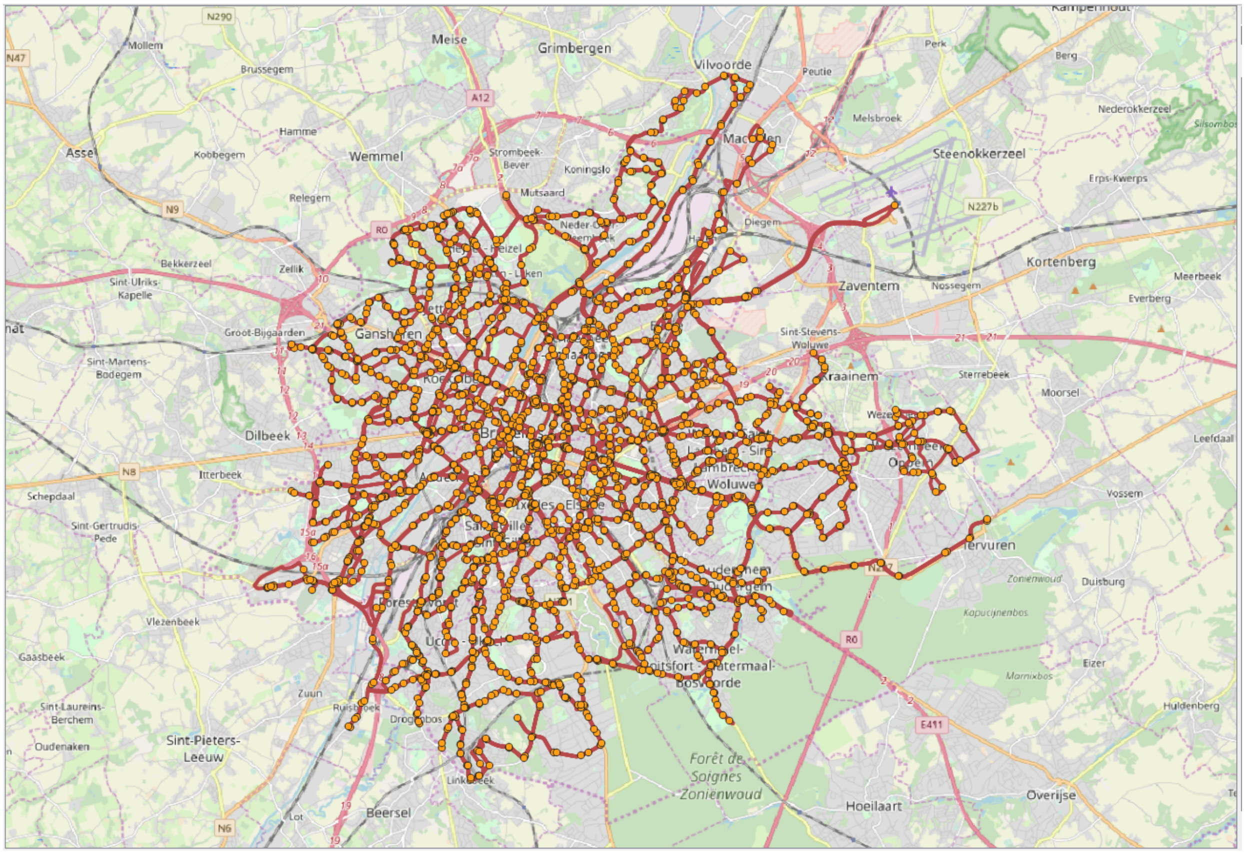


Figure 4.1: Visualization of the routes and stops for the GTFS data from Brussels.

4.2 Transforming GTFS Data for MobilityDB

We start by creating a table that contains couples of `service_id` and `date` defining the dates at which a service is provided.

```
DROP TABLE IF EXISTS service_dates;
CREATE TABLE service_dates AS (
SELECT service_id, date_trunc('day', d)::date AS date
FROM calendar c, generate_series(start_date, end_date, '1 day'::interval) AS d
WHERE (
  (monday = 1 AND extract(isodow FROM d) = 1) OR
  (tuesday = 1 AND extract(isodow FROM d) = 2) OR
  (wednesday = 1 AND extract(isodow FROM d) = 3) OR
```

```
  (thursday = 1 AND extract(isodow FROM d) = 4) OR
  (friday = 1 AND extract(isodow FROM d) = 5) OR
  (saturday = 1 AND extract(isodow FROM d) = 6) OR
  (sunday = 1 AND extract(isodow FROM d) = 7)
)
EXCEPT
SELECT service_id, date
FROM calendar_dates WHERE exception_type = 2
UNION
SELECT c.service_id, date
FROM calendar c JOIN calendar_dates d ON c.service_id = d.service_id
WHERE exception_type = 1 AND start_date <= date AND date <= end_date
);
```

