Implementing Rigid Temporal Geometries in Moving Object Databases

Maxime Schoemans[1], Mahmoud Sakr[12], Esteban Zimányi[1]
[1]Ecole Polytechnique de Bruxelles, Université Libre de Bruxelles, Belgium - [2]Ain Shams University, Egypt
{maxime.schoemans, mahmoud.sakr, ezimanyi}@ulb.ac.be

***Abstract*—Various applications process geospatial trajectories of moving objects, such as cars, ships and robots. There is thus a need for a common conceptual framework to model and manage these objects, as well as to enable data interoperability across tools. The International Organization for Standardization ISO® has responded to this need and created the standard ISO 19141–Schema for moving features. Among its types, it defines a schema for rigid temporal geometries, which represent the movement of spatial objects translating and rotating over time, while preserving a fixed shape. Despite the abundance of these objects in real-world, there exists no reference implementation of this type of data in a common system, which causes them to usually be represented as temporal points without taking into account their spatial extents and shapes. In this paper, we aim to provide an implementation of rigid temporal geometries into MobilityDB, an open-source moving object database, that extends PostgreSQL and PostGIS. We provide a data model for rigid temporal geometries and propose efficient algorithms for the operations defined in ISO 19141. A use case on real AIS ship trajectories is illustrated to validate the proposed implementation. A synthetic data generator for temporal geometries is also proposed. Finally, we review the standard from an implementation point of view and provide insights on possible improvements.**

I. Introduction

Large amounts of location data of mobile objects are produced by positioning systems such as the GPS. The processing of this big data is a requirement of many modern application domains, including autonomous vehicles, social computing, contact tracing, intelligent transportation, maritime, etc. Therefore, the demand for an ecosystem of software tools handling moving object big data is rapidly increasing. Tools are required to assist the tasks of the data science cycle including acquisition, storage, cleaning, transformation, analysis, visualization, privacy preservation, etc. Standards play a foundational role in building such an ecosystem, as the key to interoperability.

The International Standard ISO 19141—Schema for moving features [1], issued in 2008 and reviewed and confirmed in 2017, specifies an abstract data representation of a motion consisting of translation and/or rotation of a geographic feature. The schema is based on the concepts of *foliation*, *prism*, *trajectory*, and *leaf*. Fig. 1 illustrates a 2D rectangle which moves and rotates. A leaf is the snapshot of a moving feature at a given time instant, hence a geographic feature. A prism is all the points contained in all leaves in the whole continuous time range of the movement. A trajectory is the path traversed by any point of the feature as it evolves in time. The set of leaves that represent the moving feature forms a foliation.

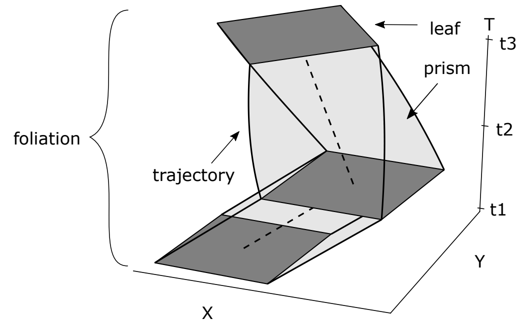


Fig. 1: Schema for Moving Features.

In its current form, ISO 19141 is an abstract schema that defines the spatiotemporal characteristics that are needed to describe moving features. It neither describes an implementation-level data model, nor methods for constructing, storing, and querying moving features in a database. Two main classes of this schema, MF_TemporalTrajectory and MF_RigidTemporalGeometry, can be used to describe moving points and fixed-shape moving geometries respectively. Temporal trajectories have been further detailed in successive implementation-level standards, and have been implemented in multiple common systems. Rigid temporal geometries, in contrast, lack implementation standards and reference implementation. As such, developers have no means of consistently developing applications that manage this kind of data.

Rigid temporal geometries are essential data abstractions for many applications. Safety in sea applications involves analyzing the movement of ships close to each other and close to infrastructures like ports and wind farms. The ability to model the full extent, or approximate it with the main dimensions of the ship, is a requirement to accurately capture the interaction with nearby infrastructure and other ships [2]. In the domain of autonomous vehicles, it is important to include the extent of the vehicles and the nearby objects to analyze the geometric layout of a scene, and the interactions in the scene flow [3]. Similar requirements arise in robot motion management in factories and logistics facilities.

This paper contributes to solving this problem by proposing an implementation of rigid temporal geometries in MobilityDB [4]. MobilityDB is an OSGeo community project[1], that extends PostgreSQL [5] and PostGIS [6] for temporal and spatiotemporal data management. This paper describes

[1]https://www.osgeo.org/projects/mobilitydb/

the implementation of a new algebra for rigid temporal geometries, which allows representing and querying moving geometries with three or six degrees of freedom. Concretely, the contributions of this paper are as follows:

- Implementing the ISO 19141 rigid temporal geometries in MobilityDB by presenting a 2D and 3D data model allowing efficient storage and data manipulation.
- Proposing efficient data management algorithms for rigid temporal geometries. Multiple novel algorithms are described (3D encoding, rotating bounding box, normalization of 2D and 3D rotations), as well as modified versions of existing algorithms (2D encoding, 2D and 3D decoding and interpolation, 2D traversed area).
- Implementing a synthetic data generator for 2D and 3D rigid temporal geometries based on the BerlinMOD generator.
- Assessing ISO 19141 from an implementation point of view. Standard operations on temporal geometries are also implemented and discussed.

Our implementation is publicly available on GitHub.[2]

The rest of this paper is organized as follows. The related work is presented in Section II. Section III describes essential background on ISO 19141 and MobilityDB. The new data model for rigid temporal geometries is then presented in Section IV and the data management algorithms are given in Section V. Section VI then describes the implementation of the standard operations, with a practical use-case on AIS data in Section VII. A synthetic data generator is presented in Section VIII and a revision of the standard is done in Section IX. Finally, Section X concludes this paper.

II. Related Work

A. Standards

International standardization in the field of geographic information has been active for two decades, as mainly coordinated by the ISO Technical Committee 211 (ISO/TC211 2007[3]) and the Open Geospatial Consortium (OGC 2007[4]). A large number of standards (more than 80) has already been published as part of the ISO191xx suite. This is complemented by more than 160 OGC implementation standards, that are written for a more technical audience, and detail the interface structure between software components.

The baseline of these standards are the ones that contribute at various levels to the modelling of geographic information:

- ISO/TS 19103 Conceptual Schema Language, defining the notation for the conceptual modelling of geographic information. It specializes UML, restricting features like multiple inheritance, and adding geographic modelling constructs.
- ISO 19107 Spatial Schema is the main abstract model for geographic features. It specifies a conceptual schema for the geographic types, and basic methods to manage geographic features. It can be seen as a metamodel, that abstracts the different physical data models of geographic applications.
- ISO 19109 Rules for Application Schema defines a standard way of mapping ISO 19107 into an application schema of a given geographic application or class of applications. An application schema is still at the conceptual level, yet it adds specifications relevant to a certain class of applications.
- OGC 06-103r4 Simple feature access - Part 1: Common architecture is an Implementation Standard. It implements a profile of the spatial schema described in ISO 19107. It defines with technical details the structures and operations used to represent and query geographic features in databases. This standard is commonly followed by spatial database developers, including ESRI, Oracle spatial, and PostGIS.
- OGC 06-104r4 Simple feature access - Part 2: SQL option is also an implementation standard. It defines a schema that supports storage, retrieval, query and update of feature collections via the SQL Call-Level Interface. It bases on the ISO 19107 and describes a set of SQL Geometry Types together with SQL functions on those types. This standard is also commonly followed by spatial database developers.

In contrast with this mature list of standards for simple features, the standards for moving features are still in infancy:

- ISO 191141 Schema for moving features is the abstract (metamodel) for data representation.
- OGC 16-120r3 Moving Features Access specifies methods specifies abstract methods to access a database storing temporal trajectory data. Rigid temporal geometries are not in the scope of this standard.

