University of Piraeus
Department of Digital Systems

Project

Project Title	Manage Navigation Data using MongoDB
Full Name:	Panagiotakopoulos Georgios
Registration Number	me2030
Supervisor	Christos Doulkeridis
Deadline	31/01/2021

Contents

1. Abstract

International trade depends heavily on maritime transport especially as the world's population continues to grow. Since maritime transport is so essential tracking of those ships becomes a necessity in order to ensure safety. The automatic identification system (AIS) is an automatic tracking system that uses transceivers on ships and is used by vessel traffic services (VTS). This study focuses on handling AIS data from a timespan of six months in the areas of Celtic sea, Channel & Bay of Biscay. The data were stored after some preprocessing on the non-relational MongoDB.
The architecture of the database was designed factoring the capability to correspond on relational, spatial, spatio-temporal and trajectory queries. The main design decision that was taken was the creation of an embedded document which contains all the navigational and vessel related data. In order to further implement the navigational collection, we linked it with a collection that is filled with grid documents.
The grid collection was the second big design decision since we separated the seas in grids of 10x10 km so that we can easier handle the coordinates of ships. Each location of a ship is assigned to a grid and as a result the respond time of distance queries is reduced by a lot. In terms of sharding we created a multi-compound index combing timestamp and hashed mms. The compound index assisted in blunting the future scaling issue of adding more data as time goes by. The architecture of the designed system ensures the effectiveness of responding to all the studied query types. Results were presented for relational queries, for k-nn or distance queries, for queries that given a trajectory, can find similar trajectories and for complex ones that can calculate where a ship will be positioned after a certain time and certain locations in between.
Future projects could be the design of a user interface, the implementation of further resources so that we could handle a larger influx of data and the implementation and usage of further information such as environmental data.

2. Introduction

2.1 Introductory Report: Shipping has a leading role in the transfer of world trade. According to a survey by the European Community Shipowners 'Associations, about 80% of world trade is transported by sea [1.]. As a result, the huge number of ships that move daily at sea and the need to locate and track these ships is realized. Most of the technology used is an Automatic Identification System known as AIS. The use of AIS seeks to identify and locate ships by extracting real-time data.

Main reasons for using AIS:

- Avoiding collisions between ships.
- Enhancing security.
- Better communication between the ship and the competent bodies on the shore.
- Better communication between ships.
- The improvement of Navigation.

Its usefulness in preventing accidents and collisions makes the use of AIS mandatory:

- On cargo ships of a gross weight exceeding 300 tonnes making international voyages.
- On cargo or passenger ships over 500 tons for all types of voyages.
- In fishing vessels that are longer than 15 meters [2.].

The exchange of data using AIS would be said to have the following structure. Digital messages are exchanged using sensors, gps, or manually by a ship user in order to exchange information such as ship position, name, voyage information, etc. *Figure 1* illustrates that the exchange of digital messages can occur bidirectionally between ships, between a station-ship, between a station-satellite and between a satellite-ship.

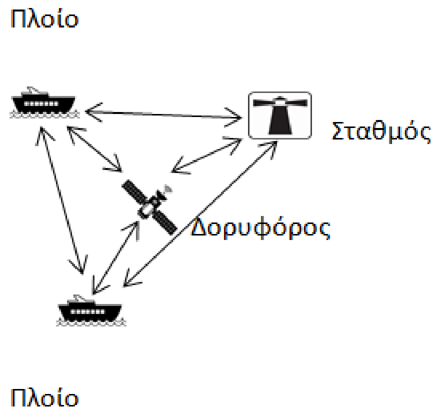


Figure 1

The storage, utilization and analysis of all this data serves in the design of the orbits of ships, in finding their distance from points of interest (eg ports), in ensuring the safety distance between moving ships, etc.

2.2 Problem Report

: The previous section highlighted the usefulness of the automatic identification system technology for various areas such as accident prevention and ensuring the safety of sea voyages. Therefore there is a constant addition of AIS systems from ships of different sizes and types regardless of whether their installation is mandatory or not. Millions of digital messages are exchanged monthly across the seas of the world, making their storage and management extremely demanding. Therefore the use of relational databases to manage such a large amount of data leads to high storage and management costs. One solution would be to delete older messages, a solution that is not appropriate as historical data can be utilized in various applications. As a result, a problem of storage and management of large geospatial data is created. The aim is to find a non-relational database in which geospatial data can be stored and used properly.

2.3 Project goals

: The constant need for better storage and management of AIS data is great. Finding or approaching a solution will be able to meet this need. This paper aims to store and analyze this data using the non-relational MongoDB database, expecting an understanding of the advantages and disadvantages of this option. Understanding the mentioned problem is a source of inspiration for designing solutions that answer geospatial questions based on efficiency and speed of implementation.

3. Related Projects

An An application template that analyzes navigation data is Marine Traffic [3.] The Marine Traffic platform manages data in real time, achieving the tracking of ships, their distance from points of interest, the intended arrival at their destination, etc. Most of the data to be analyzed is AIS data and historical data collections. A study conducted in collaboration with Marine Traffic and Harokopio University compared relational and non-relational databases in terms of data storage efficiency [4.]. In particular, he made a comparison between MongoDB (non-relational database) and PostgreSQL (relational database) as to which is the most efficient data storage system based on the query response time. Tests have shown that PostgreSQL in most queries is up to 4 times more effective in response time than MongoDB. Also, the Indexing application in the database dropped the times significantly and especially in MongoDB almost by half.
In the field of exporting routes from AIS data, Pan Sheng and Jingbo Yin [5] attempted to arrive at ship route patterns using clustering techniques in the large volume of AIS data. The ultimate goal is to better understand and improve ship management systems. The article shows that using machine learning techniques can be achieved simplification and dilution of huge and complex data volumes. Specifically, the reduction of the data volume is

achieved by using the principles of MDL while maintaining a high degree of consistency of the original data. Through experiments they concluded that a variant of the DBSCAN algorithm is the best application for the aggregation of AIS data in order to predict the possible orbit of a ship. The experimental results presented prove the effectiveness of clustering in terms of successful course design and short execution time.

4 System Architecture:

4.1 Database design and data Processing.

The The design of the base was carried out with the following objectives:

- Existence of substantial and sufficient information of the original data.
- The ability to answer all possible questions. To be more specific, to be able to answer Relational, Spatial, Spatio-temporal and Trajectory queries
- The response speed.
- Utilizing the capabilities of MongoDB.
- Possible Scalability.

Evaluating Evaluating the objectives and understanding of the data we came to the conclusion that the main pillar that will be designed around the whole base will be the navigation data. In particular, special emphasis will be given to the dynamic data of ships. The database designed in MongoDB and named ‘marine_trafic’ includes the following collections:

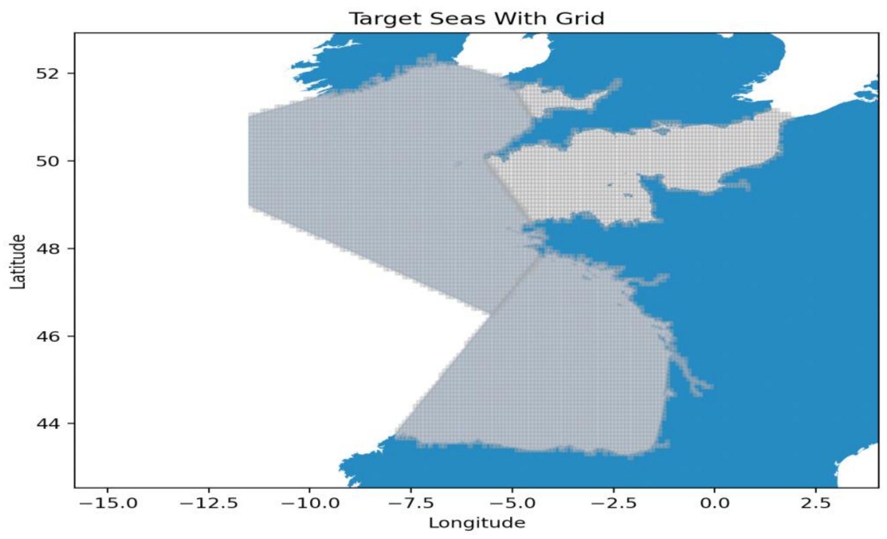
- ais_navigation
- countries
- fishing_port
- query_polygons
- target_map_grid
- world_port_geo
- world_port_information
- world_seas

In more detail, the design can be divided into the following stages:

1. Design, pre-process and upload the ais_navigation collection to MongoDB. Includes navigation data as well as ship information. Given the MongoDB architecture, some relational database features such as join are not supported or appropriate [9.]. The pairing operation or $ lookup is considered very expensive so the pre-processing of the data was chosen by creating a nested document that contains the necessary information. We basically created a document that contains other documents inside. We used PostgreSQL and Python to create this document. The file follows the Json format and was created taking in the first phase from the table Ais_data.dynamic_ships the: mmsi, ts, longtitude, latitude (the coordinates are

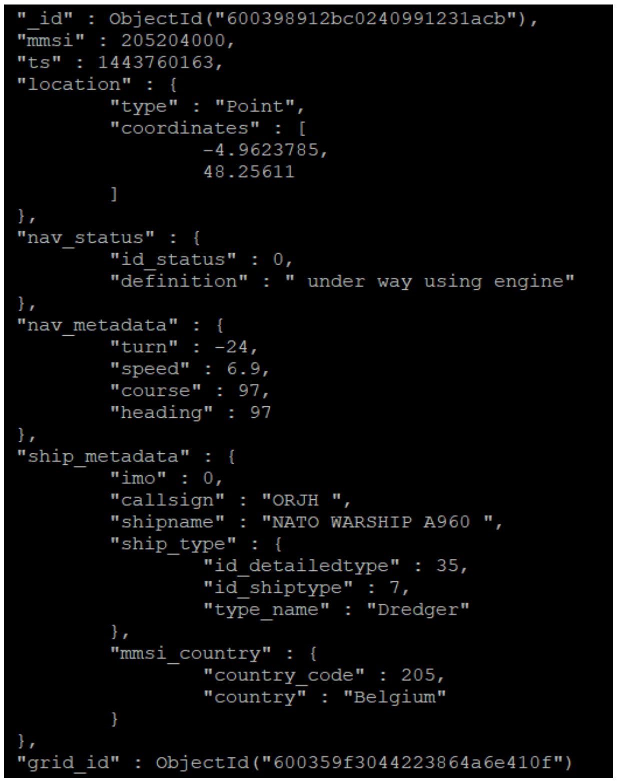
inserted nested in field location), turn, speed, course, heading. In the second phase we extracted the static data (coupling of mmsi-sourcemmsi) from the table Ais_data.static_ships and the navigational status from the table Ais_status_codes_types.navigational_status (coupling of status-id.status). We nested the type data of the ships from the tables Ais_status_codes_types.types.ship_types & ship_types_detailed (pairing in id_shiptype). Finally nested we entered the data from the table Ais_status_codes_types.mmsi_country_codes to show the country of origin of the ships (pairing the first 3 digits of mmsi with the country code). Therefore ship information was entered into a nested document which we uploaded to MongoDB (using the "pymongo" library). MongoDB automatically assigns a unique key (ObjectId) to each document in the collection. The main advantage of nesting is that by accessing the same disk block we have the ability to recover data from more 'tables'. At the same time we can update a document with access to only one disk block [10.].

2. In the second stage of design we thought to divide the requested seas into grids for better efficiency of the base in terms of answering geospatial and spatio-temporal questions. Specifically, we divided the seas Celtic Sea, Bay of Biscay, English Channel, Bristol Channel, St. George's Channel in grids with a radius of ten kilometers (10x10 km). Figure 3 shows the grids of the seas. The separation was done using Python and then the documents were uploaded to Mongo in the target_map_grid collection. Each grid document contains point coordinates, a unique grid key and the unique sea key to which it belongs.



Picture 3: Sea grids

3. The seas were separated into grids in order to connect them to the ais_navigation collection. Using the geopandas library we connected each document of the ais_navigation collection to the grid of the target_map_grid collection to which it corresponds. As a result, each ais_navigation document contains a new grid_id field to which a unique grid key of the target_map_grid collection corresponds. Therefore, all the coordinates of the examined ships are categorized in grids, making the implementation of distance queries or similar routes faster. The technique we have used and offered by MongoDB is Linking / Referencing [11.], thus avoiding the retrieval of data that is not necessary in every query. After this step, the final document of the ais_navigation collection is shown in Figure 4.



```
"_id" : ObjectId("600398912bc0240991231acb"),
"mmsi" : 205204000,
"ts" : 1443760163,
"location" : {
        "type" : "Point",
        "coordinates" : [
                -4.9623785,
                48.25611
        ]
},
"nav_status" : {
        "id_status" : 0,
        "definition" : " under way using engine"
},
"nav_metadata" : {
        "turn" : -24,
        "speed" : 6.9,
        "course" : 97,
        "heading" : 97
},
"ship_metadata" : {
        "imo" : 0,
        "callsign" : "ORJH ",
        "shipname" : "NATO WARSHIP A960 ",
        "ship_type" : {
                "id_detailedtype" : 35,
                "id_shiptype" : 7,
                "type_name" : "Dredger"
        },
        "mmsi_country" : {
                "country_code" : 205,
                "country" : "Belgium"
        }
},
"grid_id" : ObjectId("600359f3044223864a6e410f")
```

Picture 4. ais_navigation document

4. The collections countries, fishing_port, world_seas, world_port_geo, world_port_information and query_polygons were selected for possible queries that may

be asked about countries, seas or ports. No special pre-processing was performed, just the corresponding '.shp' files of the dataset were uploaded to MongoDB.

Figure 1 shows the modeling of the database, indicating what was mentioned in this chapter.

