Summary

This paper presents the problem of compressing the GPS data of unrestricted networks, such as the GPS data of ships. Previous studies have been conducted on restricted networks, such as roads for cars, using reference trajectories to describe the movement of objects. The issue with doing this in unrestricted networks lies in deriving meaningful references from the historical data, while maintaining a good coverage of the domain, and avoiding redundancies in the references. The solution presented in this paper as “STREAM” utilises the concept of model compression, as presented in its predecessors: ModelarDB and MOBY. The models used for STREAM include the vector-based model, a model presented in MOBY which uses a vector to predict future positions of the data, and the reference model which is a contribution of this paper.

The reference model suggested in this paper builds on a combination of the model based compression, as well as reference based compression suggested by REST. The model is designed to use a training set of historical data to construct a reference set. The historical data is split into sub-trajectories. Then each of these sub-trajectories are compared with the reference set which is indexed by an R-tree. If a sub-trajectories’ overlap with the reference set is below a threshold, it is added to the reference set. When compressing new data, the reference model tries to find a reference trajectory in the reference set which is within a certain error bound. If a reference trajectory is a candidate to represent the data, then when the next point of the data is given to the model, it compares it with the next index of the reference, to ensure that they are still within the error bound. This continues until either the end of the reference trajectory or until the data is further away from the correct point in the reference, then the error bound. When this happens the model of the data is saved as the id of the reference trajectory used, as well as the start and end indexes of the reference trajectory. The model can also handle data where every new point of data is within the error bound of the start index in the reference trajectory. In this case we record the amount of points in a row that can be represented with this one point in the reference trajectory, and save this count instead of the end index.

STREAM is split into two parts, a compressor which is intended to run on an edge node, and a cloud which runs on a server. The compressor's job is to compress all of the data it receives on the edge node, and then send it to the cloud. Meanwhile the cloud collects and stores the data from all of the edge nodes, and uses it to calculate new reference trajectories, which it can send back to the edge nodes. The cloud is also where the decompression methods for the models are located. In this paper the cloud has been left in a theoretical stage, as the effects of the reference based compression could be tested with just the compressor, sufficing for a proof of concept.

The results of testing done on STREAM as well as its baselines paint a clear picture of its superiority in flexibility and compression. When compared to MOBY, STREAM achieves more than twice the compression ratio. The inclusion of a model to compete with the vector-based model of MOBY, leads to STREAM being able to make intelligent choices between the two in order to leverage performance, either on compression or runtime.

This paper concludes that STREAM stands as a clear improvement on the performance of ship's GPS data, when compared to MOBY.

STREAM: System for Trajectory Reference Encoding And Modeling.

Martin Opal Lykkegaard
Computer Science
Aalborg University
moly18@student.aau.dk

Daniel Vilslev
Computer Science
Aalborg University
dvilsl18@student.aau.dk

June 15, 2023

***Abstract* - This paper tackles the challenge of compressing unrestricted GPS data, particularly those of ships, by proposing a novel solution named *STREAM*. The inspiration for this paper lies in the research gap for compressing GPS data of an unrestricted network via model compression. In order to improve on this gap, the field of trajectory compression was explored. Previous compression approaches, have primarily focused on restricted networks such as roads. STREAM builds upon the foundations of model based compression and introduces a novel reference model, which utilises a combination of model-based and reference-based compression. The reference model uses historical data to construct a reference set, with which new data is compared to and compressed by, within a certain error bound. STREAM's architecture comprises an edge-based compressor and a cloud server, with the latter being responsible for storing data, calculating new reference trajectories, and decompressing models. Comparative tests reveal STREAM to outperform baselines significantly, with over twice the compression ratio, by intelligently choosing between two models for enhanced performance. Consequently, STREAM shows promising improvements in compressing ships' GPS data.**

***Index Terms* - Compression, Database, trajectories, TSMS, AIS, Maritime**

I. INTRODUCTION

Data is a rising expenditure, as most markets look to utilise it in order to leverage it in favour of analytics. Multiple studies have been done on road networks, and how to compress these via some encoding to trajectories[15][14]. The methods developed in these studies often have the privilege of testing on data, which is restricted in what paths it can take. That is, if one were to collect the GPS data of cars in transit, it would be safe to assume that the latitude and longitude coordinates would point to some place on a road. As such, it is harder to find studies, which undertake data, which is more so unrestricted, such as the GPS data of ships. While one could argue that ships can only travel by sea, and as such is restricted, the comparison to roads isn't quite fitting as ships can travel without the idea of lanes, for most of their travels. With that said ships do have defined routes, as only one route can be the shortest between any two points. Can tendencies of historic data be used to describe the data to come? It is easy to imagine that this would be the case for static elements such as shipping routes, as ships often travel in predefined routes, as the routes between major ports are well established, and as such carry heavy precedence. If a solution was able to make sure that the data transmitted for these repetitive patterns isn't stored for each repetition, and instead uses some historical data as reference, it would be ideal. But what then when the destination isn't a point, but rather an area? what happens when one must deviate from the previously determined path? Can historic data describe not only general routes in an unrestricted network, but also describe the deviations from these. Or is it the case, that any system utilising historic data in a predictive manner should be bolstered with some means to describe smaller trends?

This paper is based on MOBY [2] by Agneborn et al. which looked into the usage of definable functions, which it coined as the term models. MOBY used different variations of these models, an example being one for linear functions and another for quadratic functions. One observation made after the development of MOBY, was that ships in general travel in repetitive routes[13]. This paper aims to build upon the initial findings of MOBY, with the inclusion of the using references to save cost on storage and transmission for maritime data. This paper will detail a system called *STREAM* which utilises model based approach in order to compress GPS data of ships. The system is split in to two parts: A compressor, designed to run on an edge node, and a Cloud, designed to run on a server. The system compresses GPS data to two different types of

models. The first model type used for this compression, is the vector-based model developed in MOBY. The second model type used, is a model presented by this paper, called the Reference model inspired by the REST framework[15]. This model uses historical data and an R-tree data structure in order to represent new data, by saving the new data as reference indexes to the historical data.

II. PRELIMINARIES

A. Introduction of running example

The data shown in Fig. 1 have been constructed in order to illustrate the different concepts discussed in this paper. These data points could be eight points from four different ships, sailing in a harbour.

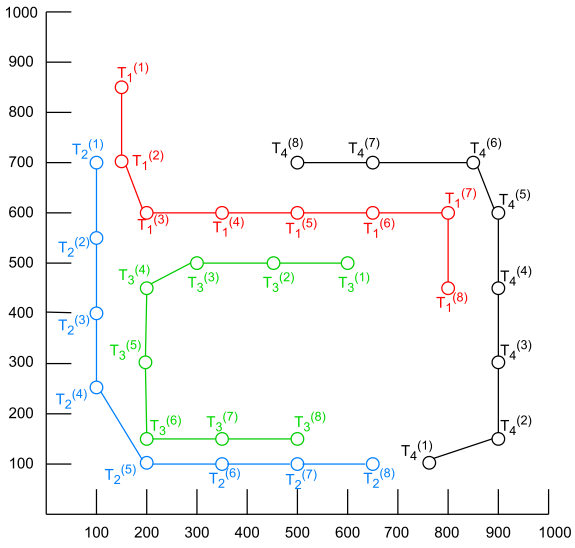


Figure 1: Example data with four movement trajectories: $\mathtt{T}_1$, $\mathtt{T}_2$, $\mathtt{T}_3$, and $\mathtt{T}_4$. The x and y axis of the figure are measured in meters.

While the data discussed in this paper is focused on AIS GPS data, which uses latitude and longitude, meters are chosen in this example as the small changes in values are hard to follow. The larger values of distances in meters are easier to work with in examples and are easier to understand. The error bound used in the examples will be 125m.

B. Definitions

The following section provides definitions of the various terms used throughout this paper.

DEFINITION 1 (TIME SERIES). *A time series* $\mathtt{TS}$ *is a sequence of data points consisting of pairs of timestamp* $\mathtt{t}$ *and the value* $\mathtt{v} \in \mathbb{R}$ *recorded at* $\mathtt{t}$*, giving us* $\mathtt{TS} = \langle(\mathtt{t}_1, \mathtt{v}_1), (\mathtt{t}_2, \mathtt{v}_2), (\mathtt{t}_3, \mathtt{v}_3), ...\rangle$*. The time series is sorted such that* $\mathtt{t}_{i+1} > \mathtt{t}_i$ *for each* $\mathtt{t}_i$*. Adapted from [2].*

DEFINITION 2 (TRAJECTORY). *A trajectory* $\mathtt{T}$ *is form of* $\mathtt{TS}$ *where* $\mathtt{v}$ *instead is a two tuple* $\mathtt{v} = (\mathtt{lat}, \mathtt{long})$*.* $\mathtt{lat}$ *and* $\mathtt{long}$ *are the latitude and longitude coordinates recorded at timestamp* $\mathtt{t}$*.*

DEFINITION 3 (SUB TRAJECTORY). *A sub trajectory is a part of a trajectory, denoted as* $\mathtt{T}^{(i,j)} = \langle(\mathtt{t}_i, \mathtt{lat}_i, \mathtt{long}_i), (\mathtt{t}_{i+1}, \mathtt{lat}_{i+1}, \mathtt{long}_{i+1}), ..., (\mathtt{t}_j, \mathtt{lat}_j, \mathtt{long}_j)$ *with* $\mathtt{i}$ *and* $\mathtt{j}$ *denoting indexes of the recordings in* $\mathtt{T}$*.*

DEFINITION 4 (REGULAR TIME SERIES). *A* $\mathtt{TS}$ *is regular if it has 2 or less data points, or the following is true:* $\mathtt{t}_i - \mathtt{t}_{i-1} = \mathtt{t}_{i-1} - \mathtt{t}_{i-2}$ *for all* $\mathtt{i} > 2$ *i the time series.*

DEFINITION 5 (IRREGULAR TIME SERIES). *A* $\mathtt{TS}$ *is irregular if it has 3 or more data points, and the following is true:* $\mathtt{t}_i - \mathtt{t}_{i-1} \neq \mathtt{t}_{i-1} - \mathtt{t}_{i-2}$ *for at least one* $\mathtt{i}$ *in the time series.*

DEFINITION 6 (MODEL). *A model over a* $\mathtt{TS}$ *is a function* $\mathtt{m}$ *which for each timestamp* $\mathtt{t}_i$ *with* $\mathtt{i} \geq 1$*, returns the value* $\mathtt{v}_i^{\mathtt{m}}$*.*

DEFINITION 7 (ERROR). *A value* $\mathtt{v}_i^{\mathtt{m}}$ *reconstructed from a model* $\mathtt{m}$ *can have a small error* $\delta = |\mathtt{v}_i - \mathtt{v}_i^{\mathtt{m}}|$*. For trajectories the* δ *will instead be the haversine distance[1], which can be found by using the formula shown in Eq.* (1)*.*

DEFINITION 8 (ERROR BOUND). *The error bound* ϵ *is the maximum allowed* δ *for the values created by a model* $\mathtt{m}$ *to the data points in the* $\mathtt{TS}$ *used to create the model.* $|\mathtt{v}_i - \mathtt{v}_i^{\mathtt{m}}| \leq \epsilon$ *for all data points in* $\mathtt{TS}$*.*

DEFINITION 9 (MODEL TYPE). *Model types are different types of functions that can be used to create* $\mathtt{m}$*. An example of this could be the linear function* $\mathtt{f(x)} = \mathtt{a} * \mathtt{x} + \mathtt{b}$*.*

III. RELATED WORK & FUNDAMENTAL THEORY

The field of time series while old and well documented, has recently seen a major up tick in interest and different undertakings. In most industries more data is being created and stored and as such the drive to improve the cost and effectiveness of data is high. This is also the case with time series data, where use cases such as monitoring of sensor readings, prediction of stock prices or saving storage costs of the data are present. To further understand the developments related to the goal of this paper, a number of inspirations and related works will be explored below.