Thus, the standardization of rigid temporal geometry data representation and functions remains at the abstract level. Furthermore, up to our knowledge, there is no implementation of it. This is a clear gap that needs to be filled by implementation standards and reference implementations, to support the interoperability of rigid temporal geometry data.

B. Moving Object Data Management

The research in moving object data management has been active since early 2000, covering all aspects of modelling, indexing, query operations, analysis, visualization, data uncertainty, etc. Multiple research prototypes exist, including SECONDO[7], UlTraMan[8] on spark, HadoopTrajectory[9], and TrajMesa[10]. There is also MobilityDB[4], an industrial-strength moving object database system.

These systems mainly focus on managing moving point objects. Less focus has been given to temporal geometries, also known as moving regions. A distinction is made in the literature between continuous trajectories and discrete points. TrajMesa [10], for instance, represents a trajectory as a sequence of timestamped points without assuming interpolation between points. In this sense, the problem of managing temporal geometries reduces to managing discrete polygon objects. Another distinction is whether the temporal geometry is of a fixed-shape or a deforming one. The case of deforming regions has been investigated in [11], [12], which present a model using a sliced representation. A polyhedral-based model [13] has been presented to solve some efficiency issues with the sliced representation model.

[2]https://github.com/mschoema/MobilityDB/tree/geometry

[3]https://committee.iso.org/home/tc211

[4]https://www.ogc.org/

This sliced representation of moving regions assumes linear interpolation between the start and end segments of the region, which does not accurately represent the movement of the vertices of a fixed-shape moving region. This model is therefore not valid for representing ships, aircraft, or similar objects whose shape does not change while moving. Towards addressing this requirement, a model for 2D fixed-shape moving regions has been presented in [14], which is the most related work to this paper. A major difference is that it represents the movement as a sequence of slices, where every slice stores the transformation of the temporal geometry at the two sides of the time interval. This introduces a lot of redundancy in the data representation, which decreases the overall performance. The ISO 19141, in contrast, represents the movement as a set of snapshots at discrete time instants and interpolates the transformation vectors in-between. The difference between the two models is captured in detail in [4]. In [14], a set of algorithms for fixed-shape moving regions are also presented, such as *point inside*, *region intersects* or *traversed area*. Ideas from these algorithms are reused here, for the 2D case, as will be cited in place.

The 3D temporal geometries are specified in ISO 19141, but not handled in the moving object data management literature. Therefore, ideas in this respect have to be surveyed in the image processing and animation literature. As will be explained in Section IV, the data model of 3D fixed-shape moving geometries makes use of quaternions to represent the 3D orientation/rotation of a geometry. This use of quaternions for the representation of orientations and rotations in three dimensions has already been researched [15], and they are already used, for example, to create smooth animations and movements in computer graphics.

III. Background

A. ISO 19141 Standard on Moving Features

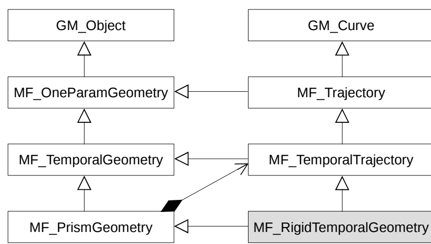


Fig. 2: Moving Features Types

The main classes of the schema for moving features are shown in Fig. 2. They form an inheritance hierarchy of three levels. At the top level are GM_Object and GM_Curve from the ISO 19107 standard — schema for spatial features. As such, a moving feature is compliant with the General Feature Model. The classes in the moving features package are denoted by the prefix MF. Their schema is based on the concept of one parameter set of geometries, which represents a function from the domain of some parameter p into a geometry object, that is $f : D_p \implies D_{GM_Object}$, where D_{type} denotes the domain of a type. A one parameter set of geometries represents an infinite set of *leaves*, where a single *leaf* $f(p)$ is the geometry at a given value of the parameter. If the leaf is a single geometry point, then this class can be specialized as an MF_Trajectory. The parameter could be any variable such as pressure, temperature or time.

The third level of the hierarchy specializes the parameter to time, expressed as TM_Coordinate in ISO 19108 [16]. The class MF_TemporalGeometry represents a mapping from time instants into geometries. During its definition interval [startTime, endTime], see Fig.3, it defines a continuous mapping from a time instant into a geometry object, i.e., the leafGeometry function. The class attributes and operations in Fig. 3 will be discussed in detail in SectionVI. MF_TemporalTrajectory specializes MF_TemporalGeometry by restricting the leaf geometry to a single point. An MF_TemporalTrajectory object can thus represent a moving point object, such as a person, a car, etc. It is the commonly used moving feature type. It is therefore the most elaborate class in this standard, in terms of attributes and functions. Corresponding abstractions to MF_TemporalTrajectory have been implemented in few database systems including PostGIS' LineStringM and trajectory functions, Informix®Spatiotemporal Search, MobilityDB temporal geometry point types and query API.

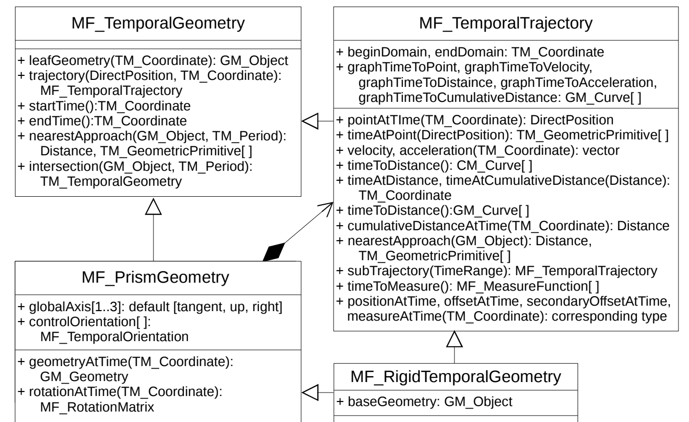


Fig. 3: Moving Features Types

The MF_PrismGeometry and MF_RigidTemporalGeometry classes restrict MF_TemporalGeometry for the case of an object that has arbitrary geometry and moves without deformation. The object may translate and/or rotate, but its shape remains congruent. Actually, ISO 19141 puts deformable moving features out of its scope. Every point in an MF_RigidTemporalGeometry is thus an MF_TemporalTrajectory, and one point acts as the centre of rotation of the MF_RigidTemporalGeometry, which explains the association relation in Fig. 2 and Fig. 3. What is not explained in the standard is the inheritance relation between MF_RigidTemporalGeometry and MF_TemporalTrajectory, especially that certain functions in MF_TemporalTrajectory cannot apply to MF_RigidTemporalGeometry. This is further discussed in Section IX. Section VI thus only contains the

operations from MF_TemporalGeometry, MF_PrismGeometry and MF_RigidTemporalGeometry.

The OGC 16-120r3 Moving Features Access standard complements ISO 19141 for the MF_TemporalTrajectory type and elaborates on its data management methods. Moreover, the few aforementioned system implementations provide references for users and developers in implementing the standard. The MF_RigidTemporalGeometry lacks such implementation standards and reference implementations.

B. MobilityDB

MobilityDB implements ADTs of moving objects in PostgreSQL and PostGIS. It further supports these types with indexes, a rich set of temporal and spatiotemporal query operations, and an SQL query interface. It is by far the only existing moving object database that supports full SQL. It is compliant with the OGC standards on moving features.

The type system of MobilityDB[4] is based on the concept of type constructors. These are functions that yield data types. It is therefore extensible. A *temporal type* is constructed using a *time type* and a *base type*. The possible time types are `timestamp`, `timestampSet`, `period` and `periodSet`, representing respectively a single time instant, a set of discrete time instants, a continuous interval, and a disconnected set of intervals. The *base type* can be any PostgreSQL and PostGIS type. The current implementation of MobilityDB has temporal types corresponding to the base types `bool`, `int`, `float`, `text`, `geometry(Point)` and `geography(Point)`.

Every combination of a base type with a time type thus creates a temporal type. For example, by combining the `geometry(Point)` base type with the time type `timestampSet` we construct a temporal type that can represent a set of disconnected spatiotemporal points, e.g., check-ins of a Foursquare user. Combining `geometry(Point)` and `period` would construct a temporal type that can represent the continuous movement of a point, i.e., a temporal point. Since the input is always discrete, such as GPS points, continuous temporal types use interpolation functions between consecutive instants. MobilityDB supports two interpolation functions: step, and linear.