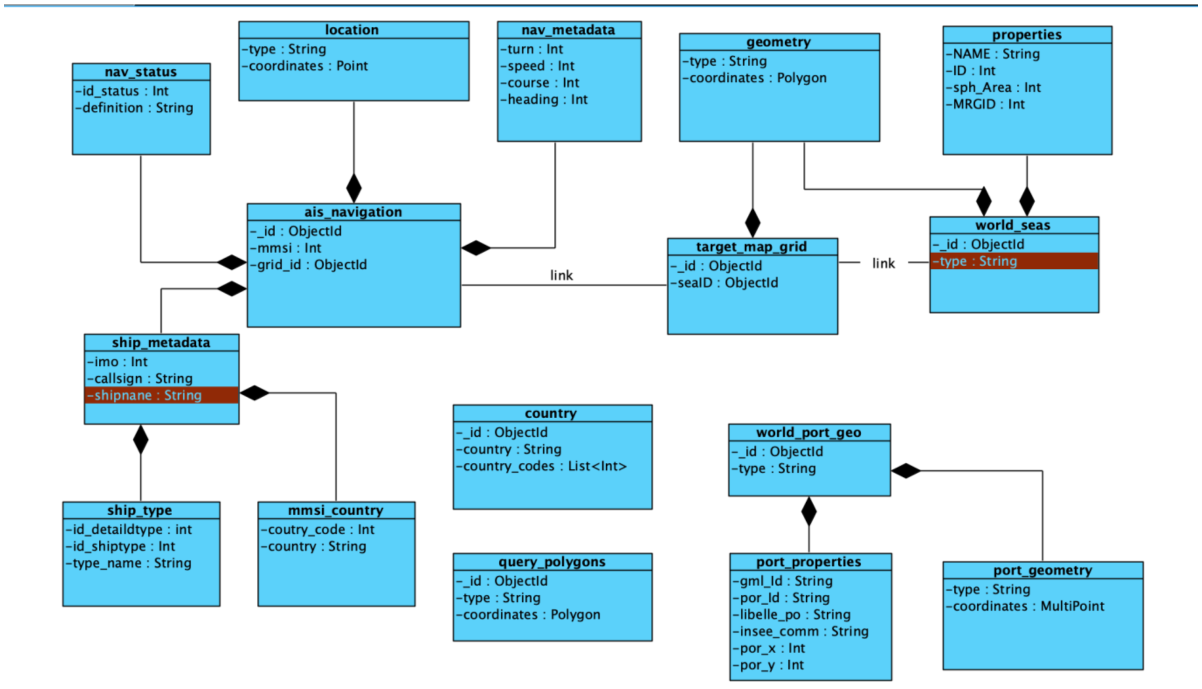


Figure 1. Database Modelling.

4.2 Indexing: Choosing the right indexes is a very important part of designing a database and is directly influenced by which use cases we want to support more efficiently. ***MongoDB*** supports indexing in ascending order (***field_id: 1***), descending order (***field_id: -1***) and hashed (***field_id: hased***). Additionally, it automatically generates an index in the «***_id***» field for each collection, however in order to optimize the performance of our system in the queries that the task targets, we resorted to creating the following indexes:

- **Indexes of the ais_navigation collection:**

Type and Index Field	**Imlementation**
MMSI: 1	Supports effective queries specific to ships.
ts: 1	Supports effective time queries.
Location: 2D-sphere	Supports effective spatial queries.
grid_id: 1	Supports efficient spatial queries and makes connection to grids more direct.
Compound index ts: 1 και mmsi: hashed	We created a combined index that combines timestamp and MMSI hashed in order to be used as a sharding key

- **Indexes of the target_map_grid collection:**

Type and Index Field	**Implementation**
geometry: 2D-sphere	Supports effective spatial queries.

Note: The 2d sphere is a special type of index that optimizes the performance of geospatial operators for executing queries in Mongo such as $geoWithin and $geoNear which will be discussed in sections 5.2 and 5.3.

4.3 Scalability And Partitioning

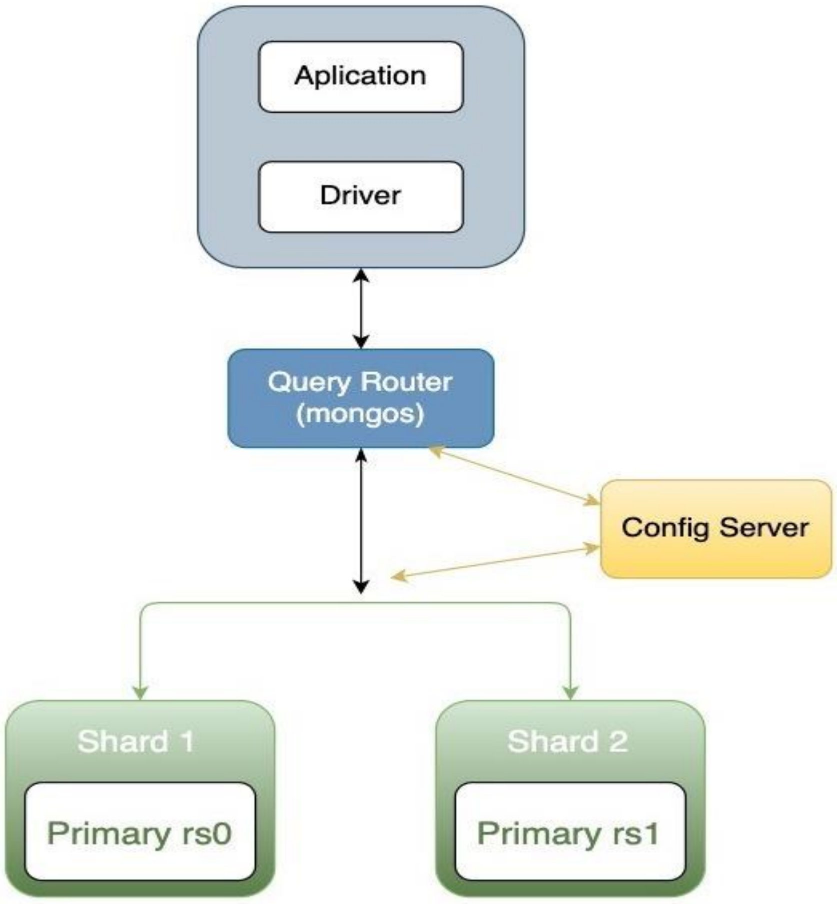
: In order for our system to be easily scalable, we configured in the Okeanos cloud a sharded cluster which consists of 4 Virtual Machines of the same specs (4 CPU, 8Gb RAM, 5gb hard disk and operating System Ubuntu Server 16.0.4). The reasons we resorted to this architecture were the following:

- **Vertical Scalability**: Vertical scaling is the process by which we vary the resources of each machine. Since our cluster consists of VMs in the cloud, we can easily customize the resources (either increase or decrease them) on each machine individually depending on the requirements of the application.
- **Horizontal Scalability:** It is called the process by which we add or remove machines (VMs) from our system. You achieve this with sharding which is a mechanism that MongoDB has and allows us to distribute our data to all the machines in the cluster. This distribution is achieved through the sharding key which is responsible for splitting the data into chanks. The chanks are a subset of the shard, they are sorted, the values they include range in a range (have min and max) and have a predefined size which by default cannot exceed 64mb/chank.
- **Cluster Architecture:** Given the very limited resources we had, we resorted to the configuration of a sharded cluster consisting of 4 VMs (minimum requirement).
    1. One query rooter which is responsible for routing queries to shards.
    2. Two sharded nodes to which our data will be shared.
    3. One configuration VM which is responsible for the configuration between the shards and the query rooter.

- **Sharding key Selection:** The selection of the sharding key was made with the criterion of the best and most balanced partition of our data in the shards. A compound index consisting of time ts in ascending order and **mmsi hashed** was selected. This choice was made because we wanted to support time segmentation to optimize spatio-temporal queries. However, choosing only a time key is considered bad practice in terms of vertical scalability. For this reason, the **mmsi hashed** was added to deal with this problem.
- **Shards Distribution Results:**

	Distribution	**Documents**	**Chanks**
Shard rs0	51.31%	9.488.244	161
Shard rs1	48.68%	9.003.165	162
Total	100%	18.491.409	323

- **Cluster Architecture Diagram:**



5. Query Database Analysis

5.1 SQL Like/Relational Queries: In this section of the paper we are asked to support on our basis various types of relational type queries.

5.1.1 Simple SQL Like Queries: Given the design of our database and more specifically the **ais_navigation** collection described above, any kind of "relational type" questions can be answered that target information about the type of ship, its navigational data (such as turn, speed, etc.), data of origin of the ship (country, imo code, callsign, etc.). Indicatively in this subsection we chose to execute the following use case:
Given a country (with its name e.g. "Greece"), we export the trajectories of all ships with the flag of that country. The requested query can be supported by our database in 2 ways. Initially the most direct way is to search for ships with the country_name which is provided as information in the ship_metadata of the ais_navigation collection. However, this approach is not optimal because it does not use any index of those we have defined in the collection. More specifically, to execute this query in this way for all ships with Greek flag, it takes 148.31s (very bad time for such a Query). For this reason, we chose to answer this question in a more complex way where its algorithmic implementation will be analyzed below.

- **Main Idea:** The main idea for executing this query is to take advantage of the fact that the country code (mmsi_country_code) is contained in the first 3 digits of the mmsi index. The problem with this implementation is that the use case provides us with the name and not the password. At the same time, each country can have more than one code.

- **Step 1st , export all the codes of the target-country:** First of all, we search the *countries collection* where the countries are registered in order to export all the codes of the target-country.

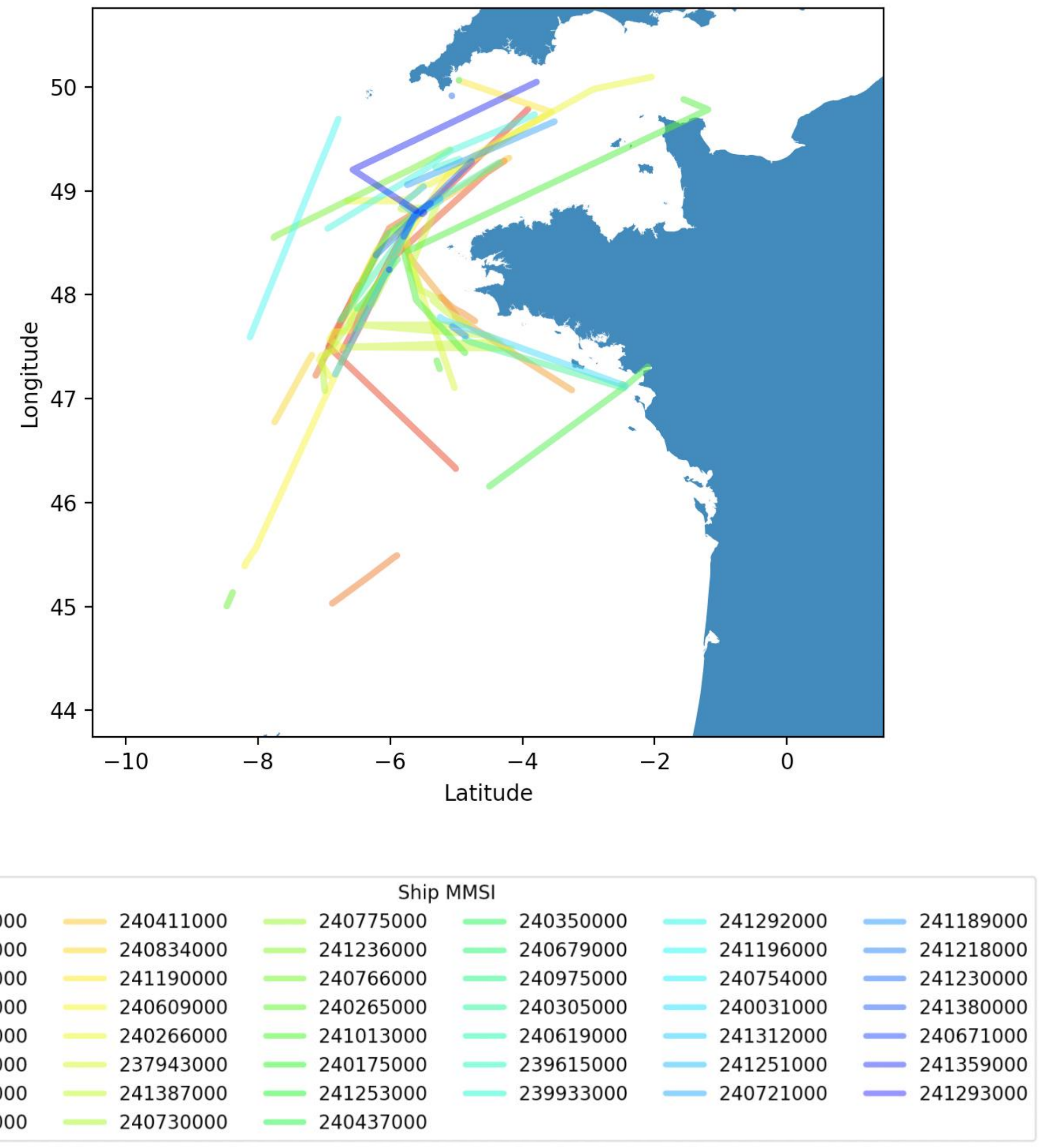
- **Step 2nd export and filter distinct MMSI:** Then, we extract from ais_collection all the distinct MMSI's and filter them in order to discard what does not contain in the first 3 digits any of the codes we exported in the previous step.

- **Step 3rd trajectories export:** Finally, we extract the trajectories for the MMSI of the previous step using $in aggregation and group the results in terms of MMSI.

- **Query Planner results for executing the query above, in MongoDB:**
  The following results were calculated for the export of all tracks of ships with Greek flag. The queries were executed on the server and the ais_navigation (sharded) and countries collections were used.

