ECOLE
POLYTECHNIQUE
DE BRUXELLES

Managing Rigid Temporal Geometries in Moving Object Databases

Thesis presented by Maxime SCHOEMANS
in fulfilment of the requirements of the PhD Degree in Engineering and technology (”Doctorat en Sciences de l’ingénieur et technologie”)
Année académique 2024-2025

Supervisor : Prof. Mahmoud SAKR
Co-supervisor : Prof. Esteban ZIMANYI
Computer and Decision Engineering Department

Thesis jury :

Prénom NOM (Université libre de Bruxelles, Chair)
Name SURNAME ([University], Secretary)
Name SURNAME ([University])
Name SURNAME ([University])
Name SURNAME ([University])

Preface

This work concludes the last five years of my life. Everything began with a meeting with Prof. Esteban Zimányi to discuss a potential master's thesis topic. This meeting, along with an infamous pandemic you might have heard about, led to a first research year at ULB, during which I continued working on this topic and published my first research paper. With the unwavering support of both Mahmoud Sakr and Esteban Zimányi, this initial work evolved into a full-fledged PhD. Over these years, I've had the privilege of collaborating with talented researchers, attending insightful conferences, and growing both as a researcher and as an individual.

First, I want to thank Prof. Esteban Zimányi for introducing me to the world of database research and for the wonderful collaboration throughout my PhD. Whether during technical meetings, code discussion, or even late-night email exchanges, his energy, encouragement, and relentless support have consistently pushed me to produce my best work.

I would also like to express my gratitude to Prof. Mahmoud Sakr, who has been an incredible supervisor throughout these years. His guidance, whether technical or personal, has been invaluable, and his support, especially during deadline pressures, was essential to my progress and well-being.

A major highlight of this journey was the third year of my PhD, when I traveled to Purdue University for a six-month research visit. I am deeply grateful to Prof. Walid Aref for inviting me to his lab, for a wonderful collaboration, and for his wisdom and guidance throughout my stay.

I am especially thankful to my family and friends, who have supported me during these years and graciously endured the ups and downs along the way.

Lastly, I want to acknowledge the support of the Fonds de la Recherche Scientifique – FNRS under grants No 40005226 and 40018136.

Maxime Schoemans
Brussels, Belgium
November 2024

Abstract

Temporal geometries, also called moving objects, are objects whose geometrical representation evolves over time. 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.

The objective of the thesis is to investigate the problem of modeling and querying rigid temporal geometries in moving object databases, with the ultimate goal to achieve an implementation of respective data types, functions, operators and indexing capabilities in the MobilityDB open-source database. Additionally, we present a tool to visualize temporal geometries stored in a database. Specifically, the four main contributions of this thesis are as follows:

- We propose a new data model for rigid temporal geometries, alongside efficient algorithms for the operations defined in ISO 19141. This model is implemented within MobilityDB, a moving object database extension for PostgreSQL, enabling robust and efficient handling of temporal geometries. Additionally, we review the ISO 19141 standard from an implementation perspective, providing insights for potential improvements.
- An efficient algorithm is developed for computing the temporal distance between moving bodies and other static or moving geometries. This approach extends the V-Clip and Lin-Canny closest features algorithms from computational geometry, allowing for accurate tracking of the temporal evolution of closest feature pairs in moving objects.

- We introduce MGiST and MSP-GiST, multi-entry generalized search trees based on the GiST and SP-GiST structures. These new search trees are designed to partition objects into multiple entries upon insertion, enhancing search performance. Evaluated in a trajectory indexing scenario, these techniques yield up to an order-of-magnitude improvement in point, range, and nearest-neighbor query performance.
- MOVE (Moving Objects Visual Exploration), an open-source visualization tool, is presented as an integration of MobilityDB with QGIS. This tool allows users to query and visualize moving object data through a straightforward interface, supporting both static and animated visualizations within QGIS and enhancing the accessibility of spatial-temporal data.

Each section of this thesis is implemented and published as open-source. The data types and operations, including the temporal distance function, are implemented in MobilityDB. MGiST and MSP-GiST are implemented as new access methods for PostgreSQL and packaged into a stand-alone extension called `mest` (Multi-Entry Search Trees). Instantiations of these access methods for trajectory indexing are implemented in a second extension, called `mobilitydb_mest`. Lastly, MOVE is implemented as a Python QGIS extension and published on the QGIS Python Plugins Repository.

Contents

List of Figures

List of Tables

Acronyms

ADT abstract data type 19

DBMS database management system 17

GIS geographical information system 18

GiST generalized search tree 100

MGiST multi-entry GiST 98

MOD moving object database 1

MOVE Moving Objects Visual Exploration 119

MSP-GiST multi-entry SP-GiST 98

TIN triangulated irregular network 9

SP-GiST space-partioned generalized search tree 101

Chapter 1
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 with tasks of the data science cycle including acquisition, storage, cleaning, transformation, analysis, visualization, privacy preservation, etc.

Moving object databases (MODs) are database systems capable of storing and querying large amounts of moving object data of different nature. Moving reals can be used to represent the evolution of a real value, such as the temperature of a room; moving points store the location of cars, planes and more; moving regions are used to keep track of forest fires, glacier extents and more. A lot of research is being done on moving points since this data is abundant and there is a need to analyze it efficiently. Due to the complexity and the limited use of moving regions, this area is less developed.

Moving regions can be categorized into two subsets: deforming and non-deforming. Deforming moving regions are geometries whose shape evolves freely over time. Such a representation is particularly important when working with natural boundaries, such as forest fires, oil spills or melting glaciers. Previous research analyzed the construction of a deforming moving regions from snapshots [38, 83], or developed novel data models to represent these deforming moving regions efficiently [25, 40].

Non-deforming moving regions, also called rigid temporal geometries, are geometries that translate and rotate over time, but maintain a fixed shape throughout the movement. 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 [29]. 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 [54]. Similar requirements arise in robot motion management in factories and logistics facilities.

The sole previous work related to fixed-shape moving regions [39] presents a new data model for storing fixed-shape moving regions in a database and implements it in SECONDO [34], an extensible database developed at the FernUniversität in Hagen. However, SECONDO is primarily a research database, and is thus not a reasonable solution for users in the industry.

This thesis aims at extending the capabilities of moving object databases by presenting a new data model for rigid temporal geometries and implementing it into an industry-ready moving object database system: MobilityDB. MobilityDB [91] is an open-source moving object database that extends PostgreSQL [66] and PostGIS [65] for temporal and spatio-temporal data management. Specifically, it adds new types to store and manipulate moving points as well as moving booleans, integers, reals and strings. In this thesis, we implement a new data type `tgeompoint` to store moving regions of fixed shape, together with a large set of functions and operators to manipulate and analyze this data type.

ISO 19141 – Schema for moving features, is a standard from the International Organization for Standardization ISO®, which defines an abstract schema for moving features, such as rigid temporal geometry, together with the needed spatio-temporal characteristics of each type. The proposed implementation in MobilityDB is conformant with the abstract schema of ISO 1914 and also implements the functions descibed in the standard.

As a relational database system, MobilityDB needs to be able to cope with large amounts of data and answer queries in reasonable time. To achieve this, MobilityDB indexes its temporal and spatio-temporal types using the PostgreSQL GiST [41] and SP-GiST [4] frameworks. In this thesis, we similarly implement indices for rigid temporal geometries using GiST and SP-GiST. However, these existing frameworks are not ideal solutions for indexing moving point and moving region data, as the produced indices are not able to closely approximate the movement of the indexed data. This leads to large amounts of false positives during index scans, which in turn deteriorates the query performance. Thus, we present new indexing frameworks: MGiST and MSP-GiST, which are able to more closely approximate moving objects and provide significant improvement in spatio-temporal query performance.

Due to their size and complexity, making sense of moving object data is difficult. One crucial requirement when working with such data is to be able to visualize it using either static or animated visualizations. This can help in exploring the data, understanding it better and communicating insights efficiently. In the last part of this thesis, we present a solution to connect QGIS, a well-known open-source geospatial visualization tool, to MobilityDB. This tool allows users to interactively create static or animated visualizations of moving object data through a simple interface.

1.1 Contributions

This thesis provides four main contributions, listed below:

- We propose a new data model for rigid temporal geometries, alongside efficient algorithms for the operations defined in ISO 19141. This model is implemented within MobilityDB, a moving object database extension for PostgreSQL, enabling robust and efficient handling of temporal geometries. Additionally, we review the ISO 19141 standard from an implementation perspective, providing insights for potential improvements.
- An efficient algorithm is developed for computing the temporal distance between moving bodies and other static or moving geometries. This approach extends the V-Clip and Lin-Canny closest features algorithms from computational geometry, allowing for accurate tracking of the temporal evolution of closest feature pairs in moving objects.
- We introduce MGiST and MSP-GiST, multi-entry generalized search trees based on the GiST and SP-GiST structures. These new search trees are designed to partition objects into multiple entries upon insertion, enhancing search performance. Evaluated in a trajectory indexing scenario, these techniques yield up to an order-of-magnitude improvement in point, range, and nearest-neighbor query performance.
- MOVE (Moving Objects Visual Exploration), an open-source visualization tool, is presented as an integration of MobilityDB with QGIS. This tool allows users to query and visualize moving object data through a straightforward interface, supporting both static and animated visualizations within QGIS and enhancing the accessibility of spatial-temporal data.

These contributions are detailed in the following publications, which collectively form the foundation of this thesis:

- M. Schoemans, M. Sakr, and E. Zimányi. Implementing rigid temporal geometries in moving object databases. In *Proc. of the 37th IEEE Int. Conf. on Data Engineering, ICDE*, pages 2547–2558, 2021
- M. Schoemans, M. Sakr, and E. Zimányi. On computing the time-varying distance between moving bodies. *ACM Transactions on Spatial Algorithms and Systems*, 9:1–28, 2023
- M. Schoemans, W. G. Aref, E. Zimányi, and M. Sakr. Multi-entry generalized search trees for indexing trajectories. In *Proc. of the 32nd ACM Int. Conf. on Advances in Geographic Information Systems, SIGSPATIAL*, pages 1–5, 2024
- M. Schoemans, M. Sakr, and E. Zimányi. MOVE: interactive visual exploration of moving objects. In *Proc. of the Workshops of the 25th EDBT/ICDT Joint Conference*, pages 1–5, 2022

1.2 Structure of the Thesis

This thesis is organized into eight chapters, grouped into three main parts.

The first part, consisting of the initial three chapters, introduces the thesis. Chapter 1 outlines the thesis objectives and summarizes its contributions. Chapter 2 reviews prior research relevant to each of the thesis's contribution chapters. Finally, Chapter 3 provides an overview of the key systems and frameworks employed, and introduces the datasets used in the experimental evaluations of subsequent chapters.

The second part, which presents the thesis's primary contributions, spans four chapters. Chapter 4 introduces a new data model for rigid temporal geometries, along with associated functions and operators. Chapter 5 details the implementation of a temporal distance operator. While these first two chapters cover all the database functionality related to rigid temporal geometries, further optimization methods are essential for efficient querying. Chapter 6 describes innovative indexing techniques for temporal points and geometries, enhancing point, range, and nearest-neighbor query performance by up to an order of magnitude. Finally, Chapter 7 introduces a new QGIS plugin that facilitates the creation of static and animated visualizations of temporal geometries stored in a MobilityDB database.

The thesis concludes with Chapter 8, which summarizes the key contributions and suggests directions for future work.

1.3 Code Availability

Each chapter of this thesis is implemented and published in an open-source repository. The data types and operations, including the temporal distance functions, are implemented in the main *MobilityDB* repository. MGiST and MSP-GiST are implemented as new access methods for PostgreSQL and packaged into a stand-alone extension called `mest` (Multi-Entry Search Trees). Instantiations of these access methods for trajectory indexing are implemeted in a second extension, called `mobilitydb_mest`. Both extension are implemented in the *mest* repository. Lastly, MOVE is implemented as a Python QGIS extension in the *move* repository and is also published on the QGIS Python Plugins Repository.

Table 1.1: List of code repositories

Repository URL	**Related Chapter(s)**
`https://github.com/MobilityDB/MobilityDB`	4, 5
`https://github.com/MobilityDB/mest`	6
`https://github.com/MobilityDB/move`	7

Chapter 2
Related Work

This thesis covers multiple aspects of moving object data management, ranging from a new data model for temporal geometries, to computational geometry algorithms for temporal distance computations, new indexing techniques, and visualization systems. In this chapter we will cover the existing work related to each part of the thesis.

The research in moving object data management has been active since early 2000, covering all aspects of modeling, indexing, query operations, analysis, visualization, data uncertainty, etc. Multiple research prototypes exist, including SECONDO [34], UlTraMan [20] on Spark, HadoopTrajectory [6], and TrajMesa [49]. There is also MobilityDB [91], 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 [49], 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.

2.1 Deforming Moving Regions

Deforming moving regions form a subset of the moving regions, and can be used in a large set of real-world application, such as the monitoring of forest fires, oil leaks in the ocean, flocks of birds, and more. Depending on the application, these regions change in discrete steps, or continuously over time.

When the regions change in discrete time steps, the major issue is the storage space, but there are no real other challenges, since the region is well-

defined at every instant (using stepwise interpolation), and the usual spatial functions can be easily generalized to moving regions.

When the regions are changing continuously, however, this stepwise interpolation method cannot be used anymore and a new interpolation method has to be defined. Next to this interpolation method, a storage model that allows us to efficiently compute these interpolations has to be described as well. Since the start and end regions might look very different, an additional difficulty arises when constructing the moving region from a set of snapshots. Indeed, the start and end regions can have a different number of vertices, and even a different number of faces if the region splits or merges.

The case of deforming regions has been investigated in [15, 25], which presents a model using a sliced representation. There exist also ways of creating this sliced representation starting from snapshots of the moving region [83].

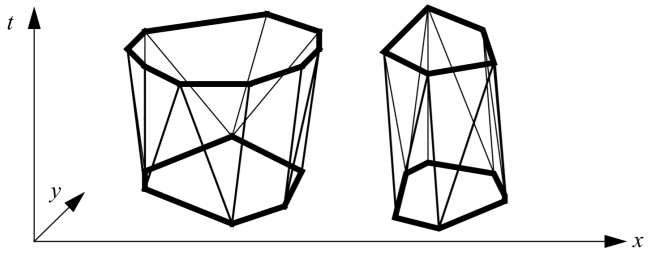


Fig. 2.1: Sliced representation of moving regions [40]

This sliced representation of moving regions makes use of *moving segments*, which are pairs of *moving points* that are co-planar in 3D space. This definition does not allow segments to rotate, but that can be avoided by representing a rotating segment using two moving segments.

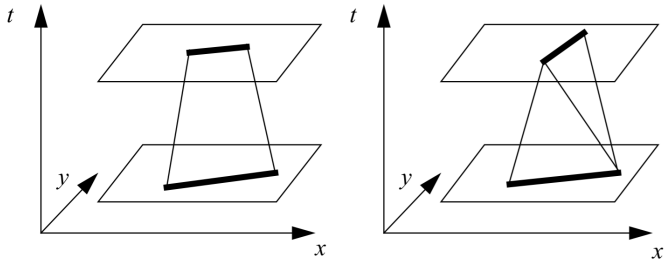


Fig. 2.2: Sliced representation of moving segments, with rotating segments being represented using two distinct moving segments [40]

To describe the complete movement of a moving region or apply operations on these regions, many of these slices are needed, and this causes storage and efficiency issues. To solve this problem, a polyhedra-based model is described in [40], which uses the same moving segments as previously, but computes the operations more efficiently by using the temporal dimension as a third dimension and using 3D geometry to compute the operations. As a result, fewer slices are created than in the previous implementations.

2.2 Fixed-Shape Moving Regions

Fixed-shape moving regions form another subset of all moving regions and can be used to represent any moving rigid body, such as cars, boats, planes, etc. When the spatial extent of the object is negligible, and the orientation of no importance, the movement of such a body is usually represented by a moving point. When these conditions are not met, however, the movement of the region as a whole has to be analyzed.

The movement of fixed-shape regions can be described much easier than the movement of a deforming region since the only possible transformations are translations and rotations. Theoretically, scaling could be taken into account too, but this case does not seem to have any real-world use case, so it is omitted.

Again, if the region moves in discrete time steps, the technical challenge only concerns the storage space, and this can be efficiently resolved by storing the moving region as a polygon followed by a list of transformations. This technique is also used when handling the continuous movement of regions and is described below.

When managing continuous moving regions, it is important to be able to compute the interpolation efficiently. The previous model for deforming regions relies on moving segments, which are assumed to be non-rotating. This assumption is contradictory to the idea of rotating (and translating) regions, and the larger the rotation of the region is, the worse the interpolation becomes. A good example of this is when a thin rectangle rotates 90 degrees without translating. The interpolation will considerably deform the region, which will at some point become a square, as can be seen in Fig. 2.3.

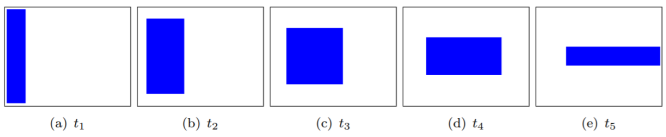
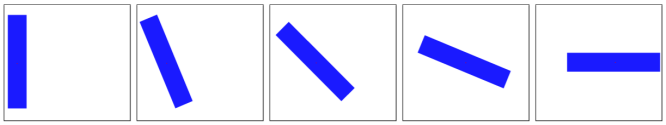


Fig. 2.3: Interpolation of a moving region using linear interpolation between the vertices [39]

The reason for this is that the interpolation method does not use the assumptions made on the possible transformations of the polygon. To solve this, a new model is proposed in [39], which stores the region as a sequence of transformations, and makes use of these transformations to compute the correct interpolation. The resulting fixed-shape moving region is called `fmRegion`, and an example of a desired interpolation for this type of region is seen in Fig. 2.4.



(a) t_1 (b) t_2 (c) t_3 (d) t_4 (e) t_5

Fig. 2.4: Interpolation of a moving region using the fixed-shape assumptions [39]

The model makes use of a `transformationUnit`, which specifies a rotation center C, an initial translation vector v_0 and rotation angle θ_0, the vector v and angle θ representing the transformation, as well as a time interval $[t_s; t_e]$, specifying the start and end instant of the transformation.

$$\texttt{transformationUnit}\ \mathcal{T} : (C, v_0, v, \theta_0, \theta, t_s, t_e)$$

An `fmRegion` is thus defined by a single classical region $\mathcal{R}$ and a set of at least one transformation unit $\mathcal{T}$.

$$\texttt{fmRegion} : (\mathcal{R}, \mathcal{T}^+)$$

When computing the position of the region at a moment between the start and end instant of a transformation unit, the translation vector and rotation angle are linearly interpolated, and then the transformation is applied to the initial region to retrieve the position of the region at the correct moment.

Note that the previous definition allows the region to rotate around an arbitrary rotation center for each transformation unit. This particularity allows the moving region to change rotation center between two transformation units, which is useful to describe a larger amount of trajectories than if the rotation center was fixed for all transformations. On the other hand, this also requires the user to input the rotation center for each transformation, since this cannot be uniquely computed by just seeing the start and end position of the region.

The same paper [39] also describes a few useful algorithms to process moving regions, such as a traversed area function, which computes the union of all places that the region went through during its movement.

2.3 3D Moving Regions

The research articles presented in the two previous sections only focused on moving regions in two dimensions, but of course we might wonder if 3D moving regions could also be manipulated similarly. Only 3D moving regions of fixed shape will be discussed here.

First of all, it is important to define what we call a 3D region. There is indeed a considerable difference between 3D polygons (polygons with points embedded in 3D space) and 3D polyhedra (3D volumetric objects). In the following, we define a 3D region as a volumetric geometry, thus representable by a 3D polyhedron.

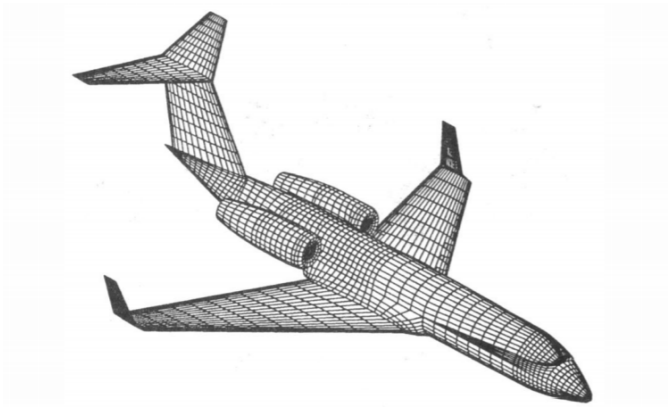


Fig. 2.5: Representation of a plane using a 3D polyhedron [26]

Support for 3D geometries is currently extremely restricted in database management systems [92]. However, PostGIS does support two types of 3D geometries: *polyhedral surfaces* and *triangulated irregular network* (TIN) surfaces, which are polyhedral surfaces where every polygon is a triangle. A few 3D functions are also implemented, such as *ST_3DClosestPoint* or *ST_3DDistance*.

In the case of discrete movement, the problem of interpolating between two instants does not arise, and the difficulty is reduced to storing the large amount of data that might be present in an efficient manner. Assuming that the regions are fixed-shape, it is again sufficient to store the region once as a 3D polyhedron, and store only the difference in position and orientation for all other positions. This technique is similar to the one described in the previous section, except that the translation and rotation are done in 3D space instead of 2D.

Looking at continuously moving regions, the same storage idea can be applied. Retrieving the position of the region between two stored instants is

then done by computing a linear interpolation of the translation, just as was done in 2D, and by computing a more advanced interpolation for the 3D rotation, based on *quaternions*. The use of quaternions to represent orientations and rotations in three dimensions has been heavily researched [17]. This technique is already being used, for example to create smooth animations and movements in computer graphics. We will further discuss the use of quaternions for manipulating 3D rigid temporal geometries in Chap. 4.

2.4 Related 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[1]) and the Open Geospatial Consortium (OGC 2007[2]). A large number of standards (more than 80) have 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 baselines of these standards are the ones that contribute at various levels to the modeling of geographic information:

- ISO/TS 19103 Conceptual Schema Language, defining the notation for the conceptual modeling of geographic information. It specializes UML, restricting features like multiple inheritance, and adding geographic modeling 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.

[1] `https://committee.iso.org/home/tc211`

[2] `https://www.ogc.org/`

- 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 is based on the ISO 19107 standard 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 its infancy:

- ISO 19141 Schema for moving features is the abstract (metamodel) for data representation.
- OGC 16-120r3 Moving Features Access 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, 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.

2.5 Distance Algorithms

The moving object database research defines three main types of moving objects: moving points, deforming moving regions and non-deforming moving regions. Moving points are commonly represented using piecewise linear functions, and computing the temporal distance between them is thus a trivial task. As the distance between two linearly moving points is a square root of a quadratic function, the difficult part consists of representing and storing this function in the database. In SECONDO [15], the data model can represent this function as a temporal float object. In MobilityDB [91], an approximation is computed and stored as a piecewise linear function that maintains the extreme points of the initial distance function. One important application of such a function is solving continuous nearest neighbor queries [27, 35, 80].