Formally, the type system consists of four type constructors, corresponding to the four time types: *INSTANT*, *INSTANTS*, *SEQUENCE*, *SEQUENCES*. Given a base type S, e.g., `geometry(Point)`, the *INSTANT* type constructor constructs a temporal type using S and *timestamp* as follows:

$$D_{\mathit{INSTANT}(S)} = D_{metavalue(S)} \times D_{timestamp} \quad (1)$$

Here *metavalue*(S) corresponds to the base type, either saved as-is or as a delta type, that is *metavalue*(S) could be D_S or another transformation. Storing a transformed form of the values of the base type, instead of the direct values, gives the possibility to compress the representation of base type at individual timestamps. It is especially useful in the case of rigid temporal geometries since the shape is preserved during the movement, and we thus don't need to store the complete geometry at every individual timestamp. This concept is referred to as delta encoding in the rest of the paper.

The *SEQUENCE* type constructor builds a temporal type using a base type S and the time type `period` as follows:

$$\begin{aligned} D_{\mathit{SEQUENCE}(S)} = \{ & (I, li, ui, \mathit{interpolation}) \mid \\ & \text{(i) } I = [(v_1, t_1), \ldots, (v_n, t_n)] \text{ is a list of } \mathit{INSTANT}(S) \\ & \text{(ii) } li, ui \in \{\mathit{true}, \mathit{false}\} \text{(in/ex)clusive bounds of period} \\ & \text{(iii) } \mathit{interpolation} \in \{\mathit{step}, \mathit{linear}\} \\ & \text{(iv) consecutive pieces of the interpolation function are} \\ & \text{not colinear} \} \end{aligned} \quad (2)$$

It represents a continuous evolution of value over a time period, that can be left/right open/closed. Individual instants are of type *INSTANT*(S). They could thus be represented using the metavalue of the base type, which is important to our implementation. Between instants $(v_i, t_i), (v_{i+1}, t_{i+1})$ the value is interpolated according to the chosen function.

To define a new temporal type in this type system, it is thus sufficient to define the domain of its metavalue $D_{metavalue(S)}$. The type constructors would then take care of constructing the corresponding temporal types. In MobilityDB, this effectively means that it can be extended with low development cost. Then the effort of development can be re-directed to implementing data management and query functions for the new types.

IV. Data Model

This section describes the data model for rigid temporal geometries and integrates it into the MobilityDB type system. To avoid redundancy, we focus the discussion on geometries represented as polygons in 2D and polyhedral surfaces in 3D. A similar development can be done for other geometry types such as linestring, points, collection types and complex types.

A. 2D geometries

For the PostGIS base type `geometry(Polygon)`, we handle the object as a 2D temporal geometry, and the metavalue thus contains the parameters for a 2D affine transformation allowing only a translation and a rotation.

$$metavalue(S) = (o, \theta, dx, dy)$$

Here o is a pointer to the original geometry. The parameters $(dx, dy) \in R^2$ represent the translation of the centre of rotation of the geometry. The rotation parameter in 2D is $\theta \in (-\pi, \pi]$. These parameters are with reference to the original geometry.

The centre of rotation is required to compute a unique transformation value between two geometries. Without loss of generality, we will fix the centre of rotation as the centroid of the geometry. It is easy afterwards to adapt the algorithms to handle an arbitrary centre of rotation, i.e., given by the user.

Accordingly, the four type constructors, *INSTANT*, *INSTANTS*, *SEQUENCE*, and *SEQUENCES*, see (1), (2), will extend the system with corresponding types. For instance, the moving feature example in Fig. 1 would be represented (in the physical storage) as the *SEQUENCE*(geometry(Polygon))

$$\Big(\big[\ ((o,0,0,0),t_1), ((o,0,1,0),t_2), ((o,\tfrac{\pi}{4},0,1),t_3) \ \big], \ true, true, linear \Big)$$

meaning that the moving feature object is defined over a continuous interval $[t_1, t_3]$, with inclusive bounds. It has three instants, each of which is a pair of a metavalue quadruple and a timestamp. The initial geometry, i.e., the rectangle shape, is stored somewhere, and it is referenced by the pointer o in the three instants. In this example, the object first only translates, then rotates and translates at the same time. The second and third instants have rotations of 0 and $\frac{\pi}{4}$ degrees respectively, with reference to the initial original geometry and translation of $(1, 0)$ and $(0, 1)$ respectively.

Two possible solutions exist for storing the original geometry; either in a separate table or inside the object. In this implementation, we choose to keep the first instant of a temporal geometry not as a delta, but as a geometry object. All following instants will then refer to this geometry in the first instant as the original geometry o.

B. 3D geometries

A three-dimensional volumetric object is represented using the PostGIS `geometry(Polyhedralsurface)` type and the corresponding metavalue contains the parameters for a 3D affine transformation allowing only a translation and a rotation.

$$metavalue(S) = (o, W, X, Y, Z, dx, dy, dz)$$

The parameters for the 3D translation are (dx, dy, dz) and, similarly to the 2D translation parameters, represent the translation of the rotation centre of the geometry.

3D rotations are handled and interpolated using unit quaternions [15], and the rotation parameters (W, X, Y, Z) thus correspond to the four parameters of a unit quaternion.

$$Q = \langle W, X, Y, Z \rangle, \ with \ \|Q\|^2 = 1$$

Similar to 2D geometries, we choose to store the original geometry in the first instant and store all subsequent instants as delta value (metavalue) with reference to the first instant. Having defined the metavalue type, the type constructors will generate the corresponding types for 3D temporal geometries.

V. Data Management Algorithms

This section describes essential internal algorithms for managing rigid temporal geometries. The operations defined by the standards (User API) will be discussed later in Section VI.

A. Encoder and Decoder

While the representation of temporal geometries consists of transformation tuples, i.e., delta encoding, the data typically arrives as a sequence of geometry snapshots. We, therefore, define encoding and decoding functions to transform from geometries into metavalues, and vice versa.

1) Encoder: The encoding function takes two geometries given either as 2D polygons or 3D polyhedra and computes the transformation from the first to the second geometry. As detailed in the data model (Section IV), this transformation consists of both translation and rotation parameters. The computed transformation is unique and always assumes a minimal rotation between the two input geometries. If the rotation between two subsequent geometries is known to be larger than π, additional intermediate instants have to be added to keep the rotation between two subsequent instants smaller than π.

Here we illustrate the more complex case of 3D. The *encode* function is illustrated in Algorithm 1. First, we compute the translation parameters, as the difference between the coordinates of the geometries centres of rotation. Then we translate both geometries to have their respective rotation centres at the origin. The remaining problem is to compute a rotation quaternion from the first geometry to the second one. For this, we make an assumption on the input geometries: the respective points of the geometries must be given in the same order. This allows, for example, different 90-degree rotations of a cube to be distinguishable from each other when given as input.