A. Related Work

The paper titled "Time Series Management Systems: A Survey", Jensen et al.[9] is a survey done on a range of time series management systems(TSMS). The motivation for the paper was based on the observation that the volume of time series data is increasing. The paper covers systems from internet of things to data analysis and evaluation. This paper is integral to understanding the recent research surrounding time series, as it serves as a collective for readily available inspiration and theory. The survey found that there was a need for an application which did not specialise into a specific type of data, but still could handle large volumes of data.

In one of the papers studied in [9], Katsis et al. presented a TSMS called Plato[10]. Plato uses statistical models in order to compress spatio-temporal data. Plato is designed such that someone with knowledge for the domain the data is from can choose a statistical model to use for the data. The model then fits to the data such that querying is done on the model instead of data, making the querying faster. According to Katsis et al. this method separates the readings from the noise, which results in the data being of higher quality than the original data [10]. The problem with this system is that it requires the system administrator to know which statistical models work well with the data they are storing.

In the paper [7], Jensen et al. presents their system ModelarDB, which is intended to fill some of the gaps found in [9]. ModelarDB is a TSMS system which utilises a model-based approach for managing times series data. They suggest compressing time series data into models which can represent the data within a certain error bound, which can be zero. In the presented solution they use an algorithm to check which model gives the best compression as the data is ingested. ModelarDB is intended to give a high compression while also having low latency. Their experiments showed that ModelarBD has a better compression ratio than the other storage options they compared with in most cases, while also being considerably faster. ModelarDB also degrades more gracefully when outliers occurs in the time series.

Jensen et al, later released a second paper [8] which presented their improved version of the system called ModelarDB+ which uses a multi model group compression which automatically groups time series, which are correlated and tries to represent them with the same model, as long as this results in a good compression. If time series becomes uncorrelated it can also split them to separate models again.

In the paper [2] Agneborn et al. proposed a system, called MOBY, based on ModelarDB with a lightweight edge intended to run on small systems, and a cloud to store the data. MOBY is in the paper presented as 3 parts; compression, decoding, and ingestion. The compression part is what is intended to be used on the small systems, such as the computer on the communication antenna on a ship. It would then be sent from the edge to the cloud, where the decoder would convert the compressed data sent from the edge, into the format which ModelarDB stores its data in. After the data is decoded it would then be ingested into ModelarDB which stores the data in a database. Agneborn et al. also introduced a few new models in order to compress different kinds of data, one of which was a vector based model, used to compress position data.

Zheng et al.[15] introduces a framework to handle spatio-temporal trajectory compression and querying called REST. The general idea behind the paper is to create a reference set, which will be used to compress incoming trajectories. The compression is achieved by saving a subset of existing trajectories, rather than saving individual points for an incoming trajectory. It explores four different methods to build this reference set, based around different principles. These are Frequent Pattern-based Approach (FPA), Segment Redundancy Reduction (SRR), Trajectory Redundancy Reduction (TRR), and lastly a Compression-based Approach (PA). It also proposes a new metric, which it dubs as MaxDTW, which describes the maximum distance, pair-wise, between the points of trajectories. This metric is used to ensure, that the maximum error between a trajectory and its reference does not exceed MaxDTW, I.e some maximum threshold. Where spatial data, is referenced within some threshold, the compression of temporal data is also saved within some threshold however, unlike with spatial, should this threshold be broken the distance from the reference point is instead saved. The paper found that compressing trajectories using references had vastly improved storage, while the running time of execution was on par with state of the art competition.

Agrawal et al. [3] is a conference paper which is referenced by Zheng et al.[15]. In the paper methods of discovering frequent patterns stored in a database are presented. In essence the problem stated in the paper is to define all sequences of events that reoccur in the system. To reach this goal, a range of algorithms are introduced and evaluated. Moreover, the paper introduces five phases which it deems as integral to solving the problem. While the paper is relatively old, and didn't necessarily take memory and computation costs into account, it serves as a good source for understanding the problems and angles of entry to finding recurrent patterns in data.

Ranu et al. [14] is a paper covering an issue associated with the push to recording more data, that being inconsistent sampling rates. This issue can make distance and similarity metrics difficult to gauge. To handle this issue a novel distance method is formulated. The method utilises a form of interpolation in order to get distance measures where the differing sampling rates, normally wouldn't allow it. The method, along with a novel index structure,

allowed for a result yielding 5 times the performance on accuracy and robustness of the competitors.

ModelarDB[7] introduced the novel idea of model based compression, which on its own showed favourable performance, however it did not consider the semantics of positional data, such as GPS, which have two correlated data points, latitude and longitude. MOBY[2] had generally good compression, as noted in its testing, the addition of the vector-based model undertook the pursuit of compressing GPS data in a semantic manner, which ModelarDB had not. REST[15] on the other hand had another approach to the compression of GPS data. The fact, however is, that for MOBY, the compression of GPS data, wasn't the main focus. It was instead the construction of a system which could handle different domains of data. A symptom of this is that there was only one model, which could handle GPS data, which lead to a poorer performance on GPS data in MOBY, where despite these only consisting of two out of 22 columns, the vector-based model took up 25-60% of all created models in order to compress them. As such this paper looks to combine the idea and practice of model based compression, with the more positionally focused REST framework.

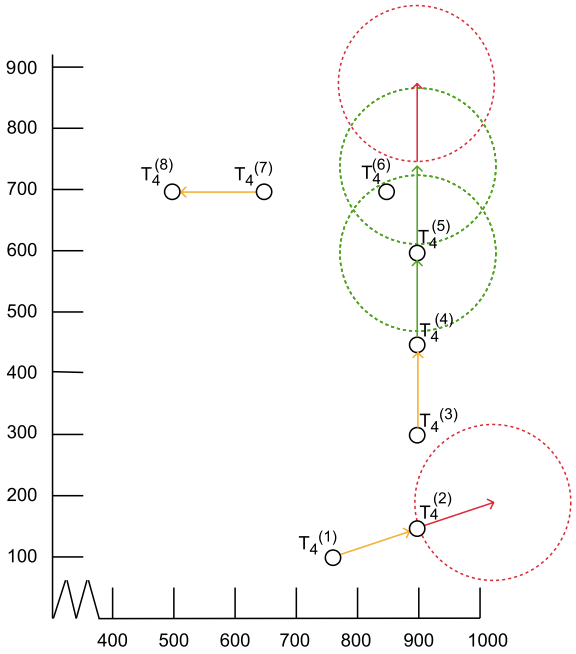


Figure 2: Visual representation of the Vector-based model, over 3 points, with $\epsilon = 125$

B. Fundamental theory

This section will cover vital components of more technical fundamentals of the paper. These will include a model of interest developed in MOBY, trajectory reference matching as developed for the REST framework, and also an R-tree data structure, also developed for the REST framework

a. Model based compression

Both MOBY[2] and ModelarDB[7], which MOBY is based on, use model based compression. During the development of MOBY and ModelarDB, a range of models were developed, for *STREAM* only the vector-based compression will be explicitly covered. In order to be utilised for large data sets without delay, all of these models need to be able to handle a stream of input values, without re-evaluating the entire domain of the function.

Vector-based model

The most titular model to *STREAM*, which was introduced in MOBY, is the vector-based model. This model was developed in order to compress spatial data points, such as GPS trajectories. Fig. 2 shows the execution of the vector-based model for T_4, as defined in Fig. 1. It should be noted that all points in the trajectory for this example are treated as having their timestamp equal to their index in the trajectory, So $T_4^{(1)}$ has timestamp = 1, $T_4^{(2)}$ has timestamp = 2 and so on. The principle of the vector based model, which can be seen in the figure, is that when two positions are consecutive, a vector can be constructed between them, denoted as a yellow arrow e.g. between points $T_4^{(1)}$ and $T_4^{(2)}$. These vectors are used to predict any future points, any such predictions are shown with red or green arrows, both of which will always be identical to the yellow arrow they are based on, but simply taking origin in the previous point. These vectors are also scaled with time, e.g. had $T_4^{(5)}$ had timestamp = 6 instead of 5 the vector predicting its position, would have doubled in length, had the timestamp been 4.5, it would instead be halved in length. Green arrows denote successful predictions, seen for points $T_4^{(5)}$ and $T_4^{(6)}$. Lastly red vectors signify failed predictions, as seen with T_4^3 which is further away from the vector constructed between $T_4^{(1)}$ and $T_4^{(2)}$, than ϵ. When predictions fail, new vectors are constructed between the point which was not found in the last prediction, and the next point on from that. For example, is the vector constructed between $T_4^{(3)}$ and $T_4^{(4)}$.

b. Haversine distance

As points in trajectories are given in latitude and longitude coordinates, as shown in Definition 2, it can be hard to tell how far apart two points from each other. To solve this issue the haversine formula[1] shown in Eq. (1) will be used to get a close approximation.

$$\begin{aligned} a &= \sin^2\left(\frac{\Delta lat}{2}\right) + \cos(lat_1) \cdot \cos(lat_2) \cdot \sin^2\left(\frac{\Delta long}{2}\right) \\ c &= 2 \cdot \operatorname{atan2}\left(\sqrt{a}, \sqrt{(1-a)}\right) \\ d &= R \cdot c \end{aligned} \tag{1}$$