A distance function for deforming moving regions is presented in [15]. It accepts a moving point and a moving region, as well as a pair of moving regions. The algorithm combines a brute-force technique, computing the distance with every segment of the regions, with a filtering technique that limits the actual number of segments to improve performance. The data model represents the deforming moving region as a set of linearly moving segments, i.e., the region edges (Sect. 2.1). This model cannot represent non-deforming moving regions that move by translating and rotating around a given rotation center. Data models have thus been proposed to represent this latter

type [39, 76] (Sect. 2.2). However, up to our knowledge, the problem of computing the temporal distance between non-deforming moving regions, i.e., moving bodies, has not been addressed before. Chapter 5 aims at developing such an algorithm.

In the field of computational geometry, there are efficient solutions for computing the distance between static 2D polygons. Using binary search and no prior information, it is possible to determine the distance between two convex static polygons in $\mathcal{O}(log(n))$ time, where n is the total number of vertices of the two polygons [13, 23, 88]. This complexity is a lower bound when no extra information is given about these two polygons. When working with non-convex polygons, the main idea is to decompose the polygon into convex parts and build a bounding hierarchy to reduce the number of operations applied to each convex part. These bounding hierarchies can be made of spheres/circles [19, 43, 68, 73] or bounding boxes [51, 62], either axis-aligned or not. [68] mentions an average complexity of $\mathcal{O}(nlog(n))$ for 2D distance computations between non-convex polygons.

Here, in the case of moving polygons, we observe that the distance is computed multiple times at close time intervals. When working with convex polygons, it is thus often correct to assume that the closest points between these two polygons will not move much between two distance computations. Lin-Canney et al. [50] describe an algorithm to compute the distance between two convex static polygons, as well as the two closest features (vertices or edges) of the two polygons that are at this minimum distance. This algorithm takes not only the two polygons as input but also an estimate of the closest features. In the worst case, the algorithm has a linear time complexity $\mathcal{O}(n)$, but can finish in constant time if the estimate of the closest features is not far from the real closest features. When iteratively computing the distance between two moving polygons, the closest feature of the previous computation is used as an estimate of the next one. This allows for significant time improvements.

Improvements to the Lin-Canney algorithm have also been made, such as the V-Clip [55] and H-Walk [32] algorithms. V-Clip is similar to Lin-Canney, but more robust and easier to implement, while H-Walk tries to improve the linear worst-time complexity of both Lin-Canney and V-Clip by representing the object as a hierarchy of convex polygons. A similar hierarchical technique is used in [64] for collision detection. An overview of existing static and iterative algorithms for collision detection and distance computation can be found in [52].

The algorithms presented in this thesis are based on ideas from V-Clip and Lin-Canney but differ from the existing algorithms by taking into account the movement parameters of the moving objects, as well as returning a temporal distance function instead of returning distance values at distinct timestamps. This setting is useful when the movement is known in advance, such as in a database setting storing historical movement data.

As detailed in Sect. 5.2, the presented algorithm starts by computing the initial closest features at the start of the movement. This is done using any one of the existing algorithms discussed above. Similarly, as discussed at the end of Sect. 5.1, the V-Clip algorithm can be used to check the validity of the result at the end of the algorithm in constant time.

2.6 Trajectory Indexing

To simplify the implementation of new indices, Hellerstein et al. [41] introduce GiST, a generalized search tree that is easily extensible for different data types and queries. GiST can instantiate balanced search trees, e.g., the B-tree, R-tree or M-tree, using minimal implementation effort through the use of pluggable modules. Behind the scenes, GiST transparently handles the complex index internals, e.g., tree balancing, concurrency, and recovery [47].

With the emergence of new database applications, new indexing solutions, e.g., k-d trees, quadtrees, tries, and other unbalanced search trees have become more prominent. However, these indices could not be implemented in GiST, as they are not balanced trees. Aref et al. [4] introduce SP-GiST, a space-partitioning generalized index, that can instantiate space-partitioning unbalanced trees in a generalized framework. Many improvements and optimizations have been proposed for both GiST and SP-GiST, e.g., [3, 7, 28] as well as their implementation in existing database systems, e.g., [8, 24]. GiST and SP-GiST are further detailed in Chap. 6.

Many specialized indices have also been developed for the indexing of spatial and spatio-temporal data. The R-tree [33] indexes spatial objects using their bounding boxes in a balanced tree structure. By storing the time as an additional dimension, spatio-temporal objects can be indexed in 3D R-trees [81] or RT-trees [87]. Another approach to index the temporal dimension is to have an R-tree for each time instant. To save space, consecutive R-trees can also share sub-trees that remain unchanged between two time instants [57, 79, 87] and there are also combinations of the above approaches [87].

A trajectory is a special type of spatio-temporal data. It combines multiple trajectory segments into a single large data object, with each segment being formed by linear interpolation of two stored instants. In the case of a primary index, each segment needs to be stored in the index, e.g., in a 3D R-tree. STR-trees and TB-trees [63] are extensions of the 3D R-tree that cluster segments of the same trajectory to improve trajectory-related queries. If the tree is used as a secondary index, not all the segments have to be stored individually. Thus, one can index fewer bounding boxes that each represent a subset of the complete trajectory. This is the scenario handled by this paper. Hadjieleftheriou et al. [37] and Rasetic et al. [70] present possible

splitting algorithms for this purpose. These algorithms are re-used in the implementation of MGiST and MSP-GiST for trajectories as in Sect. 6.3.

Another class of trajectory indices employs a 2-level approach. The first level partitions the space using a space-partitioning index, e.g., a grid or quadtree. This index will be slowly changing, as new updates commonly appear close to existing values. In a second level, each spatial cell/partition contains a temporal index that stores the temporal information of the trajectory segments covered by this spatial partition. This approach is used in [11, 16, 78, 86], with different combinations of spatial and temporal indices.

Most of the indices discussed above are mainly designed for range queries, with some being also able to answer nearest neighbor queries. Other trajectory indices have been presented to answer different types of queries. For example, the MTSB-tree [89] can answer close pair queries that search for trajectories that have been closer than a given distance during a time interval and inside a given spatial range. The TrajTree [69] index is used to match/-query similar trajectories using the Edit Distance with Projections (EDwP) measure and uses a structure similar to the R-tree. Chebyshev polynomials [9] approximate and index multi-dimensional trajectories for similarity matching. The PA-tree [59] is another index that uses Chebyshev polynomials to approximate spatio-temporal trajectories. Lastly, many indices have been proposed to index network-constrained trajectories as well as indoor trajectories. We do not detail these solutions, as they refer to a different problem. Refer to [53, 56, 58] for surveys on spatio-temporal access methods.

2.7 Moving Objects Visualization

The most comprehensive work in the visualization of mobility data is the foundation in [1]. This work provides concepts and application examples of exploratory analysis of movement data. There is, however, a lack of open-source visualization systems that allow such exploratory analysis through an interactive and powerful interface. Niche system implementations include V-Analytics [2], ST_Visions [82] and GTX [14]. The common open-source visualization tools, mainly geospatial, have little support for movement data. Users would need to integrate a complex stack of tools to obtain both rich and scalable visualizations, as assessed in [31].

In QGIS, one could use the Time-Manager plugin [30] to create animated visualizations of moving objects. It provides a user-friendly GUI for visualizing timestamped geometries (i.e., discrete point-based trajectories). Recently, it was integrated as a built-in QGIS feature, called Temporal Controller. Using client-side code, the Temporal Controller can also be used to create smoothly animated visualizations of continuous trajectories. This is, however, both complex and inefficient. Complex, because it requires users to

write client-side code in QGIS for doing interpolation. In addition, for the integration with MobilityDB, the spatiotemporal types are not understood by QGIS, and the users thus have to transform and handle these types manually. Inefficient, because this solution only scales to 10's of concurrent moving objects on the screen. The work presented in Chap. 7 leverages the capabilities of the existing Temporal Controller while hiding this complexity from the end-user and improving the scalability.

Chapter 3
Background

In this chapter, we introduce the main systems used in this thesis as well as the datasets used for the experimental sections of the following chapters. The work in this thesis has been implemented in the open-source database MobilityDB, which extends PostgreSQL and PostGIS with temporal and spatiotemporal types. Section 3.1 describes MobilityDB and its related systems. The implementation of each chapter is also tested on two main datasets: a dataset generated by the BerlinMOD synthetic data generator, and a martitime dataset consisting of real-world AIS data distributed by the Danish Maritime Authority. Section 3.2 describes these two datasets in more detail.

3.1 Systems

3.1.1 PostgreSQL

PostgreSQL [66] is an open-source object-relational database management system (DBMS) based on Postgres and was developed at the *University of California at Berkeley Computer Science Department* in 1996.

The name of the system reflects its support for most SQL standards, but the system also offers many advanced features, such as complex queries, foreign keys, triggers, multi-version concurrency control and more.

PostgreSQL is also highly extensible, which allows users to define new data types, functions, indexing methods and more, without having to modify the core database engine. This feature allows new extensions to be added, capable of handling user-defined data types while keeping the full power of a traditional DBMS.

3.1.2 PostGIS

PostGIS [65], released in 2001, is one example of an extension to the PostgreSQL DBMS. PostGIS is an open-source spatial extension to PostgreSQL that adds support for a wide range of geographic objects.

This extension allows geographical information system (GIS) objects, such as points, linestring or polygons, to be stored in the database. It also includes support for the processing and analyzing of GIS objects, by defining functions and operations that can be used to process them, such as distance and area functions.

A PostGIS type that will be used a lot in this thesis is the *polygon* type since this will be used to represent a static region. A PostGIS polygon object is defined as having an exterior ring, as well as zero or more interior rings that define holes in the polygon. The PostGIS *linestring* type is used to describe these rings. A linestring is a list of points representing a continuous line composed of multiple segments. For this linestring to be valid, it cannot intersect itself at any point (Fig. 3.1a). The only exception is that the endpoints of the chain of segments can coincide (both endpoints have the same coordinates), and in that case, the linestring is *closed* (Fig. 3.1b). Both the exterior and interior rings of a polygon are described using closed linestrings.

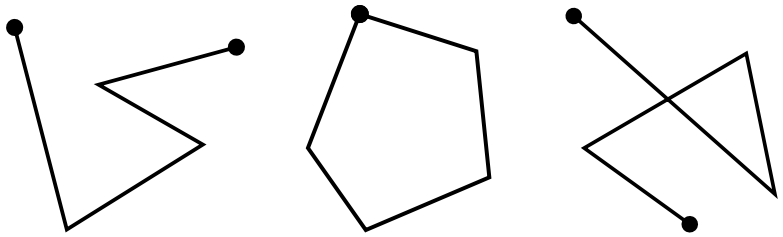
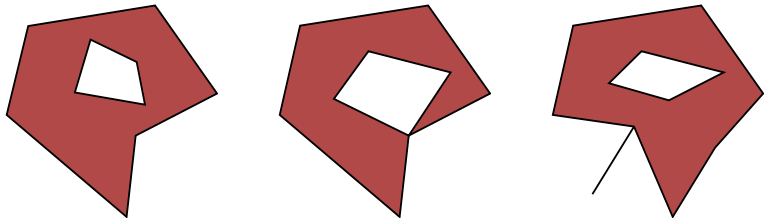


Fig. 3.1: Examples of valid (a, b) and invalid (c) PostGIS linestrings

A PostGIS polygon is valid if no two of its rings intersect (Fig. 3.2a), except in a single point on the boundary of the polygon (Fig. 3.2b). The polygon cannot have lines or spikes (Fig 3.2c), and the interior rings cannot be defined outside of the exterior ring.

Contrary to most research papers about moving regions [25, 36, 40], this definition of a region does not allow a region/polygon to have multiple faces, which is usually necessary to be able to describe regions that split or merge. PostGIS allows the creation of multi-polygons, which could theoretically be used to describe a region made of multiple faces, but since in our case we will only handle non-deforming (fixed-shape) regions, there will be no need for polygons to merge or split, so this will not be discussed further.



(a) (b) (c)

Fig. 3.2: Examples of valid (a, b) and invalid (c) PostGIS polygons

3.1.3 MobilityDB

MobilityDB [91] is an extension that implements abstract data types (ADTs) of moving objects in PostgreSQL and PostGIS. It further supports these types with indexes, a rich set of temporal and spatio-temporal 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 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 `timestamptz`, `tstzSet`, `tstzSpan` and `tstzSpanSet`, representing respectively a single time instant, a set of discrete time instants, a continuous time interval, and a disconnected set of time 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 spatio-temporal points, e.g., check-ins of a Foursquare user. Combining `geometry(Point)` and `tstzSpan` 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 continuous 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 *timestamptz* as follows:

$$D_{INSTANT(S)} = D_{metavalue(S)} \times D_{timestamptz} \tag{3.1}$$

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 the 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 thesis.

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

$$
\begin{aligned}
&D_{SEQUENCE(S)} = \{(I, li, ui, interpolation) \mid \\
&\quad \text{(i) } I = [(v_1, t_1), \ldots, (v_n, t_n)] \text{ is a list of } INSTANT(S) \\
&\quad \text{(ii) } li, ui \in \{true, false\}\text{(in/ex)clusive bounds of } \texttt{tstzSpan} \\
&\quad \text{(iii) } interpolation \in \{step, linear\} \\
&\quad \text{(iv) consecutive pieces of the interpolation function are} \\
&\quad \text{not collinear}\}
\end{aligned}
\tag{3.2}
$$

It represents an evolution of value over a time period/span, that can be left/right open/closed. This evolution is either discrete or continuous depending on the specified interpolation method. 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.

Lastly, *INSTANTS* and *SEQUENCES* construct sets of the *INSTANT* and *SEQUENCE* types respectively, with specific constraints on the allowed values. For example, two sequences in a sequence set cannot overlap in time and must have the same interpolation method.

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 costs. Then the effort of development can be re-directed to implementing data management and query functions for the new types.

Note that the *INSTANTS* and *SEQUENCE* type constructors are merged into a single *SEQUENCE* constructor (SQL function) in practice. Indeed, an instant set can be seen as a special case of a sequence, setting the interpolation method to *discrete*, both bounds to inclusive, and ignoring the collinear constraint of continuous sequences.

Bounding Box

Each temporal object also has an associated bounding box, whose type depends on the base type of the object. A bounding box is the smallest enclosing box in 1D, 2D or 4D space (depending on the type of object), that completely contains the given temporal object. The following bounding box types exist

- `TstzSpan`: for the types `tbool` and `ttext`, where only the temporal extent is taken into account.
- `Tbox` (temporal box): for the types `tint` and `tfloat`, where the value extent is stored in the X dimension, and the temporal extent in the T dimension.
- `STbox` (spatio-temporal box): for the types `tgeompoint` and `tgeogpoint`, where the spatial extent is stored in the X, Y and Z dimension, and the temporal extent in the T dimension.

These bounding boxed are used instead of the temporal object itself in some operations, in particular those related to indexing, for efficiency reasons.

Functions and Operators

MobilityDB exposes a large set of functions and operators for temporal types. These functions and operators are polymorphic, meaning they can have arguments of several types and the result type may vary depending on the input types. They can be grouped into the categories described below.

- Accessor functions:
  A multitude of accessor functions exists, that enables the retrieval of a particular instant, timestamp or value from the temporal value. The duration or memory size of the value can also be retrieved.
  Some example functions are shown below.

```
SELECT memsize(tint '[1@2001-01-01, 2@2001-01-02, 1@2001-01-03]');
-- 280

SELECT getValue(tint '1@2001-01-01');
-- 1

SELECT instantN(tint '[1@2001-01-01, 2@2001-01-02, 1@2001-01-03]', 2);
-- "2@2001-01-02"
```

- Spatial operations:
  Spatial functions can be applied to temporal points. Most of the functions support 3D points, and/or are available for geographies. These functions range from the in/output in a specific format (e.g., *EWK* or *MFJSON*), to more complicated accessor functions, such as *speed* or *nearestApproachDistance*.

```
SELECT speed(tgeompoint '[Point(1 0)@2001-01-01, Point(1 0)@2001-01-02,
    Point(1 1)@2001-01-03]')*3600*24;
-- "Interp=Step;[0@2001-01-01, 1@2001-01-02, 1@2001-01-03]"
```

- Restriction functions:
  These functions restrict a temporal value, either on its value extent or its time extent. Example of restriction functions are: *atValue*, *atMax*, *atGeometry*, etc.

  ```
  SELECT atMax(tint '[1@2001-01-01, 2@2001-01-02, 1@2001-01-03]');
  -- "{[2@2001-01-02, 2@2001-01-03]}"
  ```

- Bounding box operators:
  These are the comparison operators on the corresponding bounding boxes of temporal values. Operators exist for different kinds of relationships (for example, less than, contains, overlaps, equal, etc) and for all possible dimensions of the bounding boxes (value/X, Y, Z and T dimension). As said previously, the Y and Z dimensions are only used in case of temporal points. These functions all return a boolean value. For example, the bounding box overlap operator can be applied using the symbol `&&`.

  ```
  SELECT tint '[1@2001-01-01, 2@2001-01-02, 1@2001-01-03]' && 2::tbox;
  -- true
  ```

  The main purpose of the bounding box operators is to be used as filtering conditions in index scans to speed up operators such as textiteintersects.
- Distance operators:
  These operators compute the distance between two temporal points as a `tfloat` ('`<->`') or the smallest distance between two temporal points as a float ('`|=|`'). Since the real distance function is the square root of a quadratic function, but MobilityDB assumes linear interpolation between instants, the distance operator in MobilityDB returns a linear approximation of the real distance between its arguments.
- Spatial relationships for temporal points:
  The PostGIS spatial relationship functions (such as *ST_Intersects* and *ST_Relate*) are generalized in two different ways. In the first version, the PostGIS functions are applied to the union of all values taken by a temporal point (for example, *eintersects* and *erelate*). The second version is defined with temporal semantics and returns a temporal value. For the two example functions, the second versions are *tintersects* and *trelate*, which return a `tbool` and a `ttext` respectively.
- Aggregate functions for temporal types:
  These are the generalization of the usual aggregation functions, such as *tcount*, *tmax*, etc. The generalized form is defined with temporal semantics and returns a temporal value.

Note that this list is only there to introduce the set of functions exposed by MobilityDB, and does not consist of an exhaustive list of all available functions. For a more complete and detailed list, refer to the MobilityDB manual [90].

3.2 Datasets

This section describes the two main datasets used in this thesis: a dataset produced by the BerlinMOD synthetic data generator, and real-world AIS data openly distributed by the Danish Maritime Authority. The BerlinMOD data generator is used to construct synthetic temporal point data and is detailed in Sect. 3.2.1. We further extend its capabilities in Sect. 4.6 to construct rigid temporal geometry data in 2D and 3D. The AIS data distributed by the Danish Maritime Authority is described in Sect. 3.2.2. It is used to construct both point and 2D temporal geometry data.

3.2.1 BerlinMOD Data Generator

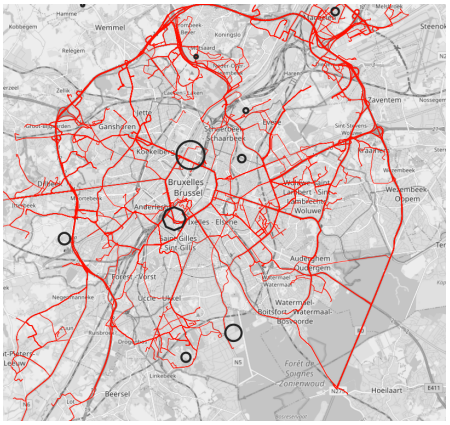


Fig. 3.3: BerlinMOD Data

BerlinMOD [22] is a benchmark for moving object databases that presents both a synthetic trajectory data generator and a set of benchmark queries. It simulates car movements 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. It has two available implementations: one in SECONDO,[1] and a second one in MobilityDB.[2] In these experiments, we use the version of the BerlinMOD data generator that is implemented in MobilityDB and thus generates the trajectories readily in MobilityDB moving

[1] `https://secondo-database.github.io/BerlinMOD/BerlinMOD.html`

[2] `https://github.com/MobilityDB/MobilityDB-BerlinMOD`

point format. The generator simulates work and leisure trips in and around the city of Brussels. Next to the trajectory data, the generator produces four other tables of interest: `QueryInstants`, `QueryPeriods`, `QueryPoints` and `QueryRegions`, which are used in the benchmark queries. Figure 3.3 displays the generated trajectories (in red) and query regions (in black). The query regions have an average size of 1 km^2. Table 3.1 summarizes the generated tables.

Table 3.1: Tables produced by the BerlinMOD generator (ScaleFactor := 1)

Name	**Columns**	**Count**
Trips	⟨vehicle_id, trip_id, trip⟩	157,549
QueryInstants	⟨id, instant⟩	100
QueryPeriods	⟨id, period⟩	100
QueryPoints	⟨id, point⟩	100
QueryRegions	⟨id, region⟩	100

The BerlinMOD benchmark describes two sets of queries: BerlinMOD/R and BerlinMOD/NN. BerlinMOD/R contains 17 range queries which test different combinations of accessors, temporal and spatial conditions, aggregation functions, and more. BerlinMOD/NN, on the other hand, describes spatio-temporal k-nearest-neighbor, reverse-nearest-neighbor, and aggregate nearest-neighbor queries. However, these spatio-temporal nearest-neighbor queries cannot be expressed using current SQL queries. In Chap. 6 we detail the range and nearest-neighbor queries used to test the newly presented multi-entry indices.

3.2.2 Maritime Dataset

The Danish Maritime Authority publishes histories of AIS ship trajectories[3] in CSV format. Each CSV file contains one full day of data (or one month for older data) and every row stores the position of a single vessel at a specific time instant. The columns of the CSV files are listed in Table 3.2. We specify in each experimental section which file (or files) has been used for that section.

When representing vessels as moving points, the columns Timestamp, MMSI, Latitude and Longitude are used to construct the trajectory of the vessels. To construct a fixed-shape region representing the geometry of the vessel, the columns Heading, Size A, B, C, and D are also used. The construction of this polygon representation of a vessel is detailed in Section 4.1.2.

[3] `https://www.dma.dk/safety-at-sea/navigational-information/ais-data`

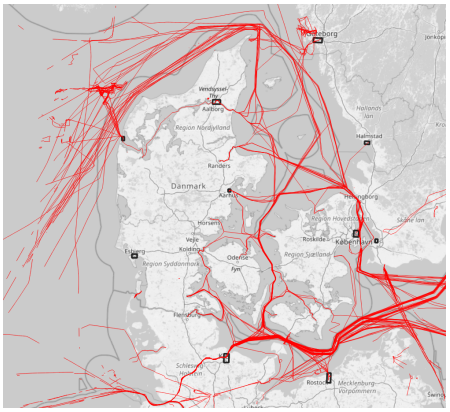


Fig. 3.4: AIS Data

Table 3.2: Columns of an AIS CSV file

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

Chapter 4
A Model for Rigid Temporal Geometries

The main objective of this thesis is to investigate the problem of modeling and querying rigid temporal geometries in moving object databases, with to goal of extending the MobilityDB database system with new types, functions, and operators to store and manipulate rigid temporal geometries. To this end, we define a new data model that can efficiently store the evolution of a rigid temporal geometry and integrate it into the MobilityDB type system. This data model is based on the `fmRegion` model discussed in Sect. 2.2 and reuses the idea of storing transformation (delta) values instead of storing the complete geometry multiple times. The main differences between the two models is that we adopt the sequence representation of MobilityDB instead of the sliced representation of SECONDO and that this model supports both 2D and 3D moving regions. We implement the new data model as a new database type called `tgeometry`. Additionally, we discuss how these transformation values (also called poses) can be used on their own without a reference geometry, whenever we are interested in storing the position and orientation of an object but not its spatial extent. This leads to the definition of a new temporal type called `tpose`, which essentially corresponds to a `tgeometry` object without specified reference geometry. An example of an object that can be represented as a `tpose` type would be a drone with a camera attached to it. Indeed, we might not be interested in the actual shape of the drone, but we might want to know the evolution of the position and orientation of the drone to be able to determine what spatial area the camera captured.