Let $\overrightarrow{P}$ and $\overrightarrow{R}$ denote two vertices (different from the centre of rotation) of the first geometry, and let $\overrightarrow{P'}$ and $\overrightarrow{R'}$ denote the corresponding points on the second geometry. As a first step, we compute the rotation axis by realising that both $\overrightarrow{PP'}$ and $\overrightarrow{RR'}$ are perpendicular to this axis. This rotation axis is represented by a unit vector $\overrightarrow{e}$.

$$\overrightarrow{e} = \frac{\overrightarrow{PP'} \times \overrightarrow{RR'}}{\|\overrightarrow{PP'} \times \overrightarrow{RR'}\|} \tag{3}$$

With $\overrightarrow{P_e}$ and $\overrightarrow{P'_e}$ denoting the projections of respectively $\overrightarrow{P}$ and $\overrightarrow{P'}$ onto the plane perpendicular to $\overrightarrow{e}$ going through the origin, we can compute the rotation angle $\theta = sign(\theta) \cdot |\theta|$.

$$\begin{aligned} \overrightarrow{P_e} &= \overrightarrow{P} - (\overrightarrow{P} \cdot \overrightarrow{e}) \cdot \overrightarrow{e} & \overrightarrow{P'_e} &= \overrightarrow{P'} - (\overrightarrow{P'} \cdot \overrightarrow{e}) \cdot \overrightarrow{e} \\ |\theta| &= \text{acos}\left(\frac{\overrightarrow{P_e} \cdot \overrightarrow{P'_e}}{\|\overrightarrow{P_e}\| \cdot \|\overrightarrow{P'_e}\|}\right) & \text{sign}(\theta) &= \text{sign}(\overrightarrow{e} \cdot (\overrightarrow{P_e} \times \overrightarrow{P'_e})) \end{aligned} \tag{4}$$

Finally, the rotation quaternion $Q = (W, X, Y, Z)$ is computed from the axis-angle representation.

$$\begin{aligned} W &= \cos(\tfrac{\theta}{2}) & X &= \overrightarrow{e_x} \cdot \sin(\tfrac{\theta}{2}) \\ Y &= \overrightarrow{e_y} \cdot \sin(\tfrac{\theta}{2}) & Z &= \overrightarrow{e_z} \cdot \sin(\tfrac{\theta}{2}) \end{aligned} \tag{5}$$

The equations above correspond to the general case where $\overrightarrow{PP'}$ and $\overrightarrow{RR'}$ have nonzero length and are not parallel. Other equations exist for the special cases, but this is not detailed in this paper.

The encoder, per se, does not work in an incremental streaming way. Indeed we assume that the movement is historical and that no update will be done to the instants composing the movement. If needed, however, the algorithm for updating is trivial, as it only needs to find the transformation vector of the updated instant w.r.t the initial instant.

2) Decoder: The decoding function takes a transformation (delta value) as input and applies this transformation to its reference geometry to recompute the initial data value. We use the PostGIS function `ST_Affine` to apply affine transformations in 2D and 3D to the reference Geometry.

We first transform the rotation parameters into a rotation matrix R.

$$R = \begin{pmatrix} cos(\theta) & sin(\theta) \\ sin(\theta) & -cos(\theta) \end{pmatrix} \tag{6}$$

Algorithm 1: encode($\mathcal{G}_1, \mathcal{G}_2, out\ \mathcal{T}$)

Input: $\mathcal{G}_1, \mathcal{G}_2$, two congruent geometries
Output: $\mathcal{T}$, 2D or 3D transformation from $\mathcal{G}_1$ to $\mathcal{G}_2$
begin
  $o \leftarrow$ pointer to $\mathcal{G}_1$;
  initialize $\overrightarrow{C}_1$ and $\overrightarrow{C}_2$ as the centroid of $\mathcal{G}_1$ and $\mathcal{G}_2$ respectively;
  $\overrightarrow{T} \leftarrow \overrightarrow{C}_2 - \overrightarrow{C}_1$;
  translate $\mathcal{G}_1$ and $\mathcal{G}_2$ to have their centroid at the origin;
  if *2D* **then**
    initialize $\overrightarrow{P}$ and $\overrightarrow{P'}$ as the first vertex of $\mathcal{G}_1$ and $\mathcal{G}_2$ respectively;
    $\theta \leftarrow$ angle between $\overrightarrow{P}$ and $\overrightarrow{P'}$;
    return *Transform2D*($o, \theta, \overrightarrow{T}$)
  else
    initialize $\overrightarrow{P}, \overrightarrow{R}, \overrightarrow{P'}$ and $\overrightarrow{R'}$ as the first two vertices of $\mathcal{G}_1$ and $\mathcal{G}_2$ respectively;
    compute $\overrightarrow{e}$, θ, Q using (3), (4), (5) respectively;
    return *Transform3D*($o, Q, \overrightarrow{T}$);

In 3D, the conversion from quaternion to rotation matrix results in the following 3×3 matrix.

$$R = 2\begin{pmatrix} \frac{1}{2} - (Y^2 + Z^2) & (XY - WZ) & (XZ + WY) \\ (XY + WZ) & \frac{1}{2} - (X^2 + Z^2) & (YZ - WX) \\ (XZ - WY) & (YZ + WX) & \frac{1}{2} - (X^2 - Y^2) \end{pmatrix} \quad (7)$$

The decoding function then works in three steps. First, it translates a copy of the reference geometry to place its rotation centre at the origin. It then applies the rotation matrix R to the geometry, which corresponds to a rotation around the origin. Lastly, it translates the geometry back while at the same time applying the needed translation from the input transformation.

B. Interpolation

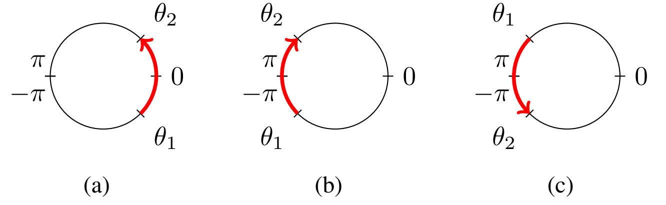
Interpolation between consecutive instants is part of the definition of *SEQUENCE* and *SEQUENCES* types, as in (2). Step interpolation is straightforward, as the geometry remains the same until the next instant. In the following, we describe the linear interpolation functions for 2D and 3D geometries.

The linear interpolation is applied to the transformation parameters. Indeed, we cannot interpolate the vertices of the geometries without deforming them. Thus we first interpolate the transformation parameters at the given time, then use the decoding function to get the interpolated geometry.

Given two transformations $\mathcal{T}_1$ and $\mathcal{T}_2$ at times t_1 and t_2, we want to compute the transformation parameters at $t_1 < t < t_2$. For both the 2D and the 3D case, the translation parameters $\overrightarrow{T} = (dx, dy, dz)$ can be interpolated using linear referencing.

$$r = \frac{t - t_1}{t_2 - t_1} \quad (8)$$

$$\overrightarrow{T} = \overrightarrow{T}_1 * (1 - r) + \overrightarrow{T}_2 * r \quad (9)$$



(a) (b) (c)

Fig. 4: Three different situations when interpolating angles of rotation: (a) ($\theta_2 > \theta_1$ and $\theta_2 - \theta_1 <= \pi$) or ($\theta_1 > \theta_2$ and $\theta_1 - \theta_2 < \pi$) (b) $\theta_1 < \theta_2$ and $\theta_2 - \theta_1 > \pi$ (c) $\theta_1 > \theta_2$ and $\theta_1 - \theta_2 >= \pi$

The difference arises when interpolating the rotation. In two dimensions, we linearly interpolate the rotation angle θ, while still keeping in mind that this angle has to stay between $-\pi$ and π. In 3D, we interpolate the rotation quaternion using the *slerp* algorithm. These interpolation methods always use the smallest angle between the two instants to construct a unique output value. This means that during construction, no two subsequent instants can have a rotation of more than π between them as explained in Section V-A.

1) 2D Case: Given two rotation angles θ_1 and θ_2 at times t_1 and t_2, we want to compute the rotation angle θ at $t_1 < t < t_2$. Since the angle has to stay between $-\pi$ and π, one of three cases can apply depending on the relative position of the start and end angles. In Fig. 4, we project the bounded linear axis on a unit circle and distinguish the three cases.

Equation (10) can then be used to compute the resulting rotation angle, according to the case.

$$\theta = \begin{cases} \text{case a: } \theta_1 + (\theta_2 - \theta_1) * r \\ \text{case b: } \theta_2 + (2\pi + \theta_1 - \theta_2) * (1 - r) \\ \text{case c: } \theta_1 + (2\pi + \theta_2 - \theta_1) * r \end{cases} \quad (10)$$

Finally, if the resulting angle is larger than π, we subtract 2π to keep the angle values between $-\pi$ and π.