Stage	**Shards**	**Query Plan Stage 1**	**Query Plan Stage 2**	**index**	**Results count**	**Stage Time**	**Total Time**
Step 1st	-	Find by country name in countries	-	-	4 country codes	0.81s	0.81s
Step 2nd	-	Find distinct MMSI	-	Mmsi	5055 distinct MMSI	0.89s	1.70s
Step 3rd	rs1, rs0	$match mmsi $in step 2nd filtred results	$group by 1) Grid_id 2) $push locations	Mmsi	65 ships 3619 pings	0.59s	2.22s

- **Graphical Representation of results.**



5.1.2 Complex SQL Like Queries**:** In this section, we aim to demonstrate that our database can effectively answer complex SQL Like queries that seek information from multiple fields of each document in the ais_navigation collection. To be more specific, we will extend the use case of section 5.1.1 in order to accept multiple countries and to search for information about the ship_type contained in ship_metadata, while also applying time constraints. The question we will aim to answer in this section is:

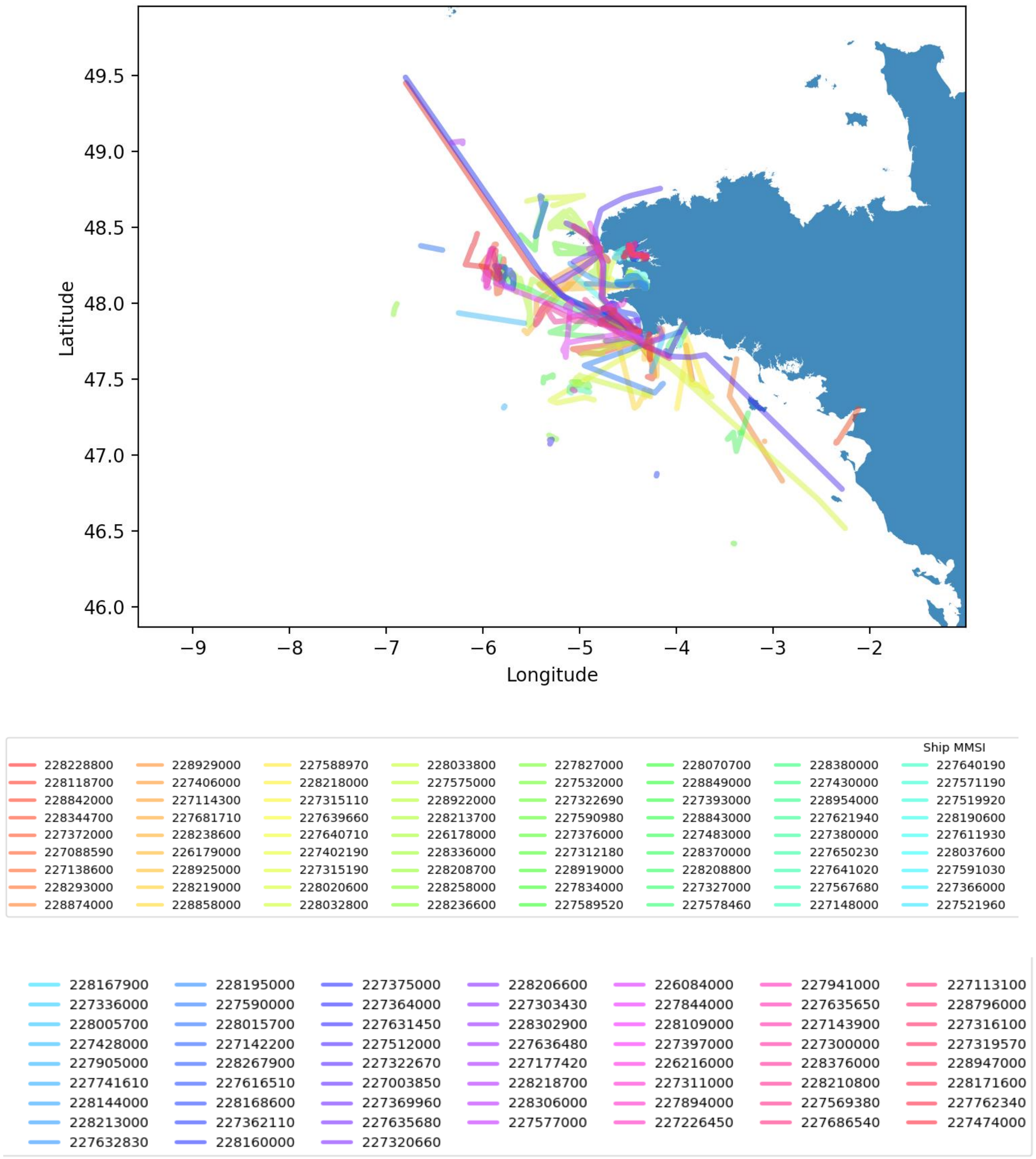
Find the trajectories for all French and Greek Dredger type ships from the time 01/12/2015 @ 4:54 pm (UTC) and for the next 72 hours.

- **Modifications in the case of the section 5.1.1:** First we modify the first step of the query so that it accepts a list of countries and then we adjust the query to the data base in step 3rd according to the requirements.

- **Results of Query Planner for executing the query above in MongoDB:**
  The queries were executed on the server and the **ais_navigation** (sharded) and **countries** collections were used.

Stage	**Shards**	**Query Plan Stage 1**	**Query Plan Stage 2**	**index**	**Results count**	**Stage Time**	**Total Time**
Step 1st	-	Find country $in country names from countries	-	-	7 country codes	0.28s	0.81s
Step 2nd	-	Find distinct MMSI	-	Mmsi	5055 distinct MMSI	0.58s	1.70s
Step 3rd	rs0	1) $match mmsi $in Step 2nd filtered results 2) $match ship_metadata. ship_type. type_name = Dredger 3) ts $le,$gt	$group by 1) Grid_id 2) $push locations	Sharding Filter ts_1	433 ships 76798 pings	5.44s	2.22s

- **Graphical Representation of results.**


49.5
49.0
48.5
48.0
47.5
47.0
46.5
46.0
Latitude
−9
−8
−7
−6
−5
−4
−3
−2
Longitude
Ship MMSI
228228800
228118700
228842000
228344700
227372000
227088590
227138600
228293000
228874000
228929000
227406000
227114300
227681710
228238600
226179000
228925000
228219000
228858000
227588970
228218000
227315110
227639660
227640710
227402190
227315190
228020600
228032800
228033800
227575000
228922000
228213700
226178000
228336000
228208700
228258000
228236600
227827000
227532000
227322690
227590980
227376000
227312180
228919000
227834000
227589520
228070700
228849000
227393000
228843000
227483000
228370000
228208800
227327000
227578460
228380000
227430000
228954000
227621940
227380000
227650230
227641020
227567680
227148000
227640190
227571190
227519920
228190600
227611930
228037600
227591030
227366000
227521960
228167900
227336000
228005700
227428000
227905000
227741610
228144000
228213000
227632830
228195000
227590000
228015700
227142200
228267900
227616510
228168600
227362110
228160000
227375000
227364000
227631450
227512000
227322670
227003850
227369960
227635680
227320660
228206600
227303430
228302900
227636480
227177420
228218700
228306000
227577000
226084000
227844000
228109000
227397000
226216000
227311000
227894000
227226450
227941000
227635650
227143900
227300000
228376000
228210800
227569380
227686540
227113100
228796000
227316100
227319570
228947000
228171600
227762340
227474000

5.2 Spatial Queries

: In this section of the paper, we are invited to study the methods provided by ***MongoDB*** for processing spatial queries. More specifically, we are called upon to implement spatial queries of range and K-nearest neighbors. MongoDB provides us with 3 basic operators with which we can apply to our database queries related to geospatial coordinates, while these 3 operators presuppose the documents we have registered in our database and follow the ***geoJSON*** format, This practically means that they must contain a ***nested document*** of the form:

```
<field>: { type: <GeoJSON type> , coordinates: <coordinates> }
```

It is important at this point to emphasize that the order of the coordinates must first be ***longitude*** and then ***latitude*** for the ***GeoJSON*** format to be valid.

The characteristics of the operators are described below:

1. ***$geoWithin Operator***: The ***$geoWithin*** operator can be applied either using the ***$geometry*** and ***$centerSphere*** properties to handle global queries or using the ***$polygon***, ***$box*** and ***$center*** properties to handle flat coordinate queries. It does not require a spatial index (neither ***2D*** nor ***2D-sphere***) to execute the queries above, however having such an index improves query performance. This operator is supported by **find** and **aggregate** queries.

2. **$geoIntersects Operator:** The **$geoIntersects** operator only supports spherical queries using only the ***$geometry*** property which accepts a **GeoJSON** argument. It does not presuppose the existence of a spatial index (neither **2D** nor **2D-sphere**), however the existence of such an index improves query performance and, finally, supports **find** and **aggregate** queries.
3. **$ geoNear Operator:** The **$geoNear** operator supports flat and spherical queries, and also accepts as arguments the point that is the center of the search, the maximum and minimum radius, and the query type (spherical or flat). To be executed, it presupposes the existence of a geospatial data index either ***2D*** or ***2D-sphere*** and it is applied only to queries of aggregate type.

5.2.1 Range Queries

: In this section we will use the operators described above in order to run range queries in the database. The questions we will support are the following:

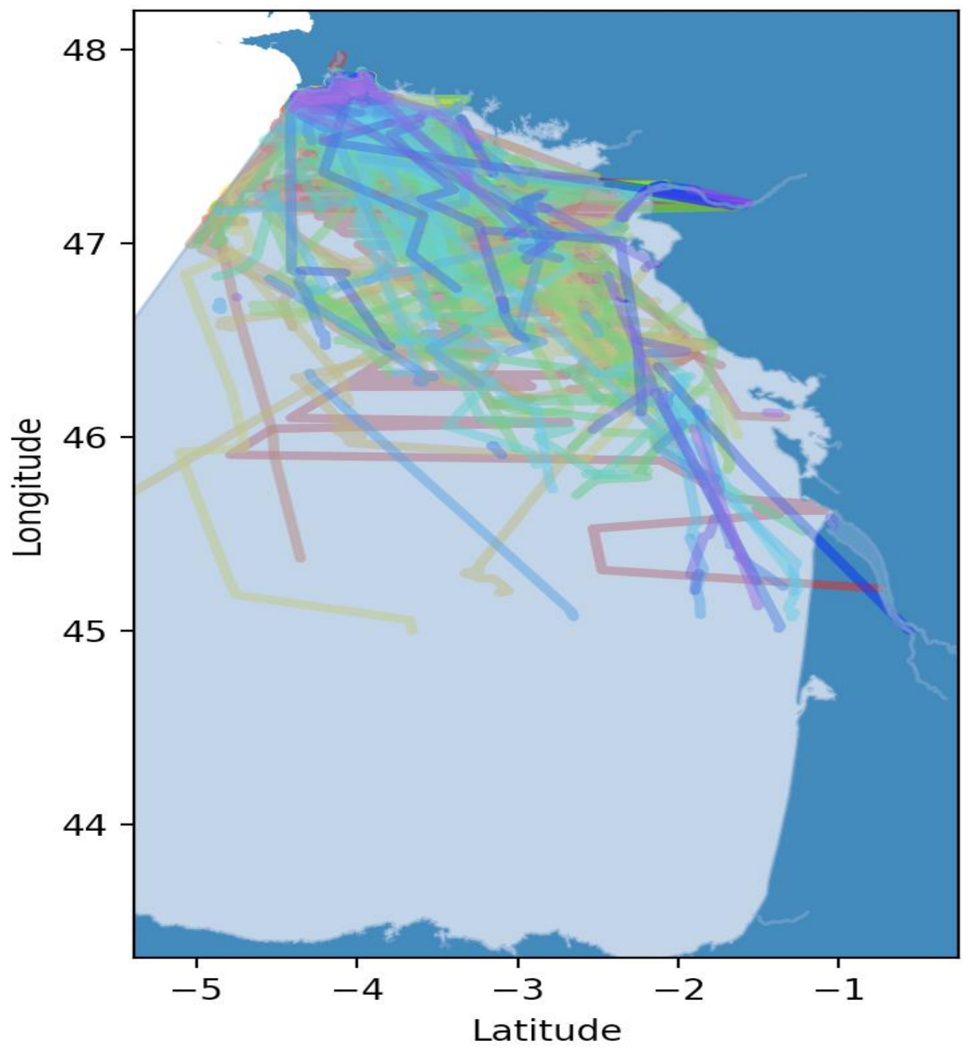
1. Find all the trajectories of the French-flagged ships recorded within the spatial boundaries (i.e., inside the polygon) of the Bay of Biscay.

- **Query Analysis:** To execute this query we will use the ship search engine based on the flag described in section *5.1* to which we will then apply the **$geoWithin** operator. This particular operator utilizes the **2D-sphere** index of the ais_navigation collection, where the query will be applied. As parameters we will submit the Bay Of Biscay polygon to the **$geometry** argument.

- **Results of the Query Planner for executing the query above, in MongoDB:** The queries were executed on the server and the *ais_navigation* (sharded), *world_seas* collections were used to find the polygon and the *countries* to find the country codes.

Stage	**Shards**	**Query Plan Stage 1**	**Query Plan Stage 2**	**index**	**Results count**	**Stage Time**	**Total Time**
Step 1st	-	Find world_sea where properties.NAME: {$eq: “Bay Of Biscay”}	-	-	1 sea	2.41s	2.41s
Step 2nd 6.1.1 query preprocessing	-	Get all county codes from countries, Find distinct MMSI from ais_navigation	-	Mmsi on ais_navigation	5055 distinct MMSI	1.70s	4.11s
Step 3rd	rs0, rs1	Mmsi $in Βημα 2ο results, $geoWithin polygon	$group by 1) Grid_id 2) $push locations	2d-sphere	719 ships 3.813.107 pings	291.01s	295.12s

- **Graphical representation of results.**



2. Find all the positions of the ships that sailed between 10 and 20 nautical miles from the port of Brest.
    - **Query Analysis:** To execute this query we will use the **$geoNear** operator which utilizes the ***2D-sphere*** index of the *ais_navigation* collection to which we will apply the query. As parameters we will submit the port point (in GeoJSON format), the maximum and minimum distance / radius from the central point, as well as that we want the query to be executed for spherical data type.