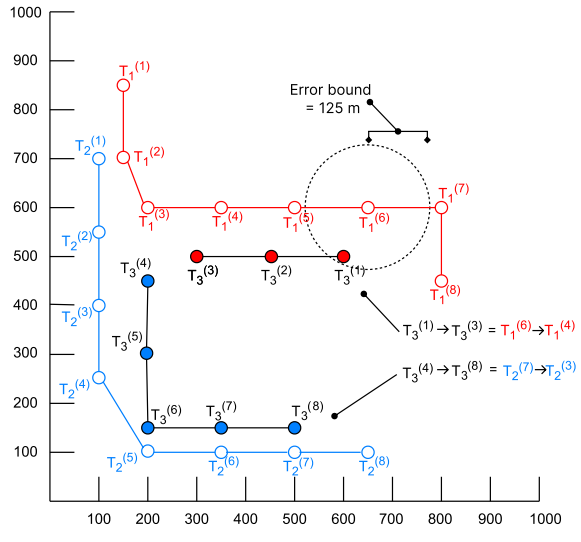


Figure 3: Trajectory matching over 2 $T_2^{(1,8)}$ and $T_1^{(1,8)}$, with input trajectory $T_3^{(1,8)}$, with $\epsilon = 125$

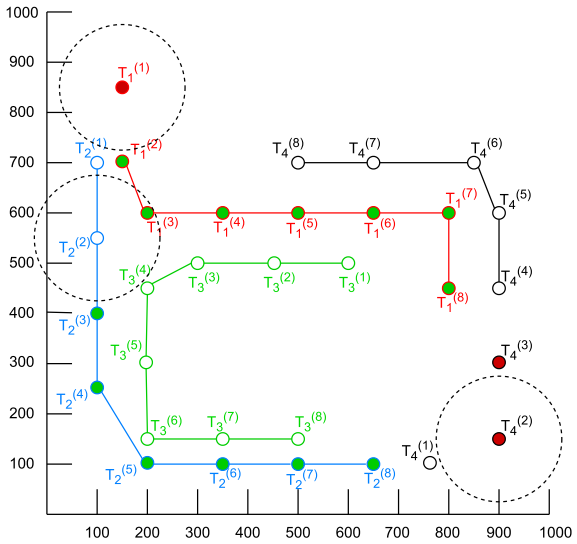


Figure 4: FPA approach with minimum support threshold $= 1$

Where $\texttt{long}_1$, $\texttt{lat}_1$, $\texttt{long}_2$, and $\texttt{lat}_2$ are the longitude and latitude coordinates of the points for which the distance is being calculated. $\Delta\texttt{long}$ is $\texttt{long}_1 - \texttt{long}_2$ and $\Delta\texttt{lat}$ is $\texttt{lat}_1 - \texttt{lat}_2$. And R is the radius of the earth, so for meters it is 6371000 and for miles it would be 3956. d is the result of the formula.

c. Trajectory reference matching

A key aspect of the trajectory reference matching process in REST is the construction of a reference set. This set holds a list of reference trajectories. It is a set of sub trajectories as defined in 3, which in turn can be used to describe new trajectories by matching them to whole or sub-trajectories stored in the reference set. The reference set should be kept small enough to be stored in memory, in order to avoid I/O operations, which in turn makes using the reference set quicker.

An example of the use of reference trajectories, being used to describe an input trajectory, can be seen in figure 3. Here two reference trajectories ($T_2^{(1,8)}$ and $T_1^{(1,8)}$) as well as an input trajectory ($T_3^{(1,8)}$) are defined. For the following example, a ϵ of 125 is used, as illustrated in the figure. Starting from $T_3^{(1,3)}$ which can be described as $T_1^{(6,4)}$. However upon reaching $T_3^{(4)}$, the nearest point from T_1 being $T_1^{(3)}$ is too far away. Now the algorithm looks to for another reference trajectory to describe $T_3^{(4)}$ with. This reference trajectory is found to be T_2, where $T_2^{(3)}$ can be used to describe $T_3^{(4)}$. From this point on T_2 is used to try and represent the coming points of T_3. Coincidentally T_3 can describe the remainder of the points meaning $T_3^{(4,8)}$ can be represented by $T_2^{(3,8)}$. This results in the equivalence $T_3^{(1,8)} = T_1^{(6,4)}, T_2^{(3,8)}$ with $\epsilon = 125$

d. Reference set construction

As mentioned in III.A, Zheng et al. explored some different methods for constructing reference sets for reference based compression[15]. The first of the explored methods are the Frequent Pattern-based Approach (FPA). This method is based on the finding that trajectories of moving objects, such as ships, cars and even animals often follow some repeated patterns. The method works by first finding a set of calculating points. A calculating point is a point in the trajectory set which can be used to represent other points in the set, within some error bound ϵ. After the calculating points have been found, they will be iterated through to find the ones that have frequencies above a threshold, in order to only get the most used calculating points. The calculating points can be used to make calculating trajectories by following the calculating points chronologically. If any of the sub trajectories of the calculating trajectories overlap, we can remove one of the sub trajectories, to get the minimum set of calculating trajectories. This will then be the final reference set. Figure 4 demonstrates the FPA approach. In the figure all of the points from the four trajectories are first added as calculating points. Then, as the minimum support is 1 all points which does not have 1 or more other calculating points are removed. The removed points are marked with a red dot in the center. The remaining points are then connected to form the calculating trajectories $T_1^{(2,8)}$, $T_2^{(1,8)}$, $T_3^{(1,8)}$, and $T_4^{(4,8)}$. Lastly, sub trajectories which have overlap with other sub trajectories are removed. This leaves the sub trajectories $T_1^{(2,8)}$ and $T_2^{(3,8)}$, marked with a green dot, as the final reference set.

The next method presented in REST is Segment Redundancy Reduction (SRR). Zheng et al. defines a sub trajectory, as a redundant segment if another sub trajec-

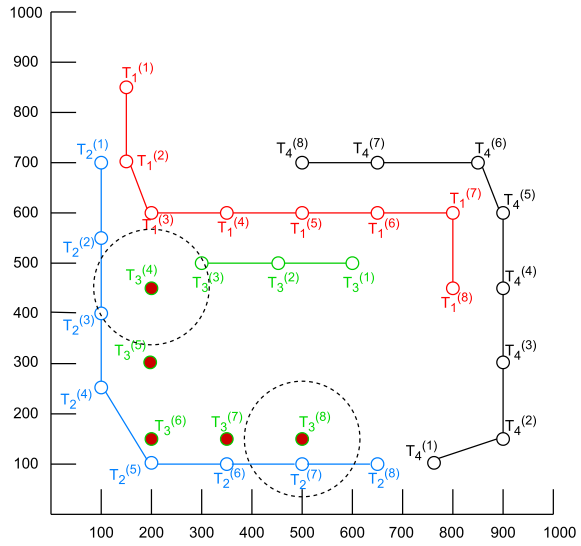


Figure 5: SRR approach with $\eta = 3$

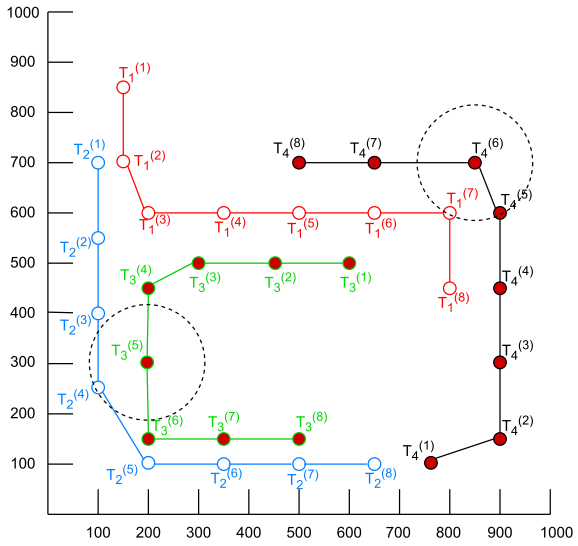


Figure 6: TRR approach with $\theta = 0.375$

tory exists, such that their lengths are equal and above a given minimum length η, and their individual points are pairwise within an error bound ϵ of each other[15]. SRR functions by taking a training set of trajectories and then removing the redundant sub trajectories, which results in a reference set with the minimum amount of redundancy. REST defines a sub trajectory $T_b^{(j,j+m)}$ as redundant if there is another sub trajectory $T_a^{(i,i+m)}$ where $m \geq \eta$ and their maximum pairwise distance $d_{max} = max_{0 \leq k \leq m} d(T_a^{(i+k)}, T_b^{(j+k)})$[15] where $d(p_1, p_2)$ is the distance between the two points given. If d_{max} is less than ϵ then all values are less than ϵ and the trajectory is redundant.

Figure 5 shows the SRR approach. The first step is to iterate over all of the sub trajectories with a length greater than $\eta = 3$, starting from the max length, which is 8 in this example. As the formula is given, the direction of the directories must be the same for them to be redundant. Thus the only redundant sub trajectory is $T_3^{(4,8)}$, which is removed. The rest of the trajectories will be the resulting reference set.

Another approach given by Zheng et al. is Trajectory Redundancy Reduction (TRR) which measures the overlap between a trajectory and the reference set. If the overlap is below a given threshold (θ) then the trajectory is added to the reference set. The overlap is measured by the following equation from[15]:

$$L(T, R) = \frac{|p \in T | \exists q \in R, d(p, q) \leq \epsilon|}{|T|} \quad (2)$$

Where T is a trajectory and R is the reference set we are checking its overlap against. $d(p, q) \leq \epsilon$ checks if the distance between the points p and q is greater than the error bound ϵ. The result of $L(T, R)$ is the amount of points in T which is within ϵ of at least one point in R, divided by the total amount of points in T. If $L(T, R) < \theta$ then T is added to R.

An example of TRR is shown in Figure 6. In this example using $\theta = 0.375$, T_1 is the first trajectory added to the reference set, thus there is no other trajectories for it to overlap with. When T_2 is inserted we need to check its overlap with the other trajectories, in this case T_1. The points $T_1^{(1)}$ and $T_2^{(2)}$ as well as $T_1^{(2)}$ and $T_2^{(3)}$ overlap. This is two out of the eight points from T_1 overlapping with existing trajectories, $2/8 = 0.25$, which is less than θ so the trajectory is added. All $8/8$ points from T_3 overlap with either T_1 or T_2. T_3 is thus discarded. And finally the sub trajectory $T_4^{(4,8)}$ overlaps with $T_1^{(5,8)}$. That is $5/8 = 0.625$ or 62.5% of the points which are already in the reference set, thus T_4 is removed as well. The final reference set consists of T_1 and T_2.

e. R-tree

An R-tree is a data structure proposed by Guttman[6] used to store multidimensional spatial data. It has a tree like structure, consisting of leaf and non-leaf nodes. The data in the tree is indexed based on its spatial location. Leaf nodes contain data records in the form $(I, Data - pointer)$ where $Data - pointer$ is a pointer to the spatial data, and I is the minimum bounding rectangle (MBR) needed to enclose the data. Non-leaf nodes contain entries in the shape $(I, child - pointer)$ where the child pointer points to a lower node in the tree and its I is the MBR needed to enclose all of its children's Is. An R-tree has a maximum and minimum amount of entries in each node. The max amount of nodes will be denoted as M, and the minimum as m. Furthermore the following must be true for M and m: $m \leq \frac{M}{2}$. The search algorithm for an R-tree takes a search area S in the form of a rectangle as an input.