2) 3D Case: Given two rotation quaternions Q_1 and Q_2 at times t_1 and t_2, we want to compute the rotation quaternion Q at $t_1 < t < t_2$. This is done using the *slerp* (spherical linear interpolation) algorithm [15], which corresponds to a linear interpolation of the rotation angle around a fixed rotation axis.

$$\begin{aligned} \cos(\theta) &= Q_0 \bullet Q_1 \\ Q &= Slerp(Q_0, Q_1, r) \\ &= \frac{Q_0 * \sin((1 - r) * \theta) + Q_1 * \sin(r * \theta)}{\sin(\theta)} \end{aligned} \quad (11)$$

C. Bounding Box

MobilityDB pre-computes and stores the bounding boxes of temporal types, as a means to increase the efficiency of operations. Computing this bounding box for temporal points is done by taking the smallest box enclosing all instants. Since the interpolation techniques used in MobilityDB are linear and stepwise, this bounding box will always be correct.

For rigid temporal geometries, in contrast, the smallest box containing all the defined instants does not correspond

Algorithm 2: interpolate($\mathcal{T}_1, \mathcal{T}_2, r, out\ \mathcal{T}$)

Input: $\mathcal{T}_1, \mathcal{T}_2, r$, two transformations and a ratio value as computed using (8)
Output: $\mathcal{T}$, the interpolated transformation
begin
  $\vec{T} \leftarrow$ linear interpolation of the translation parameters using (9);
  if *2D* **then**
    $\theta \leftarrow$ compute the rotation angle using (10);
    return *Transform2D*$(\theta, \vec{T})$;
  else
    $Q \leftarrow$ compute the quaternion using (11);
    return *Transform3D*$(Q, \vec{T})$;

to a correct bounding box. Indeed, the geometry can exit this box during its movement if it is rotating. This comes from the fact that, although the transformation parameters are interpolated linearly, the movement of the vertices is not linear. Interpolating between two instants could thus result in a geometry which is not fully contained by this naively computed bounding box, Fig. 5.

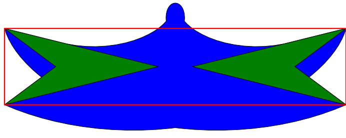


Fig. 5: The geometry in the left performs a translation to the right and a rotation of π. As illustrated, a simple union of the bounding boxes of individual instants is not a correct bounding box for the whole movement.

A first solution is to compute the bounding box exactly if we know the traversed area of the temporal geometry. Since computing this traversed area is a computationally heavy operation, we present a second solution which returns a sub-optimal, but still correct bounding box.

This second solution computes a *rotating bounding box* around every instant, and the resulting bounding box of the temporal geometries is then the smallest box enclosing all the rotating bounding boxes of the instants.

This rotating bounding box is computed by creating a square box around the rotation centre of the geometry, with sides twice the length of the longest distance between the rotation centre and any vertex of the geometry. Since the temporal geometry is rigid, this distance will be the same for all instants, and will thus be computed only once.

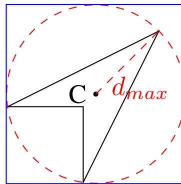


Fig. 6: Rotating bounding box around an instant

$$d_{max} = \max\{\sqrt{x^2+y^2},\ (x,y) \in V_{geometry}\} \tag{12}$$

The bounding box of a temporal geometry of any duration is then defined as a tuple of extreme values $(x_{min}, x_{max}, y_{min}, y_{max})$. With $(o, \theta, dx, dy) \in$ `tgeometry`,

$$x_{min} = \min\{dx\} - d_{max} \quad x_{max} = \max\{dx\} + d_{max}$$
$$y_{min} = \min\{dy\} - d_{max} \quad y_{max} = \max\{dy\} + d_{max}$$

The 3D case is omitted here but is analogous to the 2D case presented above.

The complexity of the algorithm is $O(m+n)$, with m being the number of vertices of the geometry and n being the number of instants in the temporal geometry. This is a significant improvement compared to the first method utilizing the traversed area of the geometry. Indeed, if the traversed area between every pair of consecutive instants is computed using the traversed area algorithm presented in [14], the overall complexity is $O(m^2 * n)$.

This algorithm computes a sub-optimal bounding box since it is, in all generality, not the smallest bounding box possible, but is still correct in the sense that the temporal geometry never exits the box during its movement. It thus can be used for indexing or filtering purposes, although less effective than an optimal bounding box. An example of a query using the bounding box indexes is shown in Section VII.

D. Normalization

In the type system of MobilityDB, temporal objects are normalized to guarantee unique representation. The normalization of a temporal sequence removes redundant/collinear intermediate instants if found. This is stated in the last condition in the definition of the *SEQUENCE* type constructor in (2).

Since the interpolation method used for temporal geometries always uses the smallest rotation between two instants, removing intermediate collinear instants when the total rotation exceeds 180 degrees will force the direction of rotation to change. An example of this is displayed in Fig. 7.

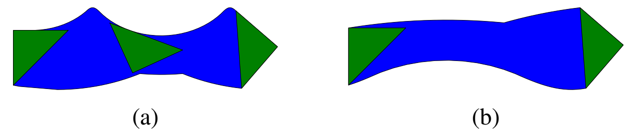


Fig. 7: Example of a collinear but not redundant instant in a moving geometry. (a) Path (in blue) taken with the collinear geometry (counter-clockwise rotation) (b) Path taken without the collinear geometry (clockwise rotation)

Keeping the intermediate collinear instants is also not a possibility, since this contradicts the constraint of having a unique representation for all identical temporal objects. To solve this issue, we replace the non-redundant collinear instants with a *dummy instant*. This dummy instant is the same for all identical temporal objects and maintains the correct direction of rotation.

Given three subsequent instants of a temporal geometry, we want to check if the middle instant is collinear and if it is redundant. Three instants are collinear if their translation and rotation parameters are collinear. The collinearity check of the translation parameters is similar to temporal points. In the following we discuss collinearity of rotation parameters.

In 3D, the *slerp* algorithm computes a linear interpolation of the angle in the axis-angle representation of the rotation. Here again, similar to the 2D case in Section V-B, the smallest angle is used. But the algorithm could be adapted to interpolate along the largest angle. Let's denote the two variants *interpolate_short* and *interpolate_long* respectively. The interpolation method used to retrieve the transformation at a given time is always *interpolate_short* (Section V-B).

During normalization, we compute the result of both interpolation methods between the first and third instant. If the middle instant is different form both results, or if the two subsequent transformations do not have the same direction of rotation, then it is not collinear and nothing has to be done. Otherwise if the middle instant is equal to the result of the *interpolate_short* function, then the instant is redundant and can be safely removed. Lastly, if the middle instant is equal to the result of the *interpolate_long*, then it is collinear since it can be recomputed from the first and third instant, but not redundant since the default interpolation method does not compute the correct value.

In this third case, we replace the second instant with a *dummy instant*, that will be the same for all temporal geometries representing the same movement, while still preserving the initial direction of rotation. We place the dummy instant such that the rotation between the first and second instant is equal to $\pi - \epsilon$, with epsilon as small as possible to minimize the number of dummy instants needed, but large enough to keep the angle below π no matter the precision errors. In practice, since the maximum allowed precision error in MobilityDB is 10^{-5}, we chose $\epsilon = 10^{-4}$.

Algorithm 3: normalize($\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3, t_1, t_2, t_3, out\ \mathcal{T}, out\ t$)

Input: $\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3, t_1, t_2, t_3$, three transformations and their corresponding time.
Output: $\mathcal{T}, t$, the new intermediate instant.
begin
- $r \leftarrow \frac{t_2 - t_1}{t_3 - t_1}$;
- $\mathcal{T}_{short} \leftarrow interpolate_short(\mathcal{T}_1, \mathcal{T}_3, r)$;
- $\mathcal{T}_{long} \leftarrow interpolate_long(\mathcal{T}_1, \mathcal{T}_3, r)$;
- **if** $\mathcal{T}_2 = \mathcal{T}_{short}$ **then**
  - **return** $NULL, NULL$;
- **else if** $\mathcal{T}_2 = \mathcal{T}_{long}$ **then**
  - compute r_{dummy} using (13);
  - $\mathcal{T}, t \leftarrow interpolate_long(\mathcal{T}_1, \mathcal{T}_3, r_{dummy})$;
  - **return** $\mathcal{T}, t$;
- **else**
  - **return** $\mathcal{T}_2, t_2$;

The r_{dummy} parameter used in Algorithm 3 is the ratio needed to compute the correct dummy instant using the *interpolate_long* function. With θ_{12} and θ_{23} being the rotation angles for the two subsequent 2D transformations, we have $sign(\theta_{12}) = sign(\theta_{23})$ and $|\theta_{12} + \theta_{23}| > \pi$. The dummy ratio is then computed using (13). In 3D, we can use the same equation with the angles of the axis-angle representation of the rotation quaternions.

$$r_{dummy} = \frac{\pi - \epsilon}{|\theta_{12} + \theta_{23}|} \tag{13}$$

E. Traversed Area

The traversed area is the set of all point traversed by the temporal geometry. It can be seen as a projection onto 2D or 3D space of the prism of a moving feature as described in Section I. We present an algorithm to compute a polygon that approximates the traversed area of a 2D rigid temporal geometry with a number of straight segments. This choice allows us to utilize the existing PostGIS spatial functions on the polygon type to analyze and manipulate the traversed area. With this assumption made initially, the algorithm differs significantly from the one presented in [14], since we do not need to define new curve types to represent the border of the traversed area.