This table transforms the service patterns in the `calendar` table valid between a `start_date` and an `end_date` taking into account the week days, and then remove the exceptions of type 2 and add the exceptions of type 1 that are specified in table `calendar_dates`.

We now create a table `trip_stops` that determines the stops for each trip.

```
DROP TABLE IF EXISTS trip_stops;
CREATE TABLE trip_stops (
  trip_id text,
  stop_sequence integer,
  no_stops integer,
  route_id text,
  service_id text,
  shape_id text,
  stop_id text,
  arrival_time interval,
  perc float
);

INSERT INTO trip_stops (trip_id, stop_sequence, no_stops, route_id, service_id,
  shape_id, stop_id, arrival_time)
SELECT t.trip_id, stop_sequence, MAX(stop_sequence) OVER (PARTITION BY t.trip_id),
  route_id, service_id, shape_id, stop_id, arrival_time
FROM trips t JOIN stop_times s ON t.trip_id = s.trip_id;

UPDATE trip_stops t
SET perc = CASE
WHEN stop_sequence =  1 then 0.0
WHEN stop_sequence =  no_stops then 1.0
ELSE ST_LineLocatePoint(g.shape_geom, s.stop_geom)
END
FROM shape_geoms g, stops s
WHERE t.shape_id = g.shape_id AND t.stop_id = s.stop_id;
```

We perform a join between `trips` and `stop_times` and determines the number of stops in a trip. Then, we compute the relative location of a stop within a trip using the function `ST_LineLocatePoint`.

We now create a table `trip_segs` that defines the segments between two consecutive stops of a trip.

```
DROP TABLE IF EXISTS trip_segs;
CREATE TABLE trip_segs (
  trip_id text,
  route_id text,
  service_id text,
  stop1_sequence integer,
```

```
  stop2_sequence integer,
  no_stops integer,
  shape_id text,
  stop1_arrival_time interval,
  stop2_arrival_time interval,
  perc1 float,
  perc2 float,
  seg_geom geometry,
  seg_length float,
  no_points integer,
  PRIMARY KEY (trip_id, stop1_sequence)
);

INSERT INTO trip_segs (trip_id, route_id, service_id, stop1_sequence, stop2_sequence,
  no_stops, shape_id, stop1_arrival_time, stop2_arrival_time, perc1, perc2)
WITH temp AS (
  SELECT trip_id, route_id, service_id, stop_sequence,
    LEAD(stop_sequence) OVER w AS stop_sequence2,
  MAX(stop_sequence) OVER (PARTITION BY trip_id),
  shape_id, arrival_time, LEAD(arrival_time) OVER w, perc, LEAD(perc) OVER w
  FROM trip_stops WINDOW w AS (PARTITION BY trip_id ORDER BY stop_sequence)
)
SELECT * FROM temp WHERE stop_sequence2 IS NOT null;

UPDATE trip_segs t
SET seg_geom = ST_LineSubstring(g.shape_geom, perc1, perc2)
FROM shape_geoms g
WHERE t.shape_id = g.shape_id;

UPDATE trip_segs
SET seg_length = ST_Length(seg_geom), no_points = ST_NumPoints(seg_geom);
```

We use twice the `LEAD` window function for obtaning the next stop and the next percentage of a given stop and the `MAX` window function for obtaining the total number of stops in a trip. Then, we generate the geometry of the segment betwen two stops using the function `ST_LineSubstring` and compute the length and the number of points in the segment with functions `ST_Length` and `ST_NumPoints`.