With that input we can start iterating through the tree from the root node T. If T is not a leaf node, we iterate through all of its entries E and check if E.I overlaps with S, and if it does overlap we use the search algorithm with the child pointed to by E as the new T. This is done in a depth first fashion, meaning that the search function is applied to the first child found, until a leaf node is hit. Then all the E in the leaf node is searched before the search continues in the parent node. If T is a leaf node, then we check for overlap with all its entries, and return the ones where S overlaps with E.I. When a entry E is added to the R-tree its MBR E.I, is compared with all of the I values of the entries in the root node. If an existing I can cover E.I then the associated pointer is followed and the same is done for each child node until a leaf node is hit. If it cannot fit into any existing I then the child is chosen where we need to expand its I the least in order to fit E.I in it. When a node is full but a new data entry fits within its minimum bounding rectangle I, it is split in two.

IV. ARCHITECTURE AND IMPLEMENTATION

This section will describe the architecture and implementation of *STREAM*, as well as the changes made to it compared to MOBY. MOBY was split into two parts, a *compressor* for the edge and a *cloud* running on a central server[2]. The system was designed to run on devices where a lot of data was collected, where MOBY's compressor would then receive the data and compress it, before it was sent to the cloud. The cloud in MOBY was further split into a decoder and an ingester. In *STREAM* these two are fused together. MOBY uses the decoder to add tags which are used by the database to keep track of where data is from, and then send it in the arrow flight[4] format to the ingester. *STREAM* instead adds these tags on the edge. And as can be seen in Fig. 7, the ingester can directly receive MQTT data sent by the compressor. At the end of this section, MQTT as the transmission layer will be covered. Here, how data is sent, the form of it, and the general flow of communication will be explained.

A. Edge

In *STREAM* each edge node reflects a ship, which is connected to the system. The job of each edge node is to compress data collected at the edge, to the models as it is received. and then choose the one which resulted in the best compression. It then transmits the chosen models to the cloud, where they will be stored.

a. Isolating the timestamp column

In MOBY a lot of both memory usage and compressed data size was caused by the timestamps. These were stored for each different data column while compressing, and also added in compressed form to each model in the output file, which was sent to the cloud node. In *STREAM* the timestamps have been separated from the data. They are instead only stored once, with every model and data column having access to it. *STREAM* also tracks how many of the timestamps each model has consumed, such that when timestamps are not needed any more, they can be compressed separately and then transferred to the cloud. The models still have the start and end timestamp, but with this method the compressed timestamps are only stored once, instead of once for each model.

b. The reference model

For *STREAM* a new model type was designed, on top of the variations made in MOBY[2]. This model type is called the reference model. The reasoning for its development was that in MOBY, GPS time series were compressed poorer than other time series. One reason for this was stipulated to be that using a vector as compression leads to poorer compression when routes swerve, rather than go in a straight line. Another reason would be, that the vector was often defined in a way that was not inclusive for the rest of the points in a trajectory, as movement is only captured between the first two points, and then predicted for the remainder. This means that if the first two points are not indicative of the remainders, then the model has no flexibility to adjust. To alleviate this the reference model was designed. Its design is heavily based on the trajectory reference matching described in III.B.c.. The algorithm used for the model is shown in Algorithm 1.

The inputs for Algorithm 1 are a trajectory point p which we attempt to fit into the model, the R-Tree RTree for the reference set, a set of candidate reference trajectories C for the current model and the list of references R in the current model. C and R are pointers as the mutations from the algorithm needs to be saved. Line 1 checks if p will be the first point added to the reference in R, if it is then line 2 retrieves the candidates (C) to be references from the R-Tree by searching the R-Tree with p. This search returns all points within ϵ of p. The candidates in C is a 3 tuple of values (Ref.ID, start, end), where Ref.ID is the id of the reference candidate point belongs to, start is the starting index of the reference and end is the end point. If the list is not empty the algorithm returns true to signify that the point could fit in the model. If the list is empty false is returned instead. If the condition on line 1 was false, then line 4 checks if there is exactly 1 point in the last reference. If there is then line 5 gets a set of candidates C' for p. Lines 6-11 loops over the candidates in C, and for each of those candidates checks there is an candidate in C' with same reference id, that also has either same index in the reference, or an index above or below the one in the reference and update the direction attribute accordingly. If there was not one in C' that fit any of those,

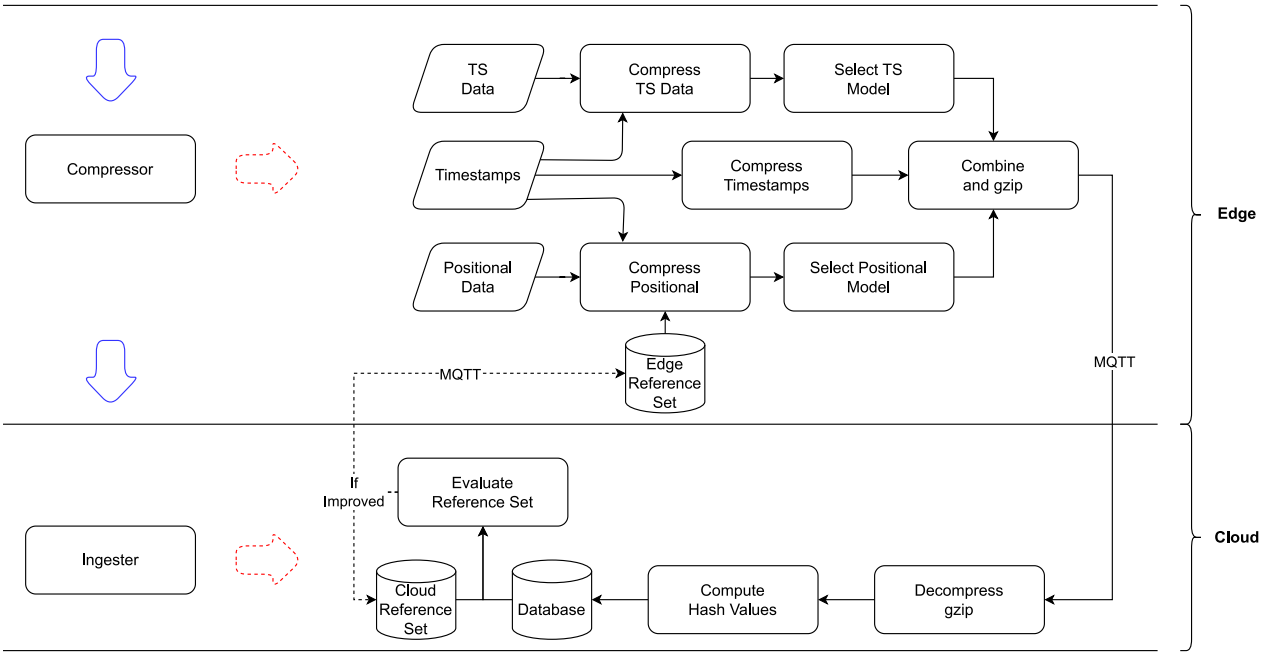


Figure 7: Architecture of *STREAM*

then c is removed from C on line 8. If C is empty after the loop, then no viable candidates are found and false is returned. Lines 13-27 is executed if there is more than two points in the currently last reference, and they simply get possibly viable next candidates, and compare them to the candidates in C, and removes them if they are no longer viable. Lines 20-26 checks if C is empty, if it is, the current reference cannot fit more. If C' is not empty then these can be used candidates for a new reference, and an empty one is created on line 24. If both C and C' are empty then the model is complete and cannot fit more, thus false is returned.

The reference set which is used in the model is generated by the TRR method as described in Section III.B.d.. When a trajectory is then compressed, it will either be compressed as a *moving reference*, or a *point reference*.

Moving reference: Moving references are used to reference trajectories of objects in movement, e.g. a ship sailing from one harbour to another or T_1 in Fig. 8. When points of a trajectory are streamed to the compressor, it looks to find the longest running reference trajectory that can represent the points of the trajectory. In the example shown in Fig. 8 $R_1^{(6,4)}$ can represent the sub trajectory $T_1^{(1,3)}$. When R_1 can no longer represent the next point of T_1, then the Ref.ID, start, and the end of the reference is saved to a list. In this case it would be $(R_1, 6, 4)$. Then the compression continues, as T1 tries to find a new reference trajectory to represent the points, which R_1 couldn't. $R_2^{(3)}$ is within ϵ of $T_1^{(4)}$, so R_2 is selected. This process repeats until The trajectory has been fully compressed into tuples of (Ref.ID, start, end). In this case the entire reference

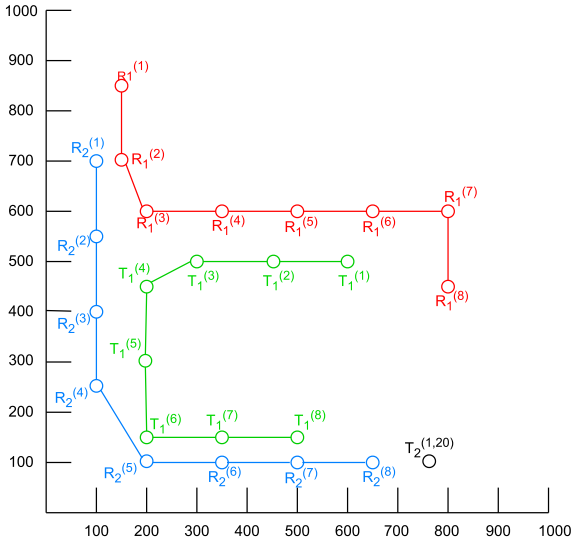


Figure 8

model needed to store T_1 would be $[(R_1, 6, 4), (R_2, 3, 7)]$. The bits required to describe this compression is as follows: reference Ref.ID = 16 bits, start and end Points = 8 bits each. This sums to 32 bits, or 4 bytes per reference sub trajectory used for the compression.