Together with the data type for rigid temporal geometries, we also implement a set of functions and operators that can be applied on this new data type. We implement functions similar to the ones already implemented for temporal points in MobilityDB, as well as functions defined in the ISO 19141 standard. However, not all functions from the standard can be implemented in the current type system. We reflect on this and discuss ways to improve the existing standard.

The strucure of this chapter is as follows. Section 4.1 presents the new data model for rigid temporal geometries. In Sect. 4.2, we describe algorithms that

are needed to internally manage the new data types. Section 3.1.3 lists the new functions and operators for rigid temporal geometries as they are defined in MobilityDB. We present the ISO 19141 standard for moving regions in Sect. 4.4 and discuss the conformance of our implementation to this standard. Lastly, Sect. 4.5 presents a use case of our implementation for maritime data and Sect. 4.6 extends the BerlinMOD generator to construct temporal geometry data.

4.1 Data Model

This section describes the data model for rigid temporal geometries and integrates it into the MobilityDB type system. Similarly to [39], our model makes use of transformation values to reduce the amount of data stored. Indeed, since the objects are non-deforming, it would be redundant to store their geometry at each timestamp. To store the evolution of a rigid temporal geometry, it is sufficient to store the evolution of their position and orientation together with a single reference geometry. The combination of a position and an orientation is also called a *pose*. Thus, a temporal geometry can be seen as a pair of a temporal pose and a reference geometry.

To integrate this data model into the MobilityDB type system, we define a new base type called `pose` and specify the metavalue of a geometry object to be a pair of a pose and a reference to the geometry.

$$D_{metavalue(S)} = \begin{cases} D_S & S = \texttt{pose} \\ D_S \times D_{pose} & S \in \{\texttt{geometry(polygon)}, \\ & \texttt{geometry(Polyhedralsurface)}\} \end{cases} \tag{4.1}$$

Accordingly, the four type constructors *INSTANT*, *INSTANTS*, *SEQUENCE*, and *SEQUENCES* (Eq. 3.1, 3.2) will extend the system with corresponding types. This produces two new temporal types: `tgeometry` and `tpose`.

Section 4.1.1 details the domain of the `pose` type in 2D and 3D and Sect. 4.1.2 explains how the reference geometry is defined in both cases.

4.1.1 Pose Type

For the PostGIS base type `geometry(Polygon)`, we handle the object as a 2D temporal geometry. In this case, the 2D pose of the object is uniquely determined by a 2D point and an angle representing the orientation of the object.

$$D_{pose} = \{(x, y, \theta) \mid \theta \in (-\pi, \pi]\}$$

A three-dimensional volumetric object is represented using the PostGIS `geometry(Polyhedralsurface)` type and the corresponding 3D pose stores both a 3D position and a 3D orientation.

$$D_{pose} = \{(x, y, z, W, X, Y, Z) \mid \sqrt{W^2 + X^2 + Y^2 + Z^2} = 1\}$$

3D orientation are stored and interpolated using unit quaternions [17], and the orientation parameters (W, X, Y, Z) thus correspond to the four parameters of a unit quaternion.

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

4.1.2 Reference Geometry

A *pose* determines the position and orientation of an object relative to a given coordinate system. This reference coordinate system could be a geographic coordinate system, a projected coordinate system, or any other chosen coordinate system. To uniquely specify the pose of an object, we need to know its position and orientation in this given coordinate system. To achieve this, we need to first specify a local coordinate system fixed to the body of the reference geometry. The stored pose then represents the transformation from our reference coordinate system to the object's local coordinate system.

The origin of the local coordinate system of an object corresponds to its center of rotation and is typically chosen to be its center of mass or centroid but can also be chose differently. For example, when constructing the reference geometry of the vessels from the maritime dataset (Sect. 3.2.2), the origin of the local coordinate system is chosen to be the position of the GPS device on the vessel.

Complementary to the position of an object, its orientation specifies the rotation needed to align the axes of the local coordinate system with the reference coordinate system. This depends on the choice of the orientation of the local coordinate system with respect to the body of the object. In 3D, there are multiple conventions for these axes based on the type of vehicle.[1] For land vehicles like cars, the x-axis, y-axis and z-axis correspond to respectively the front, left and up. For world reference frames, the conventional reference system used is the East-North-Up (ENU) coordinate system, which has the x-axis to the East, the Y-axis to the North and the z-axis pointing Up. When working with air and sea vehicles like planes or ships, the convention is slightly different. The x-axis, y-axis and z-axis correspond to respectively the front, right and down. In this case, the NED coordinate system is used, which associates North, East and Down with positive x-, y- and z-axis respectively.

[1] `https://en.wikipedia.org/wiki/Axes_conventions`

It is thus clear that the value of the pose will depend on both the choice of reference coordinate system and the convention used to for the local coordinate system of the object. As an example, Fig. 4.1 shows the construction of a vessel from the maritime dataset (Sect. 3.2.2) around its local coordinate system. Since this dataset contains data around Denmark, we use the EPSG:25832 coordinate reference system, which has its x- and y-axis pointing the the East and North respectively. We thus use the convention of placing the x- and y-axis of the local coordiante system towards the front and left of the vessel respectively. The generated geometry is a pentagon in the shape of a rectangle with an triangle-chaped front to distinguish the front from the back of the vessel.

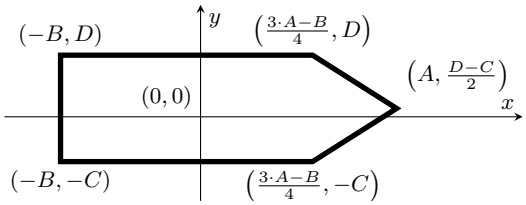


Fig. 4.1: Reference geometry of a vessel. Parameters A, B, C, and D correspond to values of the Size A, B, C, and D columns listed in Table 4.11.

In practice, we store this reference geometry as a PostGIS polygon with its coordinates defined in the geometry's local coordinate system. Two possible solutions exist for storing the geometry; either in a separate table or inside the object. In this implementation, we choose to store the reference geometry together with the rest of the object (the list of poses) to avoid an additional lookup when the geometry is needed.

4.2 Data Management Algorithms

Before listing the MobilityDB functions that can be applied on the `tgeometry` data type, we describe a set of essential internal algorithms for managind rigid temporal geometries. These algorithms manipulate poses and/or polygons and are crucial to many external functions exposed to the user.

4.2.1 Encoder and Decoder

While the internal representation of temporal geometries consists of a sequence of poses with a reference geometry, i.e., delta encoding, the data can

potentially arrives as a sequence of geometry snapshots. Therefore, we define encoding and decoding functions to transform from geometries into poses, and vice versa.

The encoding function takes two geometries as input and computes the transformation from the first to the second. If the first geometry corresponds to the reference geometry (its coordinates are in the objects local coordinate system) and the coordinates of the second geometry are in the reference coordinate system, the returned transformation can be seen as the pose of the second geometry in this reference coordinate system. Thus, in the following sections, we will use the terms pose, position, and orientation and transformation, translation, and rotation interchangeably.

In case the reference geometry is known, we can transform a sequence of geometry snapshots into a sequence of poses by calling the encoding function for all geometry snapshots, giving the reference geometry as first parameter. However, if the reference geometry is not known, we can choose it arbitrarily. In this thesis, we choose the reference geometry such that its centroid is at $(0, 0)$ (or $(0, 0, 0)$) and its first pose has a rotation of $\theta = 0$ (or $Q = \langle 1, 0, 0, 0 \rangle$).

Note that in the maritime use-case, the data already arrives as a sequence of poses (constructed from the latitude, longitude and heading columns) and this transformation is thus not needed.

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 a pose value. As detailed in the data model (Sect. 4.1), 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 Alg. 4.1. First, we compute the translation parameters, as the difference between the coordinates of the geometries centers of rotation. Then we translate both geometries to have their respective rotation centers at the origin. The remaining problem is to compute a rotation quaternion from the first geometry to the second one. For this, we assume that the respective points of the input geometries are 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 center of rotation) of the first geometry, and let $\overrightarrow{P\prime}$ and $\overrightarrow{R\prime}$ denote the corresponding points on the second geometry. As a first step, we compute the rotation axis by realizing

that both $\overrightarrow{PP\prime}$ and $\overrightarrow{RR\prime}$ are perpendicular to this axis. This rotation axis is represented by a unit vector $\overrightarrow{e}$.

$$\overrightarrow{e} = \frac{\overrightarrow{PP\prime} \times \overrightarrow{RR\prime}}{\|\overrightarrow{PP\prime} \times \overrightarrow{RR\prime}\|} \tag{4.2}$$

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

$$\begin{aligned} \overrightarrow{P_e} &= \overrightarrow{P} - (\overrightarrow{P} \cdot \overrightarrow{e}) \cdot \overrightarrow{e} & \overrightarrow{P\prime_e} &= \overrightarrow{P\prime} - (\overrightarrow{P\prime} \cdot \overrightarrow{e}) \cdot \overrightarrow{e} \\ |\theta| &= \text{acos}(\frac{\overrightarrow{P_e} \cdot \overrightarrow{P\prime_e}}{\|\overrightarrow{P_e}\| \cdot \|\overrightarrow{P\prime_e}\|}) & \text{sign}(\theta) &= \text{sign}(\overrightarrow{e} \cdot (\overrightarrow{P_e} \times \overrightarrow{P\prime_e})) \end{aligned} \tag{4.3}$$

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

$$\begin{aligned} W &= \cos(\tfrac{\theta}{2}) & X &= \overrightarrow{e_x} \cdot \sin(\tfrac{\theta}{2}) \\ Y &= \overrightarrow{e_y} \cdot \sin(\tfrac{\theta}{2}) & Z &= \overrightarrow{e_z} \cdot \sin(\tfrac{\theta}{2}) \end{aligned} \tag{4.4}$$

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

Algorithm 4.1: encode($\mathcal{G}_1, \mathcal{G}_2$, out $\mathcal{P}$)

Input: $\mathcal{G}_1, \mathcal{G}_2$, two congruent geometries
Output: $\mathcal{P}$, 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 $\overrightarrow{P\prime}$ as the first vertex of $\mathcal{G}_1$ and $\mathcal{G}_2$ respectively;
    $\theta \leftarrow$ angle between $\overrightarrow{P}$ and $\overrightarrow{P\prime}$;
    return $Pose(\overrightarrow{T}, \theta)$
  else
    initialize $\overrightarrow{P}, \overrightarrow{R}, \overrightarrow{P\prime}$ and $\overrightarrow{R\prime}$ as the first two vertices of $\mathcal{G}_1$ and $\mathcal{G}_2$ respectively;
    compute $\overrightarrow{e}$, θ, Q using Eqs. 4.2, 4.3, 4.4 respectively;
    return $Pose(\overrightarrow{T}, Q)$;

Decoder

The decoding function takes a reference geometry and a pose as input and applies this pose to the 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.

$$R = \begin{pmatrix} cos(\theta) & sin(\theta) \\ sin(\theta) & -cos(\theta) \end{pmatrix} \tag{4.5}$$

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

$$R = 2 \begin{pmatrix} \frac{1}{2} - (Y^2 + Z^2) & (XY - WZ) & (XZ + WY) \\ (XY + WZ) & \frac{1}{2} - (X^2 + Z^2) & (YZ - WX) \\ (XZ - WY) & (YZ + WX) & \frac{1}{2} - (X^2 - Y^2) \end{pmatrix} \tag{4.6}$$

The decoding function then works in two steps. First, applies the rotation matrix R to the geometry, which corresponds to a rotation around the origin. Then, it translates the geometry to place its rotation center a the position specified by the input pose.

Algorithm 4.2: decode($\mathcal{P}, \mathcal{G}out\ \mathcal{G}\prime$)

Input: $\mathcal{P}, \mathcal{G}$, input pose and reference geometry
Output: $\mathcal{G}\prime$, the resulting geometry
begin
  initialize $\mathcal{G}\prime$ as a copy of the reference geometry;
  compute R using Eq. 4.5 (2D) or Eq. 4.6 (3D);
  rotate($\mathcal{G}\prime, R$);
  translate($\mathcal{G}\prime, \overrightarrow{T}$);
  return $\mathcal{G}\prime$;

4.2.2 Interpolation

Interpolation between consecutive instants is part of the definition of *SEQUENCE* and *SEQUENCES* types, as in Eq. 3.2. 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 pose parameters. Indeed, we cannot interpolate the vertices of the geometries without deforming them, as

shown in Fig. 2.3. Thus we first interpolate the pose parameters at the given time, then use the decoding function to get the interpolated geometry.

Given two poses $\mathcal{P}_1$ and $\mathcal{P}_2$ at times t_1 and t_2, we want to compute the pose parameters at $t_1 < t < t_2$. For both the 2D and the 3D cases, the translation parameters $\overrightarrow{T} = (x, y, z)$ can be interpolated using linear referencing.

$$r = \frac{t - t_1}{t_2 - t_1} \tag{4.7}$$

$$\overrightarrow{T} = \overrightarrow{T}_1 * (1 - r) + \overrightarrow{T}_2 * r \tag{4.8}$$

The difference arises when interpolating the rotation. In two dimensions, we linearly interpolate the angle θ, while still keeping in mind that this angle has to stay between $-\pi$ and π. In 3D, we interpolate the 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 Sect. 4.2.1.

2D Case

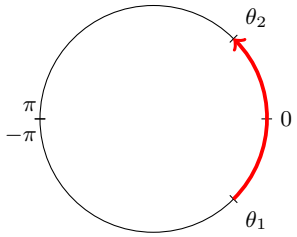
Given two angles θ_1 and θ_2 at times t_1 and t_2, we want to compute the angle θ at $t_1 < t < t_2$. Since the angle has to stay between $-\pi$ and π, one of four cases can apply depending on the relative position of the start and end angles. In Fig. 4.2, we project the bounded linear axis on a unit circle and distinguish the four cases.

Equation 4.9 can then be used to compute the resulting angle, according to the case.

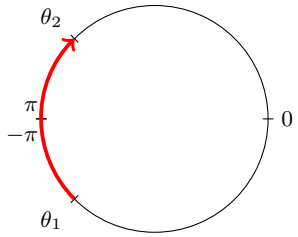
$$\theta = \begin{cases} \text{Case a: } \theta_1 + (\theta_2 - \theta_1) * ratio \\ \text{Case b: } \theta_2 + (2\pi + \theta_1 - \theta_2) * (1 - ratio) \\ \text{Case c: } \theta_2 + (\theta_1 - \theta_2) * (1 - ratio) \\ \text{Case d: } \theta_1 + (2\pi + \theta_2 - \theta_1) * ratio \end{cases} \tag{4.9}$$

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

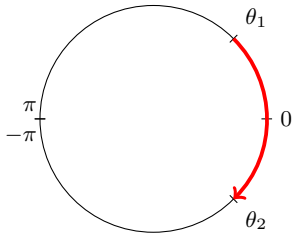
As a last remark, if the difference between the start and end angle is exactly π, we always interpolate in a counter-clockwise manner. This means that when $\theta_1 < \theta_2$, the interpolation goes through 0, and otherwise it goes through π.



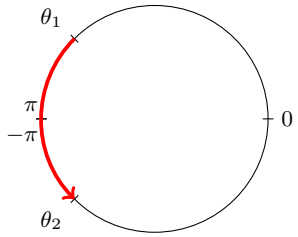
(a) $\theta_1 < \theta_2$ and $\theta_2 - \theta_1 <= \pi$



(b) $\theta_1 < \theta_2$ and $\theta_2 - \theta_1 > \pi$

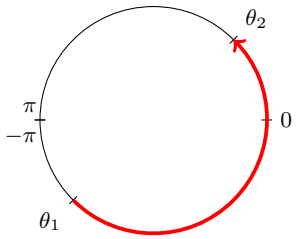


(c) $\theta_1 > \theta_2$ and $\theta_1 - \theta_2 < \pi$

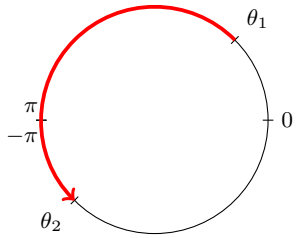


(d) $\theta_1 > \theta_2$ and $\theta_1 - \theta_2 >= \pi$

Fig. 4.2: Four different situations when interpolating angles



(a) $\theta_1 < \theta_2$ and $\theta_2 - \theta_1 = \pi$



(b) $\theta_1 > \theta_2$ and $\theta_1 - \theta_2 = \pi$

Fig. 4.3: Interpolation of a 180-degree rotation

3D Case

Given two quaternions Q_1 and Q_2 at times t_1 and t_2, we want to compute the quaternion Q at $t_1 < t < t_2$. This is done using the *slerp* (spherical linear interpolation) algorithm [17], which corresponds to a linear interpolation of the rotation angle around a fixed rotation axis.

$$\begin{aligned} \cos(\theta) &= Q_0 \cdot Q_1 \\ Q &= Slerp(Q_0, Q_1, r) \\ &= \frac{Q_0 * \sin((1-r) * \theta) + Q_1 * \sin(r * \theta)}{\sin(\theta)} \end{aligned} \tag{4.10}$$

Algorithm 4.3: interpolate($\mathcal{P}_1, \mathcal{P}_2, r, out\ \mathcal{P}$)

Input: $\mathcal{P}_1, \mathcal{P}_2, r$, two poses and a ratio value as computed using Eq. 4.7
Output: $\mathcal{P}$, the interpolated pose
begin
  $\overrightarrow{T} \leftarrow$ linear interpolation of the position parameters using Eq. 4.8;
  if *2D* **then**
    $\theta \leftarrow$ compute the angle using Eq. 4.9;
    return $Pose(\overrightarrow{T}, \theta)$;
  else
    $Q \leftarrow$ compute the quaternion using Eq. 4.10;
    return $Pose(\overrightarrow{T}, Q)$;

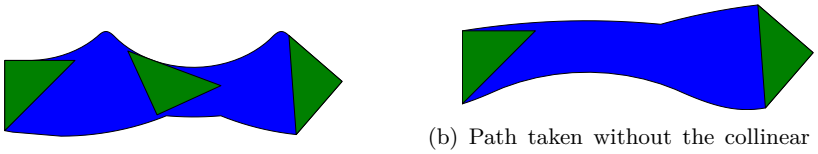
4.2.3 Normalization

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 Eq. 3.2.

Fig. 4.4: Two different representations of the same moving region

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. 4.5.

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.

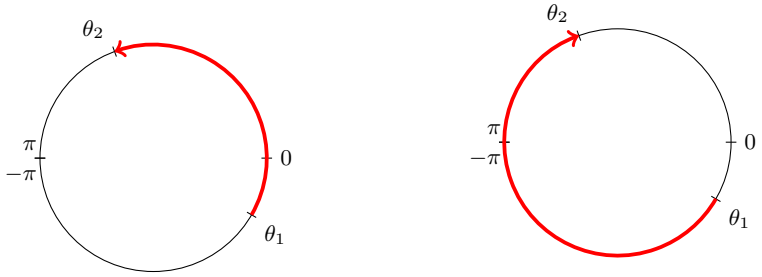


(a) Path taken with the collinear region

(b) Path taken without the collinear region

Fig. 4.5: Example of a collinear but not redundant instant in a moving region

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 poses are collinear. The collinearity check of the position parameters is similar to temporal points. In the following, we discuss the collinearity of orientation parameters.



(a) *interpolate_short*

(b) *interpolate_long*

Fig. 4.6: Interpolation of angles on the unit circle

In 3D, the *slerp* algorithm computes a linear interpolation of the rotation angle in the axis-angle representation of the quaternion. Here again, similar to the 2D case in Sect. 4.2.2, 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. Figure 4.6 displays the difference between both functions in 2D. The interpolation method used to retrieve the transformation at a given time is always *interpolate_short* (Sect. 4.2.2).

During normalization, we compute the result of both interpolation methods between the first and third instant. If the middle instant is different from 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*, which 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 the 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 4.4: normalize($\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3, t_1, t_2, t_3, out\ \mathcal{P}, out\ t$)

Input: $\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3, t_1, t_2, t_3$, three poses and their corresponding time.
Output: $\mathcal{P}, t$, the new intermediate instant.
begin
  $r \leftarrow \frac{t_2 - t_1}{t_3 - t_1}$;
  $\mathcal{P}_{short} \leftarrow interpolate\_short(\mathcal{P}_1, \mathcal{P}_3, r)$;
  $\mathcal{P}_{long} \leftarrow interpolate\_long(\mathcal{P}_1, \mathcal{P}_3, r)$;
  if $\mathcal{P}_2 = \mathcal{P}_{short}$ **then**
    return $NULL, NULL$;
  else if $\mathcal{P}_2 = \mathcal{P}_{long}$ **then**
    compute r_{dummy} using Eq. 4.11;
    $\mathcal{P}, t \leftarrow interpolate\_long(\mathcal{P}_1, \mathcal{P}_3, r_{dummy})$;
    return $\mathcal{P}, t$;
  else
    return $\mathcal{P}_2, t_2$;

The r_{dummy} parameter used in Alg. 4.4 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 Eq. 4.11. In 3D, we can use the same equation with the angles of the axis-angle representation of the rotation quaternions.

$$r_{dummy} = \frac{\pi - \epsilon}{|\theta_{12} + \theta_{23}|} \tag{4.11}$$

4.2.4 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 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 that is not fully contained by this naively computed bounding box, Fig. 4.7.

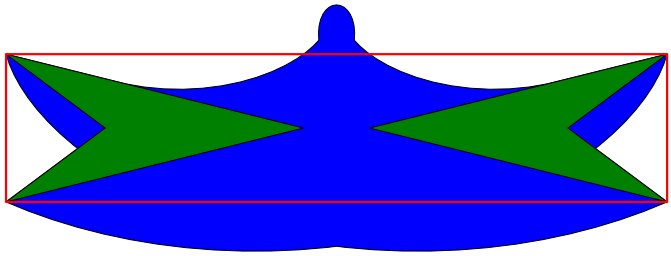


Fig. 4.7: The geometry in the left performs a translation to the right and a rotation of π. As illustrated, a naive union of the bounding boxes of individual instants is not a correct bounding box for the whole movement.

The 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 that 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 center of the geometry, with sides twice the length of the longest distance between the rotation center 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.

$$d_{max} = \max\{\sqrt{x^2 + y^2},\ (x, y) \in V_{geometry}\} \tag{4.12}$$

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 $Pose(x, y, \theta) \in$ `tgeometry`, the bounding box is computed using Eq. 4.13. The 3D case is omitted here but is analogous to the 2D case.

$$\begin{aligned} x_{min} &= \min\{x\} - d_{max} \quad & x_{max} &= \max\{x\} + d_{max} \\ y_{min} &= \min\{y\} - d_{max} \quad & y_{max} &= \max\{y\} + d_{max} \end{aligned} \tag{4.13}$$

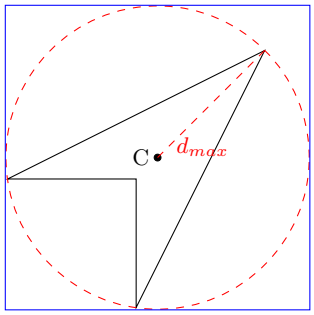


Fig. 4.8: Rotating bounding box around an instant

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 [39], 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 Sect. 4.5.