The geometry of a segment is a linestring containing multiple points. From the previous table we know at which time the trip arrived at the first point and at the last point of the segment. To determine at which time the trip arrived at the intermediate points of the segments, we create a table `trip_points` that contains all the points composing the geometry of a segment.

```
DROP TABLE IF EXISTS trip_points;
CREATE TABLE trip_points (
  trip_id text,
  route_id text,
  service_id text,
  stop1_sequence integer,
  point_sequence integer,
  point_geom geometry,
  point_arrival_time interval,
  PRIMARY KEY (trip_id, stop1_sequence, point_sequence)
);

INSERT INTO trip_points (trip_id, route_id, service_id, stop1_sequence,
  point_sequence, point_geom, point_arrival_time)
WITH temp1 AS (
  SELECT trip_id, route_id, service_id, stop1_sequence, stop2_sequence,
  no_stops, stop1_arrival_time, stop2_arrival_time, seg_length,
  (dp).path[1] AS point_sequence, no_points, (dp).geom as point_geom
FROM trip_segs, ST_DumpPoints(seg_geom) AS dp
```

```
),
temp2 AS (
SELECT trip_id, route_id, service_id, stop1_sequence, stop1_arrival_time,
  stop2_arrival_time, seg_length, point_sequence, no_points, point_geom
FROM temp1
WHERE point_sequence <> no_points OR stop2_sequence = no_stops
),
temp3 AS (
SELECT trip_id, route_id, service_id, stop1_sequence, stop1_arrival_time,
  stop2_arrival_time, point_sequence, no_points, point_geom,
  ST_Length(ST_MakeLine(array_agg(point_geom) OVER w)) / seg_length AS perc
FROM temp2 WINDOW w AS (PARTITION BY trip_id, service_id, stop1_sequence
  ORDER BY point_sequence)
)
SELECT trip_id, route_id, service_id, stop1_sequence, point_sequence, point_geom,
CASE
WHEN point_sequence = 1 then stop1_arrival_time
WHEN point_sequence = no_points then stop2_arrival_time
ELSE stop1_arrival_time + ((stop2_arrival_time - stop1_arrival_time) * perc)
END AS point_arrival_time
FROM temp3;
```

In the temporary table `temp1` we use the function `ST_DumpPoints` to obtain the points composing the geometry of a segment. Nevertheless, this table contains duplicate points, that is, the last point of a segment is equal to the first point of the next one. In the temporary table `temp2` we filter out the last point of a segment unless it is the last segment of the trip. In the temporary table `temp3` we compute in the attribute `perc` the relative position of a point within a trip segment with window functions. For this we use the function `ST_MakeLine` to construct the subsegment from the first point of the segment to the current one, determine the length of the subsegment with function `ST_Length` and divide this length by the overall segment length. Finally, in the outer query we use the computed percentage to determine the arrival time to that point.

Our last temporary table `trips_input` contains the data in the format that can be used for creating the MobilityDB trips.

```
DROP TABLE IF EXISTS trips_input;
CREATE TABLE trips_input (
  trip_id text,
  route_id text,
  service_id text,
  date date,
  point_geom geometry,
  t timestamptz
);

INSERT INTO trips_input
SELECT trip_id, route_id, t.service_id, date, point_geom, date + point_arrival_time AS t
FROM trip_points t JOIN
( SELECT service_id, MIN(date) AS date FROM service_dates GROUP BY service_id) s
ON t.service_id = s.service_id;
```

In the inner query of the `INSERT` statement, we select the first date of a service in the `service_dates` table and then we join the resulting table with the `trip_points` table to compute the arrival time at each point composing the trips. Notice that we filter the first date of each trip for optimization purposes because in the next step below we use the `shift` function to compute the trips to all other dates. Alternatively, we could join the two tables but this will be considerably slower for big GTFS files.

Finally, table `trips_mdb` contains the MobilityDB trips.

```
DROP TABLE IF EXISTS trips_mdb;
CREATE TABLE trips_mdb (
  trip_id text NOT NULL,
  route_id text NOT NULL,
```

```
  date date NOT NULL,
  trip tgeompoint,
  PRIMARY KEY (trip_id, date)
);

INSERT INTO trips_mdb(trip_id, route_id, date, trip)
SELECT trip_id, route_id, date,
  tgeompoint_seq(array_agg(tgeompoint_inst(point_geom, t) ORDER BY T))
FROM trips_input
GROUP BY trip_id, route_id, date;

INSERT INTO trips_mdb(trip_id, service_id, route_id, date, trip)
SELECT trip_id, route_id, t.service_id, d.date,
  shift(trip, make_interval(days => d.date - t.date))
FROM trips_mdb t JOIN service_dates d ON t.service_id = d.service_id AND t.date <> d.date;
```