The area traversed by a temporal geometry is in all generality not perfectly representable by a polygon, and we thus need to approximate the curved border of the real area using straight segments. This approximation is not necessary when computing the traversed area of a translating (but not rotating) polygon. We will thus assume that we can approximate the movement of a vertex of a temporal geometry by a single straight segment when the applied transformation has a rotation smaller than a given threshold θ_{max}. The smaller this θ_{max}, the better the approximation will be.

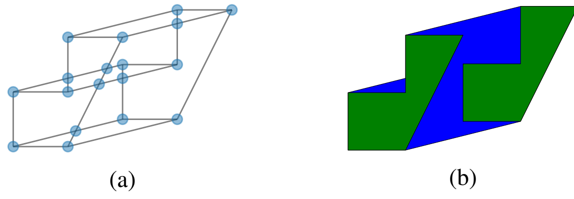
Given a sequence of transformations, all relating to the same reference polygon, Algorithm 4 computes the traversed area of this sequence as follows. As a first step, we compute the rotation θ between each pair of subsequent transformations and add n interpolated transformations between them if the total rotation θ was larger than θ_{max}. The number of intermediate transformations added is computed using (14).

$$n = \left\lceil \frac{\theta}{\theta_{max}} \right\rceil - 1 \tag{14}$$

Then we compute the traversed area between two consecutive transformations using the assumption that the geometry moves linearly. Lastly, we combine all the computed traversed areas into a single polygon using the PostGIS *ST_Union* function.

The remaining problem consists of computing the traversed area of two subsequent transformations assuming linear interpolation of vertices. We do this in three steps. First, we create a list of segments consisting of the edges of the start and end polygon as well as the straight segments representing the movement of the vertices. We then compute the intersections between every pair of segments to create a planar graph. Lastly, we compute the traversed area by starting at the left-most vertex of the graph and travelling along the border of

this graph until we arrive back at the start vertex. An example of this process is shown in Fig. 8a.



(a) (b)

Fig. 8: Process of computing the traversed area between two subsequent transformations. (a) Planar graph created from the computed segments. (b) Start and end polygons (green) with the resulting traversed area (blue).

Algorithm 4: traversedArea($\mathcal{T}_{arr}, \theta_{max}, out\ \mathcal{P}$)

Input: $\mathcal{T}_{arr}, \theta_{max}$, an array of transformations and the maximum rotation angle
Output: $\mathcal{P}$, the polygon representing the traversed area
begin
  for $(\mathcal{T}_i, \mathcal{T}_{i+1})$ *in* $\mathcal{T}_{arr}$ **do**
    compute the rotation θ from $\mathcal{T}_i$ to $\mathcal{T}_{i+1}$;
    compute n using (14);
    add n intermediate transformations computed using *interpolate*;
  /* subsequent transformations in $\mathcal{T}_{arr}$ now have a rotation smaller than θ_{max} */
  $\mathcal{P}_{arr} \leftarrow$ empty polygon array;
  for $(\mathcal{T}_i, \mathcal{T}_{i+1})$ *in* $\mathcal{T}_{arr}$ **do**
    compute the traversed area between $\mathcal{T}_i$ and $\mathcal{T}_{i+1}$ assuming linear interpolation of vertices;
    add this polygon to $\mathcal{P}_{arr}$;
  return $ST_Union(\mathcal{P}_{arr})$;

VI. Standard Operations On Rigid Temporal Geometries

The operations of the MF_RigidTemporalGeometry class of ISO 19141 are mostly inherited from its parent classes. This section presents their algorithms. Some operations that are defined in the standard cannot be evaluated, either for lack of an algorithm or because of type system limitations. We, therefore, believe that the discussion in this section can be useful in revising the standard.

A. LeafGeometry and GeometryAtTime

The operation *LeafGeometry* of the MF_TemporalGeometry class accepts a timestamp and returns the geometry object representing the leaf at that time. The MF_PrismGeometry class defines a similar function called *GeometryAtTime*.

We implement it in MobilityDB as an overload of the existing `valueAtTimestamp` function. It computes the value of the transformation at the given timestamp using Algorithm 2 of interpolation, then uses the decoder to compute the resulting PostGIS `geometry(Polygon)`. If the temporal geometry is not defined at the given timestamp, the function returns `NULL`.

B. Trajectory

The *Trajectory* operation takes a point on a leaf at a given timestamp and returns the MF_TemporalTrajectory representing the trajectory of this point. This operation can, for example, retrieve the evolving position of the bow of a ship.

We represent the result with the MobilityDB type *SEQUENCE*(geometry(Polygon)). It assumes that the movement of temporal points is piecewise linear. Therefore, we need to approximate the resulting trajectory, which can be an arbitrary curve, into a piecewise linear trajectory.

We can compute the position of the temporal point at a given instant exactly using the interpolation function on the transformations, and applying this interpolated transformation to the initial point. Note that this does not imply interpolating the whole geometry. The interpolation is done only for the specified point, which is much faster. The accuracy of the result will depend on the number of intermediate instants, which is left here as a user parameter.

When the input point corresponds to the centre of rotation, the function returns an exact solution since the rotation centre of the geometry is assumed to move linearly. The result corresponds to the *originTrajectory* attribute defined in the MF_PrismGeometry class.

C. StartTime and EndTime

These are simple getters of the first and last timestamps of the temporal geometry, i.e., t_1, t_n in (2).

D. NearestApproach

This function returns the distance and time of the nearest approach between a temporal geometry and another geometric object. An optional parameter *timeInterval* restricts the search to a given period. This operation can take both a static or a temporal geometry as the second argument.

The equations and algorithms for finding both the distance and the time of the nearest approach in both 2D or 3D are complex computational geometry problems on their own and do not fit into the scope of this paper. However, we present a solution for the case of a temporal and static geometry in 2D.

This solution consists of two steps. Firstly, we compute the traversed area of the temporal geometry using Algorithm 4. Secondly, we apply the PostGIS function *ST_Distance* between this traversed area and the fixed geometry. This function returns the shortest distance between the two geometries. In MobilityBD, we implement it as an overload of the `nearestApproachDistance` function, which is readily implemented for temporal points.

E. Intersection

When talking about static geometries, it is common to discuss their intersection, union or difference. The ISO 19141 standard also defines an *intersection* operator between a temporal geometry and any other geometry, either temporal or static. This operation returns a temporal geometry describing the intersection between both geometries at any point in time.

When looking at the intersection between a temporal and a fixed geometry, it is clear that the result will not be of fixed shape. The same goes intersecting two temporal geometries. Since deforming geometries are not modeled in ISO 19141, this function cannot be defined. Alternatively, a special definition of its semantic has to be added.

F. RotationAtTime

The standard describes an operation called *rotationAtTime*, which returns the rotation needed to correctly orient the geometry in the global frame at the given time. Assuming a (3D) rigid geometry with a natural front, left and up parallel to the x, y and z-axis respectively, this operation returns the rotation that we need to apply to its rotation centre to give it the correct orientation at the given time.