4.2.5 Traversed Area

The traversed area is the set of all points traversed by the temporal geometry. It can be seen as a projection onto 2D or 3D space of the prism of a moving feature. Examples of the traversed area of two 2D moving regions can be seen in Fig. 4.9.

An algorithm that can be used to compute the traversed area of a moving region is described in [39], and the resulting area is stored in a newly defined type called `cregion`, short for curved region, which can have three different types of segments: straight segments, trochoid segments and ravoid segments. The last two types are used to represent two different types of curves, *trochoids* and *ravoids*. A trochoid is used to represent the movement of a vertex of the region, and a ravoid is used to represent the movement of an edge of the region. These two types are described in [39], and the functions used to describe them are also listed, but they are not added here since they will not effectively be used for our solution of the problem.

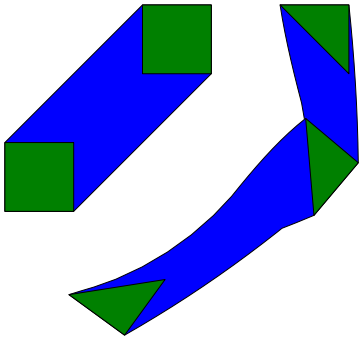


Fig. 4.9: Traversed area (in blue) of two moving regions

The solution given in the paper is also implemented in practice, and could thus be used by MobiltiyDB, but this would require us to define a new `cregion` or `cpolygon` type, and implement all of the needed geometric functions, such as `inside`, `overlap`, etc. However, one of the things that MobilityDB tries to do is to delegate geometric functions as much as possible to the existing PostGIS functions, since these are already implemented and optimized for the given types. To stay consistent with this idea, we thus want to store the traversed area in a PostGIS type, which enables us to use the PostGIS geometric functions without needing to re-implement them.

PostGIS has two different types that could be used to represent a traversed area: polygon and curvepolygon. If the moving region is not subject to any rotations, the resulting traversed area can be perfectly represented using a polygon, since there are no curves present. Otherwise, some parts of the area have to be represented using two different types of curves: trochoids and ravoids [39]. The curvepolygon, however, allows us to represent a circular curve, but neither trochoids nor ravoids. This means that no matter which polygon type is used, these curves will have to be approximated. Computing intersections between these two types of curves and straight segments is also computationally expensive, so instead of computing the traversed area as in [39] and then approximating it using a polygon or a curvepolygon, we will approximate the movement of the vertices and the edges of the region by straight segments, without ever using the trochoid or ravoid curve types. Since this solution uses only straight segments, the resulting traversed area can be stored in the polygon type. The current solution to this traversed area problem, together with remaining issues and possible improvements, is described below

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 [39], since we do not need to define new curve types to represent the border of the traversed area.

Algorithm

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.

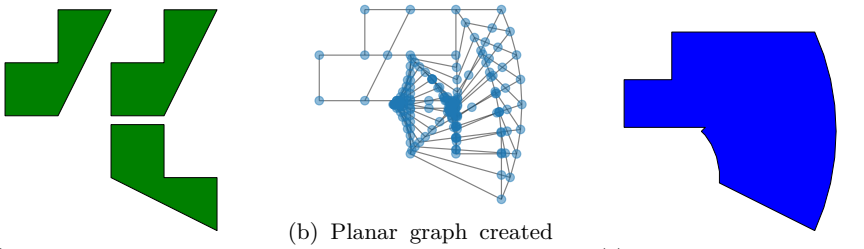
Algorithm 4.5: traversedArea($\mathcal{P}_{arr}, \mathcal{G}_{ref}, \theta_{max}, out\ \mathcal{G}$)

Input: $\mathcal{P}_{arr}, \mathcal{G}_{ref}, \theta_{max}$, an array of poses, a reference polygon and the maximum rotation angle
Output: $\mathcal{G}$, the polygon representing the traversed area
begin
  for $(\mathcal{P}_i, \mathcal{P}_{i+1})$ *in* $\mathcal{P}_{arr}$ **do**
    compute the rotation θ from $\mathcal{P}_i$ to $\mathcal{P}_{i+1}$;
    compute n using Eq. 4.14;
    add n intermediate poses computed using *interpolate*;
  `/* subsequent poses in` $\mathcal{P}_{arr}$ `now have a rotation smaller than` θ_{max} `*/`
  $\mathcal{G}_{arr} \leftarrow$ empty polygon array;
  for $(\mathcal{P}_i, \mathcal{P}_{i+1})$ *in* $\mathcal{P}_{arr}$ **do**
    compute the traversed area of $\mathcal{G}_{ref}$ between $\mathcal{P}_i$ and $\mathcal{P}_{i+1}$ assuming linear interpolation of vertices;
    add this polygon to $\mathcal{G}_{arr}$;
  return $ST\_Union(\mathcal{G}_{arr})$;

Given a sequence of poses, all relating to the same reference polygon, Alg. 4.5 computes the traversed area of this sequence as follows. As a first step, we compute the rotation θ between each pair of subsequent pose and add n interpolated poses between them if the total rotation θ was larger than θ_{max}. The number of intermediate poses added is computed using Eq. 4.14.

$$n = \lceil \frac{\theta}{\theta_{max}} \rceil - 1 \tag{4.14}$$

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



(a) Instants defining the moving regions

(b) Planar graph created from the computed segments

(c) Resulting traversed area

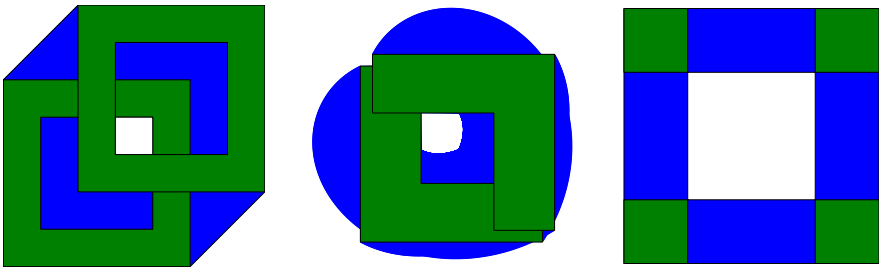
Fig. 4.10: Process of computing the traversed area of a moving region starting from a sequence of instants

The remaining problem consists of computing the traversed area of two subsequent poses 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 transforms this list of segments into a planar graph by considering the endpoints of the segments, as well as intersections of segments, as vertices of the graph. This process consists of computing all the intersections in the received list of segments, and then computing for all vertices and intersections the list of neighbors. Two points are neighbors if there exists a segment containing both points, and no other vertex or intersection is present between them on that same segment. A visual representation of this planar graph can be seen in Fig. 4.10b.

Computing all intersections of the set of segments is currently done naively (by testing all pairs of segments), but this could be improved by implementing the *Bentley–Ottmann algorithm* [18], which is a sweep-line algorithm with smaller time complexity than the naive solution when the number of intersections is not too large.

Lastly, the traversed area is computed from this planar graph. This is done by starting at the left-most vertex of the graph, and iteratively computing the next vertex of the traversed area until the start point is reached again. Computing the next vertex is done by taking the first neighbor of the current vertex when listing them in counterclockwise order starting from the previous vertex (which is one of the neighbors of the current vertex). By storing every vertex encountered in a list and stopping when the start vertex is reached, we obtain the polygon representing the traversed area of the moving region.

The polygon representing a traversed area can, in all generality, have an arbitrary number of inner rings (holes in the area). However, the described algorithm does not compute these inner rings and instead returns a polygon without any holes. Figure 4.11 shows three kinds of holes, that are created in three different ways.



(a) Type 1 (b) Type 2 (c) Type 3

Fig. 4.11: Different types of holes created during the computation of the traversed area

The first type is created from holes already present in the moving region (Fig. 4.11a). The second is created by a non-convex polygon in a single transformation, for example, when an L-shaped polygon rotates by π and translates slightly (Fig. 4.11b). The last one can be created by any type of polygon and occurs when the moving region comes back to a previous position after multiple transformations (Fig. 4.11c). For a hole like this to be present, the temporal sequence must have at least three instants, which means that the region is subject to at least two transformations.

By construction, the presented algorithm correctly takes the third type of hole into account, but does not handle the first two types of holes. The algorithm will always produce correct results if the input polygon is convex without holes, since the first two types of holes in the traversed area cannot appear in this case.

4.3 Implementation of MobilityDB Functions

The main focus of this master thesis is extending MobilityDB by adding a new type to represent moving regions and by implementing a set of functions to input, manipulate and output this new type. In Sect. 3.1.3, different types of functions and operators are presented, and most of these functions are polymorphic, meaning that they accept different input types and that their output depends on these input types. This section lists and describes most of these functions, and explains how they are used to manipulate the `tgeometry` and `tpose` types. We list the functions related to the `tgeometry` type. Most of these functions also have a equivalent version for the `tpose` type.

4.3.1 Type Definition

When defining a new SQL type, 2 functions have to be declared, and a few other functions and parameters are optional. The two required functions are *input* and *output*, converting a `cstring` (C string type) to the newly declared type and back. The optional functions and parameters are used to declare the internal storage length of the type, the storage alignment in bytes, input/out to and from byte format, type modifier specification, and more.

As explained previously, three new types have been added to MobilityDB: `pose`, `tpose`, and `tgeometry`. The `pose` type can store both 2D and 3D pose values internally. Multiple variations of the `tpose` and `tgeometry` type are also present at the C-level to manipulate objects of different temporal durations, but only these two top-level types are exposed at the SQL level. Table 4.1 lists the declared parameters and functions for the `tgeometry` type.

Table 4.1: Declared parameters and functions for the `tgeometry` type

Parameter	**Value**	**Description**
internalLength	`variable`	Internal storage length of the type in bytes.
storage	`extended`	Storage strategy of the type.
alignment	`double`	Storage alignment of the type.
Function		**Description**
input		Conversion from `cstring` to `tgeometry` type
output		Conversion from `tgeometry` to `cstring` type
receive		Conversion from `bytea` to `tgeometry` type
send		Conversion from `tgeometry` to `bytea` type

The `tgeometry` type contains PostGIS polygons (which are of variable length) and can have an arbitrary number of defined instants, so it must be defined as variable-length. Since the type is variable in length, the `plain` storage strategy, storing the values in-line without compression, cannot be used. The `extended` strategy is thus used to compress large values, and even move them out of the main table if the value is still too large.

The string representation of the `tgeometry` types depends on the duration of the value as described in Sect. 3.1.3 (instant, sequence, sequence set), and represents the values of the instants as `pose` types paired with a PostGIS polygon. These conversion functions from and to a string are defined together with the conversion functions from and to a byte list.

4.3.2 Constructors

As mentioned in Sect 4.2.1, a `tgeometry` value can be constructed in two main ways, depending on how the data was collected. If the data is collected as a set of position and orientations at different timestamps, we first use the `tpose` constructors to create a temporal pose object. Then, we combine this temporal pose with the reference geometry to produce the `tgeometry` value. However, if the data is collected as a sequence of polygon values, for example retrieved from satellite images, a different set of `tgeometry` constructors should be used. These constructors make use of the encoding algorithm (Alg. 4.1) to compute the sequence of poses. Table 4.2 lists the different constructors functions for the `tgeometry` types.

Table 4.2: Constructor functions for `tgeometry`

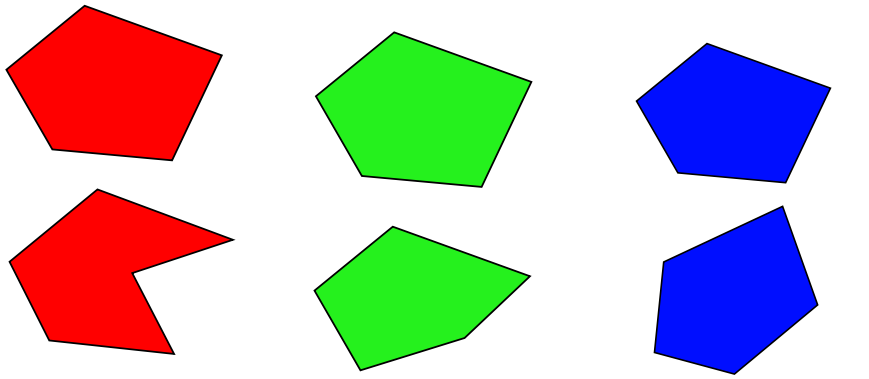
Function	**Signature**		
tgeometry	`tpose` × `geometry`	→	`tgeometry`
tgeometry	`geometry` × `timestamptz`	→	`tgeometry(Instant)`
tgeometrySeq	`ARRAY tgeometry(Instant)` × `text` × `bool` × `bool`	→	`tgeometry(Sequence)`
tgeometrySeqSet	`ARRAY tgeometry(Sequence)`	→	`tgeometry(Sequence Set)`

When creating a `tgeometry(Instant)` using the second set of constructors, the input geometry is checked to make sure that it is either a 2D polygon or a 3D polyhedron. The sequence constructor receives an array of `tgeometry(Instant)` as input, while the sequence set constructor requires an array of `tgeometrys(Sequence)`. Different checks are done to make sure that this input array corresponds to a valid temporal geometry.

First of all, when creating a sequence, the usual timestamp checks are done, meaning that the instants must have strictly increasing timestamps. Lastly, the regions are compared to make sure that they have the same shape.

This comparison between regions is done in two steps. A first quick check compares the number of inner rings of the polygons and if both polygons have the same number of rings, it compares the number of points on each ring. If two polygons pass this check, the transformation between the two polygons is computed from the first two points of the outer ring of both polygons, and the transformation is then applied to one of the polygons. Only polygons where the corresponding points are close enough to each other (their distance is smaller than a fixed ϵ) after the transformation are considered fixed-shape. Figure 4.12 shows examples of polygons that fail the first test (a), pass the first test, but fail the second (b), or pass all tests because they are indeed fixed-shape (c).

If all polygons are indeed fixed-shape, the input array is accepted and transformed into a sequence of poses using Alg. 4.1.



(a) Different number of vertices

(b) Different shape

(c) Same shape, only transformed

Fig. 4.12: Polygons failing or passing the different constructor checks

The sequence constructor has three additional parameters. The first parameter specifies the interpolation strategy for the sequence, either discrete (no interpolation), stepwise or linear. The last two specify if the start and end instants are inclusive or not and by default both are considered inclusive. The sequence set constructor has checks similar to the two previous ones, but since the input durations are already assumed to be valid sequences, only the comparisons between distinct sequences have to be done. During these comparisons, checks are done to ensure continuously increasing timestamps, and fixed-shape polygons across the whole sequence set.

Every `tgeometry` instance of either sequence or sequence set duration also computes and saves a bounding box used for optimization purposes. Section 4.3.10 lists a few bounding-box specific functions, and Sect. 4.2.4 explains how these bounding boxes are computed.

Lastly, the arrays of instants and sequences stored are normalized during the input process to allow for efficient comparisons later on. This normalization is necessary to ensure that identical paths have the same internal representation. This topic has already been discussed in Sect. 4.2.3.

4.3.3 Transformation Functions

Table 4.3 lists the transformation functions that can be applied to `tgeometry` objects. The first three transformation functions are used to change the duration of a temporal geometry. These transformations are only possible when the correct number of instants and/or sequences are present in the input `tgeometry`.

Next to changing their duration, temporal geometries can also be transformed by appending a new instant/sequence to the end or by merging two temporal geometries, during which similar checks as in the constructors are applied.

Table 4.3: Transformation functions for `tgeometry`

Function	**Signature**		
tgeometryInst	`tgeometry`	→	`tgeometry(Instant)`
tgeometrySeq	`tgeometry`	→	`tgeometry(Sequence)`
tgeometrySeqSet	`tgeometry`	→	`tgeometry(Sequence Set)`
appendInstant	`tgeometry` × `tgeometry(Instant)`	→	`tgeometry(Sequence)`
appendSequence	`tgeometry` × `tgeometry(Sequence)`	→	`tgeometry(Sequence Set)`
merge	`tgeometry` × `tgeometry`	→	`tgeometry`
merge	`ARRAY tgeometry`	→	`tgeometry`

4.3.4 Accessors

A large list of accessor functions exists to retrieve the duration, memory size, individual values, instants, timestamps, and more. These functions are listed in Table 4.4. The *tpose* function retrieves the `tpose` from a `tgeometry` disarding the reference geometry. This function is also used to create a SQL cast in MobilityDB from `tgeometry` to `tpose`.

The *tempSubType* function returns one of {`"Instant"`, `"Sequence"`, `"Sequence Set"`}, *interp* returns one of {`"None"`, `"Discrete"`, `"Linear"`, `"Stepwise"`}, and *memsize* returns the internal memory size in bytes.

Other accessor functions are defined to retrieve individual values, timestamps or sequences from the temporal geometry. In Table 4.4, all functions containing the symbol '*' are part of a set of functions returning a certain type of element. These functions return the number of that element in the `tgeometry`, the first, last or n-th element, and an array containing all elements respectively. This set is defined for the `Value`, `Timestamp`, `Instant` and `Sequence` elements. A complete list of these functions can be found in the MobilityDB manual [90].

Table 4.4: Accessor functions for `tgeometry`

Function	Signature		
tpose	`tgeometry`	→	`tpose`
tempSubType	`tgeometry`	→	`text`
interp	`tgeometry`	→	`text`
memSize	`tgeometry`	→	`integer`
getValue	`tgeometry(Instant)`	→	`geometry`
getTimestamp	`tgeometry(Instant)`	→	`timestamptz`
valueAtTimestamp	`tgeometry` × `timestamptz`	→	`geometry`
getTime	`tgeometry`	→	`tstzspanset`
timespan	`tgeometry`	→	`tstzspan`
*num**	`tgeometry`	→	`integer`
*start**	`tgeometry`	→	`*`
*end**	`tgeometry`	→	`*`
**N*	`tgeometry` × `integer`	→	`*`
**s*	`tgeometry`	→	`array *`

4.3.5 Always/Ever Comparison Functions and Operators

These functions and operators compare a temporal geometry with a fixed region value (Polygon or Polyhedron) and return the result of the ever equals (*ever_eq*), always equals (*always_eq*), ever not equals (*ever_ne*) or always not equals (*always_ne*) comparisons.

Table 4.5: Always/Ever comparison functions and operators

Function	Operator	Signature		
ever_eq	`?=`	`tgeometry` × `geometry`	→	`boolean`
always_eq	`%=`	`tgeometry` × `geometry`	→	`boolean`
ever_ne	`&<>`	`tgeometry` × `geometry`	→	`boolean`
always_ne	`%<>`	`tgeometry` × `geometry`	→	`boolean`

The only way a `tgeometry` object is ever equal to a given polygon (or polyhedron if 3D) is if this polygon is congruent to the reference geometry of the `tgeometry`. Thus, these functions first check if this is the case. If so, the encoder is used to compute the `pose` of the given polygon and the comparison function is then applied on the `tpose` of the temporal geometry.

4.3.6 Restriction and Difference Functions

These functions are used to restrict a temporal geometry to a given value (restriction functions), or to the complement of that value (difference functions).

The given value can range from a single region Polygon or Polyhedron) value to a timestamp, or even a set of tstzspan values. The set of restriction and difference functions is listed in Table 4.6. The symbol $\mathcal{T}$ = {`timestamptz`, `tstzset`, `tstzspan`, `tstzspanset`} is used to represent a time value of any duration.

Table 4.6: Restriction and difference functions

Function	**Signature**		
atValue	`tgeometry` × `geometry`	→	`tgeometry`
minusValue	`tgeometry` × `geometry`	→	`tgeometry`
atValues	`tgeometry` × `ARRAY geometry`	→	`tgeometry`
minusValues	`tgeometry` × `ARRAY geometry`	→	`tgeometry`
atTime	`tgeometry` × $\mathcal{T}$	→	`tgeometry`
minusTime	`tgeometry` × $\mathcal{T}$	→	`tgeometry`
atGeometry	`tgeometry` × `geometry`	→	`tgeometry`

The *at-* and *minusValue(s)* functions work similarly to the ever/always comparison functions. They first compute the pose of the given geometry value(s) and then apply *at-* or *minusValue(s)* on the temporal pose of the temporal geometry. The *atGeometry* function on the other hand, recieves any type of geometry as input and restrict the `tgeometry` to the times when its rotation center intersects the given geometry. This is thus equivalent to applying *atGeometry* to the `tpose` representation of the `tgeometry`.

4.3.7 Comparison Operators

A set of comparison operators is defined in PostgreSQL, and they have been extended to accept arguments of type `tgeometry`. In practice, only the equals (`=`) and not equals (`<>`) operators are useful for real-world applications, but the other operators are used internally to create B-tree indexes or to allow `GROUP BY` or `DISTINCT` clauses on the `tgeometry` type.

4.3.8 Temporal Comparison Operators

The previous equality and inequality operators returned a single boolean value, but we might also be interested in knowing when two temporal geometries are equal or when they differ as a function of time. The following two operators return a temporal bool representing the result over time of the equality or inequality operator on the input values.

Table 4.7: Comparison functions and operators

Function	Operator	Signature		
tgeometry_lt	`<`	`tgeometry` × `tgeometry`	→	`boolean`
tgeometry_le	`<=`	`tgeometry` × `tgeometry`	→	`boolean`
tgeometry_eq	`=`	`tgeometry` × `tgeometry`	→	`boolean`
tgeometry_ne	`<>`	`tgeometry` × `tgeometry`	→	`boolean`
tgeometry_ge	`>=`	`tgeometry` × `tgeometry`	→	`boolean`
tgeometry_gt	`>`	`tgeometry` × `tgeometry`	→	`boolean`
tgeometry_cmp		`tgeometry` × `tgeometry`	→	`integer`

Table 4.8: Temporal comparison functions and operators

Function	Operator	Signature		
tgeo_eq	`#=`	{`tgeometry, geometry`} × {`tgeometry, geometry`}	→	`tbool`
tgeo_ne	`#<>`	{`tgeometry, geometry`} × {`tgeometry, geometry`}	→	`tbool`

Again, the comparison is actually being applied on the pose representation in case the geometries are congruent. If they are not, the values are assumed to always be non equal.

4.3.9 Spatial Functions

As has been done for the temporal points previously in MobilityDB (briefly described in Sect. 3.1.3), multiple spatial functions similar to those defined in PostGIS are exposed. Table 4.9 shows a non-exhaustive list of spatial functions that can be applied to the `tgeometry` type.

Table 4.9: Spatial functions

Function	Signature		
traversedArea	`tgeometry`	→	`geometry(Polygon)`
rotationSpeed	`tgeometry`	→	`tfloat`
nearestApproachDistance	{`tgeometry, geometry`} × {`tgeometry, geometry`}	→	`float`
nearestApproachInstant	{`tgeometry, geometry`} × {`tgeometry, geometry`}	→	`tgeometry(Instant)`
shortestLine	{`tgeometry, geometry`} × {`tgeometry, geometry`}	→	`geometry(LineString)`
tDistance	{`tgeometry, geometry`} × {`tgeometry, geometry`}	→	`tfloat`

The *traversedArea* function returns a polygon representing the area traversed by the moving region. The computation of this area is described in Sect. 4.2.5.

Computing the translation or rotation speed of a polygon can also be of interest, but since the translation speed can be retrieved by applying the *speed* function (defined for temporal points) on the rotation center of the moving region, only the rotation speed has to be handled. This is returned by the *rotationSpeed* function, which returns a temporal float. Since we assume linear interpolation between instants, the rotation speed is assumed to be constants between two instants.