In the first `INSERT` statement we group the rows in the `trips_input` table by `trip_id` and `date` while keeping the `route_id` atribute, use the `array_agg` function to construct an array containing the temporal points composing the trip ordered by time, and compute the trip from this array using the function `tgeompointseq`. As explained above, table `trips_input` only contains the first date of a trip. In the second `INSERT` statement we add the trips for all the other dates with the function `shift`.

Chapter 5

Managing Google Location History

5.1 Loading Google Location History Data

By activating the Location History in your Google account, you let Google track where you go with every mobile device. You can view and manage your Location History information through Google Maps Timeline. The data is provided in JSON format. An example of such a file is as follows.

```
{
"locations" : [ {
  "timestampMs" : "1525373187756",
  "latitudeE7" : 508402936,
  "longitudeE7" : 43413790,
  "accuracy" : 26,
  "activity" : [ {
    "timestampMs" : "1525373185830",
    "activity" : [ {
      "type" : "STILL",
      "confidence" : 44
    }, {
      "type" : "IN_VEHICLE",
      "confidence" : 16
    }, {
      "type" : "IN_ROAD_VEHICLE",
      "confidence" : 16
    }, {
      "type" : "UNKNOWN",
      "confidence" : 12
    }, {
      "type" : "IN_RAIL_VEHICLE",
      "confidence" : 12
...
```

If we want to load location information into MobilityDB we only need the fields `longitudeE7`, `latitudeE7`, and `timestampMs`. To convert the original JSON file into a CSV file containing only these fields we can use jq, a command-line JSON processor. The following command

```
cat location_history.json | jq -r ".locations[] | {latitudeE7, longitudeE7, timestampMs}
| [.latitudeE7, .longitudeE7, .timestampMs] | @csv" > location_history.csv
```

produces a CSV file of the following format

```
508402936,43413790,"1525373187756"
508402171,43413455,"1525373176729"
508399229,43413304,"1525373143463"
508377525,43411499,"1525373113741"
508374906,43412597,"1525373082542"
508370337,43418136,"1525373052593"
...
```

The above command works well for files of moderate size since by default jq loads the whole input text in memory. For very large files you may consider the `--stream` option of jq, which parses input texts in a streaming fashion.

Now we can import the generated CSV file into PostgreSQL as follows.

```
DROP TABLE IF EXISTS location_history;
CREATE TABLE location_history (
latitudeE7 float,
longitudeE7 float,
timestampMs bigint,
date date
);

COPY location_history(latitudeE7, longitudeE7, timestampMs) FROM
'/home/location_history/location_history.csv' DELIMITER ',' CSV;

UPDATE location_history
SET date = date(to_timestamp(timestampMs / 1000.0)::timestamptz);
```

Notice that we added an attribute `date` to the table so we can split the full location history, which can comprise data for several years, by date. Since the timestamps are encoded in milliseconds since 1/1/1970, we divide them by 1,000 and apply the functions `to_timestamp` and `date` to obtain corresponding date.

We can now transform this data into MobilityDB trips as follows.

```
DROP TABLE IF EXISTS locations_mdb;
CREATE TABLE locations_mdb (
date date NOT NULL,
trip tgeompoint,
trajectory geometry,
PRIMARY KEY (date)
);

INSERT INTO locations_mdb(date, trip)
SELECT date, tgeompoint_seq(array_agg(tgeompoint_inst(
ST_SetSRID(ST_Point(longitudeE7/1e7, latitudeE7/1e7),4326),
to_timestamp(timestampMs / 1000.0)::timestamptz) ORDER BY timestampMs))
FROM location_history
GROUP BY date;

UPDATE locations_mdb
SET trajectory = trajectory(trip);
```

We convert the longitude and latitude values into standard coordinates values by dividing them by 10^7. These can be converted into PostGIS points in the WGS84 coordinate system with the functions `ST_Point` and `ST_SetSRID`. Also, we convert the timestamp values in miliseconds to `timestamptz` values. We can now apply the function `tgeompointinst` to create a `tgeompoint` of instant duration from the point and the timestamp, collect all temporal points of a day into an array with the function `array_agg`, and finally, create a temporal point containing all the locations of a day using function `tgeompointseq`. We added to the table a `trajectory` attribute to visualize the location history in QGIS is given in Figure 5.1.