We define three versions of the *rotationAtTime* operation that vary on the return type: angle, quaternion and matrix. In our model, the rotation angles/quaternions, which are stored in the instants, are expressed in reference to the original geometry. Based on this, we implement the function `quaternionAtTimestamp`, that returns the rotation quaternion at a given time. It represents the rotation we need to apply to the first instant to orient the geometry correctly at that timestamp. An optional argument *initialQuaternion* can be used to describe the initial orientation of the geometry. The function will then combine the rotation quaternion, defined with respect to the first instant, with this initial quaternion. The returned quaternion, as well as the optional quaternion argument, is given as an array of four doubles. A similar function `angleAtTimestamp` is implemented for 2D rotations and returns a single double value.

A third version, `rotationMatrixAtTimestamp`, converts the result from rotation angle or quaternion to rotation matrix, i.e., equations (6) and (7). The optional argument is also received as a matrix. If the received matrix does not correspond to a pure rotation, an error is thrown.

VII. Use Case

The Danish Maritime Authority publishes histories of AIS ship trajectories[5] in CSV format. Each row contains a single point of a ship track. The relevant columns of this data are listed in Table I. We transform into SRID 25832 (European Terrestrial Reference System 1989), which is the CRS advised by the Danish Maritime Authority. We use the CSV file from September 29th, 2020. The file size is 1.9GB. It contains 8M AIS points, in 1810 ship tracks.

T	Timestamp	MMSI	ID of the vessel
x, y	projected coordinates	heading	Angle between 0 and 359
sizea	GPS to bow	sizeb	GPS to stern
sizec	GPS to starboard side	sized	GPS to port side

TABLE I: AIS data columns

Assuming that this data is loaded in a table called `AISInput`, we can then create the rigid temporal geometries using the following SQL query.

[5]www.dma.dk/SikkerhedTilSoes/Sejladsinformation/AIS/Sider/default.aspx

```
CREATE TABLE Ships(MMSI, Trip) AS
SELECT MMSI, tgeometryseq(array_agg(tgeometryinst(
      ST_Rotate(ST_Envelope(ST_MakeLine(
         ST_MakePoint(x - sized, y + sizea),
         ST_MakePoint(x + sizec, y - sizeb)
      )), - radians(heading), ST_MakePoint(x, y)),
      T) ORDER BY T))
FROM AISInput GROUP BY MMSI;
```

The red section in the query corresponds to the creation of the geometry that represents the vessel. We represent the vessels using rectangles, and we size and rotate these rectangles according to the given data. Fig. 9 illustrates the construction of the rectangle for a single record.

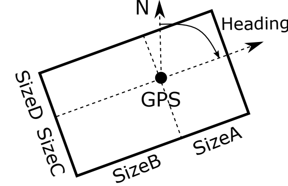


Fig. 9: Construction of the geometry of a vessel.

After adjusting the input geometries, single instants are constructed using the `tgeometryinst` function. All instants are then aggregated into an array and passed to the `tgeometryseq` constructor which is equivalent to *SEQUENCE*(geometry(Polygon)). This function will thus construct the rigid temporal geometry object by iteratively calling the encoder function for all instants.

We compare this construction with the construction of the temporal point representation of the AIS ships, as a baseline, and a second comparison with an equivalent construction of fixed-shape moving regions in SECONDO. While SECONDO contains a fixed-shape moving region algebra, not all operations are implemented, and specifically, the constructor only allows for two input instants instead of a sequence of instants. For this comparison, we thus apply the constructor on every pair of subsequent instants, which is the closest approximation of the MobilityDB `tgeometryseq` constructor.

The construction query in MobilityDB completes in 51 seconds for temporal points and 2 minutes 37 seconds for temporal geometries, compared to the 45 minutes needed in SECONDO. This construction operation creates a total of 1810 trips, starting from 8M AIS records. The constructed temporal geometry data is also about 1.2 times the temporal points. Compared to SECONDO there is a 50% gain in storage space.

Although this comparison is done on a small data set, the results are conclusive enough to show the improved performance of our implementation compared to SECONDO. Section 5 presents a generator for temporal geometry data based on the BerlinMOD generator and displays similar performance comparisons between temporal points and geometries. This generator also uses the 3D temporal geometry implementation.

A. Data Retrieval and Visualization

With the table `Ships(MMSI, Trip)` containing the temporal geometries, we can now use the implemented operations

to analyze these temporal geometries. In the following query, we compute the starting geometry of the vessels that were in the region (, created using `ST_MakeEnvelope`,) between the ports of Rødby and Puttgarden when they first started emitting data, as well as their nearest approach to one end of the breakwater of the port of Rødby.

```
SELECT MMSI,
   valueAtTimestamp(Trip, startTimestamp(Trip)),
   nearestApproachDistance(Trip, geometry 'SRID
       =25832;POINT(651337.45 6058377.98)')
FROM Ships
WHERE ST_Contains(ST_MakeEnvelope(640730, 6058230,
     654100, 6042487, 25832), valueAtTimestamp(Trip,
     startTimestamp(Trip)));
```

The result type of `valueAtTimestamp` is a PostGIS geometry and we can thus visualize these results in QGIS. The area close to the port of Rødby is shown in Fig. 10a, with the vessels returned by the query. Fig. 10b is a zoomed-in version of the previous figure and displays the rectangular geometry of one of the ships on the top-right of Fig. 10a.

The query also returns the nearest approach distance to one end of the port of Rødby. The results show that the ship with MMSI = 219000429 came closest with a distance of 20.3 meters. The equivalent query on temporal points gives a distance of 40.5 meters instead, which shows the importance of using the ship geometry for this operation.

B. Indexing on Temporal Geometries

We extend the GiST and SP-GiST indexes of MobiltiyDB, implementing respectively R-tree and Oct-tree on temporal geometries. These indexes store the bounding boxes of the temporal type and can be used to accelerate queries containing operators such as `overlaps (&&)`, `contained by (<@)`, etc.

The following two queries create a GiST index on the `Trip` column and lists all the vessels whose trip started after 12 pm and ended before 4 pm. We can see in the query plan of the second query that the index is indeed used.

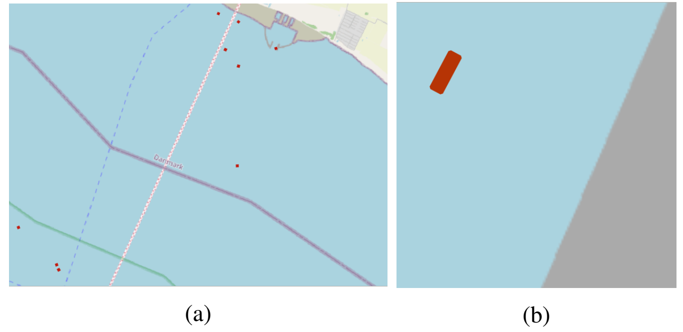
```
CREATE INDEX ShipsIndex ON Ships USING GiST(Trip);

EXPLAIN SELECT MMSI FROM Ships WHERE Trip <@ period
     '[2020-09-29 12:00:00, 2020-09-29 16:00:00)';
---------------------------------------------------
Bitmap Heap Scan on ships
   Recheck Cond: (trip <@ 'STBOX T((,,2020-09-29
   12:00:00+02),(,,2020-09-29 16:00:00+02))'::stbox)
   -> Bitmap Index Scan on shipsindex
         Index Cond: (trip <@ 'STBOX T((,,2020-09-29
         12:00:00),(,,2020-09-29 16:00:00))'::stbox)
```

VIII. A Data Generator for Rigid Temporal Geometries

To the end of facilitating the research and development work with Rigid Temporal Geometries, this section proposes a data generator. Such a synthetic data generator alleviates the effort for collecting real data, and provides the possibility to scale the generated data size as per the experiments needs.

We base this data generator on the BerlinMOD benchmark data generator [17]. BerlinMOD simulates car movements



(a) (b)

Fig. 10: Visualization of ship geometries retrieved from temporal geometries using *valueAtTimestamp*.

in a city capturing normal life scenarios: drive to work, drive to home, and afternoon and weekend leisure trips. The simulation can be controlled by a scale factor parameter, which decides the number of simulated cars, and days. In its original proposal, BerlinMOD generates temporal point objects representing car trajectories. It has two available implementations: one in SECONDO[6], and a second one in MobilityDB[7].