Note that the rotation center of a temporal geometry can be retrieved by casting the `tgeometry` into a `tgeompoint`. Effectively, this does a first casting of `tgeometry` to `tpose` using the *tpose* function defined in Sect. 4.3.4 and then casts the `tpose` into a `tgeompoint` by simply discarding the orientation values of each pose.

The last four functions return respectively the smallest distance possible between the two (temporal) regions, the instant from the first temporal region at which this distance was attained, a segment going from the first region to the second at their closest instant, and finally a temporal float representing the evolution of the distance between the two regions during their movement.

The first three functions can be implemented easily if the `tDistance` function exists. Indeed, if we have a `tfloat` value representing the result of the distance function applied to two (moving) regions, we can use already defined functions on the `tfloat` type to find the minimum value, or the instant at which this minimum value is obtained, which solves the first two problems. After computing the position of both regions when they were at their closest, we can then also use a PostGIS spatial function to compute the smallest line joining these polygons, which is the result of the third question.

This leaves us with the implementation of the `tDistance` function, which is detailed in Chap. 5.

4.3.10 Bounding Box Functions and Operators

Bounding boxes are precomputed for temporal geometries of sequence and sequence set duration. These boxes are used in multiple functions to prune some input values with fast checks applied before the function itself. For example, if we want to compute the instants at which two regions were within 5 meters of each other, we can first compute the minimum distance between the bounding boxed and return `NULL` if this distance is larger than 5 meters.

Computing this bounding box is relatively tricky when the temporal geometry is a sequence or a sequence set since the bounding box is not simply the union of the bounding boxes of all instants. Section 4.2.4 explained how the bounding boxes of moving regions can be computed in two different ways.

Some bounding box operators are also exposed to the user. These operators are polymorphic and allow all types of input that can be cast into a spatio-temporal box (`STbox`).

Table 4.10: Bounding box functions and operators

Function	Operator	Signature		
stbox		`tgeometry`	→	`stbox`
expandSpace		`tgeometry` × `float`	→	`stbox`
expandTime		`tgeometry` × `interval`	→	`stbox`
temporal_contains	`@>`	{`tgeometry`, `geometry`, `stbox`} × {`tgeometry`, `geometry`, `stbox`}	→	`bool`
temporal_contained	`<@`	{`tgeometry`, `geometry`, `stbox`} × {`tgeometry`, `geometry`, `stbox`}	→	`bool`
temporal_overlaps	`&&`	{`tgeometry`, `geometry`, `stbox`} × {`tgeometry`, `geometry`, `stbox`}	→	`bool`
temporal_same	`~=`	{`tgeometry`, `geometry`, `stbox`} × {`tgeometry`, `geometry`, `stbox`}	→	`bool`
temporal_adjacent	`-|-`	{`tgeometry`, `geometry`, `stbox`} × {`tgeometry`, `geometry`, `stbox`}	→	`bool`

4.4 Conformance with the ISO 19141 Standard

The International Standard ISO 19141—Schema for moving features [45], 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*. Figure 4.13 illustrates a 2D rectangle that 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 full 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.

In its current form, ISO 19141 is an abstract schema that defines the spatio-temporal 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 im-

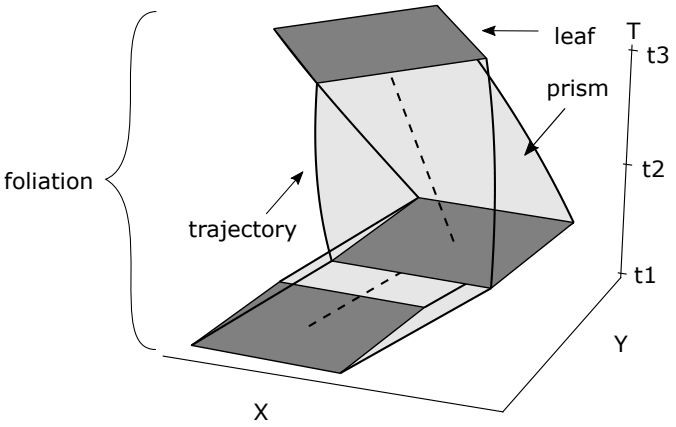


Fig. 4.13: Schema for moving features [45]

plementation standards and reference implementation. As such, developers have no means of consistently developing applications that manage this kind of data.

This thesis contributes to solving this problem by proposing an implementation of rigid temporal geometries in MobilityDB which conforms to the abstract model described in the standard. Section 4.4.1 introduces the ISO 19141 standard in more detail and Sect. 4.4.2 then explains how the standard operations on rigid temporal geometries are implemented in MobilityDB.

4.4.1 ISO 19141 – Schema for Moving Features

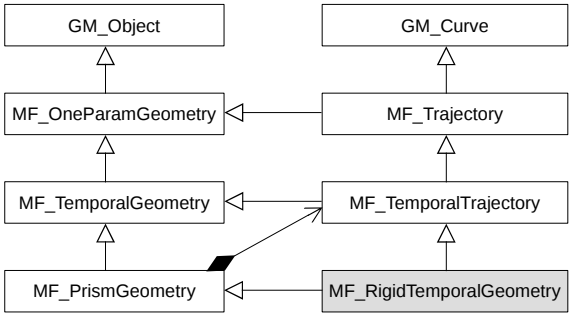


Fig. 4.14: Moving features types [45]

The main classes of the schema for moving features are shown in Fig. 4.14. 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 $f : D_p \Longrightarrow D_{GM\_Object}$, 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 [44]. The class `MF_TemporalGeometry` represents a mapping from time instants into geometries. During its definition interval [startTime, endTime], see Fig. 4.15, 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. 4.15 are discussed in detail in Sect. 4.4.2. `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 a few database systems including PostGIS' LineStringM and trajectory functions, Informix® Spatio-temporal Search, and MobilityDB temporal geometry point types and query API.

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. 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 center of rotation of the `MF_RigidTemporalGeometry`, which explains the association relation in Fig. 4.14 and Fig. 4.15. What is not explained in the standard is the inheritance relation between `MF_RigidTemporalGeometry` and `MF_TemporalTrajectory`, especially since certain functions in `MF_TemporalTrajectory` cannot apply to `MF_RigidTemporalGeometry`. This is further discussed in Sect. 4.4.3. Section 4.4.2 thus only contains the operations from `MF_TemporalGeometry`, `MF_PrismGeometry` and `MF_RigidTemporalGeometry`.

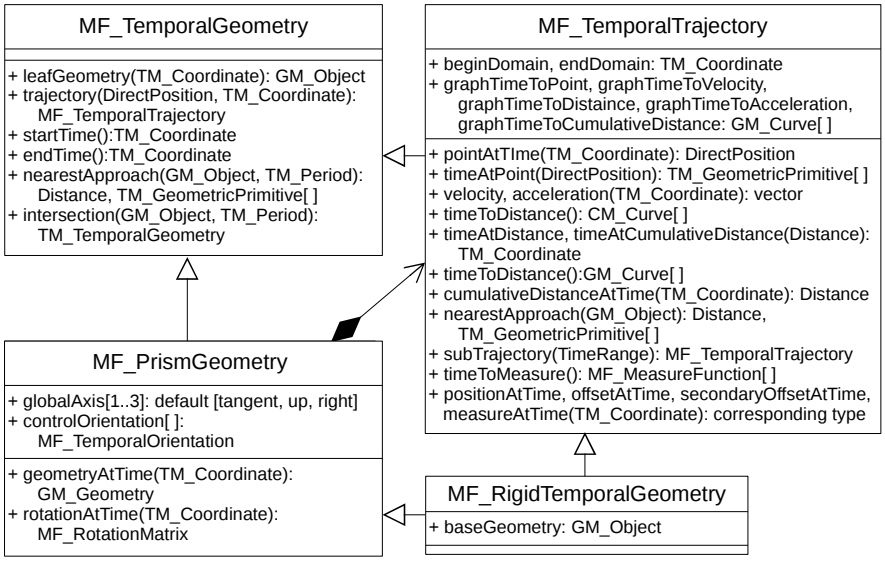


Fig. 4.15: Moving features types [45]

4.4.2 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 and in Sect. 4.4.3 can be useful in revising the standard.

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 Alg. 4.3 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`.

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 `tgeompoint`. 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 center of rotation, the function returns an exact solution since the rotation center of the geometry is assumed to move linearly. The result corresponds to the *originTrajectory* attribute defined in the `MF_PrismGeometry` class.

StartTime and EndTime

These are simple getters of the first and last timestamps of the temporal geometry, i.e., t_1, t_n in Eq. 3.2. These functions are called *startTimstamp* and *endTimestamp* in MobilityDB.

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 and a temporal geometry as the second argument.

The equations and algorithms for finding both the distance and the time of the nearest approach in 3D are complex computational geometry problems on their own and do not fit into the scope of this thesis. However, in Chap. 5 we present algorithms for 2D temporal distance operations. The implementation of the nearest approach functions is detailed in Sect. 5.5.2.

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 a fixed shape. The same goes for intersecting two temporal geometries. Since deforming geometries are not modeled in ISO 19141, this function cannot be defined. Alternatively, a special definition of its semantics has to be added.

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 center 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 with respect to the reference geometry. Based on this, we implement the function *quaternionAtTimestamp*, which returns the rotation quaternion at a given time. It represents the rotation we need to apply to the reference geometry to orient the geometry correctly at that timestamp. The returned quaternion 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 a rotation matrix, i.e., Eqs. 4.5 and 4.6.

4.4.3 Discussion 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 Sect. 4.4.2.

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.

Lastly, the standard gives as an annex a method to interpolate 3D rotations using quaternions, which has been the basis of the 3D *interpolate* algorithm. This addition is thus valuable. A small weak note might be that other than in this annex, there is no real mention of the interpolation process for temporal geometries. In MobilityDB, both stepwise and linear interpolation methods are available, but in the literature even more processes exist, and it might be interesting to either discuss this in the standard, or specify that this is out of scope.

4.5 Maritime Use Case

In this section we showcase and evaluate our implementation on the maritime dataset presented in Sect. 3.2.2. This dataset contains real-world AIS records distributed by the Danish Maritime Authority. As a reminder, the relevant columns of this data are listed in Table 4.11. We transform the lat/lon points into projected x/y coordiantes with SRID 25832 (European Terrestrial Reference System 1989), which is the CRS advised by the Danish Maritime Authority. We use the CSV file from September 29, 2020. The file size is 1.9 GB. It contains 8 M AIS points, in 1,810 ship tracks.

Assuming that this data is loaded in a table called `AISInput`, we can then create the rigid temporal geometries using Query 4.1.

Table 4.11: Relevant AIS columns

Name	Description
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

Query 4.1 *Constructing rigid temporal geometries from AIS data*

```
CREATE TABLE Ships(MMSI, Trip) AS
SELECT MMSI, tgeometrySeq(array_agg(tgeometry(
       ST_Rotate(ST_MakePolygon(ST_MakeLine(
           ST_MakePoint(x - sizeB, y - sizeC),
           ST_MakePoint(x + (3*sizeA - sizeB)/4, y - sizeC),
           ST_MakePoint(x + sizeA, y + (sizeD - sizeC)/2)
           ST_MakePoint(x + (3*sizeA - sizeB)/4, y + sizeD),
           ST_MakePoint(x - sizeB, y + sizeD),
           ST_MakePoint(x - sizeB, y - sizeC)
       )), pi() / 2 - radians(heading), ST_MakePoint(x, y)),
       T) ORDER BY T))
FROM AISInput GROUP BY MMSI;
```

As mentioned in Sect. 4.1.2, we represent the vessels using pentagons, and we size and rotate these rectangles according to the given data. Figure 4.16 illustrates the construction of the rectangle for a single record. Unlike in Fig. 4.1, we do not directly create a reference geometry, but rather produce the actual geometry of the vessel at each timestamp.

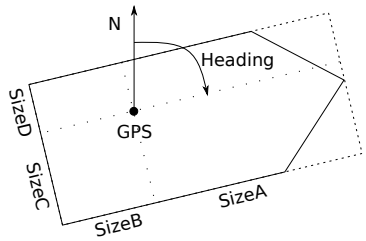


Fig. 4.16: Construction of the geometry of a vessel

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.

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 8 M AIS records. The constructed temporal geometry data is also about 1.2× larger than the temporal points. Compared to SECONDO there is a 50% gain in storage space.

Although this comparison is done on a small dataset, 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.

4.5.1 Data Retrieval and Visualization

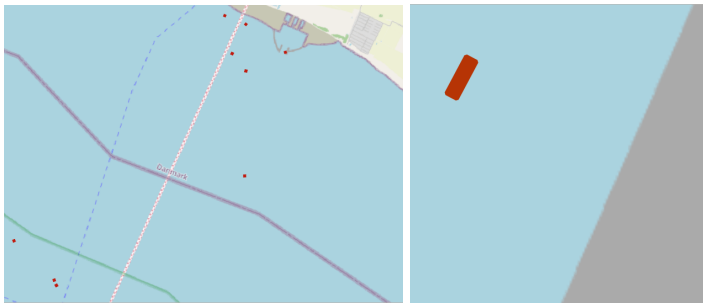
With the table `Ships(MMSI, Trip)` containing the temporal geometries, we can now use the implemented operations to analyze these temporal geometries. In Query 4.2, we compute the starting geometry of the vessels that were in the region (created using *ST_MakeEnvelope*) between the ports of Rødby and Puttgarden when they first started emitting data, as well as their nearest approach to one end of the breakwater of the port of Rødby.

Query 4.2 *Compute starting geometry and nearest approach distance*

```
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. 4.17a, with the vessels returned by Query 4.2. Figure 4.17b is

a zoomed-in version of the previous figure and displays the geometry of one of the ships on the top-right of Fig. 4.17a.



(a) (b)

Fig. 4.17: Visualization of ship geometries retrieved from temporal geometries using *valueAtTimestamp*

Query 4.2 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.

4.5.2 Indexing on Temporal Geometries

We extend the GiST and SP-GiST indexes of MobiltiyDB, implementing respectively an R-tree and a quadtree for 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.

Queries 4.3 and 4.4 create a GiST index on the `Trip` column and list all the vessels whose trip started after 12 pm and ended before 4 pm. We can see in the query plan of the Query 4.4 that the index is indeed being used.

Query 4.3 *Create R-tree index on the Ships table*

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

Query 4.4 *Query plan using the R-tree index*

```
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)
```

Although this index returns correct results, its efficiency can be limited when used for spatio-temporal filtering. This is due to the fact that the spatio-temporal boxes of temporal geometries cover a large extent and are thus not ideal for filtering purposes. This problem is also present with temporal points. Chapter 6 discussed this problem further and presents new indexing methods to improve the performance of indexes in MobilityDB.

4.6 A Data Generator for Rigid Temporal Geometries

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

We base this data generator on the BerlinMOD benchmark data generator (Sect. 3.2.1). In its original proposal, BerlinMOD generates temporal point objects representing car trajectories. 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. Queries 4.5 and 4.6 illustrate a sample to experiment with the generated data.

Query 4.5 *Geometry at a random timestamp*

```
SELECT valueAtTimestamp(Trip, startTimestamp(Trip) +
 random() * (endTimestamp(Trip) - startTimestamp(Trip)))
FROM Trips;
```

Query 4.6 *Trajectory of the center of rotation*