    - **Query Planner results for executing the query above, in MongoDB:**
    The queries were executed on the server and the *ais_navigation* (sharded)

and *world_port_geo* collections were used to find the coordinates of the port.

Stage	**Shards**	**Query Plan Stage 1**	**index**	**Results count**	**Stage Time**	**Total Time**
Step 1st	-	Find port where properties.libelle_po: {$eq: "Brest"}	-	1 port	0.78s	0.78s
Step 2nd	rs0, rs1	Find points using $geoNear	2d-sphere	719 ships 3.813.107 pings	327.60s	328.38s

- **Graphical representation of results.**
  The results of the query above, are too many to be represented graphically. In the next section we repeat the same query using the limit parameter in order to return the **k** closest entries to the point.

5.2.2 k-nearest neighbors Queries:

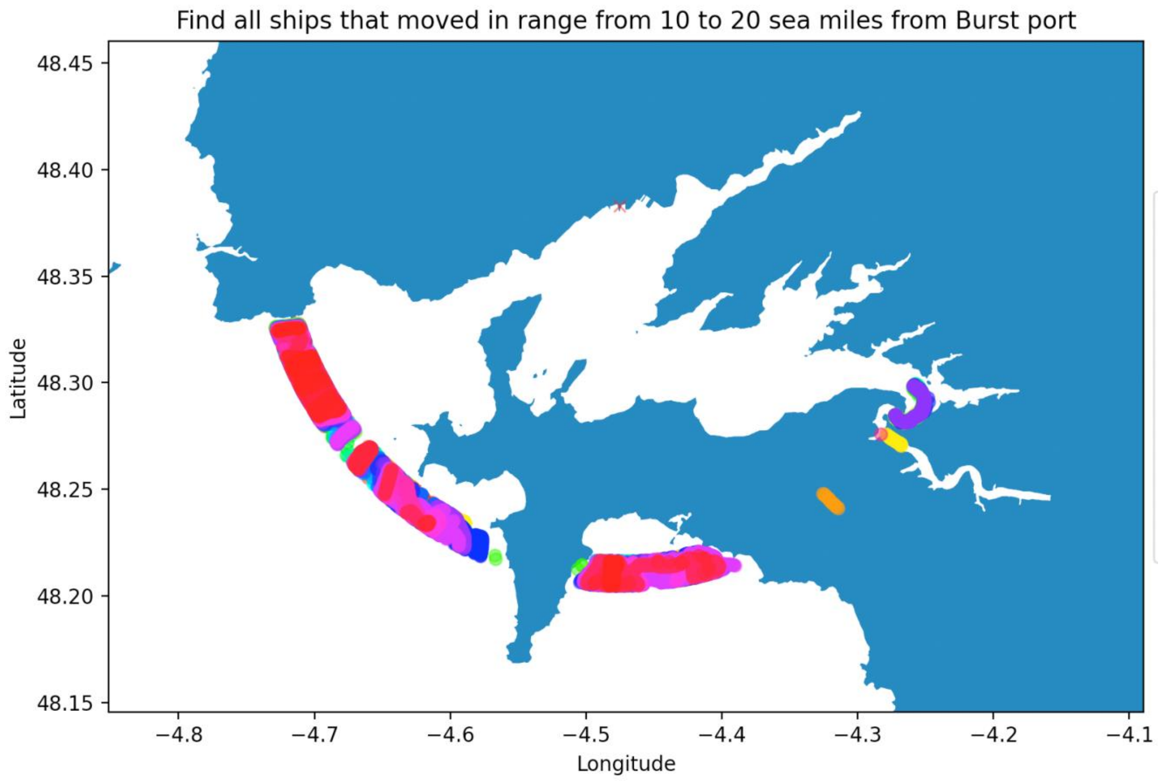
Using the ***$geoNear*** operator we described earlier, we can set him a property limit (K) that returns the **k** nearest records from the target point. In order to demonstrate this capability of the ***$geoNear*** operator, we repeat the query for the port of Brest with limit=100,000 points

- **Results of the Query Planner for the execution of the query above, in MongoDB:**
  The queries were executed on the server and the *ais_navigation* (sharded) and *world_port_geo* collections were used to find the coordinates of the port

Stage	**Shards**	**Query Plan Stage 1**	**index**	**Results count**	**Stage Time**	**Total Time**

Step 1st	-	Find port where properties.libelle_po: {$eq: “Brest”}	-	1 port	0.70s	0.70s
Step 2nd	rs0, rs1	Find points using $geoNear min dist 10, max dist 20 km	2d-sphere	100.000 pings	18.47s	19.08s

- **Γραφική αναπαράσταση αποτελεσμάτων.**



- **Note:** The 10 to 20 nautical mile range was deliberately chosen to reduce the search range of the **$geoNear** operator in which its performance is observed to be significantly affected as we increase the distance.

5.3 Spatiotemporal Queries

: Spatiotemporal queries are the largest part of queries overcome to spatial databases. In this section of the work we will apply time constraints in conjunction with the Mongo spatial operators we described earlier. The aim is to show that the incorporation of time constraints achieves a significant increase in the performance of Mongo spatial operators. In particular, the **$geoNear** operator makes comparisons for the entire search target range.

5.3.1 Range Queries

: In this section we will use the operators described in section *5.1* in conjunction with time constraints to execute range queries in the database. The queries we will support are the following:

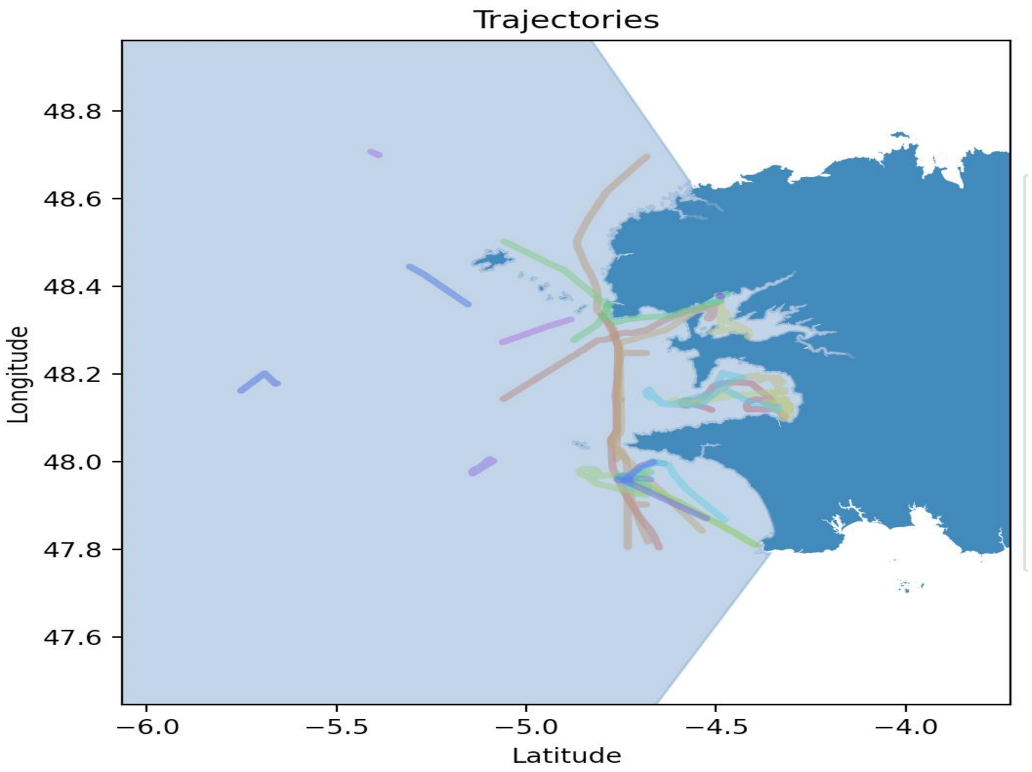
1. Find the trajectories of all French-flagged ships in the Celtic Sea for one hour.
    - **Query Analysis:** To execute this query as in the corresponding query in section *5.2.1*, we will use the flag search engine based on the flag described in section 5.1, where we will then apply the **$geoWithin** operator. In this particular operator we will submit as parameters the polygon of Bay of Biscay in the **$geometry** argument.

    - **Results of Query Planner for executing the query above in MongoDB:** The queries were executed on the server and the *ais_navigation* (sharded), *world_seas* collections were used to find the polygon and the *countries* to find the country codes.

Stage	Shards	Query Plan Stage 1	Query Plan Stage 2	index	Results count	Stage Time	Total Time
Step 1st	-	Find world_sea where	-	-	1 sea	2.62s	2.62s

		properties.NAME: {$eq: "Celtic Sea"}					
Step 3rd 5.1.1 query preprocessing	-	Get all county codes from countries, Find distinct MMSI from ais_navigation	-	Mmsi on ais_navigation	5055 distinct MMSI	1.15s	3.77s
Step 2nd	rs0	**1)** ts $le,$gt **2)** $in target_mmsi **3)** $geoWithin polygon	$group by 1) Grid_id 2) $push locations	**1)** sharding-filter **2)** index ts_1 **3)** 2d-sphere	39 ships 13.747 pings	14.76s	18.53s

- **Graphical representation of results**



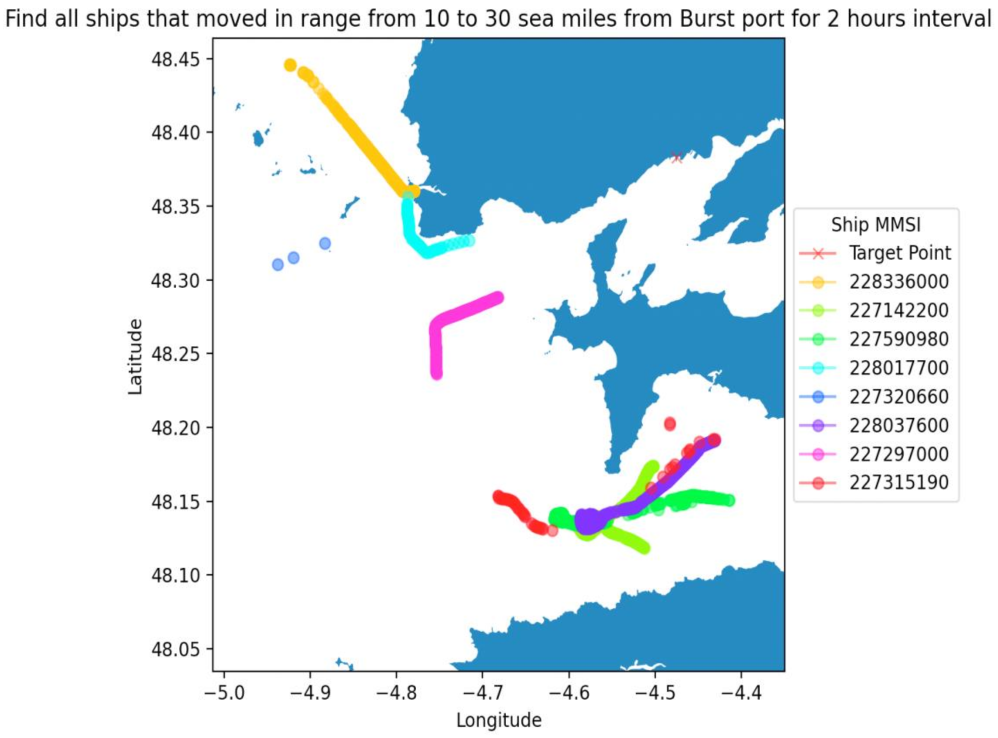
2. Find all ship blips observed within 10 to 30 nautical miles of Brest Harbor.
   - **Query Analysis:** In this query we will practically repeat the procedure we performed in the corresponding query of section 5.2.1 applying a time limit. Our purpose is to check if the query speed is actually improving.

- **Results of the Query Planner for executing the query above in MongoDB:** The queries were executed on the server and the *ais_navigation* (sharded) and *world_port_geo* collections were used to find the port coordinates.

Stage	Shards	Query Plan Stage 1	index	Results count	Stage Time	Total Time
Step 1st	-	Find port where properties.libelle_po: {$eq: “Brest”}	-	1 port	0.35s	0.35s
Step 2nd	rs0, rs1	Find points using $geoNear	2d-sphere	34 ships 9744 pings	220.03s	220.38s

- **Note:** It is noticed that geoNear is always run first by the query planner and in fact is the only option. This is explained by the fact that it must be the first stage of aggregation. This results in the other indexes not being utilized and whatever filtering is performed, is performed in the next step of aggregation. Additionally, in each case from the moment we use **$geoNear** both shards are accessed. From the execution of the above query but also from the execution of the corresponding query of section 5.2.1 there is an improvement in the execution time greater than 1.5 minutes. However, this reduction did not come from the execution of the first step of aggregation. So, we conclude that it is very important to choose the starting point and the distance for **$geoNear**, and that if there is no important reason to get accurate and classified information about the distance, it is better to choose the **$geoWithin** operator.