We extended BerlinMOD to output 2D or 3D rigid temporal geometries, where every car is represented using a randomly selected 2D/3D geometry from a given list of shapes.

The generator is implemented in PL/pgSQL (Procedural Language/PostgreSQL). Constructing the data is done by calling the SQL function `berlinmod_generate(dim, scale factor)`. The first argument decides the dimension of the vehicle shape. (0 = point, 2 = polygon and 3 = polyhedron). Setting dim = 0 corresponds to the original BerlinMOD. The scale factor argument varies the size of the generated data. A scale factor of 1.0 simulates 2000 vehicles for 28 days, which corresponds to a total of 157635 trips having on average 593 instants per trip after normalization. The generator stores the simulated trips in the table `Trips(vehicle, day, seq, source, target, trip)`, where the `trip` column is the temporal geometry. The following two queries illustrate a sample to experiment with the generated data.

```
Query 1 -- geometry at a random timestamp:
SELECT valueAtTimestamp(Trip, startTimestamp(Trip) +
 random()*(endTimestamp(Trip)-startTimestamp(Trip)))
FROM Trips;

Query 2 -- trajectory of the centre of rotation:
SELECT trajectory(Trip) FROM Trips;
```

	Point	2D Geometry	3D Geometry
Generation time (minutes)	33	83	360
Data size (MB)	2461	2873	3621
Index Size (MB)	28	28	25
Query 1 (s)	31.4	26.8	36.1
Query 2 (s)	N/A	56.9	58.8

TABLE II: Data statistics for scale factor 1.0.

Table II shows the duration of the generation for points as well as 2D and 3D geometry types, together with the

[6]http://dna.fernuni-hagen.de/secondo/BerlinMOD/BerlinMOD.html
[7]https://docs.mobilitydb.com/MobilityDB-BerlinMOD/master/

size of the `trip` column and the size of the R-Tree index constructed on this same column. All these values correspond to a scale factor of 1.0. These results show that there a difference in the generation time, which is expected due to the increasing complexity of computing the transformation vectors and normalization. The contrast in data size is not high, thanks to the delta encoding. The index size is almost the same for all types, because the index stores bounding boxes. For the two queries, the runtimes of the different types are also similar, because the processing is done on the transformation vectors, rather than on the geometries themselves.

IX. Comments On The Standard

During this implementation of the ISO 19141 standard, both strong and weak points of the standard were exposed. This section presents our implementer insights.

First of all, the structure of the standard is well-done and it presents all the necessary types in an understandable hierarchy. The scope of the standard is also clearly defined in the introduction. The inheritance hierarchy, however, results in two inconsistencies. Firstly, the *intersection* operation is also present for rigid temporal geometries, which contradicts the decision of keeping deforming regions out of scope of the standard, as is discussed in Section VI-E.

Secondly, the MF_RigidTemporalGeometry type inherits from MF_PrismGeometry and forces the base geometry to be of fixed-shape, which is correct. However, the MF_RigidTemporalGeometry also inherits from MF_TemporalTrajectory. Since this type represents the movement of a single point, and an object of this type is already present in MF_PrismGeometry to represent the movement of the centre of rotation of the geometry, we believe that this second inheritance should not be present. Indeed, multiple operations on temporal point trajectories are not possible or do not make sense when talking about rigid temporal geometries.

In its current form, the ISO 19141 schema does not allow for temporal gaps within the object representation. Temporal gaps represent durations where we have no information about the moving object (e.g., missing sensor readings), or when we selectively want to remove certain durations. An example for the latter case is when we want to only keep the parts of the trip, where the speed is greater than some threshold. Because the standard assumes that an MF_TemporalGeometry object is a continuous mapping from time to geometry, gaps cannot be represented. One possible solution is to change it into an array of such mappings, so that every inner array represents a continuous piece of the movement, and between two consecutive arrays there is a temporal gap.

X. Conclusion

This paper presented a data model and an implementation of the ISO 19141 rigid temporal geometries, in the open-source moving object database MobilityDB. Delta encoding has been used to compact the representation into a list of transformations in the form of translation and rotation parameters.

A baseline of algorithms was proposed, that are necessary to manipulate this new data type. We further implemented the operations in the ISO 19141 standard.

The experimental validation showed that the data representation size and the query performance are comparable to temporal points. The integration with PostGIS and MobilityDB also yields the advantage of exploiting their ecosystems and functions, such as QGIS visualization, and GiST indexes.

We also implemented a data generator for rigid temporal geometries in 2D and 3D, based on the BerlinMOD generator for temporal points. It is available as open source in PL/pgSQL. Thus its scenario can be further tailored to the R&D needs.

We look forward in the future to determining interesting operations on rigid temporal geometries and implementing them into MobilityDB, and to enrich the user API similar to what is currently possible with temporal points.

References

[1] ISO/TC 211 Geographic Information/Geomatics, "Geographic information – Schema for moving features," International Organization for Standardization, Standard, 2008.

[2] F. Goerlandt and P. Kujala, "Traffic simulation based ship collision probability modeling," *Reliability Engineering System Safety*, vol. 96, pp. 91–107, 2011.

[3] M. Menze and A. Geiger, "Object scene flow for autonomous vehicles," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015, pp. 3061–3070.

[4] E. Zimányi, M. Sakr, and A. Lesuisse, "MobilityDB: A Mobility Database based on PostgreSQL and PostGIS," *ACM Transactions on Database Systems*, vol. 45, p. 42, 2020.

[5] (1996-2020) PostgreSQL: The World's Most Advanced Open Source Relational Database. [Online]. Available: https://www.postgresql.org/

[6] (2001-2020) PostGIS – Spatial and Geographic objects for PostgreSQL. [Online]. Available: https://postgis.net/

[7] R. Güting, T. Behr, and C. Düntgen, "SECONDO: A Platform for Moving Objects Database Research and for Publishing and Integrating Research Implementations," *IEEE Data Eng. Bull.*, vol. 33, pp. 56–63, 2010.

[8] X. Ding, L. Chen, Y. Gao, C. S. Jensen, and H. Bao, "UlTraMan: A Unified Platform for Big Trajectory Data Management and Analytics," *Proc. VLDB Endow.*, vol. 11, pp. 787–799, 2018.

[9] M. Bakli, M. Sakr, and T. H. A. Soliman, "HadoopTrajectory: a Hadoop spatiotemporal data processing extension," *Journal of Geographical Systems*, vol. 21, pp. 211–235, 2019.

[10] R. Li, H. He, R. Wang, S. Ruan, Y. Sui, J. Bao, and Y. Zheng, "TrajMesa: A Distributed NoSQL Storage Engine for Big Trajectory Data," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, 2020, pp. 2002–2005.

[11] L. Forlizzi, R. H. Güting, E. Nardelli, and M. Schneider, "A Data Model and Data Structures for Moving Objects Databases," in *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, 2000, pp. 319–330.

[12] J. A. Cotelo Lema, L. Forlizzi, R. H. Güting, E. Nardelli, and M. Schneider, "Algorithms for Moving Objects Databases," *The Computer Journal*, vol. 46, pp. 680–712, 2003.

[13] F. Heinz and R. H. Güting, "A polyhedra-based model for moving regions in databases," *International Journal of Geographical Information Science*, vol. 34, pp. 41–73, 2019.

[14] F. Heinz and R. Güting, "A data model for moving regions of fixed shape in databases," *International Journal of Geographical Information Science*, vol. 32, pp. 1737–1769, 2018.

[15] E. B. Dam, M. Koch, and M. Lillholm, "Quaternions, Interpolation and Animation," Department of Computer Science, University of Copenhagen, Tech. Rep., 1998.

[16] ISO/TC 211 Geographic Information/Geomatics, "Geographic information – Temporal schema," International Organization for Standardization, Standard, 2002.

[17] C. Düntgen, T. Behr, and R. Güting, "BerlinMOD: A benchmark for moving object databases," *VLDB J.*, vol. 18, pp. 1335–1368, 2009.