```
SELECT trajectory(Trip) FROM Trips;
```

Table 4.12: BerlinMOD generator statistics for scale factor 1.0

	Point	**2D Geometry**	**3D Geometry**
Generation time (minutes)	33	83	360
Data size (MB)	2461	2873	3621
Index Size (MB)	28	28	25
Query 4.5 (s)	31.4	26.8	36.1
Query 4.6 (s)	N/A	56.9	58.8

Table 4.12 shows the duration of the generation for points as well as 2D and 3D geometry types, together with the 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 is 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.

Chapter 5
A Temporal Distance Operator

When working with rigid temporal geometries, the time-varying (temporal) distance is of importance in many domains. For instance, it will help analyze near-collision cases among sea vessels or autonomous vehicles. It will also help to analyze the interaction of logistics robots and their surroundings. As mentioned in the previous chapters, existing solutions in the domain of moving object databases approximate the temporal body by a temporal point, e.g., its centroid. This representation, although memory-efficient and easier to manipulate, completely disregards the spatial extent of the moving bodies and can result in inaccurate or wrong results when computing distances. In the example shown in Fig. 5.1, the difference between computing the distance using the moving point presentation v.s. using the moving body representation is illustrated. The nearest approach distance between the objects is in reality less than half as much as what is computed using the moving point representation.

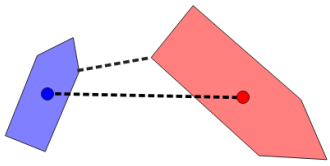


Fig. 5.1: Nearest approach distance in moving point and moving body representation. The moving point distance is 9.8m, compared to the 3.7m computed using moving bodies.

In the domain of computer graphics, computing the distance between two rigid bodies is a well-known and heavily researched problem. Some of the presented algorithms are optimized for distance computations on moving bodies [32, 50, 55]. These solutions, however, compute the evolution of the distance by iteratively calling a static distance function at subsequent snap-

shots. None of these solutions makes use of the parameters of the movement, mostly because they do not assume that the motion is known in advance. These solutions are thus not viable for tracking the continuous distance when the complete movement of the bodies is known, as is the case when analyzing historical moving objects data in a moving object database. For example, when computing the nearest approach distance between two moving bodies, the minimum distance can be missed if it is reached between two calls of the static distance algorithm. To clarify, the term *distance* denotes the smallest Euclidean distance between two static geometries, and the *minimum distance* thus relates to the smallest distance value reached during the movement of the geometries.

In this chapter, we present an efficient algorithm to compute the temporal distance between two moving bodies in 2D, when both the geometry and the movement of the bodies are known. The algorithm computes a list storing the evolution of the closest features (vertices or edges) between the moving bodies. This list, together with the moving bodies themselves, completely defines the distance function between these objects. It can then be used to compute the distance at any time during the movement. This algorithm can be used to compute the distance between a pair of moving objects or between a static and a moving object. Note that the latter case is obtained by simply setting all the movement parameters of the static object to zero. We also implement the proposed algorithm in MobilityDB and evaluate it on our maritime dataset.

The rest of the chapter is organized as follows. We define a representation of moving objects in Sect. 5.1, which is compatible with the existing representations of moving points and moving bodies in moving object databases [39, 76], such as the one presented in Chap. 4. Section 5.2 describes the proposed algorithm for computing the evolution of closest features between convex moving bodies and Sect. 5.3 explains the solution for non-convex polygons. Section 5.4 then discusses possible optimizations in case the movement is non-rotating and Sect. 5.5 presents the implementation of the proposed algorithms in MobilityDB. Lastly, Sect. 5.6 validates the theoretical complexity of the proposed algorithms and assesses their running time in an actual database use case.

5.1 Preliminaries

In this section, we present a simple model for moving points and moving bodies that will be used throughout the chapter. This model agrees with more general models described in Chap. 4 and in previous research [15, 25, 39, 76, 91], but restricts the movement of an object to a single segment. Indeed, if the movement is composed of a sequence of these segments (as is usually the case), the algorithm presented in this chapter can be applied

independently to each segment. We discuss how this can be done for the sequence-based data model presented in Chap. 4 at the end of this section.

A 2D moving object is described using a static geometry and associated movement parameters. In this chapter, the static geometry will be either a point $p = (x_p, y_p)$ or a simple polygon $\mathcal{R}$.

A simple polygon $\mathcal{R}$ is represented using a list of n vertices stored in counter-clockwise order, together with a rotation center (x_c, y_c) assumed to be inside the polygon. The presented algorithm does not take into account holes in the polygon.

$$\mathcal{R} = [(x_1, y_1), ..., (x_n, y_n)|(x_c, y_c)] \tag{5.1}$$

To simplify the notation, we will assume that $(x_{n+i}, y_{n+i}) = (x_i, y_i)$. The modulo operations will thus be omitted in the rest of the equations. To further simplify the notation, when talking about a single vertex we will use v_i and (x_i, y_i) interchangeably. Edges of the polygon are denoted e_i and represent the linear segment between v_i and v_{i+1}.

We define a point on the edge e_i as $e_i(s) = (x_i(s), y_i(s)) = v_i*(1-s)+v_{i+1}*s$, with $s \in [0, 1]$. The notation $v_i * s$ corresponds to a scalar multiplication and $v_i + v_{i+1}$ is the standard vector addition. This parametric definition of a point along an edge is the same as in [55]. The distance between a point $p = (x_p, y_p)$ and an edge e_i of the polygon can thus be computed using Eq. 5.2. The notation $d(e_i, p)$ is an abuse of notation as e_i is not a vector, but it is to be understood as the minimum Euclidean distance between the edge e_i and p.

$$d(e_i, p) = min_{s\in[0,1]}\{\sqrt{(x_i(s) - x_p)^2 + (y_i(s) - y_p)^2}\} \tag{5.2}$$

As mentioned before, a moving object combines a static geometry with movement parameters related to that geometry. In the case of a moving point, its movement is defined by a single translation (dx, dy). The coordinates of a moving point $p(t) = (x_p(t), y_p(t))$ can thus be computed at a given $t \in [0, 1]$ using Eq. 5.3. The parameter t represents time, normalized to $[0, 1]$. Thus, $t = 0$ corresponds to the start of the movement and $t = 1$ corresponds to the end of the movement. This will hold for all future equations in this chapter.

$$\begin{aligned} x_p(t) &= x_p + t * dx \\ y_p(t) &= y_p + t * dy \end{aligned} \tag{5.3}$$

Similarly, the movement of a polygon is defined by a translation (dx, dy) and a rotation θ around its rotation center (x_c, y_c). In theory, the algorithms described in this chapter work for any value of θ. In practice, however, we will assume that θ is a (sufficiently small) constant and we can thus omit it from the complexity estimations of Sect. 5.2. In the model presented in Chap. 4, for example, the value of θ for a single segment is restricted between $-\pi$ and π.

A moving polygon $\mathcal{R}(t)$ is thus represented by a list of moving vertices. At any given $t \in [0,1]$, the coordinates of the vertices are computed using Eq. 5.5.

$$\mathcal{R}(t) = [v_1(t), ..., v_n(t)|v_c(t)] \tag{5.4}$$

$$\begin{aligned} x_i(t) =& (x_i - x_c) * cos(t * \theta) - (y_i - y_c) * sin(t * \theta) \\ &+ x_c + t * dx \\ y_i(t) =& (x_i - x_c) * sin(t * \theta) + (y_i - y_c) * cos(t * \theta) \\ &+ y_c + t * dy \end{aligned} \tag{5.5}$$

We define the moving edge $e_i(t)$ as being the moving linear segment between the moving vertices $v_i(t)$ and $v_{i+1}(t)$. Note that Eq. 5.5 guarantees non-deformation. The length of the segments is thus constant during the movement. Like in the static case, we define a point on this edge as $e_i(s,t) = (x_i(s,t), y_i(s,t)) = v_i(t) * (1-s) + v_{i+1}(t) * s$, with $s \in [0,1]$.

These definitions of a moving point and moving polygon only represent a single *linear* segment of the movement. In practice, however, the full movement of an object is composed of a sequence of segments, which are either combined into a single data object [76, 91] or stored individually [15, 25, 39]. In Chap. 4, for example, the data is stored as a list of poses with a reference geometry. Each consecutive pair of poses represents a linear segment of the movement. The starting geometry of a specific segment is computed by applying the decoder algorithm (Alg. 4.2) with the reference geometry and the start pose of the segment. The transformation parameters are then computed as the difference between the start and end pose of the segment.

Note that the algorithm requires an initialization step, as described in Sect. 5.2. This initialization step computes the initial set of closest features at time $t = 0$ (of the current segment), using existing algorithms. In case the movement is composed of a sequence of segments, the initial set of closest features of the next segment will be the same as the set of closest features at the end of the last segment. The initialization step is thus only required for the very first segment. Between two segments, the validity of the closest features can be tested in constant time using the V-Clip algorithm. In case of invalid features, the result of the V-Clip algorithm will be used as input to the next segment. This avoids propagating errors when working will long trajectories.

Lastly, when computing the distance between two segments having different start and end timestamps, the algorithm can only be applied on the period during which these segments overlap. Let's imagine that the movement of a first object is defined between t_1 and t_2, and the movement of a second object is defined between t_3 and t_4, with $t_1 < t_3 < t_2 < t_4$. The algorithm can thus only be applied on the period $[t_3, t_2]$. In the rest of this chapter,

we always assume that this period is normalized to $[0, 1]$ for simplicity. All equations thus implicitly assume $t \in [0, 1]$.

5.2 Evolution of Closest Features

The problem of computing the distance between two static geometries can be reduced to finding the two closest features of these geometries. The features of a geometry are its vertices and edges. For a point geometry, the only existing feature is the point itself. Given the closest features of two geometries, their distance can then be computed in constant time. Indeed, both the distance between two points and the distance between a point and an edge can be computed in constant time.

The same conclusion can be made when computing the temporal distance between two moving objects. If the closest features are known at any time $t \in [0, 1]$, returning the distance between these objects given their closest features can be done in constant time. Knowing this, the remaining problem consists of computing the evolution of the closest features of the two moving objects from $t = 0$ to $t = 1$. In this section, we present an algorithm that, given a pair of moving objects (points or polygons) returns a list representing the evolution of closest features during the movement of the objects.

The rest of this section is structured as follows. Section 5.2.1 first details how we can compute the distance between a moving point and a moving edge in constant time. This section also presents a fundamental equation (Eq. 5.8) used to detect changes in the closest features. Then, Sect. 5.2.2 presents the algorithm to find the evolution of the closest features between a moving point and a convex moving polygon and Sect. 5.2.3 describes an equivalent algorithm applied on two convex moving polygons.

5.2.1 Point-to-Edge Distance

This section describes the equations needed to compute the distance between a moving point $p(t) = (x_p(t), y_p(t))$ and a moving edge $e(t)$. These equations are crucial for the algorithms described in Sects. 5.2.2 and 5.2.3. For simplicity, we omit the subscript of the edge and denote its start and end vertices as $v_s(t)$ and $v_e(t)$ respectively. A point on the moving edge is thus defined using Eq. 5.6. As mentioned, this follows from the static parametric definition of a point along an edge, as used in [55].

$$e(s, t) = v_s(t) * (1 - s) + v_e(t) * s, \; s \in [0, 1] \tag{5.6}$$

The equations of the moving point $p(t)$ are not specified here, as they can in practice be any parametric function of time. In the context of this chapter, however, we can assume that they correspond to either Eq. 5.3 or Eq. 5.5.

As described in Sect. 5.1, the distance between a point and an edge can be computed by minimizing the distance between this moving point and any point on the moving edge (Eq. 5.7). Again, $d(e(t), p(t))$ is an abuse of notation and denotes the minimum Euclidean distance between the edge $e(t)$ and $p(t)$.

$$d(e(t), p(t)) = min_{s \in [0,1]}\{d(e(s,t), p(t))\} \tag{5.7}$$

By differentiating $d(e(s,t), p(t))$ with respect to s and equating to 0, we can find the values of s where this minimum is obtained. This results in a function $s(t)$, given in Eq. 5.8.

$$s(t) = \frac{(p(t) - v_s(t)) \cdot (v_e(t) - v_s(t))}{L^2}, \tag{5.8}$$

where $x \cdot y$ represent the standard dot operator and $L^2 = (x_e - x_s)^2 + (y_e - y_s)^2$ is the squared length of the edge.

Since we require $s \in [0,1]$ for the point $e(s,t)$ to be on the edge, we can distinguish three cases:

- $s(t) <= 0$: Point p is closest to the start vertex v_s.
- $s(t) >= 1$: Point p is closest to the end vertex v_e.
- $0 < s(t) < 1$: Point p is closest to $e(s(t), t)$.

Using Eq. 5.8 we can thus determine the point on the moving edge closest to the moving point. Equation 5.8 has two main uses. Firstly, it can tell us when the closest point on the edge is one of the end vertices or a point inside the edge. This will be used to determine changes in the closest features in the following sections. Secondly, knowing $s(t)$ we can compute the distance between a moving point $p(t)$ and a moving edge $e(t)$ at any given time $t \in [0,1]$ in $\mathcal{O}(1)$ time using Eq. 5.9.

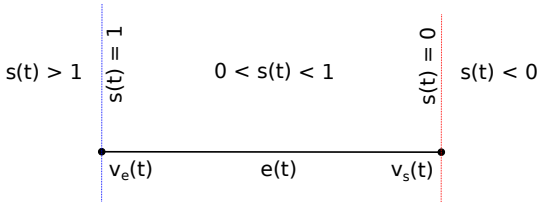


Fig. 5.2: Visualization of the value of $s(t)$ depending on the position of the point with respect to the edge

$$d(e(t), p(t)) = d(e(max(min(s(t), 1), 0), t), p(t)) \tag{5.9}$$

Figure 5.2 displays the lines where $s(t) = 0$ (in red) and $s(t) = 1$ (in blue) for a moving edge. The position of the moving point with respect to these lines will thus determine where the closest point on the edge lies. It is important to understand that these lines are not actually parametrized by t. Indeed, if the equation $s(t) = 0$ ($s(t) = 1$) is true for a given t^*, this simply means that the moving point is on the red (blue) line at $t = t^*$. The actual equations of the two lines are not given here as they are of no practical importance.

5.2.2 Point-to-Polygon Distance

In this section, we present an algorithm to compute the evolution of closest features between a moving point and a convex moving polygon in $\mathcal{O}(log(n) + k)$ time, where n is the number of vertices of the polygon and k is the size of the result. This algorithm returns a new data object of size k that allows successive computations of the distance to be done in $\mathcal{O}(log(k))$ time. The relation between k and n is discussed at the end of the section.

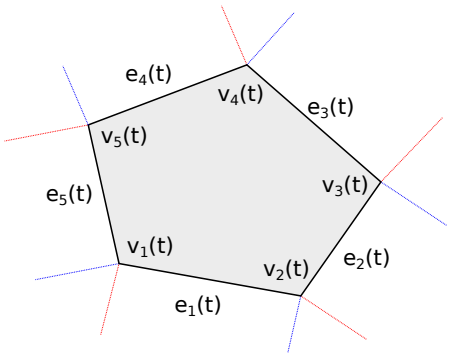


Fig. 5.3: Outer Voronoi diagram of a convex polygon

The general idea of the algorithm is to track the closest feature of the moving polygon to the moving point. The features of a polygon are its vertices and edges, and the closest feature of a polygon to a point is thus the vertex or the edge closest to this point. The outer Voronoi diagram of a convex polygon can be used to determine the closest feature given the position of the point. An example of an outer Voronoi diagram is shown in Fig. 5.3. In this figure, the Voronoi regions outside of the polygon are bounded by a red and a blue line, as well as an edge in half of the cases. The red and blue lines are identical to the ones in Fig. 5.2 but applied to the different edges of the polygon. This representation assumes that the vertices of the polygon are

listed in counter-clockwise order, as mentioned in Sect. 5.1. This assumption will hold in all future figures as well.

The return value of the algorithm is a mapping from time to feature as a list of tuples. Each tuple contains a timestamp $t \in [0, 1]$ and a feature $\mathcal{F}$, which can be either a vertex ($\mathcal{F} = v_j$) or an edge ($\mathcal{F} = e_j$) of the polygon. This list is sorted by increasing timestamp t. Note that this sorting does not have to be done in practice, as the list is already being computed in sorted order.

$$\mathcal{L} = [(t_0 = 0, \mathcal{F}_0), ..., (t_{k-1}, \mathcal{F}_{k-1}), (t_k = 1, \mathcal{F}_{k-1})] \tag{5.10}$$

At every timestamp t, with $t_0 \leq t < t_1$, the closest feature to the moving point is $\mathcal{F}_0$. At $t = t_1$, the closest feature becomes $\mathcal{F}_1$, and so on. In practice, the last tuple in $\mathcal{L}$ of Eq. 5.10 will be omitted since it is only there to describe the fact that the last feature $\mathcal{F}_{k-1}$ is valid from $t = t_{k-1}$ to $t = 1$.

With this data, the distance between the moving point and the moving polygon can be computed in $\mathcal{O}(log(k))$ time. Indeed, given any timestamp $t \in [0, 1]$, we can use binary search on the list to find the closest feature at that instant in $\mathcal{O}(log(k))$ time. With the closest feature and the moving point, we can then compute the distance in constant time. If the closest feature is a point, the Euclidean distance is used. If it is an edge, the distance is computed using Eq. 5.9. Note that in this case, we are certain that $0 < s(t) < 1$, and the *min* and *max* functions of Eq. 5.9 can thus be omitted.

Next, we describe the algorithm to compute the list $\mathcal{L}$ starting from a convex moving polygon and a moving point.

Algorithm

The algorithm consists of two parts. Firstly, the initial closest feature $\mathcal{F}_0$ is computed. This is a static problem and can be solved using known algorithms in $\mathcal{O}(log(n))$ time [88]. Secondly, given $(t_i, \mathcal{F}_i)$, the algorithm finds the next pair $(t_{i+1}, \mathcal{F}_{i+1})$ as described below. Starting from $(t_0 = 0, \mathcal{F}_0)$, the second part is then called repeatedly using the output of the previous call as input to the next, and this continues until no new closest feature is found at a time $t_i < t < 1$. The remaining problem consist thus of computing $(t_{i+1}, \mathcal{F}_{i+1})$, given $(t_i, \mathcal{F}_i)$ and both moving objects ($p(t)$ and $\mathcal{R}(t)$).

The input feature is either a vertex ($\mathcal{F}_i = v_j$) or an edge of the polygon ($\mathcal{F}_i = e_j$). Figure 5.4 displays both cases. The four possible transitions ((a) to (d)) of closest features are shown in Fig. 5.5. If the input closest feature is a vertex of the polygon $\mathcal{F}_i = v_j$ (Fig. 5.5, left), then the next closest feature will be either one of the two edges adjacent to the vertex: $\mathcal{F}_{i+1} = e_j$ or e_{j-1}. This corresponds to cases (a) and (b) of Fig. 5.5 respectively. The boundary lines are defined as in Sect. 5.2.1, and determining when either case (a) or (b)

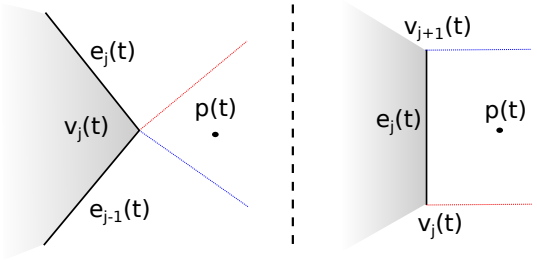


Fig. 5.4: Examples where the moving point is closest to a vertex (left) or an edge (right) of the moving polygon.

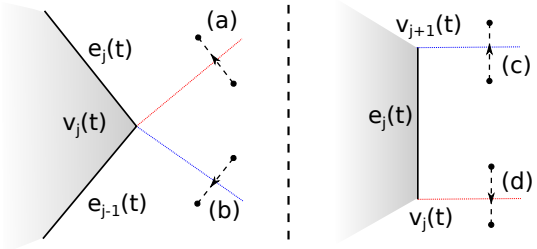


Fig. 5.5: Two possible transitions per type of initial closest feature.

happens can be done by solving Eq. 5.11 with respect to t. In these equations, $s_j(t)$ corresponds to Eq. 5.8 applied to the edge $e_j(t)$ and the point $p(t)$.

$$(a): \ s_j(t) = 0 \qquad\qquad (b): \ s_{j-1}(t) = 1 \tag{5.11}$$

If at least one of these equations has a solution in $[t_i, 1]$, then there will be a change in the closest feature. Let's denote the respective solutions t^a (Eq. 5.11, (a)) and t^b (Eq. 5.11, (b)). The algorithm will add (t^a, e_j) to the list $\mathcal{L}$ if $t^a < t^b$, and (t^b, e_{j-1}) otherwise. If neither equations have a solution in $[t_i, 1]$, this means that the closest feature will not change again before the end of the movement, and the algorithm will thus terminate and return the current list $\mathcal{L}$.

Similarly, if the initial closest feature is an edge of the polygon $\mathcal{F}_i = e_j$ (Fig. 5.5, right), then the next closest feature will be either the end or the start vertex of this edge $\mathcal{F}_{i+1} = v_{j+1}$ or v_j. This corresponds to cases (c) and (d) of Fig. 5.5 respectively. In these cases, the equations to solve are listed in Eq. 5.12.

$$(c): \ s_j(t) = 1 \qquad\qquad (d): \ s_j(t) = 0 \tag{5.12}$$

As in the previous case, we denote the solutions t^c and t^d respectively. The algorithm appends (t^c, v_{j+1}) to the list $\mathcal{L}$ if $t^c < t^d$, and (t^d, v_j) otherwise. If neither equations have a solution in $[t_i, 1]$, the algorithm terminates and returns $\mathcal{L}$.

From the definition of $s(t)$, these equations are nonlinear if the rotation of the polygon is nonzero. To solve these equations, numerical methods have to be utilized to find numerical approximations of the solutions. Possible methods include the Newton-Raphson method [10] or bracketing methods such as the false position method or the ITP method [61]. The implementation used in Sect. 5.6 uses the false position method to solve these nonlinear equations. In case the movement of the polygon contains no rotation, these equations become linear. Section 5.4 describes direct solutions to the equations under the assumption that the polygons move without rotation.

The examples and equations described in this section do not take into account special cases. For the point-to-polygon distance, three special cases have to be taken into account.

1. The moving point is on a boundary line of the Voronoi diagram at t_0.
2. The moving point enters the polygon through the edge e_j in the right image of Fig. 5.5.
3. The moving point enters the polygon through the vertex v_j in the left image of Fig. 5.5.

All three of these cases can be detected and handled accordingly. In case 1, the algorithm will have to decide if $\mathcal{F}_0$ corresponds to the left or right case of Fig. 5.4. This can be done by looking at a timestamp $t_0^* = t_0 + \epsilon$, right after $t_0 = 0$. Both Cases 2 and 3 correspond to an intersection between the moving polygon and the moving point. This is not allowed in the algorithm and, if one of these two cases is detected, the algorithm will fail. This behavior also exists in the polygon-to-polygon algorithm and can thus be used to detect if two moving objects intersect or not. Case 2 can be detected using Eq. 5.13, and Case 3 will happen if the solutions of both (a) and (b) in Eq. 5.11 are true for the same value of t (the moving point crosses the two boundary lines of the Voronoi region at the same time):

$$(p(t) - v_j(t)) \times (v_{j+1}(t) - v_j(t)) = 0, \tag{5.13}$$

where $v_1 \times v_2$ represents the cross product between two vectors.

Algorithm 5.1 summarizes the process of computing the list of closest features between a moving point and a convex moving polygon (without special cases). The complexity of computing the first closest feature is $\mathcal{O}(log(n))$ [88]. Assuming that solving the equations using numerical methods takes constant time, the complete algorithm will finish in $\mathcal{O}(log(n) + k)$ time, where k is the length of the list returned by the algorithm. This follows from the fact that the loop only solves two equations per iteration. The cost of running this loop will thus be the number of iterations times the cost of an iteration. As

Algorithm 5.1: Point to Polygon Distance Algorithm

Input: $\mathcal{R}(t), p(t)$
Output: $\mathcal{L}$, the list of closest features with timestamps.
begin
  $\mathcal{F}_0 \leftarrow$ Compute closest feature at $t_0 = 0$ [88];
  $\mathcal{L} \leftarrow [(0, \mathcal{F}^0)]$;
  while *True* **do**
    $(t_i, \mathcal{F}_i) \leftarrow$ Last element of $\mathcal{L}$;
    if $\mathcal{F}_i = v_j$ **then**
      $t^a, t^b \leftarrow$ Solve Eq. 5.11;
      if $t_i < t^a <= 1$ *and* $t^a < t^b$ **then**
        Append (t^a, e_j) to $\mathcal{L}$;
      else if $t_i < t^b <= 1$ **then**
        Append (t^b, e_{j-1}) to $\mathcal{L}$;
      else break ;
    else
      $t^c, t^d \leftarrow$ Solve Eq. 5.12;
      if $t_i < t^c <= 1$ *and* $t^c < t^d$ **then**
        Append (t^c, v_{j+1}) to $\mathcal{L}$;
      else if $t_i < t^d <= 1$ **then**
        Append (t^d, v_j) to $\mathcal{L}$;
      else break ;
  return $\mathcal{L}$;

mentioned, we assume that one iteration (solving one or two equations) takes constant time. Each iteration adds a single change of the closest feature to the list, and we exit the loop when no new change is found. The number of iterations is $\mathcal{O}(k)$, the result size. This gives us a total complexity of $\mathcal{O}(log(n)+k)$. The assumption that solving the nonlinear equations takes constant time is a simplification made to keep the notation of the complexity the same for both the rotating and the non-rotating cases. As can be seen in Sect. 5.6.1, this simplification is coherent with the actual run time of the algorithm. We observe that the nonlinear solution is about 5 to 8 times slower than the direct solution, which does not influence the global complexity of the algorithm. The value of k corresponds to the number of times the closest features switched during the movement and is thus dependent on the translation and rotation parameters of the moving objects. With the assumptions that $\theta \in [-\pi, \pi]$ for the moving polygon and that the moving point either translates linearly or has the equations of a vertex of another moving polygon, the complexity of k will in the worst case be linear in n. In practice, it is clear that if the polygon rotates at most 180 degrees, there cannot be more than n changes in the closest features. The worst-case complexity of this algorithm is thus $\mathcal{O}(n)$, and the best-case complexity is $\mathcal{O}(log(n))$ (if $k = 1$).

5.2.3 Polygon-to-Polygon Distance

This section describes an algorithm to compute the distance between two moving polygons $\mathcal{R}^A(t)$ and $\mathcal{R}^B(t)$, each having their own translation and rotation parameters $(dx^A, dy^A, \theta^A, x_c^A, y_c^A)$ and $(dx^B, dy^B, \theta^B, x_c^B, y_c^B)$. The algorithm is similar to the one described in Sect. 5.2.2 in the sense that it also tracks the closest feature between the moving objects. In the case of the distance between two polygons, however, we track two closest features instead of one.

The algorithm processes two moving polygons having n and m vertices respectively in $\mathcal{O}(log(n) + log(m) + k)$ time and returns a data objects that allows successive computations to be done in $\mathcal{O}(log(k))$ time. As for the point-to-polygon distance algorithm, the value of k corresponds to the size of the result set and depends heavily on n, m and the movement of both polygons. In particular, $k = \mathcal{O}(n + m)$ in the worst case, assuming that the rotation parameter of both polygons is a constant (e.g., $\theta^A \in [-\pi, \pi]$ and $\theta^B \in [-\pi, \pi]$).

The object returned by the algorithm is a list $\mathcal{L}$ of tuples, where each tuple contains a timestamp t and the closest features $\mathcal{F}^A$ and $\mathcal{F}^B$ of the two polygons at that instant.

$$\mathcal{L} = [(t_0 = 0, \mathcal{F}_0^A, \mathcal{F}_0^B), ..., (t_{k-1}, \mathcal{F}_{k-1}^A, \mathcal{F}_{k-1}^B)] \tag{5.14}$$

The first step of the algorithm consists of computing the closest features $\mathcal{F}_0^A$ and $\mathcal{F}_0^B$ at $t_0 = 0$. This is a well-known static problem and can be solved in $O(log(n) + log(m))$ time [88]. The rest of the tuples in the list are then computed by iteratively determining when the next change in closest features happens, and what the new closest features are. This process is similar to the one described in Alg. 5.1. In this case, however, the closest feature transitions are more complex than in the point-to-polygon case. Each change in the closest feature will still be detected in constant time, and the algorithm will exit when no new change in the closest features is detected. The loop in the algorithm has thus a complexity of $\mathcal{O}(k)$, which results in an algorithm of complexity $\mathcal{O}(log(n) + log(m) + k)$.

The initial state of the closest features pair can be either vertex-vertex, vertex-edge, edge-vertex or edge-edge, where the first and second terms denote the types of the closest feature on polygons $\mathcal{R}^A$ and $\mathcal{R}^B$ respectively. The vertex-edge and edge-vertex cases can be handled similarly by simply swapping the two polygons in the equations, and the edge-edge case is a special case that will be discussed later. Figure 5.6 thus displays the two main cases for the types of initial closest features: vertex-vertex (left) and edge-vertex (right). Figure 5.7 shows two examples of the special case where the two closest features are parallel edges of the polygons.

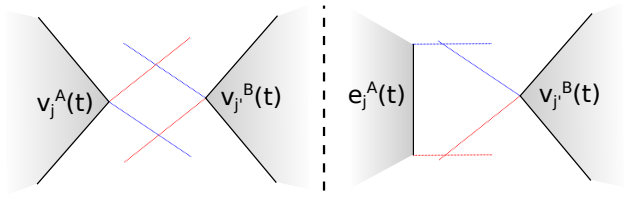


Fig. 5.6: Initial closest features between two moving polygons: vertex-vertex (left) or edge-vertex (right).

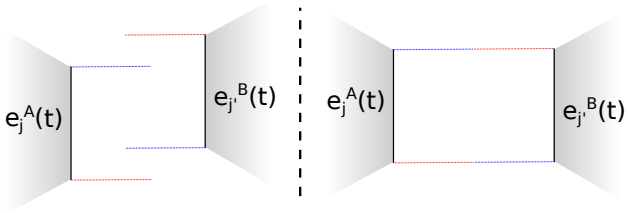


Fig. 5.7: Two examples of parallel edges as closest features between two polygons. (Right) is a special case of (left).

In this section, the indices $j \in \{1, ..., n\}$ and $j\prime \in \{1, ..., m\}$ will be used for the vertices and edges of polygons $\mathcal{R}^A$ and $\mathcal{R}^B$ respectively. The index $i \in \{0, ..., k-1\}$ will still be used to denote the different tuples of $\mathcal{L}$.

Starting from one of the two states in Fig. 5.6, three transitions in the closest features are possible.

- Vertex-vertex to edge-vertex
- Edge-vertex to vertex-vertex
- Edge-vertex to another edge-vertex

These transitions and their corresponding equations are explained in the following subsections. Since all of the transitions keep the closest feature pairs in either the vertex-vertex or edge-vertex state, the final list $\mathcal{L}$ will only contain vertex-vertex or edge-vertex pairs. This is important for future sections and this constraint can be met even when handling special cases, e.g., when two edges are parallel and closest.

Vertex-Vertex ↔ Edge-Vertex

Figure 5.8 shows the possible transitions that will cause the closest features to go from the vertex-vertex case to the edge-vertex case or back. For clarity, we change the notation of Eq. 5.8 once again and write function $s(t)$ as $s(v, e)(t)$, to specify which moving vertex/point and moving edge are being used. The cases (a) to (f) in Fig. 5.8 can thus be detected by solving Eqs. 5.15 to 5.17.

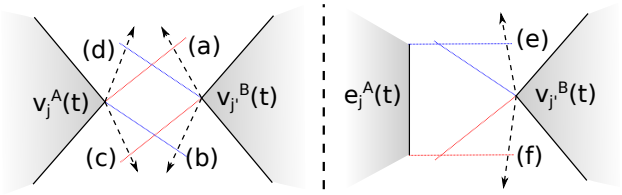


Fig. 5.8: Evolution of the closest features: vertex-vertex to edge-vertex (left) or edge-vertex to vertex-vertex (right).

$$(a): \ s(v_{j'}^B, e_j^A)(t) = 0 \qquad (b): \ s(v_{j'}^B, e_{j-1}^A)(t) = 1 \tag{5.15}$$

$$(c): \ s(v_j^A, e_{j'}^B)(t) = 0 \qquad (d): \ s(v_j^A, e_{j'-1}^B)(t) = 1 \tag{5.16}$$

$$(e): \ s(v_{j'}^B, e_j^A)(t) = 1 \qquad (f): \ s(v_{j'}^B, e_j^A)(t) = 0 \tag{5.17}$$

If $(\mathcal{F}_i^A, \mathcal{F}_i^B) = (v_j^A, v_{j'}^B)$, Eqs. 5.15 and 5.16 have to be solved to determine what the next pair of closest features will be. The edge-vertex pair used in the function $s(v, e)(t)$ corresponds to the next pair of closest features in that particular case. The equation in Eqs. 5.15 and 5.16 that has the earliest solution $t \in [t_i, 1]$ will thus determine uniquely the next pair of closest features. For example, if the equation corresponding to case (a) has the earliest solution $t^a \in [t_i, 1]$, then the next pair of closest features is $(\mathcal{F}_{i+1}^A, \mathcal{F}_{i+1}^B) = (e_j^A, v_{j'}^B)$.

In the edge-vertex case, the process is similar. Equation 5.17 has to be solved for cases (e) and (f) to determine what the next pair of features will be. If the equation for case (e) has the earliest solution between t_i and 1, the next pair of features is $(\mathcal{F}_{i+1}^A, \mathcal{F}_{i+1}^B) = (v_{j+1}^A, v_{j'}^B)$. Otherwise, the next pair of features is $(\mathcal{F}_{i+1}^A, \mathcal{F}_{i+1}^B) = (v_j^A, v_{j'}^B)$.

Special cases similar to the ones discussed in Sect. 5.2.2 can also happen here. Since they are solved in the same way as explained in Sect. 5.2.2, they are not discussed further.

Edge-Vertex ↔ Edge-Vertex

Another possibility when starting from the edge-vertex case is that the edge e_j^A becomes parallel to one of the edges adjacent to the vertex $v_{j'}^B$. If this happens, the edges will be parallel during a single timestamp t, and the next closest features will be a new edge-vertex pair. Figure 5.9 shows an example of this, where the edge e_i^A becomes parallel to the edge $e_{j'}^B$ during a single timestamp, and the closest features thus evolve from $(e_j^A, v_{j'}^B)$ to $(v_j^A, e_{j'}^B)$ at that timestamp. This is only 1 of the 4 possible transitions starting from $(e_j^A, v_{j'}^B)$. The four possible transitions are listed below.

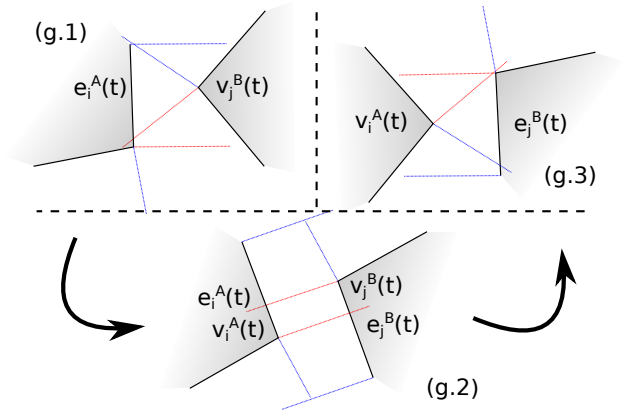


Fig. 5.9: Evolution from an edge-vertex pair (g.1) to another edge-vertex pair (g.3), by transitioning through an edge-edge pair (g.2).

i $(e_j^A, v_{j'}^B) \rightarrow (v_j^A, e_{j'}^B)$
ii $(e_j^A, v_{j'}^B) \rightarrow (v_{j+1}^A, e_{j'-1}^B)$
iii $(e_j^A, v_{j'}^B) \rightarrow (e_j^A, v_{j'+1}^B)$
iv $(e_j^A, v_{j'}^B) \rightarrow (e_j^A, v_{j'-1}^B)$

To determine when edge e_j^A becomes parallel to either $e_{j'}^B$ or $e_{j'-1}^B$, Eqs. 5.18 and 5.19 have to be solved.

$$(v_{j+1}^A(t) - v_j^A(t)) \times (v_{j'+1}^B(t) - v_{j'}^B(t)) = 0 \tag{5.18}$$
$$(v_{j+1}^A(t) - v_j^A(t)) \times (v_{j'}^B(t) - v_{j'-1}^B(t)) = 0 \tag{5.19}$$

If Eq. 5.18 is true for some t, then e_j^A is parallel to $e_{j'}^B$ at that timestamp, and either case (i) or (iii) of the transitions listed above will happen. These two cases can be distinguished by determining which of the two vertices v_i^A or $v_{j'+1}^B$ is closest to the other edge at t. In the example given in Fig. 5.9, v_i^A is closer to $e_{j'}^B$ than $v_{j'+1}^B$ is to e_i^A, and this corresponds to transition (i) in the list. With two possible transitions per adjacent edge of $\mathcal{R}^B$, we thus get to the four possible transitions listed above.

Just like in Sect. 5.2.2, the equations listed in this section are nonlinear in the general case and have to be solved using numerical methods. For all equations, only the first solution between t_i and 1 is required, with $0 \leq t_i < 1$. If the movement of the polygons does not involve a rotation, these equations become linear and Sect. 5.4 describes how they can be solved directly without using numerical methods.

Special Cases

Next to the two start states shown in Fig. 5.6 and the transitions detailed in Sects. 5.2.3 and 5.2.3, a multitude of special cases exist. Similarly to the ones in Sect. 5.2.2, they can be detected and handled in constant time. Below is a non-exhaustive list of these special cases.

- Two edges of the polygons are parallel and closest at $t_0 = 0$, as shown in Fig. 5.7.
- The two polygons have the same rotation speed, and two edges are parallel during the complete movement instead of at a single timestamp.
- The polygons do not rotate. See Sect. 5.4.
- An edge-vertex pair evolves to a vertex-vertex pair by transitioning through an edge-edge pair. This is a special case of Sect. 5.2.3.

The first special case essentially corresponds to starting from the state (g.2) in Fig. 5.9. In this case, the problem consists of determining the next pair of closest features without knowing what the previous pair was. Notice that we are looking for state (g.3) without knowing state (g.1). This can be solved by looking at the closest features at a timestamp $t_0^* = t_0 + \epsilon$, right after $t_0 = 0$, similarly to what was done for a special case of Sect. 5.2.2.

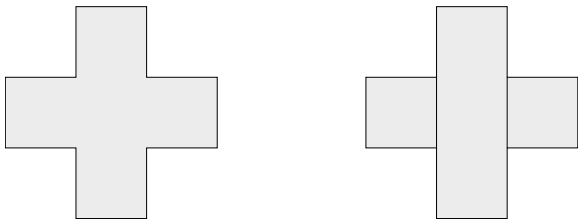
The rest of the special cases are not detailed here, but can also be solved in constant time using methods similar to the ones presented here and in Sect. 5.2.2. Note that these cases will for the most part never happen when running the algorithm on random or real-world data. For example, even when handling fast-rotating polygons with up to 500 vertices as in the experiments of Sect. 5.6.1, more than 99% of the runs did not encounter a single one of these special cases.

5.3 Non-Convex Polygons

The algorithms presented in Sects. 5.2.2 and 5.2.3 assume that the moving polygons are convex, which allows solutions in linear time in terms of number of vertices. In this section, we present an algorithm for non-convex polygons that makes use of the previously presented algorithms for convex polygons. The algorithm for non-convex polygons consists of three steps. Firstly, the polygons are decomposed into convex parts (Sect. 5.3.1). Secondly, partial solutions are computed on the convex parts (Sect. 5.3.2). Lastly, the partial solutions are merged into a final solution (Sect. 5.3.3).

5.3.1 Convex Decomposition

The first step of the algorithm consists of decomposing the non-convex polygons into convex parts. This has to be done for both polygons in the case of the polygon-to-polygon distance operation. Even if the polygons are moving, the decomposition only has to be done on the static polygon at $t = 0$. The movement of the convex parts will then be determined using the same parameters as the movement of the initial polygon. Specifically, if the initial polygon has a nonzero rotation, all convex parts will also have a nonzero rotation. The rotation parameters of each convex part are then defined using the same rotation center and rotation angle as the initial polygon.



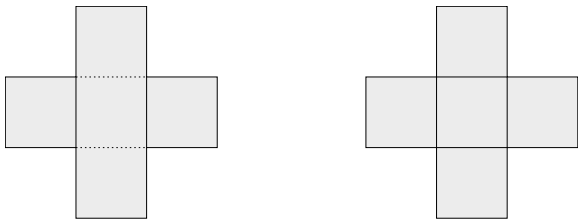
(a) (b)

Fig. 5.10: Non-convex object (a), with an optimal decomposition in 3 convex parts (b)

Convex polygon decomposition is a well-known problem in computation geometry. Chazelle [12] and Keil [46] present two optimal solution in respectively $\mathcal{O}(n + r^3)$ and $\mathcal{O}(r^2 nlog(n))$ time, where r corresponds to the number of reflex vertices (with an inside angle $> 180°$) and n is the total number of vertices. A more practical algorithm is the Hertel-Mehlhorn algorithm [42], which returns a convex decomposition in $O(n + rlog(r))$ time, with at most 4 times the optimal number of convex parts. We will write the number of convex parts of a polygon using the uppercase letter of its number of vertices. For example, a polygon with n vertices will have N convex parts when decomposed using Hertel-Mehlhorn. Figure 5.10 shows an example of a non-convex polygon, as well as its optimal decomposition in convex parts. In the example shown, we have $N = 3$.

Note that the decomposition required to compute the temporal distance is less restrictive than a traditional convex decomposition problem. For example, Fig. 5.11a, shows an example of a decomposition in only 2 overlapping convex parts. This is less than the optimal convex decomposition without overlap, which is beneficial for the next step. Another case is when a convex part does not contain any edges of the initial non-convex polygon. In this case, that specific convex part can be omitted, as there will always be at least one other part with a smaller distance to the second object. An exam-

ple of such a case is shown in Fig. 5.11b, where the center square can be omitted. In this specific case, the omitted part is convex as well, but this is not a requirement since it does not participate in the distance computation. Determining a good (or optimal) convex polygon partitioning, taking into account overlapping and superfluous parts, is left as future work.



(a) (b)

Fig. 5.11: (a) Overlapping or (b) superfluous parts in the convex decomposition

5.3.2 Computing Partial Solutions

With the moving polygons being decomposed in convex parts, the next step consists of computing partial solutions using the previously explained algorithms for convex polygons. For the point-to-polygon distance, this has to be done once per convex part. For the polygon-to-polygon distance, every part from one polygon has to be combined with every part of the other polygon. With the two polygons having N and M convex parts respectively, the algorithms for convex polygons will thus be called N and $N * M$ times, for the point-to-polygon and polygon-to-polygon problems respectively. As discussed in the previous subsection, only convex parts that share an edge with the initial non-convex polygon need to be part of this computation.

The related work in Sect. 2.5 mentions techniques for non-convex polygons making use of bounding hierarchies to reduce the total number of computations necessary. Similar techniques could be used in this step to improve the running time of the algorithm, but this is left as future work.

5.3.3 Merging Partial Solutions

The last step of the non-convex algorithm consists of merging the partial solutions into a final output. Let's denote the number of partial solutions

L. L will either be equal to N or $N \times M$, depending on which problem is being solved. Every partial solution consists of a list of $k_l (l \in \{0, ..., L-1\})$ pairs of closest features with their corresponding timestamps. We will also assume that in the point-to-polygon distance problem, the second feature in the closest feature pair always contains the moving point. The two problems can thus be solved using the same algorithm. The remaining problem consists of finding for every timestamp between 0 and 1, which solution contains the actual closest features of the non-convex problem. Figure 5.12 shows an example of when a closest feature changes from one convex part to another. The green line is a type of boundary in the Voronoi diagram of the non-convex polygon that does not exist in the convex case (Fig. 5.3).

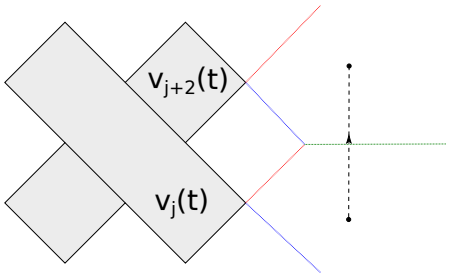


Fig. 5.12: Evolution of closest features for a non-convex polygon

The problem is solved in multiple steps. Firstly, for every partial solution, the distance is computed at every timestamp in the list of closest features, as well as at $t = 1$. In total these are $K = \sum_{l=0}^{L-1} k_l + 1$ distance computations. We now have L lists of tuples, where each tuple contains a timestamp, a pair of closest features, and a distance value. The timestamp and distance values can thus be seen as piecewise-linear functions, where every linear piece has an associated pair of closest features. In this representation, the distance between two computed instants is assumed to be linearly interpolated between the previous and the next instant. Figure 5.13 shows an example with two partial solutions.

In the second step, we track the minimum of these piecewise-linear solutions using a sweep-line algorithm such as the Bentley–Ottmann algorithm [18]. The sweep-line will have to maintain a binary search tree with L elements (the L solutions) throughout the complete algorithm. An event will be either a change in segments from a single solution or a crossing between two solutions. The main purpose of this step is to determine when the actual closest features switch from one convex part to another. This corresponds in the sweep-line algorithm to a crossing between the two lowest elements in the binary search tree. These crossings can be detected using the linear approximation of the distance, but the actual timestamps at which these solutions swap still have to be computed. In Fig. 5.13, for example, one swap in convex

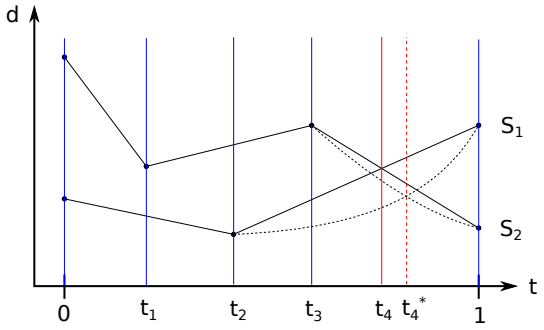


Fig. 5.13: Piece-wise linear functions of two partial solutions. The dotted red line corresponds to the actual timestamp t_4^* (computed using Eq. 5.22) at which the solutions switch.

parts takes place at the red vertical line. To determine when the actual swap takes place (the red dotted line), however, the real distance functions (the black dotted lines) have to be used instead of the linear approximation.

Since every linear segment of the linear approximation is associated with a pair of closest features, this can be done by finding the timestamps where the distance between the pair associated to the first segment is equal to the distance between the pair associated to the second segment. Depending on the type of closest features, the distance functions will be different. If the closest features are two moving vertices $v^A(t) = (x^A(t), y^A(t))$ and $v^B(t) = (x^B(t), y^B(t))$, then the (squared) distance function corresponds to Eq. 5.20. The distance between a moving vertex $v^A(t) = (x^A(t), y^A(t))$ and a moving edge $e^B(t)$ is equal to the distance between the vertex and the closest point on the edge. From Sect. 5.2.1, we know that the closest point on the edge is given by $e^B(s(t), t) = (x^B(s(t), t), y^B(s(t), t))$, where $e^B(s, t)$ is an arbitrary point on the edge, and $s(t)$ is given by Eq. 5.8. The (squared) distance is thus given by Eq. 5.21.

$$d_{AB}^2(t) = (x^B(t) - x^A(t))^2 + (y^B(t) - y^A(t))^2 \tag{5.20}$$

$$d_{AB}^2(t) = (x^B(s(t), t) - x^A(t))^2 + (y^B(s(t), t) - y^A(t))^2 \tag{5.21}$$

To determine when the closest features change from $(\mathcal{F}^A, \mathcal{F}^B)$ to $(\mathcal{F}^C, \mathcal{F}^D)$, we thus have to solve Eq. 5.22 between t_i and t_{i+1}, where the actual distance function of both pairs of features is either Eq. 5.20 or 5.21.

$$d_{AB}^2(t) - d_{CD}^2(t) = 0 \tag{5.22}$$

The final step of the merging part consist of creating the final list of closest features. This process is straightforward, since after solving Eq. 5.22 for the different switches between minimum solutions, the minimum/closest solution

is known for every instant between 0 and 1. In the example in Fig. 5.13, S_1 is closest when $t \in [0, t_4^*]$ and S_2 is closest when $t \in [t_4^*, 1]$.

To summarize the more complex case of polygon-to-polygon distance, the first step consists of decomposing both polygons into convex parts. This will create N and M convex parts respectively. The second step calls the algorithm for convex polygons using every combination of convex parts of the two initial polygons. This results in $N * M$ partial solutions. Finally, these solutions are merged together. A sweep-line algorithm is used to compute the intersections between the solutions, and the exact timestamps of the intersections between the two minimum solutions are computed using Eq. 5.22. The final solution is then created by combining the partial solutions at the times where they are the actual closest solution in the non-convex problem.

5.4 Non-Rotating Polygon Optimization

Section 5.1 assumes that the movement of the polygons is a combination of a translation and a rotation. The rotation is the cause of the non-linearity of the equations in Sect. 5.2. If the movement of the polygons only contains a translation, then the movement of its vertices is linear, and most of the equations presented in this chapter become linear or polynomial. If this is the case, then the equations present direct solutions and can thus be solved without numerical methods. Without directly impacting the complexity of the algorithms, this can still significantly improve their performance. Real-world examples where this could be of use include logistics robots, objects moving on fixed tracks, cars on a highway and more. Whenever an object has a rotation $\theta < \epsilon$ between two timestamps, these direct solutions can be used to speed up the algorithms.

With the movement of the polygons being defined using only translation parameters, the equations of the moving vertices defined in Eq. 5.5 can be simplified to Eq. 5.23.

$$\begin{aligned} x_i(t) =& x_i + t * dx \\ y_i(t) =& y_i + t * dy \end{aligned} \tag{5.23}$$

With the equations of the vertices being linear, Eq. 5.6 can be rewritten as Eq. 5.24.

$$\begin{aligned} e(s,t) &= v_s(t) * (1-s) + v_e(t) * s, \ s \in [0,1] \\ &= v_s * (1-s) + v_e * s + t * (dx, dy) \end{aligned} \tag{5.24}$$

Two important equations that need solving are $s(t) = 0$ and $s(t) = 1$, with $s(t)$ as defined in Eq. 5.8. As a reminded, $p(t) = (x_p(t), y_p(t))$ is the moving point, and $v_s(t)$ and $v_e(t)$ are the start and end points of the moving segment. We will call (dx_p, dy_p) the translation of the moving point, and (dx, dy) the

translation of the edge. Examples where these equations are required for different combinations of moving point and moving edge are Eqs. 5.11, 5.12, 5.15, 5.16, and 5.17. The direct solutions for these equations are shown in Eqs. 5.25 and 5.26. These equations only have solutions if the movement of the point is not perpendicular to the edge ($(dx_p - dx) * (x_e - x_s) + (dy_p - dy) * (y_e - y_s) \neq 0$).

$$s(t) = 0 \Leftrightarrow$$

$$t = \frac{(x_s - x_p) * (x_e - x_s) + (y_s - y_p) * (y_e - y_i)}{(dx_p - dx) * (x_e - x_s) + (dy_p - dy) * (y_e - y_s)} \tag{5.25}$$

$$s(t) = 1 \Leftrightarrow$$

$$t = \frac{(x_e - x_p) * (x_e - x_s) + (y_e - y_p) * (y_e - y_s)}{(dx_p - dx) * (x_e - x_s) + (dy_p - dy) * (y_e - y_s)} \tag{5.26}$$

When working with rotating polygons, it is possible that two edges become parallel, and that the closest features change at that moment (Sect. 5.2.3). In this case, since the polygons do not rotate, two edges of different polygons are either always or never parallel. Equations. 5.18 and 5.19 thus do not need to be solved for t anymore.

Using Eqs. 5.25 and 5.26, the algorithms of Sects. 5.2.2 and 5.2.3 can thus be run without using numerical methods. The experimental evaluation on maritime data in Sect. 5.6.2 shows that more than half the movement segments are non-rotating, thus the impact of this performance optimization is significant.

5.5 Implementation in a Moving Objects Database

In Sect. 5.2 we presented a general solution to the problem of computing the temporal distance between two moving points or polygons. The described algorithms return a list of k closest features with their associated timestamps. This list of closest features, together with the corresponding moving objects, completely defines the equation of the temporal distance between these moving objects. Indeed, the temporal distance is a piece-wise defined function, where every piece is defined using either Eq. 5.20 or Eq. 5.21. To compute the distance value at a given timestamp t, we can thus determine the closest features at that timestamp in $\mathcal{O}(log(k))$ time using a binary search on the list of features. Given the closest features at t, we then use the corresponding equation (Eq. 5.20 or 5.21) to determine the distance value in $\mathcal{O}(1)$ time.

When using these algorithms to implement a distance operator in a moving object database, the returned distance function still needs to be transformed

into the data model used in the database. In this section, we describe how the algorithm of Sect. 5.2 has been used to implement various distance operators in MobilityDB.

5.5.1 Linear Approximation of the Distance

MobilityDB defines the `tfloat` type, which is used to store the temporal evolution of a real-valued parameter as a piecewise linear function. This temporal type is used for example to store temperature measurements, the speed of a moving object, etc. It is also the return type of the distance operators that involve temporal objects, e.g., the operator computing the distance between two moving points.

Since the temporal distance between two moving objects is not piecewise linear, we need to compute and store its linear approximation. This linear approximation will accurately store the correct start and end distance values of every movement segment, as well as the extreme points of the function, i.e., all minima and maxima. An example of such a linear approximation of a nonlinear function is shown in Fig. 5.14 as the red line. This choice of linear approximation is the same as the one already done in MobilityDB when computing the distance between two moving points [91]. Since this approximation maintains the extreme points of the distance function it can be used to compute the exact nearest approach distance between two moving bodies. This is also explained in Sect. 5.5.2.

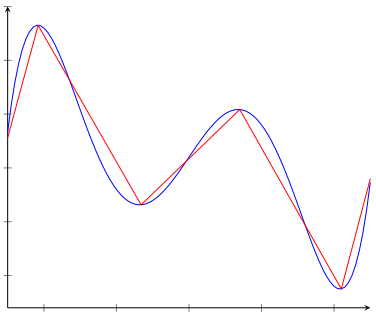


Fig. 5.14: Approximation of a nonlinear continuous function (in blue) using a piecewise-linear function (in red)

The previously presented distance algorithm returns a list $\mathcal{L}$ representing the evolution of closest features between two moving objects. In case the distance is computed between a moving point and a moving polygon, we can assume that the second feature always corresponds to the moving point itself.

$$\mathcal{L} = [(t_0 = 0, \mathcal{F}_0^A, \mathcal{F}_0^B), ..., (t_{k-1}, \mathcal{F}_{k-1}^A, \mathcal{F}_{k-1}^B), (t_k = 1, \mathcal{F}_{k-1}^A, \mathcal{F}_{k-1}^B)] \quad (5.27)$$

As explained in Sect. 5.2.3, the closest feature pairs stored in the list can either be two moving vertices or a moving vertex and a moving edge. Between every pair of timestamps $[t_i, t_{i+1}]$, the closest features remain the same. In case these closest features are two moving vertices, the (squared) distance function is described by Eq. 5.20. If the two features are a moving vertex and a moving edge, the function corresponds to Eq. 5.21.

The linear approximation of the distance function is computed in three steps. Firstly, for every period $[t_i, t_{i+1}], i \in \{0, ..., k-1\}$, the time coordinates of the extreme points are computed for the corresponding distance function. This is done by computing the derivative of the (squared) distance functions and finding the roots of this new function between t_i and t_{i+1}. Again, this has to be solved using numerical methods if the rotation is nonzero. In this case, however, we are looking for all the roots, instead of only the first. If there is no rotation, the distance functions will be quadratic, and there will thus be a single extreme point. The developments for this direct solution are omitted here.

In the second step, the distance is computed at every $t \in \{t_0, ..., t_k\}$, as well as at every timestamp returned by the first step. These values are stored in a list of time-value pairs, sorted by increasing timestamp. Finally, the intermediate timestamps $t \in \{t_1, t_{k-1}\}$ that do not correspond to an extreme point are removed from the list. The extreme point condition can be checked by looking at the values of the previous, current and next timestamps. The current point will then be extreme if its value is either smaller or larger than both other values. This last check is required in case the switch from one closest feature to another does not happen at an extreme point.

This algorithm results in a list of time-value pairs that stores a linear approximation of the distance between two moving bodies. This is also the MobilityDB representation of a `tfloat` value that does not contain temporal gaps. The temporal distance operator in MobilityDB will thus combine the algorithms described in Sect. 5.2 with the linear approximation algorithm to return a `tfloat` representing the temporal distance between its two operands.

5.5.2 Distance Functions and Operators

The previous section describes how we implement the *tDistance* operator in MobilityDB. Additionally, we also define three additional operators: *nearestApproachDistance*, *nearestApproachInstant* and *shortestLine*, with the signature in Table. 5.1. All three of these operators are defined in the OGC Moving Features Access standard [60]. The types `tgeompoint` and `tgeometry`

are the types representing moving points and moving polygons respectively in MobilityDB. For brevity, let's also use the notation $\mathcal{A}$ to denote the set of types `{tgeompoint, tgeometry, point, polygon}`.

Table 5.1: List of implemented distance operators

tDistance	`tgeometry` $\times \mathcal{A}$	$\rightarrow$	`tfloat`
nearestApproachDistance	`tgeometry` $\times \mathcal{A}$	$\rightarrow$	`float`
nearestApproachInstant	`tgeometry` $\times \mathcal{A}$	$\rightarrow$	`timestamptz`
shortestLine	`tgeometry` $\times \mathcal{A}$	$\rightarrow$	`geometry`

NearestApproachDistance is an operator that, given two moving objects, returns the smallest distance ever obtained between them during their movement. This operator is important as it is used for nearest-neighbor searches. Its implementation first computes the distance function using the *tDistance* operator and then finds the minimum value of the result using the MobilityDB *minValue* function. Indeed, as described in Sect. 5.5.1, the linear approximation of the distance maintains all the extreme points of the function. The *nearestApproachDistance* operator will thus return the correct value despite the linear approximation step.

The *nearestApproachInstant* operator is complementary to the *nearestApproachDistance* operator, as it computes the timestamp at which the nearest approach distance is obtained. If this happens more than once, the first timestamp is returned. Lastly, the *shortestLine* operator computes the shortest line string between the two moving objects at the nearest approach instant.

5.6 Experimental Validation

In this section, we evaluate the performance of the distance operator as it is described in Sects. 5.2 and 5.5. The section is divided into two parts. Firstly, we validate the analytical complexity of the algorithms described in Sect. 5.2. This is done on synthetic data, as this allows us to control the size and the movement parameters of the polygons. In the second step, we compare the existing MobilityDB distance function for moving points to the newly implemented distance operator for moving polygons. Since moving points have a more simple data model and distance function, we use them as a baseline to get an insight into the performance of the proposed moving body distance function in practice. This second experiments section uses the maritime dataset introduced in Sect. 3.2.2.

5.6.1 Validation of Computational Complexity

In this section, we evaluate the complexity of the algorithms of Sects. 5.2.2 and 5.2.3, as well as the speed-up received by the optimized version when the polygons are non-rotating. The algorithms are coded in Python 3.6 and the experiments are repeated for both the point-to-polygon and the polygon-to-polygon problems. The code and explanations to replicate these results are available on GitHub.[1]

In the point-to-polygon case, a random convex polygon with n vertices is generated, using the algorithm in [84],[2] in the box $(x_{min}, y_{min}, x_{max}, y_{max}) = (0, 0, n, n)$, with a translation $(dx, dy) = (0, n)$ and a rotation θ. The random point is then generated in the box $(2n, n, 3n, 2n)$, with a translation $(0, -n)$ and the first algorithm is applied to these two moving objects. For the polygon-to-polygon problem, we apply the same generation technique but generate a second polygon with $m = n$ vertices in the box $(2n, n, 3n, 2n)$, with a translation $(0, -n)$ and the same rotation θ as the other polygon. In both cases, the rotation center of the polygons corresponds to their centroid. These starting and movement conditions allow us to vary the number of vertices n and the rotation θ of the polygons while making sure that the moving objects do not intersect. We vary the number of vertices n between 3 and 500 and the rotation θ between 0 and π.

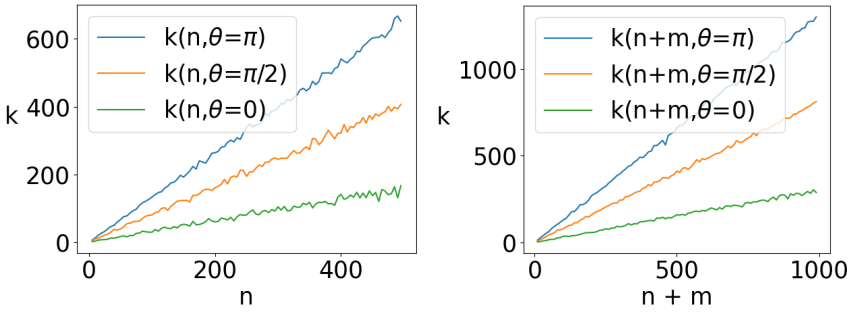
Two elements of the algorithms can be analyzed: the size k of the result and the runtime t of the algorithm. Firstly, we analyze the size of the result with respect to the parameters θ and n (remember that $m = n$ in the experiments). Figure 5.15 shows the graphs of k as function of n (point-to-polygon problem) or $n + m$ (polygon-to-polygon problem), for varying values of θ. We can see that the size of the result is linear in the number of vertices, but that the actual value of k also heavily depends on θ. The results are averaged over multiple runs with the same values for n and θ.

Secondly, we analyze the runtime t of the algorithms with respect to the result size k. This is done by running the algorithms for random values of n and θ, and storing the size of the result and runtime of the algorithms as (k, t) pairs in a list. The list is then sorted by k and displayed on the graph. For this experiment, only the *While* loop of Alg. 5.1 is timed. The computation of the initial closest feature in $\mathcal{O}(log(n))$ is omitted, as it is done using existing algorithms. We thus expect the time t to be linear in k as detailed in Sects. 5.2.2 and 5.2.3.

The results of these experiments can be seen in Fig. 5.16. The blue and orange lines correspond to the point-to-polygon and the polygon-polygon cases respectively, and Figs. 5.16a and 5.16b show the result for the non-optimized ($\theta \neq 0$) and optimized ($\theta = 0$) algorithms respectively. As expected, the durations of the different algorithms are all linear with respect to their result

[1] `https://github.com/mschoema/tgeometry_python`

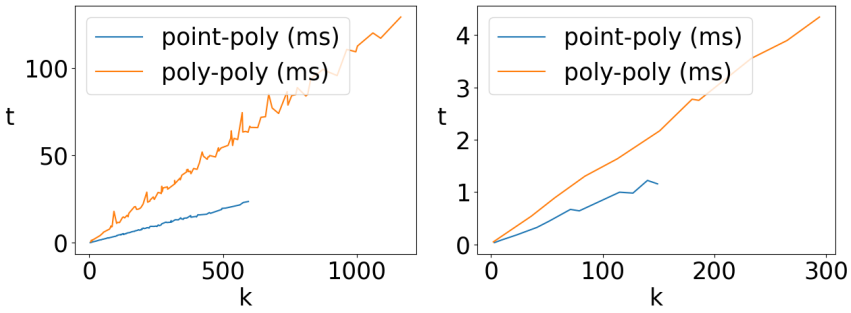
[2] `https://cglab.ca/~sander/misc/ConvexGeneration/convex.html`



(a) Point-to-polygon (b) Polygon-to-polygon

Fig. 5.15: Result size k as a function of the number of vertices n or n+m for fixed value of θ.

sizes ($t = \mathcal{O}(k)$). The average duration per result (t/k) for the non-optimized algorithm is $t/k = 38$ μs and 100 μs for the point-to-polygon and polygon-to-polygon problems respectively. The respective average duration per result for the optimized algorithms is $t/k = 7$ μs and 13 μs. The optimized algorithms are thus about 5 to 8 times faster than the non-optimized ones for identical result sizes.



(a) Non-optimized ($\theta \neq 0$) (b) Optimized ($\theta = 0$)

Fig. 5.16: Duration t (in ms) as a function of the result size k for the non-optimized and optimized algorithms.

5.6.2 Maritime Use Case

With Sect. 5.6.1 confirming the computational complexities of the algorithms presented in Sect. 5.2, it remains to assess the running time of a distance query in a real-world use case. For this, we test the MobilityDB implementation of the *tDistance* operator on a real-world AIS dataset from the Danish Maritime Authority (Sect. 3.2.2). We use the CSV files from September 2020, which contain a total of 250 M AIS points in 40 K ship tracks spread over 30 days, for a total of 61.6 GB of raw data. The data is cleaned and loaded in MobilityDB in both moving point and moving polygon format. The moving point data is constructed using the Timestamp, Longitude and Latitude fields of the AIS data. We constuct the moving polygons using additionally the Size A, B, C, D, and Heading fields like detailed before. This construction can be seen in Fig. 5.17. Since the position, size, and orientation information is extracted from the AIS data, we can assume that the moving polygon closely approximates the real movement and geometry of the vessel.

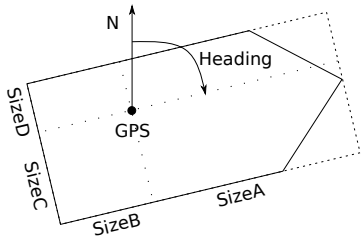


Fig. 5.17: Construction of the geometry of a vessel from its position, size and heading information

The experiments in this section thus correspond to the case where $n = 5$. The two constructed tables are the following:

Query 5.1 *Constructed tables*