- **Graphical representation of results.**



5.3.2 K-Nearest Neighbors Queries.

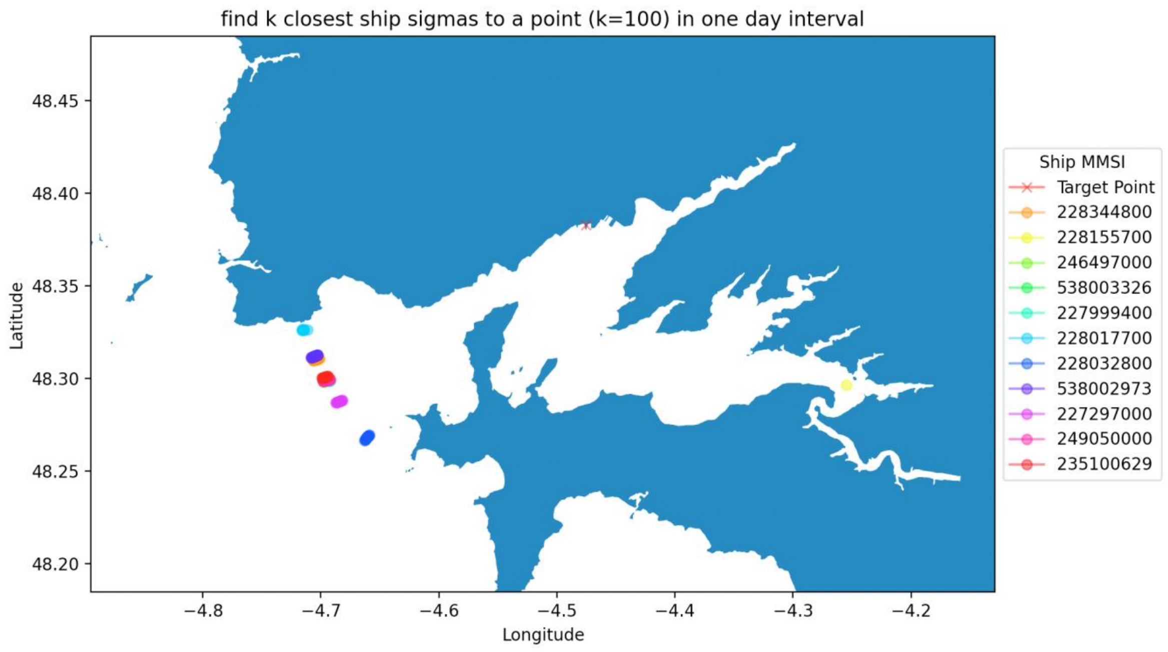
We repeat the previous question about the port of Brest with limit = 100 spots and a time period of one day.

- **Results of the Query Planner for executing the above query in Mongo:**
  The queries were executed on the server and the *ais_navigation* (sharded) and *world_port_geo* collections were used to find the port coordinates.

Stage	Shards	Query Plan Stage 1	index	Results count	Stage Time	Total Time

Step 1st	-	Find port where properties.libelle_po: {$eq: “Brest”}	-	1 port	0.70s	0.70s
Step 2nd	rs0, rs1	Find points using $geoNear min dist 10, max dist 20 km	2d-sphere	11 ships 100 pings	18.47s	19.08s

- **Γραφική αναπαράσταση αποτελεσμάτων.**



5.4 Distance Join Queries:

Distance Join Queries are a category of queries that geospatial databases are often asked to support and are used to define pairs of geospatial points that satisfy a particular condition. More precisely, the exact definition of Distance Join Queries is as follows:
Given two geospatial data sets **DA** and **DB** and a condition ϑ, the result of the Distance Join query is the set of pairs *oa*, *ob* such that *oa* belongs to **DA**, *ob* belongs to **D**B and condition ϑ (*oa, ob*) must be true.

5.4.1 Distance Join Using Map Grid:

In this category of queries, we have chosen to support the following use case:
Given a Fishing Constraint (i.e., a sea area e.g., Bay of Biscay), the MMSI of a specific ship and a distance greater than 10 km (supported by our grid), the issue is to calculate and return all the positions of the ships found in distance less than or equal to ϑ from the target-ship within the boundaries of the marine area.
Practically the above question is a ***self join*** of the navigational data collection (*ais_collection*) with itself. The Fishing Constraint factor simply acts as a filter to show that we can perform the most targeted search on the locations of the target-ship but as you will see below it could easily be omitted. This use case is described algorithmically in the steps below.

- **Step 1st Finding the Target Grid:** To prepare the above query we first load from MongoDB all the grids (*target_map_grid* collection) which do intersect with the given *fishing constraint* which is polygon **geoJson** type.

- **Step 2nd Detection of target ship blips:** We search from the collection with dynamic data (*ais_collection*) all the spots of the target ship that have been observed within the specific Fishing Constraint using the **$geoWithin** command of MongoDB and we group the results of this search in terms of *grid_id*. The results of this step consist of the total **DA** for our query.

- **Step 3rd Υπολογισμός Expanded MultyPolygon:** From the results we extracted in step 2nd we calculate an expanded Multypolygon which consists of the whole grid in which a position of the target ship was observed magnified by ϑ on each side. The calculation of this multyPolygon is very useful because it is practically the maximum limits at which you can find a Ping of another ship that can be considered valid.

- **Step 4th Search for blips within the Expanded MultyPolygon:** We search for blips observed by ships with a different MMSI from the target-ship within the Expanded MultyPolygon using the **$geoWithin** command (stage 1). With the above search we manage to access all the possible spots that can satisfy the ϑ condition, however there is a risk that the Expanded MultyPolygon will go beyond the limits of the original Fishing Constraint. For this reason, we performed another *aggregation stage* (stage 2) in which we filter the results of stage 1 so that they exist in the Fishing Constraint. Finally in this query we perform a **$group** stage during which we group the search results in terms of *grid_id*. The results of this step are the total **DB** for our query.

- **Step 5th Common Grid Base Filtering:** Since in the definition of use case we considered that ϑ> = 10 which is the minimum distance supported by our grid, we can conclude that all the observations of the 2 sets (**DA, DB**) are in the same grid, satisfy the ϑ condition. So, we performed the first filtering of the data we extracted from the previous step.

- **Step 6th Filtering Residual Results:** At this point in the query, we have drawn all the conclusions we can draw from the grid. So, to determine which of the remaining points of **DB** (**DB’**) satisfy the ϑ condition, we calculate the haversine distance as a Cartesian product of **DA** with **DB'**.

- **Results of the Query Planner for the execution of the queries above in Mongo as well as time results of the inApp calculations:**
  The following results were calculated for the ship with mmsi 228762000 within the Bay of Biscay. The queries were executed on the server and the *ais_navigatio*n (sharded) and *target_map_grid* collections were used.
  Bay of Biscay Fishing Constraints loaded with another query from the *world_seas collection* which lasted 1.60s and is not included in the following analysis.

Stage	**Query Plan Stage 1**	**Query Plan Stage 2**	**Query Plan Stage 3**	**index**	**Results count**	**Stage Time**	**Total Time**
Step 1st	$geointersects	-	-	2d sphere	2710	4.95s	4.58
Step 2nd	1) Filter by mmsi 2) $geowithin	$group by 1) Grid_id 2) $push locations	-	Mmsi	17 grids 952 pings	32.18s	37.13s
Step 3rd	-	-	-	-	Calculate 17 expanded grids by ϑ And the multyPolygon	0.65s	37.78s
Step 4th Shard rs0	**1)** $ne target_mmsi **2)** $geowithin multyPolygon	$geowithin Fishing Constraint	$group by 1) Grid_id 2) $push locations	2d sphere	39499 pings in 74 grids	17.13s	54.91s
Step 4th Shard rs1							
Step 5th	-	-	-	-	Detect observations (**DA, DB**) on the same grid	0.01s	54.92s
Step 6th	-	-	-	-	We calculate the haversine distance as a Cartesian product of ***DA*** with ***DB'***. (***DB'*** count= 22376)	576.62s	631.77s

- **Distance Join Using Map Grid Optimization:** Observing the results from the analysis of the previous section, it is easy to understand that step 6th, which calculates the distance of the points as a Cartesian product, has a great impact on the execution performance of the query. So since the Cartesian product is a very "expensive operation", as well as the fact that it is not appropriate to implement join between collections (in our case **Self Join**) in non-relational databases such as MongoDB, we had to use the GeoPandas library which is a Python library suitable for processing geospatial data as it supports Spatial Join queries and also has an R index for data uploaded to GeoDataframes.

The main idea is to extend the algorithm described in section 5.4.1 to perform a Spatial Join between the *expanded_grid* calculated in step 3rd and the non-matching ***DB*** locations calculated in step 5th. You can accomplish this by adding one more step between steps 5th and 6th that will do the following:

- **Processing with GeoDataframes:** First of all, we create the GeoDataframes for the expanded grids and for the **DB'** spots. And then we perform Spatial Join between the 2 GeoDataframes we produced. In this way we manage to determine the expanded grind that corresponds to each position of **DB'**.
- **Optimized Results Filtering:** By performing the above step we are able to determine exactly which expanded grid corresponds to each point of the ***DB***' set, and that each expanded grid comes from a grid. In this way we can reduce the complexity of the Cartesian product we performed above by performing multiple Cartesian products between the points of the target ship within each grid with the dots of the ***DB'*** set located in the corresponding expanded grid.

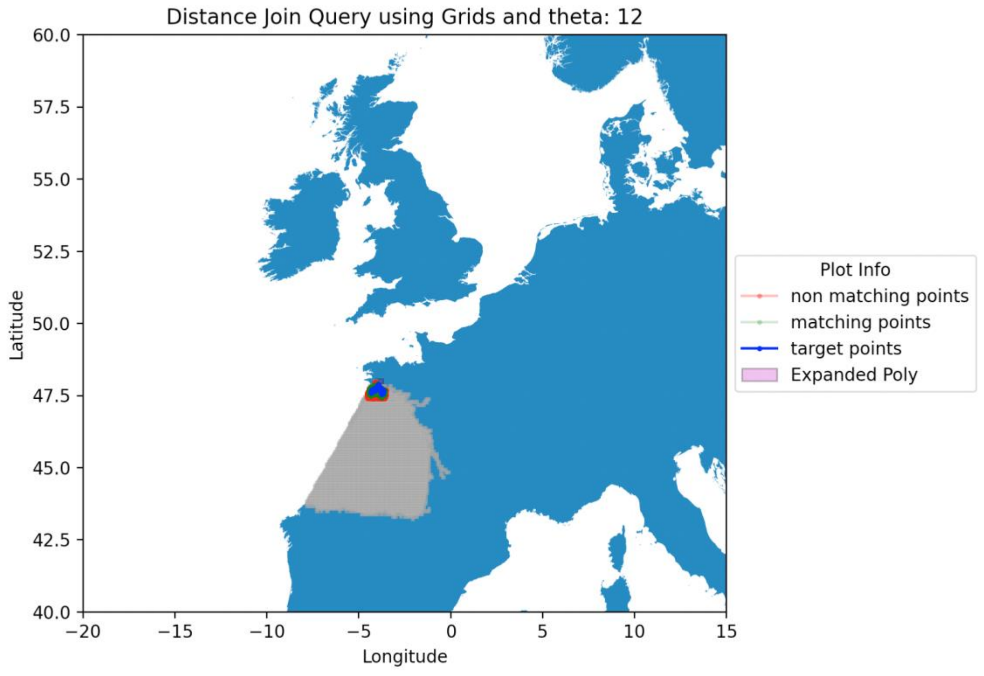
- **Results measures the query optimization:** After the optimization performed in the previous step the resulting table of results is the following:

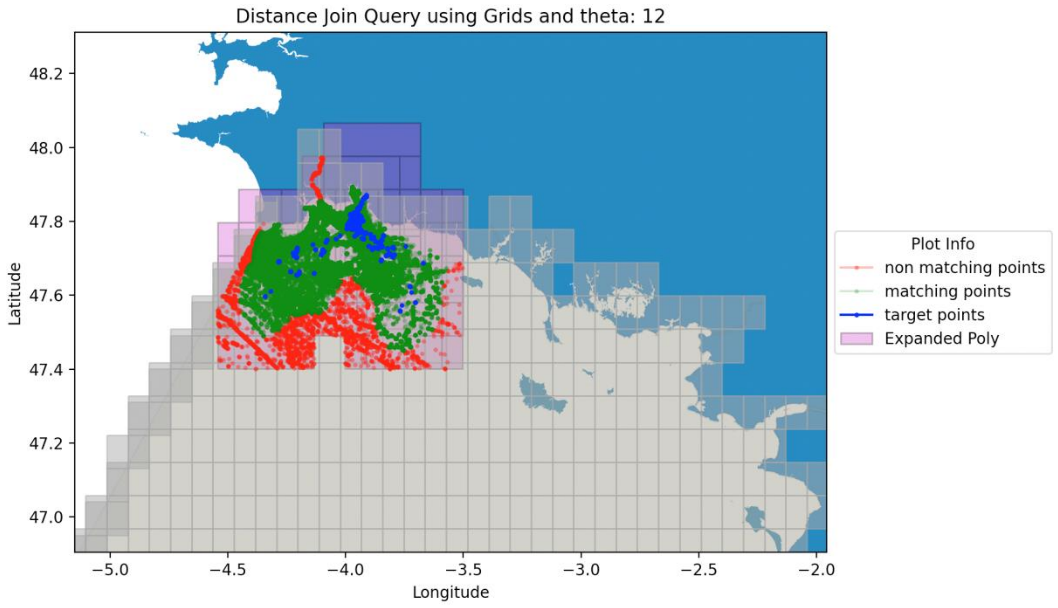
Stage	**Query Plan Stage 1**	**Query Plan Stage 2**	**Query Plan Stage 3**	**index**	**Results count**	**Stage Time**	**Total Time**
Step 1st	$geointersects	-	-	2d sphere	2710	4.58s	4.58s
Step 2nd	1) Filter by mmsi 2) $geowithin	$group by 1) Grid_id 2) $push locations	-	Mmsi	17 grids 952 pings	32.18s	37.13s
Step 3rd	-	-	-	-	Calculate 17 expanded grids by θ And the multyPolygon	0.65s	37.78s
Step 4th Shard rs0	**1)** $ne target_mmsi	$geowithin	$group by 1) Grid_id	2d sphere		17.13s	54.91s

Step 4th Shard rs1	**2)** $geowithin multyPolygon	Fishing Constraint	2) $push locations		39499 pings in 74 grids		
Step 5th	-	-	-	-	Detect (**DA, DB**) observations located on the same grid	0.01s	54.92s
Step 6th	-	-	-	-	Spatial join calculation between the results of steps 5th and 3rd	2.86s	57.77s
Step 7th	-	-	-	-	we calculate the haversine distance as a Cartesian product between the points of **DA** in a grid with the points of **DB**' in its expand	145.28	203.04s

- **Note:** It is worth noting that it measures the optimization, the largest Cartesian implemented was (525 x 5338) while with the previous implementation of the Cartesian product was (952 x 22376).