Point reference: Where a moving reference is used to map a moving object, point references serve to represent objects which are standing still, E.g. a ship which is currently anchored. Point references are used when a trajectory such as T_2 can continuously be represented by the same point of a reference trajectory. In order to represent

Algorithm 1 Fit value to reference model

Input: p, *RTree*, C, R
Output: *success*

if *R.last.count* == 0 **then**
  C = *RTree.search(p)*
  return $C \neq \emptyset$
else if *R.last.count* == 1 **then**
  C' = *RTree.search(P)*
  for $c \in C$ **do**
    if *SetCandidateDirection(C, C')* == **false then**
      $C = C \setminus c$
    end if
  end for
  return $C \neq \emptyset$
else
  C' = *RTree.search(P)*
  for $c \in C$ **do**
    if (c.RID, c.to + c.direction) $\notin$ C' **then**
      $C = C \setminus c$
    end if
  end for
end if
if C == $\emptyset$ **then**
  if C' == $\emptyset$ **then**
    return false
  end if
  R.appendNewEmpty
  $C = C'$
end if
return true

such a case, the only thing to be changed is that instead of saving the end index of the reference, a count is saved. This count represents the amount of succeeding points there is in a trajectory, which can be represented by the point denoted by the start index of the reference. $\mathtt{T}_2$ from Fig. 8 has all 20 of its points at the coordinates $(750, 100)$, this is within ϵ of $\mathtt{R}_2^{(8)}$. The model for this would then be $[(\mathtt{R}_2, 8, 20)]$. Seeing as this is the only change, the difference in number of bits required for the compression of a point reference is any additional bits needed for the count. In our case count is described with 16 bits, resulting in an additional byte in total.

The bits given, are delimited to fit a system where there can be up to 65536 reference trajectories, each of which can have up to 127 points, and each point reference can represent 32767 subsequent points. The reason for it being 127 points rather than 256, which is the maximum integer an unsigned byte can represent, is that the first bit of the byte representing the end index is used as a control to determine if the reference is a moving or point reference. An example of a bit representation of a reference model can be seen in 9. This example visualises a single trajectory reference, that of reference trajectory 2 from point 7

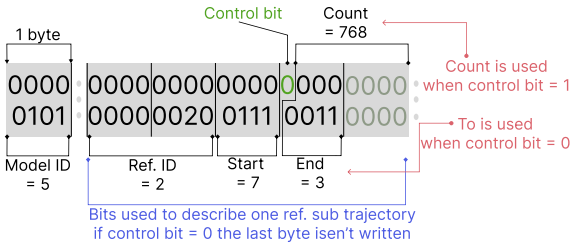


Figure 9: Bit representation of a sub trajectory reference using the Reference model

and to `end` point `3`. The first byte is the model type `ID`, which is supplied to the cloud, so that it can identify the model. The next two bytes are used for the `Ref.ID` of the reference trajectory in the cloud's reference set. After the `ID`, the index of the `start` point in the reference trajectory is supplied. Then the first bit of the next byte, is the control bit, seeing as it is 0, that means the current reference is a moving reference, and as such the next 7 bits must be read as the index of the `end` point. Had the control bit been set to 1 the next 15, rather than 7, bits would have been read. Had the model featured additional references, these would be found before the second byte I.e. before the `Ref.ID` bytes, or after the last byte (byte 5 is the last when the control bit 0 is, otherwise byte 6 is last). This is shown by the three grey dots on both sides of the trajectory reference. Any such additional references would not need to supply the model type ID, as this is only needed once per model.

B. Cloud

The focus of this paper is on the effect of references for compression compared to the existing vector based model. As these references can be made, and the reference model can be tested without completing the cloud part of the system, it has been left in a conceptual stage.

The Cloud in *STREAM* is responsible for both storage and management of the compressed data, as well as serving edge nodes with proposed reference trajectories.

a. Storage of models on the cloud

As seen in Fig. 7 the cloud will receive data from the edge nodes, which it in turn will ingest into its database. For *STREAM* this database is a columnar database, which uses a ship's MMSI and the column of the incoming data to compute a hash, which is used as a uni-variate ID. The MMSI and the column index are supplied by the transmitting edge node. Where the column index is derived from the shape of the data, when the edge node receives sampled data from its sensors.

b. Proposal of reference set

In order to save space on the edge nodes, which often have limited memory and computational power[2], the construction of reference sets for all edge nodes is done on the cloud. This raises two issues: 1) if only the cloud can make references, how can the compressor benefit from reference-based compression? and 2) There is potentially an enormous amount of edge nodes, how would the cloud be able to serve them all?

To answer problem 1. the compressor can benefit of reference-based compression via communication with the cloud. This communication runs through MQTT, where a broker would publish a message, which the compressor would subscribe to. The contents of this message would include the `Ref.ID`, and the entire collection of its points.

To answer problem 2. it is important to understand that the key aspect of recurrent routes, is that they are not necessarily unique to one ship, this was already alluded to in cite [13]. As such a base assumption of this paper is that the reference sets needed to supply each of the edge nodes individually would have overlaps with each other. The expectation here being that by having one reference set on the cloud for all edge nodes, it would be possible to have multiple ships use the same references. The universality of the reference set could be enhanced by further modification of the reference set. An example of such a modification could be to remove outliers from established routes. i.e a ship follows an established reference trajectory, but at some point for a small period deviates before returning to the same reference trajectory. Such modifications are not included in this iteration of *STREAM*, but candidates will be discussed in future workVII.

The reasoning of having the construction of the reference set on the cloud node, is that edge nodes are expected to have low computational power and available memory. The construction of the reference set is expensive, and as mentioned earlier it is expected that the reference sets needed for the compression of individual edge nodes data will have overlaps, and as such any computation of these overlaps should rather be done only once.

c. Decompression of reference models on cloud

When data is ingested onto the cloud component of *STREAM* it should be query-able. In order for this to be possible, said data must have a method of decompression. This section will cover the decompression process for data compressed via a reference model. When a reference model is transmitted from an edge node to the cloud, it is transmitted as a model type ID and a list of tuples containing the `Ref.ID`, `start` point and `end` point for each of the sub trajectories used to compress the data. The `Ref.ID` points to a reference trajectory, which the cloud at some point facilitated the compressor of an edge with. As such, the decompression is done by querying the cloud's reference set for the specific `Ref.ID`. This query will return the trajectory which is being referenced, and the data points can be remade through the corresponding sub trajectory, delimited by the `start` and `end` points. This process is then repeated for every tuple in the model. A case where this decompression is shown is 8 where the reference model used to decompress $\mathtt{T}_1^{(1,8)}$ would look like $5, (\mathtt{R}_1, 6, 4), (\mathtt{R}_2, 3, 7)$, with each separate tuple being enclosed in parentheses, for ease of understanding. Also note here that 5 is the model type ID for reference models. The decompressed $\mathtt{T}_\mathtt{d}$ would then have the points $\mathtt{T}_\mathtt{d}^{(1,3)} = \mathtt{R}_1^{(6,4)}$ and $\mathtt{T}_\mathtt{d}^{(4,8)} = \mathtt{R}_2^{(3,7)}$. So $\mathtt{T}_\mathtt{d}$ is not exactly $\mathtt{T}_1$. but it holds that $\mathtt{T}_\mathtt{d}^{(\mathtt{i})} - \mathtt{T}_1^{(\mathtt{i})} \leq \epsilon$ for every index $\mathtt{i}$ in $\mathtt{T}_1$.

Should the reference model include an instance of a point reference it would have the same form, however now the endpoint in either tuple would instead signify the number of points to repeat on a specific point in the reference trajectory. A quick example of this would be $\mathtt{T}_2$ in Fig. 8 which compressed would be $5, 1(\mathtt{R}_2, 8, 20)$. Decompressing this simply gives us index 8 of the reference trajectory $\mathtt{R}_2$ 20 times as $\mathtt{T}_\mathtt{d}$, where again every point is not exactly $\mathtt{T}_2$ but still within ϵ for it. Thus there is accuracy guarantee within the ϵ for the model.

C. MQTT as transmission layer

The architecture of *STREAM* features MQTT as its communication between the edge and cloud layers[12]. MQTT is a lightweight publish and subscribe messaging protocol. MQTT is centred around internet of things devices, and as such is made to scale with large number of agents. It features Bi-directional communication, which is needed as the edge will transmit its data to the cloud, and the cloud will effectively transmit a reference set for the compressor component of an edge to use.

a. The flow of communication

The flow of communication between edge and cloud is in a push-based manner from both ends. The edge will push its compressed data to the cloud, and the cloud will push proposal reference trajectories to the edge, for its compressor component to use. The reasoning for not having proposal reference trajectories on a pull basis, is that only the cloud can determine, when a reference trajectory is done, and viable for an edge node. This is due to the fact, that the edge does not hold on to its data after transmission, while the cloud would have access to all the data the edge ever sent.

b. The form of transmission data

From edge to cloud:

When compressed data is sent from the compressor on the edge to the cloud, it is sent as comma separated values with the header of form: *Model type ID, Column index, End index, Min value, Max value, Values, First timestamp, last timestamp*. These values on transmissions are further compressed using gzip, as can be seen on Fig. 7.

From cloud to edge:

When the cloud wants to send a reference trajectory for an edge node to use, it transmits a message of the form *Reference ID, and a list of GPS locations*. The size of this message varies on how many points a reference trajectory can have. For *STREAM* this length is at a most 127, due to the size limitations set in the end of IV.A.b.. Each GPS point is recorded with float precision for both latitude and longitude. Which translates to 4 bytes, or 32 bits, each. Like the edge, the content of the transmissions made by the cloud are further compressed using gzip, as can be seen on Fig. 7.

V. EVALUATION

This section will cover the evaluation of *STREAM*, this includes an introduction of the metrics used for experimentation, a list of experiments as well as a discussion of the findings.

A. introduction for experiments

For the experiments, there are some fundamentals which needs to be defined. This includes metrics and their units, the data sets, and the focus of the experiments.

a. Metrics

The *Compression Ratio* (*CR*) is the difference in size between an uncompressed and a compressed file.

$$\texttt{CR} = \frac{\texttt{Original size}}{\texttt{Compressed size}} \tag{3}$$

Like MOBY[2] the compression in *STREAM* at transmission is further enhanced by using gzip[5], on the already model compressed files. Then *CR* is measured again for the compressed file(*CR.gz*)

Runtime is the amount of time it takes the system to compress the entire data set measured in seconds.