```
CREATE TABLE ships_point(mmsi integer, trip tgeompoint);
CREATE TABLE ships_poly(mmsi integer, trip tgeometry);
```

After this transformation into a MobilityDB data format, the table sizes are respectively 4.4 GB and 5.5 GB. We now compare the existing distance operator for moving points with the newly implemented operator for moving polygons.

As a first example use-case, let us analyze the entrance of the port of Helsingborg. For security reasons, when entering the port, the vessels should not come too close to the dyke at the entrance of the port. Queries 5.2 and 5.3 thus computes the temporal distance between the vessels stored as moving points or polygons and a fixed point at the end of the dyke at the entrance

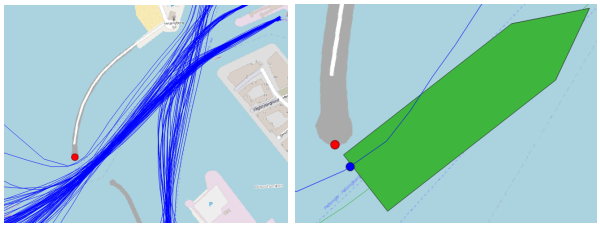
of the Helsingborg port. Figure 5.18a shows the position of this point as well as the trajectories of the vessels around it. For demonstration purposes, the queries are only run on the first day of data. The running time of Query 5.2 is 3.65 seconds (average of 10 runs), while the same query on moving bodies (Query 5.3) takes 5.06 seconds. Note that this is the total running time of applying the algorithm on the 6.5 M segments that make up the movement of the vessels. The distance operator for moving polygons is thus about 1.4 times slower than the operator for moving points, which is to be expected due to the increased complexity of the algorithm. Another thing to note here is that for the trips of table `ships_poly`, 65.7% of the segments have a rotation angle of 0. This indicates that the optimized solution for non-rotating polygons is heavily used in real-world use cases.

Query 5.2 *Temporal distance query on temporal point data*