- **Graphical representation of results.**





5.4.2 Spatiotemporal Distance Join Using Map Grid:

In this category of questions we will support the use case that we analyzed in section 5.4.1 by adding a time constraint. In more detail, the updated definition of use case for this category is the following:

Given a Fishing Constraint (i.e. a sea area eg Bay Of Biscay), the MMSI of a particular ship, a distance ***d*** greater than **10 km** (supported by our grid) and a time interval ***T*** (which will be determined by the initial and the final timestamp of the space and will be counted in UNIX EPOCH) to calculate and return all the positions of the ships that during the time ***T*** were at a distance less than or equal to ***d*** from the target ship within the boundaries of the sea area.
Practically the above question as well as that of category 5.4.1 is a ***self-join*** of the collection of navigational data (ais_collection) with itself. More specifically, to fit this question into the Distance Join definition, the total points of the target ship that constitutes the ***DA***, the total of the other ais_collection ships that constitute the ***DB***, while the constraints of the Fishing Constraint, of the time interval ***T*** and the distance ***d***, compose the condition ***ϑ***.
This use case is described algorithmically using the same methodology as described in the previous section 5.4.1 with the only difference that in steps 2nd and 4th the time limit for submitting queries to the database has been added.

- **Query Planner results for the execution of the above queries in Mongo as well as time results of inApp calculations:**
  Below are three tables that describe the execution of the query using both the normal and the optimized algorithmic implementation we described earlier in order to compare results in terms of query execution time.
  The following results were calculated for the ship with mmsi 227300000 for the period from 02/12/2015 to 02/12/2015 (1448988894 and 1449075294 in UNIX timestamp) within the limits of the Bay of Biscay. The queries were executed on the server and the *ais_navigation* (sharded) and *target_map_grid* collections were used.
  Bay of Biscay Fishing Constraints loaded with another query from the *world_seas* collection which lasted 1.60s and is not included in the following analysis.

Joint steps of executing the two implementations.

Stage	Query Plan Stage 1	Query Plan Stage 2	Query Plan Stage 3	index	Results count	Stage Time	Total Time
Step 1st	$geointersects	-	-	2d sphere	2710	4.79s	4.79s
Step 2nd Shard rs0	**1)** Filter by mmsi **2)** Filter by ts **3)** $geowithin	$group by 1) Grid_id 2) $push locations	-	Mmsi	2 grids 77 pings	6.21s	11.00s

Step 3rd	-	-	-	-	Calculate 2 expanded grids by d and the multyPolygon	0.01s	11.01s
Step 4th Shard rs0	**1)** ts $le,$gt **2)** $ne target_mmsi **3)** $geowithin multyPolygon	$geowithin Fishing Constraint	$group by 1) Grid_id 2) $push locations	1) Sharding Key, ts: 1, mmsi: hashed 2d sphere	151 pings in 15 grids	10.96s	21.961s
Step 5th	-	-	-	-	Detect observations **(DA, DB)** located on the same grid	0.01s	21.962s

Execution of simple implementation:

Βήμα 6ο	-	-	-	-	υπολογίζουμε την haversine distance ως καρτεσιανό γινόμενο του DA με το DB'. (DB' count= 114)	0.31s	22.27s

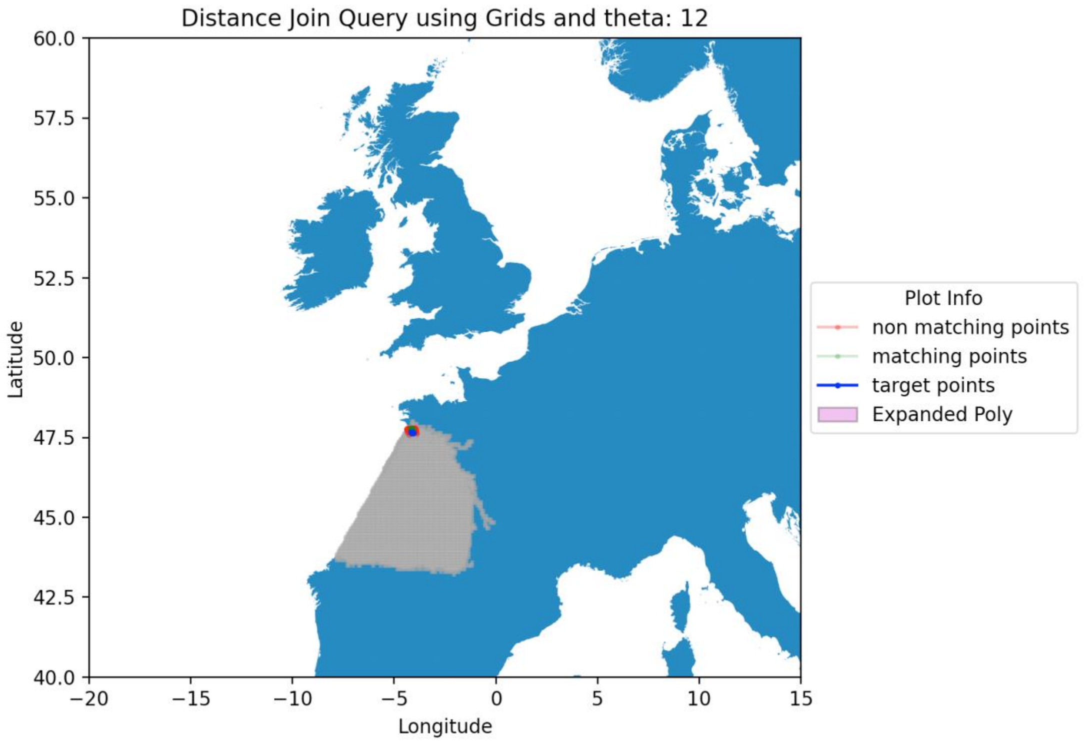
Execution of optimized implementation:

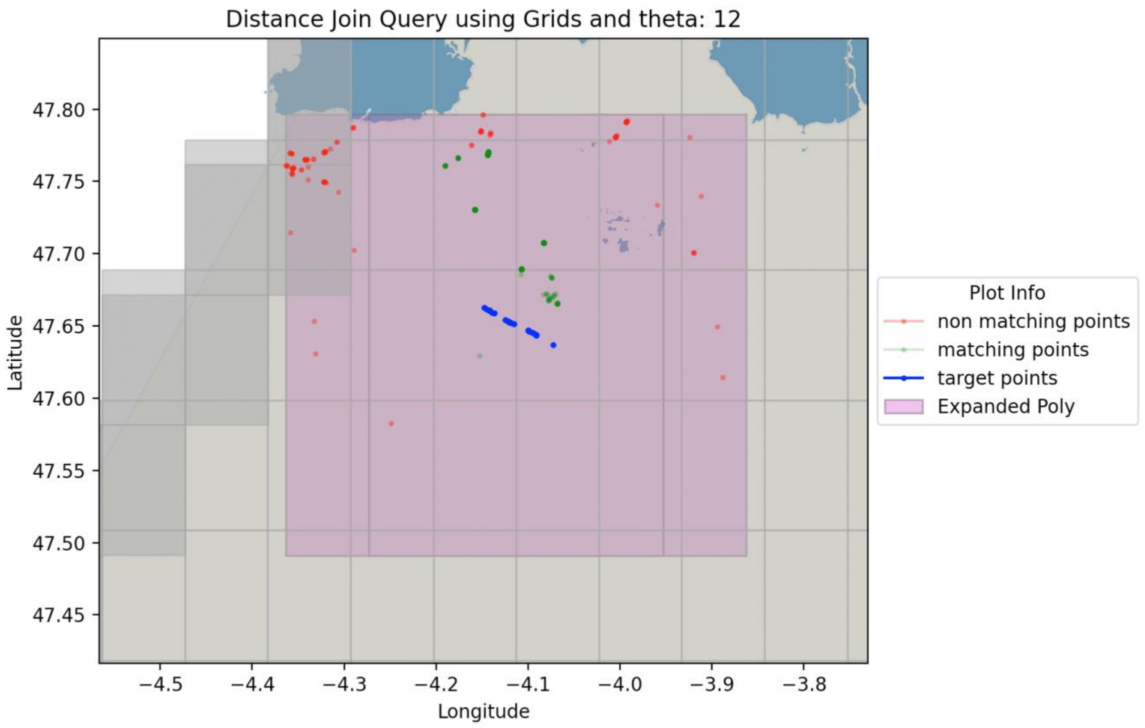
Step 6th	-	-	-	-	Spatial join calculation between the results of steps 5 and 3	0.03s	21.992s
					we calculate the haversine distance as a		

Step 7th	-	-	-	-	Cartesian product between the points of ***DA*** in a grid with the points of ***DB'*** in its expand	0.22s	22.21s

- **Note:** From the analytical execution of the two methods, we can observe that the times of the two methods are quite close, which is considered logical, as we limit the number of calculations to be done in step 5th of the 1st method significantly reduces the complexity of the Cartesian product. It is also worth noting that when performing step 4th the MongoDB uses the sharding key in order to limit the search to the ships that meet the time condition ***T*** and then applies the **2D** sphere index to look for a result within the polygon.
  Note: If we wanted to run the use cases of sections 5.4.1 and 5.4.2 without applying the Fishing Constraint constraint, we would simply skip step 1st and 2nd of Query Planer and go to step 4th of the methodology. More specifically, it means that the performance of our query would be improved by about 5s, as it is considered reasonable, because we are simplifying the ***ϑ*** condition .

- **Graphical representation of results.**


Distance Join Query using Grids and theta: 12
Latitude
Longitude
60.0
57.5
55.0
52.5
50.0
47.5
45.0
42.5
40.0
−20
−15
−10
−5
0
5
10
15
Plot Info
non matching points
matching points
target points
Expanded Poly


Distance Join Query using Grids and theta: 12
Latitude
Longitude
47.80
47.75
47.70
47.65
47.60
47.55
47.50
47.45
−4.5
−4.4
−4.3
−4.2
−4.1
−4.0
−3.9
−3.8
Plot Info
non matching points
matching points
target points
Expanded Poly

5.5 Execution of Trajectories Calculation Queries.

Trajectory calculation queries are the most basic type of query that a database of dynamically moving object data is required to support. In previous sections we have already implemented some export trajectory queries such as the trajectory of all Greek-flagged ships, however in this category of queries we are asked to calculate more complex questions, such as exporting trajectories that satisfy some spatiotemporal constraints, to compare trajectories and to detect similar as well as to find trajectories passing from point to point (within a radius ρ) at a specific time.

5.5.1 Detection of trajectories in a spatiotemporal Box:

The term Box in the processing of geospatial data refers to a square polygon which is defined by its lower left and upper right sides. MongoDB has a special **$box** operator that can be used in **$geoWithin** aggregation which supports such polygons generated by these 2 boxes. However, when you use aggregation **$geoWithin** with the **$box** operator it does not use the **2D-sphere** index, the **$box** operator only uses the **2D index** which is not supported by the ais_navigation collection. In the following example we will run 2 different scenarios for 2 different time periods (for one day and for one month) and compare the execution results between the **$box** and **$geometry** operators utilized by the **2D-sphere** index (to do this requires that we will transform inApp the coordinate pair to geoJson). The points that will be used to define the polygon are Bottom left (-5.084,48.161), Top right (-4.939,48.377).