Model Usage (*ModU(x)*) is the percentage of compressed data that uses a given model x. (*ModU.p(x)*) is the percentage of data points, which are covered by a given model x. For both *ModU(x)* and *ModU.p(x)* shorthand for the models will be used, so the reference model = "ref" and vector-based model = "vec".

b. Experimental environment

The experiments was performed on a Linux virtual machine with a Intel(R) Core(TM) i5-8600K CPU @ 3.60GHz processor, and 11.2GB RAM.

c. focus of experiments and opposition

The general focus of *STREAM* is to improve upon the shortcomings of GPS compression found in MOBYIII.A. As such, the main goal of experiments is to uncover whether it stands as an improvement. Due to this, MOBY will be used for baseline while a list of variations will be used as opposition. These variations include: *STREAM* with reference model only and with vector-based model only. As an additional baseline, gzip will be used. Gzip is a compression tool, which uses frequent sub-sequences in data to compress said data[5]. Due to the niche nature of, *STREAM* finding other oppositions is difficult. An example of such is REST[15], which was introduced in Section III.A. REST uses an already established network, in their case the road network, however the AIS data used in *STREAM* does not have such a network, not all routes are identifiable. Another example is ModelarDB [7] also introduced inSection III.A. ModelarDB has no model to describe moving objects, as the vector-based model was introduced later, in MOBY[2], which was inspired by ModelarDB.

d. Data sets

The data sets used for the experiments come in two forms: 1) different subsets of American coastal AIS data from the years 2021 and 2022[11]. 2) Danish AIS data from the years 2018 and 2019. For the American coastal AIS data, a score system was implemented. This score was found for each ship by:

$$\texttt{MG score} = \frac{\texttt{sum(delta of delta)}}{\texttt{number of rows}} \tag{4}$$

Delta here referring to the difference between two timestamps, and delta of delta in turn referring to the difference between two consecutive deltas. A lower score in this case mean that the time series of a ship, is closer to being regularDefinition 4, than one with a higher score. This score was implemented to find ships, which had more regular transmissions than others, since this was found to be an important factor in MOBY [2]. The MG score was used to find the 20 ships, which had the highest position in a list sorted by MG score(MGHigh), as well as the 20 ships with positions 80-100(MGLow). For the danish AIS data two sets are included, the first being the ship "Friheden" and the second being the ship "Copenhagen". These ships were selected due to the fact that they were included in the testing of MOBY, where Friheden was the ship which MOBY performed the worst on, and Copenhagen the one,

where performance was best, while also containing positional data. As such, the idea of testing *STREAM* on these is to see how *STREAM* performs in contrast to the former best and worst cases of MOBY.

For testing, the data was split into training and testing data. For *STREAM* this means that training data is used to construct the reference set, and testing data is compressed using the set. For MOBY only the latest year is used, as it has no training in its process. For all data sets, the rows which had duplicate timestamps, rows with latitude $= 91$ and longitude $= 181$, as well as all columns not listed in Table 1 were removed. This results in all data sets having the same format, which can be seen in Table 1. Here MMSI covers the integer identifier of a ship, Timestamp is the recorded timestamp of a data point, Latitude and Longitude is the captured GPS point. The sizes and number of data rows in the data sets are shown in Table 2.

MMSI	**Timestamp**	**Latitude**	**Longitude**
56781234	1678522800	48.856613	2.352222

Table 1: The format for all datasets used

Dataset	Rows	Size
MGHigh	4355357	178.102 MB
MGHighTrain	4140257	169.313 MB
MGLow	7298989	298.140 MB
MGLowTrain	7780736	317.529 MB
Friheden	14791	0.615 MB
FrihedenTrain	21838	0.907 MB
Copenhagen	4274643	175.804 MB
CopenhagenTrain	4030270	165.660 MB

Table 2: The data sets, with their row count and sizes

e. Testing parameters

The parameters used to evaluate *STREAM* are shown in Section V.A.e.. The default values are the values which are underlined in the table.

Parameter	Values
Reference trajectory length	5, 10, 20, 30, 60, <u>90</u>, 100, 110, 120
Max overlap (θ)	10, 20, 30, 40, 50, 60, <u>70</u>, 80, 90, 100
Error bound ϵ	1, 10, 20, 30, 40, 50, 75, <u>100</u>, 125, 150, 175, 200, 250, 300, 400

Table 3: Parameters for evaluation of *STREAM*

B. Experiments and findings

A series of experiments have been made in order to figure out the best parameters for *STREAM*. The tests were conducted on the data sets MGHigh and MGLow with MGHighTrain and MGLowTrain used to generate the reference sets. The first experiment was to vary the length of reference trajectories. Figure 10 show the *CR* and *CR.gz* of *STREAM* for this experiment. The *CR* increases along with the length of trajectories, but the increase diminishes as the reference trajectory length gets near 100. This is the case for both data sets these tests were performed on. Figure 11 and Figure 12 shows that the *Runtime* and *ModU(ref)* of the reference model also increased with the reference trajectory length. The correlation between reference trajectory length increasing and the *ModU(ref)* of the reference model is easily explained, as when reference trajectory length is increased the same reference can be followed longer, which uses fewer bits when compressing. This in turn also explains the better compression. The *Runtime* shown in Figure 11 could in part be explained by how the reference trajectory length and the θ values are connected. If the reference trajectory length for example was 3 with $\theta = 0.30$, then if a single point and there are two vectors with a single point overlapping between them, then the second vector is discarded and only 3 points are added to the reference set. If on the other hand the reference length is 4 with the same θ and the two vectors still only overlap with one point, then both vectors are added to the reference set, as $1/4 = 0.25$, which is less than θ. This results in more points being returned in the R-Tree when querying, thus making the system slower.

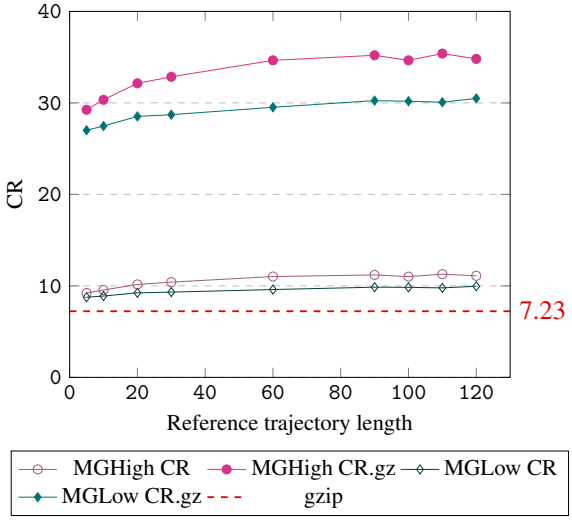


Figure 10: The impact of reference trajectory length on CR and CR.gz

Figure 13, Figure 14, and Figure 15 shows the results from experimenting with different values for θ. Increas-

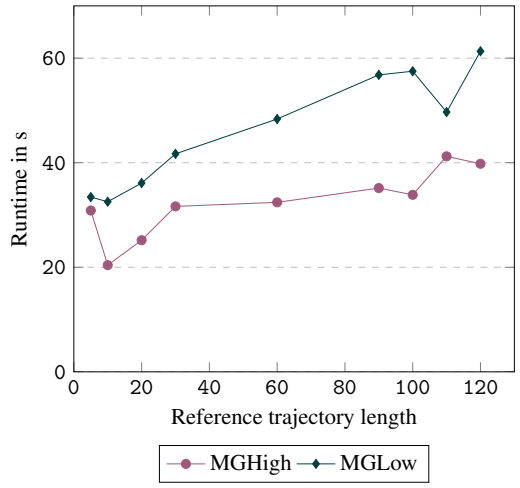


Figure 11: The impact of reference trajectory length on Runtime

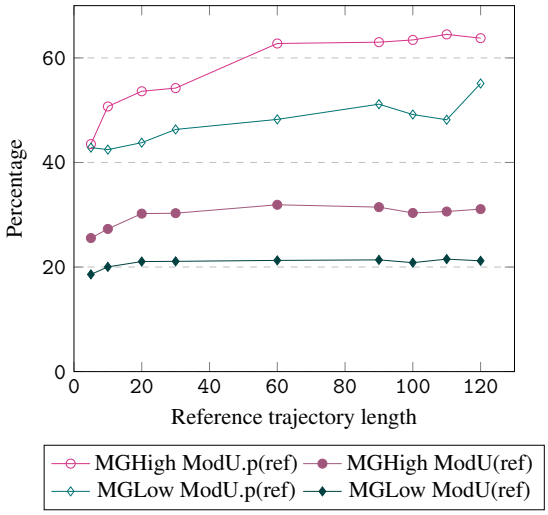


Figure 12: the impact of reference trajectory length on ModU(ref) and ModU.(ref)

ing θ increased the *CR* as allowing more overlap results in more references to use for compression, but it also increases the amount of references that needs to be stored. The goal of this experiment is to find a value that gives a good *CR* while keeping the number of reference trajectories low. It can be seen in Figure 13 that as θ is increased, the *CR* increases as well as there are more reference trajectories to fit the data to. Figure 15 shows that the *ModU(ref)* increases in a mostly linear fashion, yet the *CR* increased exponentially. One of the reasons that the *CR* increases faster than the *ModU.p(ref)* is that when the new reference models replace multiple vector-based models. Figure 14 shows that there is a huge spike in *Runtime* when $\theta = 100$, as at this point every trajectory is added to the reference set, even though it is already covered. This means that whenever it is checked if there is a point close to one that is fitted to a reference model, then it returns all of the points within the window, whereof some could be duplicates. Iterating over all of these points to see if one of them is a neighbour, is very slow.

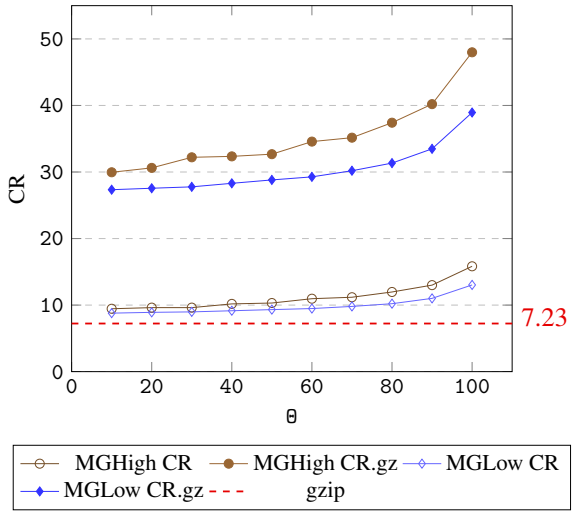


Figure 13: the impact of θ on CR and Cr.gz

Figure 16 shows what effect changing the ϵ would have on *CR*. The figure shows that the *CR* increased drastically between $\epsilon = 1$ and $\epsilon = 150$. It can be observed that there are diminishing returns on the *CR* from increasing ϵ. Figure 17 show that the *Runtime* decreases by magnitudes as the ϵ increases, though also with diminishing returns. As the ϵ increases the amount of reference trajectories in the r-tree decreases as it affects when trajectories are counted as overlapping in Eq. (2). The effect on *ModU(ref)* and *ModU.p(ref)* is shown in Fig. 18. We can see that the *ModU(ref)* increases on both datasets, in the beginning. But after $\epsilon = 250$ the *ModU(ref)* for MGLow starts to decrease again. With the reference model the ϵ only prolongs how long the data can follow a reference with a few