Figure 5.1: Visualization of the Google location history loaded into MobilityDB.

Chapter 6

Managing GPX Data

6.1 Loading GPX Data

GPX, or GPS Exchange Format, is an XML data format for GPS data. Location data (and optionally elevation, time, and other information) is stored in tags and can be interchanged between GPS devices and software. Conceptually, a GPX file contains tracks, which are a record of where a moving object has been, and routes, which are suggestions about where it might go in the future. Furthermore, both tracks and routes and composed by points. The following is a truncated (for brevity) example GPX file.

```
<?xml version='1.0' encoding='UTF-8' standalone='yes' ?>
<gpx version="1.1"
xmlns="http://www.topografix.com/GPX/1/1"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.topografix.com/GPX/1/1
http://www.topografix.com/GPX/1/1/gpx.xsd"
creator="Example creator">
<metadata>
  <name>Dec 14, 2014 4:32:04 PM</name>
  <author>Example creator</author>
  <link href="https://..." />
  <time>2014-12-14T14:32:04.650Z</time>
</metadata>
<trk>
  <name>Dec 14, 2014 4:32:04 PM</name>
  <trkseg>
    <trkpt lat="30.16398" lon="31.467701">
      <ele>76</ele>
      <time>2014-12-14T14:32:10.339Z</time>
    </trkpt>
    <trkpt lat="30.16394" lon="31.467333">
      <ele>73</ele>
      <time>2014-12-14T14:32:16.00Z</time>
    </trkpt>
    <trkpt lat="30.16408" lon="31.467218">
      <ele>74</ele>
      <time>2014-12-14T14:32:19.00Z</time>
    </trkpt>
    [...]
  </trkseg>
  <trkseg>
    [...]
  </trkseg>
```

```
  [...]
</trk>
<trk>
  [...]
</trk>
[...]
<gpx>
```

The following Python program called `gpx_to_csv.py` uses `expat`, a stream-oriented XML parser library, to convert the above GPX file in CSV format.

```
import sys
import xml.parsers.expat

stack = []
def start_element(name, attrs):
stack.append(name)
if name == 'gpx' :
  print("lon,lat,time")
if name == 'trkpt' :
  print("{},{},".format(attrs['lon'], attrs['lat']), end="")

def end_element(name):
stack.pop()

def char_data(data):
if stack[-1] == "time" and stack[-2] == "trkpt" :
  print(data)

p = xml.parsers.expat.ParserCreate()

p.StartElementHandler = start_element
p.EndElementHandler = end_element
p.CharacterDataHandler = char_data

p.ParseFile(sys.stdin.buffer)
```

This Python program can be executed as follows.

```
python3 gpx_to_csv.py < example.gpx > example.csv
```

The resulting CSV file is given next.

```
lon,lat,time
31.46032,30.037502,2015-02-09T08:10:16.00Z
31.460901,30.039026,2015-02-09T08:10:31.00Z
31.461981,30.039816,2015-02-09T08:10:57.00Z
31.461996,30.039801,2015-02-09T08:10:58.00Z
...
```

The above CSV file can be loaded into MobilityDB as follows.

```
DROP TABLE IF EXISTS trips_input;
CREATE TABLE trips_input (
  date date,
```

```
  lon float,
  lat float,
  time timestamptz
);

COPY trips_input(lon, lat, time) FROM
'/home/gpx_data/example.csv' DELIMITER ',' CSV HEADER;

UPDATE trips_input
SET date = date(time);

DROP TABLE IF EXISTS trips_mdb;
CREATE TABLE trips_mdb (
  date date NOT NULL,
  trip tgeompoint,
  trajectory geometry,
  PRIMARY KEY (date)
);

INSERT INTO trips_mdb(date, trip)
SELECT date, tgeompoint_seq(array_agg(tgeompoint_inst(
  ST_SetSRID(ST_Point(lon, lat), 4326), time) ORDER BY time))
FROM trips_input
GROUP BY date;

UPDATE trips_mdb
SET trajectory = trajectory(trip);
```