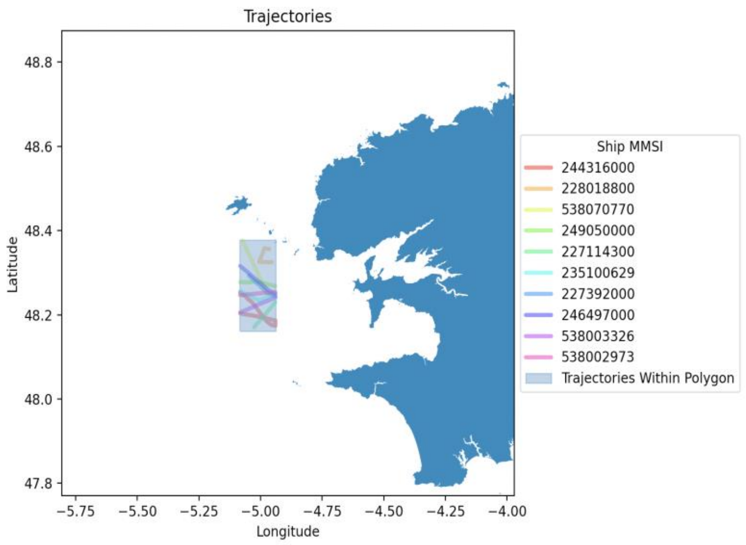
- **Results of the Query Planner for the execution of the above query for one day.**

Stage	**Shards**	**Query Plan Stage 1**	**Query Plan Stage 2**	**index**	**Ship 1 Results count**	**Ship 2 Stage Time**
Execution with $box	rs0-ship1	1) ts $le,$gt 2) $geoWithin using $box operator	$group by 1) MMSI 2) $push locations	ts_1	11 ships 1378 pings	1.63s
Execution with $geometry		1) ts $le,$gt 2) $geoWithin using polygon geoJson		ts_1		3.19s

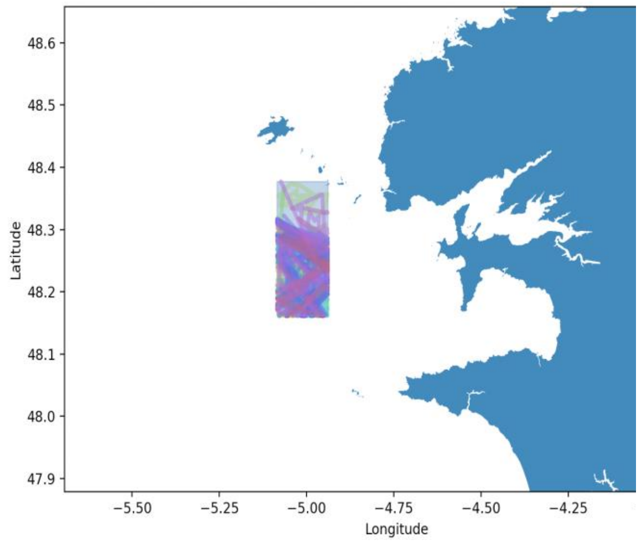
- **Results of the Query Planner for the execution of the above query for one month**

Stage	Shards	Query Plan Stage 1	Query Plan Stage 2	index	Ship 1 Results count	Ship 2 Stage Time
Execution with $box	rs0-ship1	1) ts $le,$gt 2) $geoWithin using $box operator	$group by 1) MMSI 2) $push locations	ts_1	92 ships 34343 pings	24.65s
Execution with $geometry		1) ts $le,$gt 2) $geoWithin using polygon geoJson		2D_sphere ts_1		4.88s

It is observed that in the execution of the query for one month the winning Plan of queryPlanner in the case of **$box** uses projection default and then applies a filter to the index ts_1. In the case of **$geomety**, on the other hand, it first applies Fetch based on the polygon and then utilizes the index in ts. As a result, there is a significant difference in execution time. On the other hand, in the data of one day it is observed that the execution of the query is faster although both queries use PRJECTION_DEFAULT and index ts_1.



The trajectories contained in the “Box” for a period of one day



The trajectories contained in the “Box” for a period of one month

5.5.2 Calculation of Similar Trajectories

In this category of queries, we are called to calculate queries of similar trajectories. More specifically, taking as an argument the trajectories of a ship, which is determined by exporting the total points of the ship (according to the MMSI) for a specific period of time (timestamp from, timestamp to in UNIX). We are called to export all the trajectories of the ships that took place during this time, they are at a specific distance ***d*** determined by the user from the target-trajectory and they satisfy a similarity threshold which we will define later. Our query is also called to support the finding of **K** more similar trajectories in terms of the target trajectory. The following is the algorithmic description of the query above.

- **Step 1st Extraction of trajectory-target:** First of all, we export the target-trajectory using the MMSI and the time period given by the user. While executing the query in Mongo, we sort the results using the **$sort** command and group them in terms of blips per grid on the **$group** stage of aggregation. Then we use an auxiliary function that we have constructed in order to group the points of the trajectory we exported and convert them to **lineString**. The fact that our results from the search of the previous step are chronologically sorted, guarantees us that the trajectory represented by the **lineString** will be correct. The reason we convert the trajectory to **lineString** is because we want our trajectory to be continuous and not to consist of distinct points. This makes it easier for us to export to the next step the grids with which it intersects.

- **Step 2nd, Export the grid:** Then, we export the grids with which the **lineString** trajectory, we exported in the previous step, interacts. That can be implemented by searching the **collection target_map_grid** for all the grids through which our trajectory passes using the MongoDB **$geoIntersects** command.

- **Step 3rd, MultyPolygon Calculation (optional):** In this step we join the grids we previously exported to produce a multiPolygon, also in case the user enters a distance **d** greater than **10km** in the system, before joining the grids to create the multiPolygon, we expand those by (d-10) / 2 to each side and thus create an expanded multiPolygon. Then a query about the collection with the grids (**target_map_grid**) in order to export all the grids (**target_grids**) that interact with the polygon we created in the previous step.

- **Step 4th, export trajectories within the target_grids:** Then we use the grids we exported previously to determine all the trajectories of the ships (excluding the trajectory of the target-ship) that were observed in it for the period of time that concerns us. This method was preferred over exporting trajectories from the expanded polygon because the base search of grid_id which is an indexed field in our database is faster than the search using the **2D-sphere** index. Finally, we convert all trajectory search results to **lineString** to be used in the next query.

- **Step 5th, Cleaning and evaluating the similarity of trajectories:** At this stage our goal is to determine which trajectories are quite similar to the target trajectories and to eliminate those that either do not meet the similarity criteria or create noise. Since the trajectories we have extracted so far, come from the expanded polygon, they all satisfy its criterion of being within a distance ***d*** from the target-trajectory. So in order to determine the similarity of the trajectories, we will use as a pointer the number of target_grids with which each trajectory interacts (including the target trajectory). You do this by using the **GeoPandas** library which is a **Python library** that we described earlier. The reason we use this library is to implement a Spatial join between the *lineString* of all the trajectories we have exported so far and the target_grids to determine the exact number of grids that intersect each trajectory. We then utilize this information to equate trajectories that interact with more than a percentage of the grids that the target-trajectory interacts with (default value= 50% of the target-trajectory grids) and less than a defined noise threshold (150% of the target trajectory grid). We also check that the total point-to-point distance of each *lineString* is greater than the corresponding percentage of the target-trajectory distance (default value=50% of the target-trajectory distance). In case the user has defined a number K of more similar trajectories, we return from the previous trajectories those that gathered the largest Similarity Index.

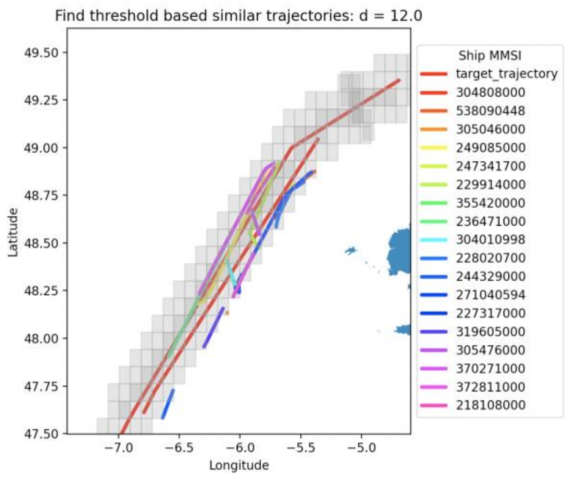
- **Note**: Given that our ship is moving and that the width of the polygon produced is specific and defined by ***d***, the majority of grids will be in the direction in which our ship is moving. So by applying the filters above, we succeed in rejecting vertical trajectories that consist of very few points (using the lower similarity barrier) and trajectories which make circles within the target-polygon created.

- **Results of the Query Planner for the execution of the queries above in MongoDB as well as time results of inApp calculations:** The following table shows the results of the Query for finding similar trajectories for two different ships. The first trajectory that will be analyzed is for the ship with mmsi 240266000 in the period from 01/12/2015 to 02/12/2015 (time From: 1448988894, time To: 1449075294 in Unix) which was sailing off the Celtic sea, where the parameters used were **d= 0**,

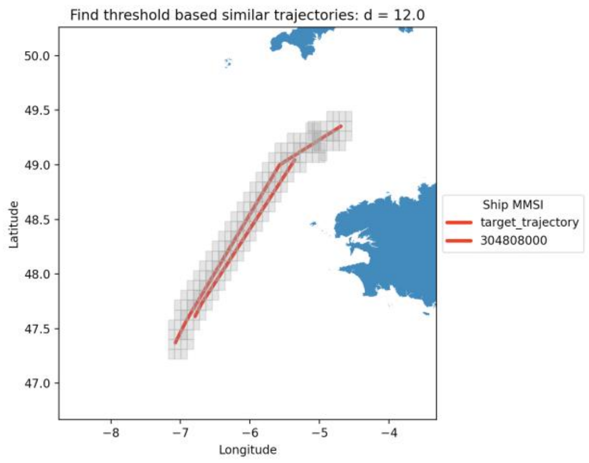
Similarity Threshold = 50% and **K= 0**. The second trajectory to be analyzed is for the ship with mmsi 227574020 in the period from 01/10/2015 5:16 am to 01/10/2015 : 6:06 am (time From: 1443676587, time To: 1443679590 to Unix) which was sailing inside the port of Brust. The queries were executed on the server and the **ais_navigation** (sharded) and **target_map_grid** collections were used.

Stage	**Shards**	**Query Plan Stage 1**	**Query Plan Stage 2**	**Query Plan Stage 3**	**index**	**Ship 1 Results count**	**Ship 2 Results count**	**Ship 1 Stage Time**	**Ship 2 Stage Time**
Step 1st	rs0-ship1 rs0-ship2	Fetch by target MMSI	$sort by timestamp (ts)	$group by 1) Grid_id 2) $push locations	ts_1_mmsi _hashed	24 pings	347 pings	0.77s	1.17s
Step 2nd		$geointersects with target ship lineString trajectory	-	-	2d sphere	51 grids	2 grids	0.13s	0.74s
Step 3rd		Calculate expanded grids from previous stage results	$geointersects with polygon consists of expanded grids	-	2d sphere	Returned 109 more grid_ids (so total grid_ids = 160)	-	0.65s	-
Step 4th Ship 1 Step 4th Ship 2	rs0-ship1 rs0-ship2	**1)** ts $le,$gt **2)** $grid_id $in target_grid_ids **3)** $ne target_mmsi	$group by 1) Grid_id 2) $push locations	-	1) sharding filter on ts 2) grid_id	689 pings from 46 ships	1853 pings From 21 ships	2.13s	0.13s
Step 5th		-	-	-	-	Detection of trajectories that satisfy the Similarity Criteria (optional κ-most similar)		0.06s	0.03s
						Με trajectory similarity 50% έχουμε 1 ενώ με trajectory similarity 20% έχουμε 4	Με trajectory similarity 50% έχουμε 3		
Total execution time		-	-	-	-	-		3.75s	2.43s

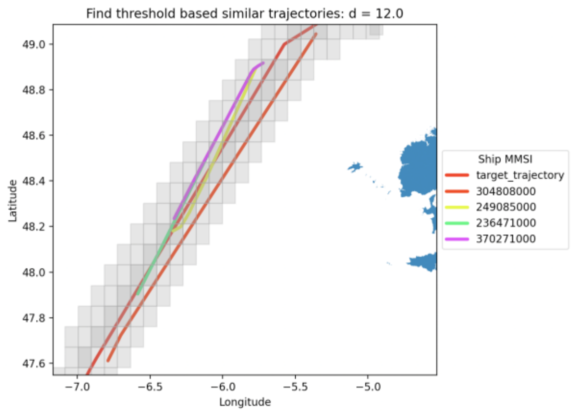
- **Graphical representation of results for the ship 1.**



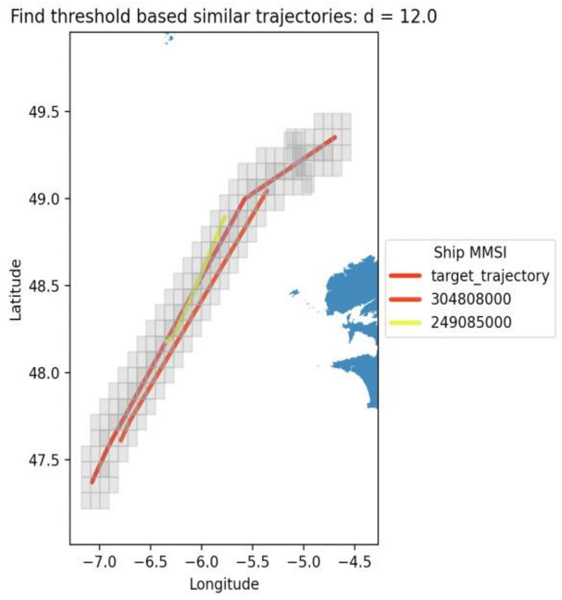
Όλες οι τροχιές που αλληλεπιδρούν με το πολύγωνο του πλοίου 1



Οι τροχιές που προκύπτουν από την εκτέλεση του αλγορίθμου με δείκτη ομοιότητας 50%

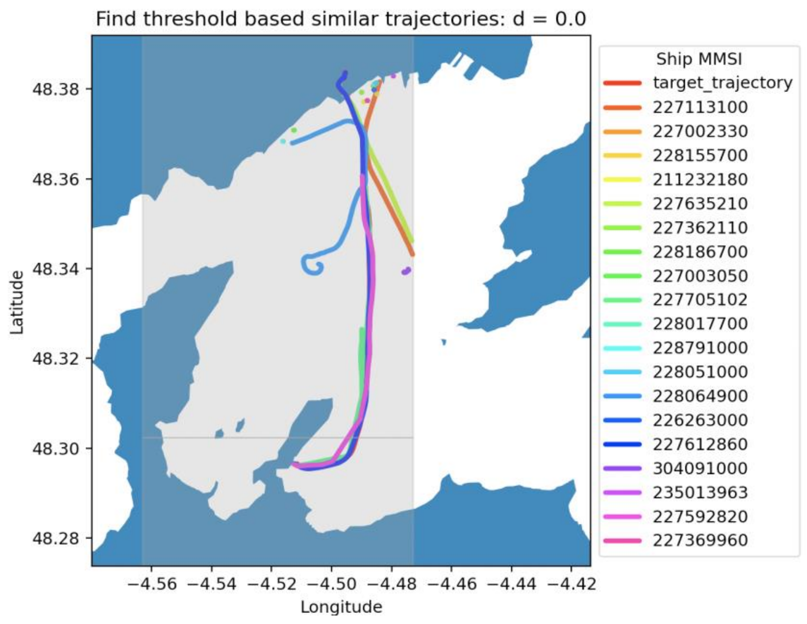


The trajectories resulting from the execution of the algorithm with a similarity index of 20%