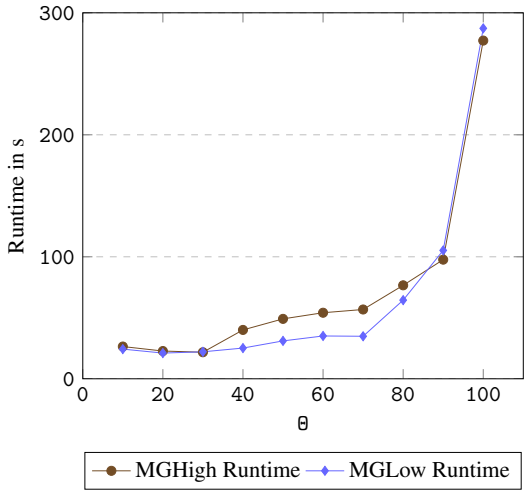


Figure 14: the impact of θ on Runtime

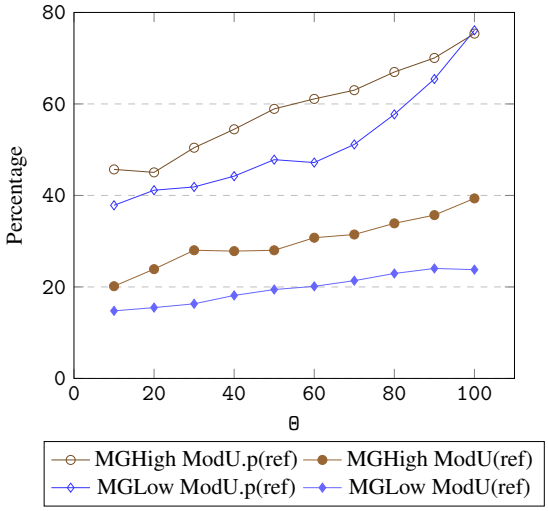


Figure 15: The impact of θ on ModU(ref) and ModU.(ref)

points. If it is sailing with another speed than the reference is recorded with, the deviation from the recorded point to the reference becomes greater for each point. But for the vector-based model, as long as the speed it is sailing with is about the same as between the first two points, then the model will continue to be expanded. This makes the higher ϵ more beneficial to the vector based model, whereas it becomes less and less beneficial for the reference model.

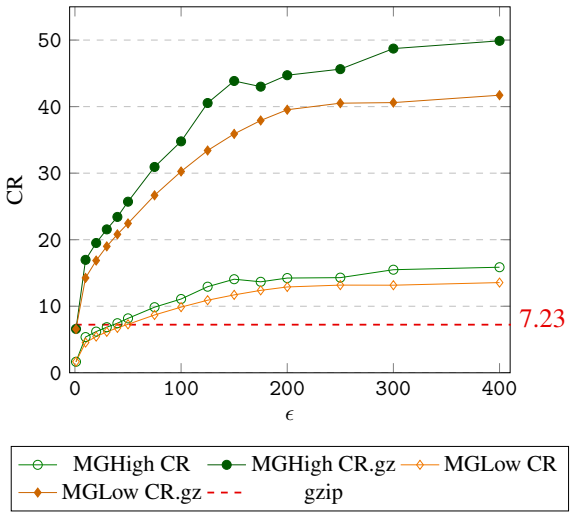


Figure 16: The impact of ϵ on CR and CR.gz

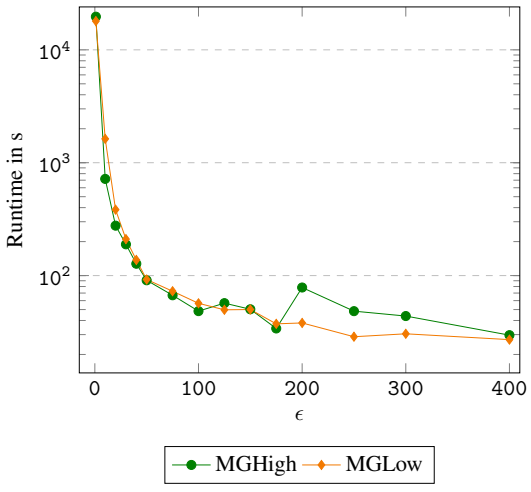


Figure 17: The impact of ϵ on Runtime

Experiments were made on different sets of data, in order to test *STREAM* against the baselines, with the values: reference length $= 90$, $\theta = 90$ and $\epsilon = 100$. The result of these experiments can be seen in Fig. 19.

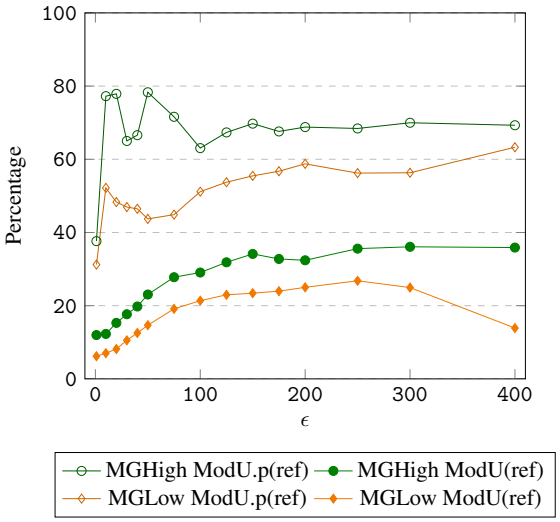


Figure 18: The impact of ϵ on ModU(ref) and ModU.(ref)

Note that "Ref" refers to the variant only using reference models, and "Vec" refers to the variant only using vector-based models. For the MGhigh data set Figure 19a it can be seen that MOBY has the poorest *CR* at 6.12 and *CR.gz* at 17.02. *STREAM* and the variant only using reference models have the highest compression. However, which among them has the highest *CR* depends on whether the data has been further compressed with gzip. The best *CR.gz* was *STREAM* with a *CR* of 46.7. When looking at the result for the MGLow data set shown in Fig. 19b,*STREAM* is ahead with a *CR.gz* of 40.12, compared to the nearest competitor, being the variant using only vector-based models, with a *CR.gz* of 26.65. Even without using gzip to further compress, *STREAM* with a *CR* of 13.23 doubles the *CR* of MOBY which was 6.17. Another interesting observation is the *Runtime* for the reference model variant which, relative to *STREAM* and the vector-based variant, skyrockets to 226.7 seconds. This is unusual as for all the other datasets this variant has seen a *Runtime* below *STREAM*, but here it is instead 3 times slower than *STREAM*'s *Runtime* of 47.9 seconds When testing on the Copenhagen data in Fig. 19c, the reference-only variant was worse than than MOBY. The reason for this, is likely that the training data for Copenhagen had a low overlap with the test data. The best method on this data set seems to be the Vector only model, with a *CR.gz* of 50.7, though *STREAM* is right below it with a *CR.gz* of 50.62. But looking at the results without gzip *STREAM* has a *CR*, that is slightly higher than the vector only variant. Figure 19d shows the results of the four methods used on the Friheden data set. The best method on this data was the reference-only variant, which had a *CR.gz* of 8.62. *STREAM* is again just below this method though, with a *CR.gz* of 8.41. And again *STREAM* has a slightly better *CR* than the method with the best *CR*.

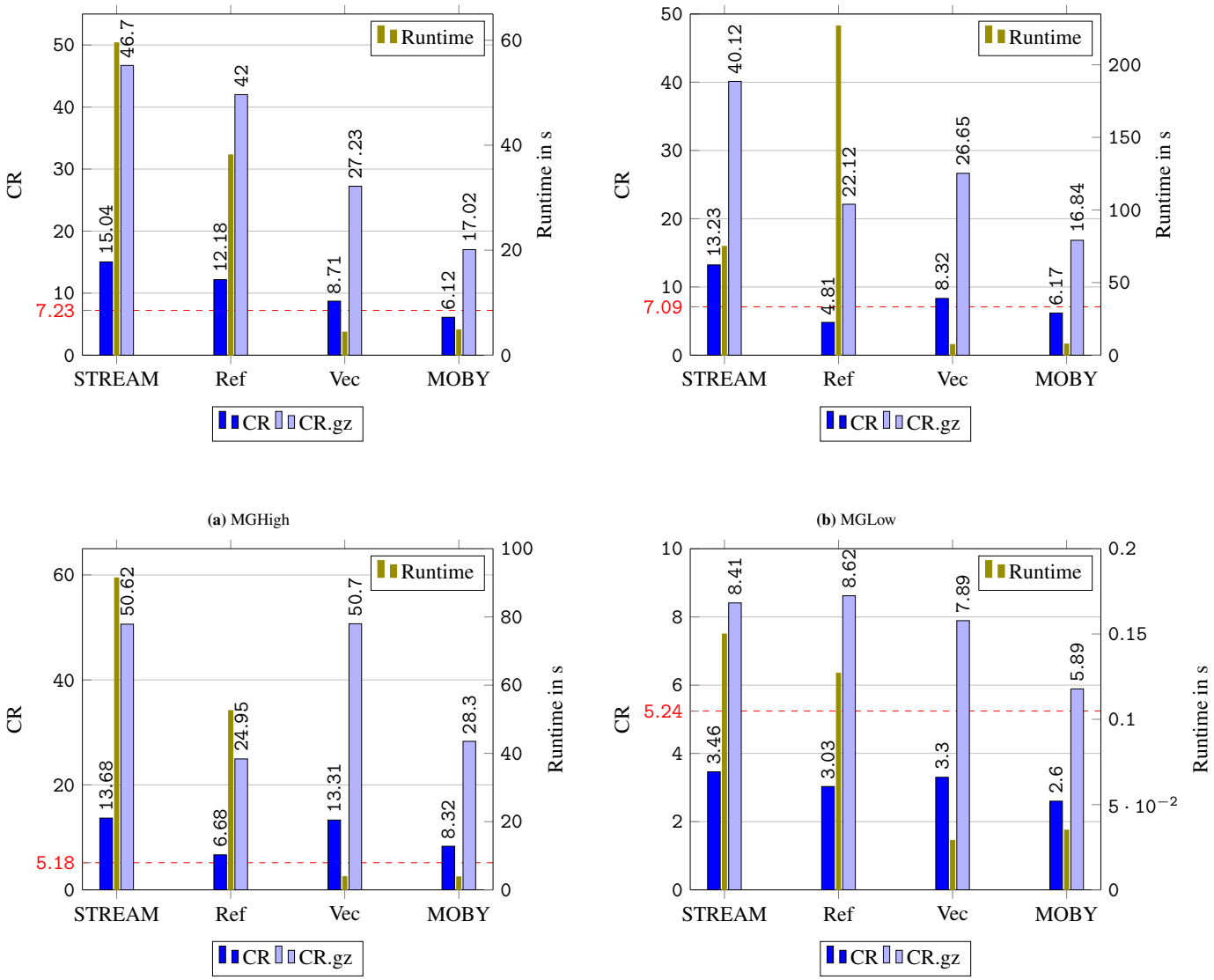
C. Discussion

In Fig. 19 a range of interesting observation were made. Firstly, the *CR* of *STREAM* was better than the *CR* of the other methods. Secondly, it is evident that among the variants the one using only vector-based models is faster, especially on the MGHigh data set. It can also be observed that while *STREAM* generally has among the best *CR* and *CR.gz*, it is also the slowest, unless it is run on MGLow where it was the second slowest. The *Runtime* could easily be improved at the expense of some *CR* by lowering the θ value as shown in Fig. 14

The higher *Runtime* of *STREAM* is to be expected, as the reference model needs to evaluate all found candidates. However, when put into the perspective that these are *Runtimes* for a year worth of data the values are quite acceptable. A more explicit example of this is *STREAM* on MGHigh where it has a *Runtime* of 59.5 seconds for 20 ships worth of data. This means that it would take 530.368, assuming the same average size as MGHigh, ships to fill out a year's worth of *Runtime*. This along with the fact, that in the complete American coastal AIS data[11], which MGHigh and MGLow were extracted from, there were 71334 ships in total, paints a clear picture of a *Runtime* within reasonable boundaries.

When it comes to the aspect of compressing data, it is clear that the reference model is a viable contender to the vector-based model. This is seen in the *ModU(ref)* and *ModU.p(ref)* shown in Fig. 18, Fig. 15 and Fig. 12. Where for each of the parameter values selected to test *STREAM* against MOBY, the *ModU.p(ref)* is above 50%. While this is the case, at no point is *ModU(ref)* above 50%, the highest it ever gets is around 40%. This naturally means that on average, the reference model, fits more points than the vector-based model. For Fig. 15 and Fig. 12 it was observed that in general *ModU(ref)* and *ModU.p(ref)* rose with θ and the trajectory length of reference trajectories. This is almost trivial, as these are parameters, which only affect the reference model. On Fig. 18 however, a higher ϵ also affects the vector-based model, and small deviations from the general tendency of *ModU.p(ref)* can be seen, as the two models gain advantages over each other, until $\epsilon = 100$ and the values become more stable.

An interesting feature of *STREAM* shown by the variant which only uses reference models, is the synergy with gzip and reference models. This synergy is seen in Fig. 19, where the relative growths from *CR* to *CR.gz* for this variant is $3,442$, $4,592,84$ and $3,73$. this compared to the variant, which only uses the vector-based model, relative growths of $3,12$, $3,20$, $2,39$ and $3,80$, shows a trend, only broken by the last value, of the variant using reference models proportionally being compressed more by



(a) MGHigh

(b) MGLow

(c) Copenhagen

(d) Friheden

Figure 19: Experiments showing performance of *STREAM*, MOBY, and the two variants Ref and Vec, on MGHigh, MGLow, Copenhagen and Friheden. *CR* and *CR.gz* on the left y-axis, Runtime on the right. The CR achieved by gzip alone, is marked by a red dashed line

gzip. This is most likely due to the usage of id's for trajectories and points in the reference model. The whole idea of the reference set is to capture frequent patterns, so logically it would have better synergy with gzip, as gzip utilizes frequent subsequences of symbols[5].

STREAM set out to built upon the foundation of MOBY[2]. Through the testing done in this section, it is clear, *STREAM* outperforms MOBY in terms of compression of AIS GPS data, even on the AIS data which MOBY had performed best on itself in its own testing. While some of this *CR* comes from the performance of the reference model, it is also simply due to the ability to choose. This can be seen, especially in Fig. 19a and Fig. 19b where first the reference-only variant performs better than the vector-only variant but then the performance switches and the vector-only variant performs better. In both these cases *STREAM* outperforms both its variants, proving the advantage of being able to choose between the two during compression. The superiority in *CR* is even more apparent, as between MOBY and *STREAM*, MOBY had poorer *CR* than gzip, on all except the Copenhagen dataset, while *STREAM* outperformed gzip on all except Friheden. While MOBY had superior *Runtime* to *STREAM*, it was also underlined, that the *Runtime* of *STREAM* was low enough to facilitate a multitude of ships. This superiority is even more apparent once it is understood, that if one wants lower *Runtime*, *STREAM* already facilitates this through its variant, which only uses the vector-based model. This variant in all tests had comparable and even lower *Runtime*, while maintaining higher *CR* for all tests.

VI. CONCLUSION

This paper has detailed *STREAM*, a system focusing on facilitating model based compression for GPS time series in an edge to cloud environment. It is a successor of MOBY[2], where the focus was also on model based compression, however the compression on location data fared poorly. As such *STREAM* first and foremost looks to improve on this. *STREAM* takes inspirations from the REST framework[15], a compression framework specialising in compressing GPS data of moving cars, by viewing them as sub-trajectories over a network of trajectories. *STREAM* uses historic data to establish reference trajectories, which in turn will be used to compress incoming GPS data. The paper presents both an edge and a cloud component. The cloud is responsible for managing and distributing reference trajectories to its edge nodes, as well as storing and managing the compressed data. whereas the edge nodes are responsible for collecting time series data, compress said data into models, favouring the models which achieve higher compression. Both components make use of a reference set, which is a R-tree data structure, mapping reference trajectories to 2d space. The tests put forth examine the performance of *STREAM* focused on compression ratio and runtime. The tests showed that over 4 data sets, *STREAM* always outperformed MOBY in terms of compression, and in two cases does so with a factor of 2.

VII. FUTURE WORK

A. Enhanced vector-based model

One of the downsides of the vector-based model is that it offers no flexibility in terms of the construction. In Jensen et al.[7] as explaining inSection I, a number of models were designed for ModelarDB, one of these models was called Swing. Swing as a model uses planes to represent moving upper and lower bounds, and any point between those two planes can be expressed by the model. For each new point in the model the upper and lower bounds converge based on where the new point is, if it is closest to the lower bound, the upper bound converge faster towards the lower bound and vice versa. This repeats until an incoming point isn't in the space between the two bounds, as the space becomes more narrow with each convergence. If the vector-based model was enhanced to use this method, it still wouldn't be able to swerve from side to side, but the prediction vector wouldn't only be constructed from the first two points.

B. Proposal reference sets

In section, Section IV.B.b. it was mentioned that the reference set constructed by the cloud component could be modified, in order to increase performance. It would be interesting to explore such modification in future work. An example of such a modification could be distributed exchange of the reference set. Rather than edge nodes transmitting only to the cloud, they could also communicate with each other. This of course would put more load on the edge nodes, so any such communication should be used sparingly. An example of a scenario where such a transmission could be interesting, is when two ship are in close proximity to each other, far out into an ocean. Communication here between any edge and cloud node, is more expensive, as it often runs through satellites[2]. It would then be more monetarily effective for these ships to facilitate these two edge nodes, to communicate. In summary a modification to the way the cloud manages and distributes proposal reference sets, could be halted for edge nodes which are too far away. Instead the cloud node could put another ship it deems to intercept the ship which is too far away, upon reaching close proximity the two edge nodes can fulfil the cloud's intention for it.

C. Tests on other data domains

STREAM was developed with AIS data in mind, REST[15], the inspiration behind the reference model, was developed for and tested on vehicular road networks. These two domains, location-wise, have a clear distinction in that one(road networks) takes place in a defined network, and this network can be used to form the reference trajectories. The other(AIS GPS data) while it has defined routes and right of way laws, aren't restricted to said routes. Despite this dichotomy, traffic in both domains have successfully been compressed using some form of reference trajectory matching. It would be interesting not only to further develop on this method, but also gauge its applicability to other domains. Examples for this could be something like weather balloons but also things less governed by humans, such as animal packs. It would also be interesting to explore data which isn't necessarily movement data, but other readings such as temperature, predicted states of a device etc.

VIII. ACKNOWLEDGEMENT

L.V. thank Torben Bach Pedersen for his supervision and valuable inputs throughout the project.

References

[1] Abhilash. *How to calculate distance using the haversine formula*. URL: https://www.educative.io/answers/how-to-calculate-distance-using-the-haversine-formula. (accessed: 05.06.2023).

[2] Frederik Agneborn et al. *MOBY: MOdel-Based compression sYstem*. AAlborg University, 2023.

[3] R. Agrawal and R. Srikant. "Mining sequential patterns". In: *Proceedings of the Eleventh International Conference on Data Engineering*. 1995, pp. 3–14. DOI: 10.1109/ICDE.1995.380415.

[4] *Arrow Flight RPC*. URL: https://arrow.apache.org/docs/format/Flight.html. (accessed: 13.06.2023).

[5] *GNU Gzip*. URL: https://www.gnu.org/software/gzip/manual/gzip.html. (accessed: 09.06.2023).

[6] Antonin Guttman. "R-Trees: A Dynamic Index Structure for Spatial Searching". In: *SIGMOD Rec.* 14.2 (June 1984), pp. 47–57. ISSN: 0163-5808. DOI: 10.1145/971697.602266. URL: https://doi.org/10.1145/971697.602266.

[7] Søren Kejser Jensen, Torben Bach Pedersen, and Christian Thomsen. "ModelarDB: Modular Model-Based Time Series Management with Spark and Cassandra". In: *Proc. VLDB Endow.* 11.11 (July 2018), pp. 1688–1701. ISSN: 2150-8097. DOI: 10.14778/3236187.3236215. URL: https://doi.org/10.14778/3236187.3236215.

[8] Søren Kejser Jensen, Torben Bach Pedersen, and Christian Thomsen. "Scalable Model-Based Management of Correlated Dimensional Time Series in ModelarDB+". In: *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. 2021, pp. 1380–1391. DOI: 10.1109/ICDE51399.2021.00123.

[9] Søren Kejser Jensen, Torben Bach Pedersen, and Christian Thomsen. "Time Series Management Systems: A Survey". In: *IEEE Transactions on Knowledge and Data Engineering* 29.11 (2017), pp. 2581–2600. DOI: 10.1109/TKDE.2017.2740932.

[10] Yannis Katsis, Yoav Freund, and Yannis Papakonstantinou. "Combining Databases and Signal Processing in Plato". In: *Conference on Innovative Data Systems Research*. 2015.

[11] MarineCadastre.gov. *Vessel Traffic Data*. 2022. URL: https://marinecadastre.gov/ais/. (accessed: 31.05.2023, the data is found under the points menu, where data can be downloaded per year).

[12] *MQTT*. URL: https://mqtt.org/. (accessed: 13.06.2023).

[13] International Maritime Organization. *Ships' routeing*. URL: https://www.imo.org/en/OurWork/Safety/Pages/ShipsRouteing.aspx. (accessed: 19.05.2023).

[14] Sayan Ranu et al. "Indexing and matching trajectories under inconsistent sampling rates". In: *2015 IEEE 31st International Conference on Data Engineering*. 2015, pp. 999–1010. DOI: 10.1109/ICDE.2015.7113351.

[15] Kai Zheng et al. "Reference-Based Framework for Spatio-Temporal Trajectory Compression and Query Processing". In: *IEEE Transactions on Knowledge and Data Engineering* 32.11 (2020), pp. 2227–2240. DOI: 10.1109/TKDE.2019.2914449.