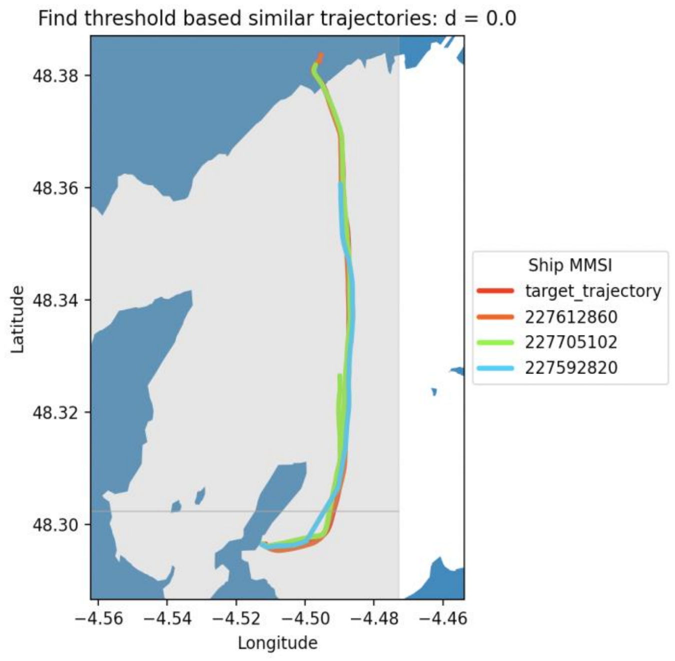


The trajectories resulting from the execution of the algorithm with a similarity index of 20% and k=2

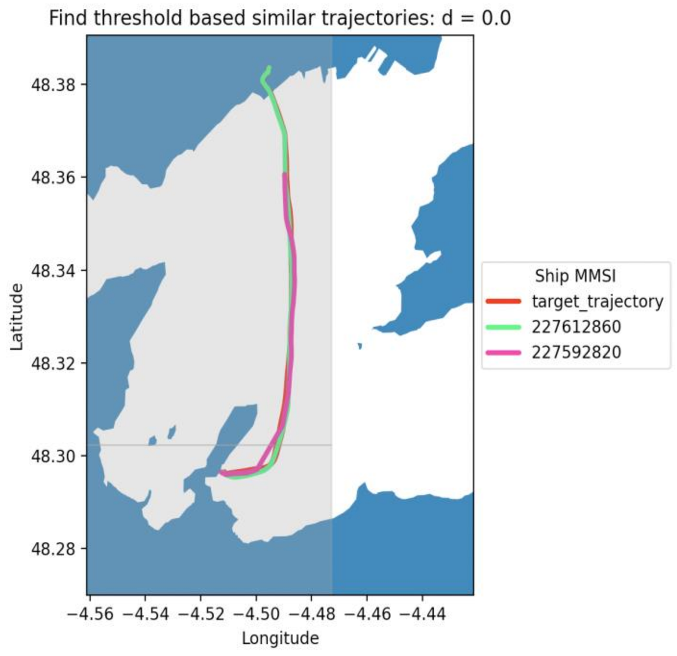
Graphical representation of results for the ship 2.



All trajectories that interact with the ship 2's polygon



The trajectories resulting from the execution of the algorithm with a Similarity Index of 50%



The trajectories resulting from the execution of the algorithm with a similarity index of 50% and k=2

- **Conclusions:** From the execution of the examples above, we observe that the algorithm works optimally when it comes to search trajectories within **10km** which is the default supported by the *grid*. It should also be noted that the algorithm works well for predicting similar trajectories given a "clean trajectory" i.e., it does not contain circles. In this case the initial trajectory must be pre-processed to avoid cycles and then the algorithm to be applied. It is also observed that due to the fact that the main element of the sharding key is time, all searches target a **shard**.

5.5.3 Complex Trajectory Queries:

In this category of queries we are called to execute complex queries that are related to trajectories. In more detail, the use case we are called to support is: Given three points defined by the user, we can export all the trajectories that passed from point A then from point B (within X hours) and finally from point C (less than 6 hours).
We have developed the above so that it can answer for more than 3 points, providing the user with the data of the time constraints from point to point.The following is the algorithmic description of the query above.

Note: Since the determination of whether a ship passes through a particular point is very specific, we chose to consider that a ship passes through a particular point only when it is within a radius of *ρ* kilometers from it. This distance is a configuration and can be changed if specified by the use case.

- **Step 1st Extract blips passing through the points:** First we extract all the ship positions observed within radius *ρ* for each of the target points using **$geoNear** aggregation, then we group the blips we exported to the MMSI and enter them in a list using of **$push** aggregation all timestamps observed and its coordinates for all pings (locations).

- **Step 2nd First stage of filtering:** Then to reduce the number of ships returned to us from the previous step, we check all ships that have a remark in the first point to have a valid observation in the last point of the search. In this way we manage to reject all ships that do not pass through all points.

- **Step 3rd Check point-by-point time constraints:** Next, we check for each vessel passing through the starting point (and maintained by filtering the previous step) if it has moved to the next point in less than the user-specified limit for that transition. This process is repeated for all points so that we retain only the ships that satisfy the condition in question.

- **Step 4th Export of trajectories for ships that satisfy the condition:** After identifying in the previous step all the ships that meet the initial condition, we perform for each of them, a query at the base in order to export its trajectory for the time period determined from the time the ship's position was located at the first point + the total time period set by the user in order for the ship to have passed through the last point.

- **Query Planner Results:** The following table lists the results of requests to the base of the Complex Trajectory Query for finding trajectories passing through 3, 4 and 5 points respectively. In this Query the inApp calculations are done in parallel for steps 2 and 3 and are not easy to represent in a table. The points as well as the time constraints used for the above question are listed in the following table. Finally, the distance ***d*** used is ***10km*** around each point.

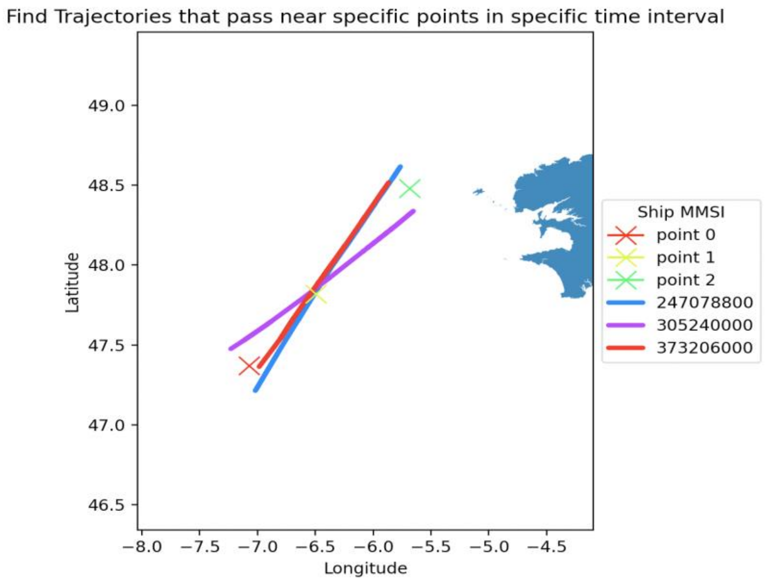
	Coordinates X	**Coordinates Y**	**Time Constraint**
1st Point	-7.072965	47.371117	(as initial point)
2nd Point	-6.494	47.820	point to point time = 3 hours
3rd Point	-5.683	48.480	point to point time = 3 hours
4th Point	-4.911	48.984	point to point time = 4 hours
5th Point	-3.683	49.465	point to point time = 4 hours

Stage	**Execution**	**Shards**	**Requests**	**Query Plan Stage 1**	**Query Plan Stage 2**	**index**	**Request Results**	**Stage Time**
Step 1st	3 points	rs0-ship1 rs0-ship2	3	$geoNear	$group by 1) mmsi 2) $push locations 3) $push ts	2d sphere	150 for point 1 825 for point 2 842 for point 3	3.53s
	4 points		4				504 for point 4	3.60s
	5 points		5				108 for point 5	4.22s
	3 points	After looking for a specific **id**, we	3	**1)** $grid_id $in target_grid_ids	$group by 1) Grid_id	1) sharding filter on ts 2) mmsi	1) 113 pings 2) 207 pings 3) 1293 pings	0.26s

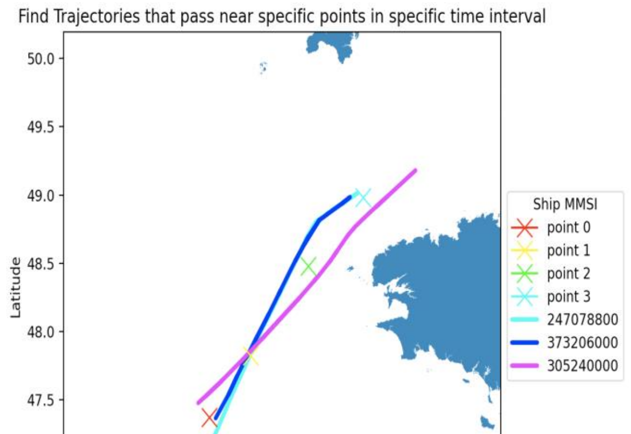
Step 4th	4 points	always target a **shard**	3	**2)** $ target_mmsi	2) $push locations		1) 296 pings 2) 255 pings 3) 1572 pings	0.31s
	5 points		2				1) 305 pings 2) 1598 pings	0.23s
Total execution time	3 points			-	-	-	-	3.69s
	4 points							3.94s
	5 points							4.51s

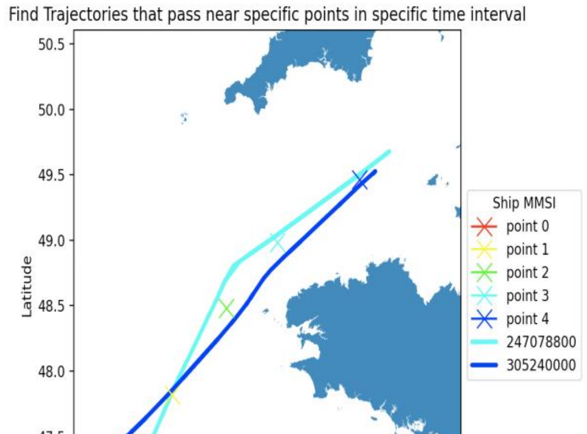
- **Result analysis:** As can be seen from the above results, in order to answer this query, multiple queries are performed in the database. However, it is observed that the response times of the question are quite satisfactory. You justify this with the fact that all requests are favored by **indexes** and the **sharding key**, as these are individual registrations. This question could be answered with a ***look up*** if it was not addressed to a ***sharded collection***.

- **Graphical illustration of results:**



All trajectories that pass through all 3 points within the time constraints





6. Conclusions and Future Extensions:

From the study and the analytical processing that was submitted in the field of geospatial data analysis, we came to create a system that can satisfactorily meet the needs of the work. Through dealing with the object, we understood concepts related to the modeling and management of non-relational databases. We explored the capabilities of **MongoDB** in both ***CRUD Operations*** and the scalability of our system through **Sharding**. Finally, this work enabled us to make a first contact with the management and processing of navigation data.
As future extensions we would aim at the configuration of a more complete *Grid* which consists of **super** and **sub layers** in order to optimize the performance in *Distance Join*, as well as queries to find similar trajectories. We would also like to configure a more complete cluster that also supports replication for our shards. Another worth mentioning is that it would also be interesting to use more data from the original Dataset such as *weather data* in order to get information on how these affect the ship's trajectories. Finally, we would aim to build a web user interface in order to optimize user interaction with our application.

7. Bibliography

1. Shipping and Global Trade: Towards an EU external shipping policy. Available online: https://www.ecsa.eu/sites/default/files/publications/2017-02-27-ECSA-External-Shipping-Agenda-FINAL.pdf.
2. International Convention for the Safety of Life at Sea (SOLAS), Chapter V: Safety of Navigation, Regulation 19, 13 December 2002.
3. MarineTraffic site. https://www.marinetraffic.com
4. Antonios Makris, Konstantinos Tserpes, Giannis Spiliopoulos, Dimosthenis Anagnostopoulos. Performance Evaluation of MongoDB and PostgreSQL for spatio-temporal data.
5. Pan Sheng and Jingbo Yin. Extracting Shipping Route Patterns by Trajectory Clustering Model Based on Automatic Identification System Data.
6. RAY, Cyril Naval Academy Research Institute DRÉO, Richard Naval Academy Research Institute; CAMOSSI, Elena; JOUSSELME, Anne-Laure, Heterogeneous Integrated Dataset for Maritime Intelligence, Surveillance, and Reconnaissance.
7. Zenodo site. Zenodo - Research. Shared.
8. Commission of the European Communities. Common position adopted by the Council with a view to the adoption of a Directive of the European Parliament and of the Council amending Directive 2002/59/EC establishing a Community vessel traffic monitoring and information system, Document COM 2008 310 final–2005/0239 COD; Brussels, Belgium, 11 June 2008; Available online: http://eurlex.europa.eu/LexUriServ/LexUriServ.do?uri=COM:2008:0310:FIN:EN:pdf (accessed on 30 May 2013).
9. Shannon Bradshaw, Eoin Brazil, Kristina Chodorow. MongoDB: The Definitive Guide, 3rd Edition. O'Reilly Media (ISBN: 9781491954461), December 2019
10. Χρήστος Δουλκερίδης: Διαχείριση Δεδομένων για Σχεσιακές και Μη-σχεσιακές Βάσεις Δεδομένων: MongoDB.
11. MongoDb documentation for Database References site: https://docs.mongodb.com/manual/reference/database-references.