```
SELECT tDistance(trip, 'Point(729493 6216766)') FROM ships_point;
```

Query 5.3 *Temporal distance query on temporal geometry data*

```
SELECT tDistance(trip, 'Point(729493 6216766)') FROM ships_poly;
```



(a) (b)

Fig. 5.18: (a) Entrance to the port of Helsingborg, with the query point in red and the vessel trajectories in blue. (b) Visualization of the vessel with the closest approach in both the point (blue) and polygon (green) representation.

Let us also query for the vessel that came closest to the query point during its movement. Queries 5.4 and 5.5 compute the nearest approach distance of every vessel to the query point and returns the vessel ID (mmsi) and distance of the vessel with the smallest nearest approach distance. Again, this query is run with the vessels being represented as moving points (Query 5.4) and moving polygons (Query 5.5) for the first day of data. Table 5.2 shows the running time and results of both queries. Consistently with the results obtained in Queries 5.2 and 5.3, the running times of Queries 5.4 and 5.5 are similar since the processing of the distance algorithm takes the majority of the query time. Looking at the result values, we can see that the smallest distance is 10.42 m when computed using moving points, and 5.21 m

when computed using moving polygons, which illustrates the gain in precision. More important is that the vessels (identified by mmsi) returned by the queries are also different. This means that the impression introduced by computing distances using the moving point approximation led to returning a wrong vessel ID in this query. This result is also visualized in Fig. 5.18b.

Query 5.4 *Nearest approach distance query on temporal point data*

```
SELECT mmsi,
   nearestApproachDistance(trip,
       'Point(729493 6216766)') AS min_dist
FROM ships_points
ORDER BY min_dist LIMIT 1;
```

Query 5.5 *Nearest approach distance query on temporal geometry data*

```
SELECT mmsi,
   nearestApproachDistance(trip,
       'Point(729493 6216766)') AS min_dist
FROM ships_polys
ORDER BY min_dist LIMIT 1;
```

Table 5.2: Running time and result of NAD queries

Query	Representation	Running time (s)	mmsi	min_dist (m)
5.4	Point	3.59	265610940	10.42
5.5	Polygon	4.94	265041000	5.21

In a second experiment, we demonstrate the scalability of the algorithms. As reference geometries, we generate a point and 5 convex polygons of varying sizes that we arbitrarily placed below the island of Læsø. Table 5.3 lists the generated geometries with their number of vertices and Fig. 5.19 shows four of these geometries on the map. Note that, since we are working with convex polygons, the polygon with 75 vertices already has a smooth contour. We thus limit ourselves to polygons with at most 75 vertices. To link this experiment to a practical problem, the generated polygons can be seen as the extent of an offshore wind farm. European member states and, in particular, Denmark already possess many offshore wind farms[3] and are planning on adding more to be able to meet ambitious goals related to renewable energy[4][5]. For security reasons, vessel traffic cannot come within a certain distance of these wind farms [71]. When planning a new project, it is thus important to know the distance between the planned wind farm and the surrounding vessel traffic.

[3] https://www.4coffshore.com/windfarms/denmark/

[4] https://energy.ec.europa.eu/news/member-states-agree-new-ambition-expanding-offshore-renewable-energy-2023-01-19_en

[5] https://www.offshorewind.biz/2023/02/20/denmark-to-auction-off-9-gw-of-offshore-wind-in-2023/

Table 5.3: Sizes of the generated geometries

Geometry Name	Point A	Poly B	Poly C	Poly D	Poly E	Poly F
# of Vertices	1	5	10	25	50	75

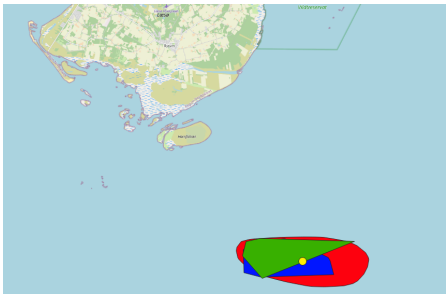


Fig. 5.19: Visualization of four generated geometries: Point A (yellow), Poly B (green), C (Blue) and F (red).

In this experiment, we compute the temporal distance between the vessel trajectories in moving polygon representation and each of the generated reference geometries. Additionally, we also compute the temporal distance between the moving point representation of the trajectories and the first point geometry, since this is currently the only existing distance function for moving objects in MobilityDB. Figure 5.20 shows the query duration for data sizes varying from 1 day to a full month of data. For the full data size, computing the temporal distance between the moving polygons and a static geometry takes about 80 seconds.

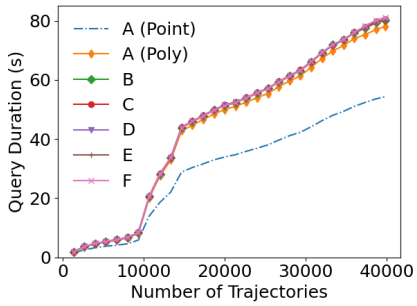
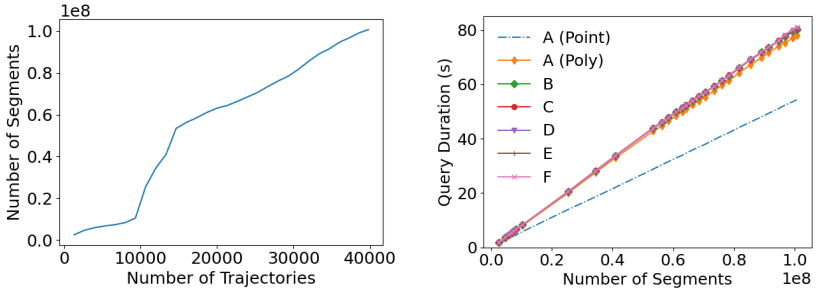


Fig. 5.20: Duration of temporal distance queries in function of the number of trajectories. Queries A (Poly) and B-F use the polygon representation of the vessels, while query A (Point) uses the point representation.

The first thing we note in the figure is that computing the distance for the polygon representation of the vessels is at most 1.5 times slower than for the point representation. This is coherent with the result from the previous experiment. Secondly, the polygon-to-point algorithm (A Poly) is ever so slightly faster than the polygon-to-polygon algorithm (B-F), which is expected since the polygon-to-polygon algorithm needs to solve more complex equations to determine changes in closest features. Lastly, the number of vertices of the second polygon does not visibly influence the duration of the query. Indeed, the duration of the algorithm is linked to the number of changes in the closest features. For many trajectories that are relatively far away from the wind farm geometry, increasing the number of vertices of the geometry will not directly increase the number of changes in the closest features during the movement.

The last remaining particularity of Fig. 5.20 is the shape of the graph. Notice that the data size is given in terms of number of trajectories. Each trajectory represents a sequence of linear segments, as discussed at the end of Sect. 5.1. The distance algorithm is thus called subsequently on each segment. Querying the distance for trajectories with more segments would thus take longer than for trajectories with fewer segments. Looking at Fig. 5.21a, we can indeed see that the total number of segments does not increase linearly with the number of trajectories. This means that some trajectories contain more segments than others. Plotting now the query duration as a function of the total number of segments (Fig. 5.21b), we can see that the query duration is linear in terms of trajectory segments. The average duration per segment is 539 ns, 775 ns and 800 ns for the point-to-point, polygon-to-point and polygon-to-polygon algorithms respectively. The last value is computed as the average of the four polygon-to-polygon queries since they all have similar duration.



(a) Relation between the number of trajectories and the number of segments

(b) Duration of `tDistance` queries as a function of the number of segments

Fig. 5.21: Impact of the number of segments on the query duration

Chapter 6
Trajectory Indexing in MobilityDB

GiST [41] and SP-GiST [4] are software engineering solutions that facilitate the creation of custom-tailored indices inside a database management system. They allow users to easily implement a new index by specifying a set of interface parameters and external methods specific to that index. Using these parameters and methods, GiST or SP-GiST will take care of the construction, storage, and search, as well as the concurrency and crash recovery of the index. For example, the GiST framework can be used to implement most types of balanced search trees, e.g., the B+-tree, the R-tree, and the RD-tree using minimal implementation effort. On the other hand, SP-GiST can be used to construct space-partitioning trees, e.g., the quadtree, the k-d tree, and the Trie, and their variants. In PostgreSQL, these frameworks allow database extensions, e.g., PostGIS or MobilityDB to easily construct R-trees, k-d trees, and quadtrees for their spatial and spatio-temporal data.

In MobilityDB, the GiST and SP-GiST frameworks are used to index the six existing temporal types. An interval tree is used to index the temporal extent of the `tbool` and `ttext` types. For the `tint` and `tfloat` types, a 2D R-tree is used, indexing both the time and the value dimensions. Lastly, the `tgeompoint` and `tgeogpoint` types are indexed using either and R-trees, a quadtree or a k-d tree of the appropriate dimension depending on the dimension of the data (2+1D or 3+1D). In Sect. 4.5.2, we show how the framework can be used to implement equivalent indices for the new `tgeometry` data type.

Table 6.1: Classes of generalized indices

Single-Entry	B-tree	GiST	SP-GiST
Multi-Entry	GIN	MGiST	MSP-GiST

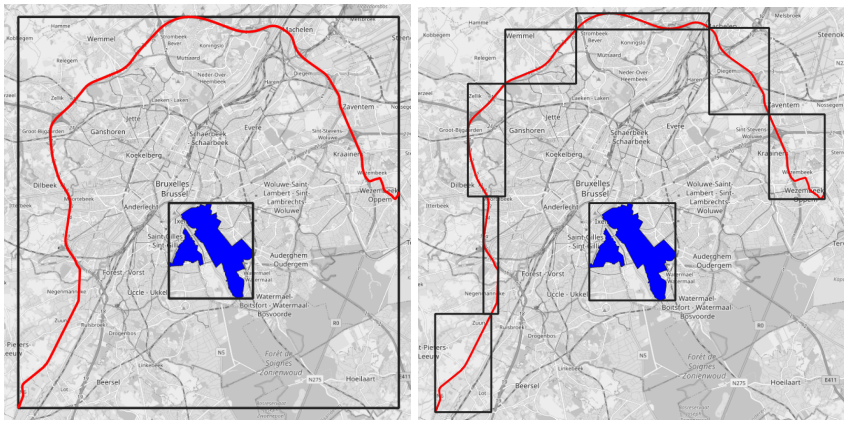
However, in their current implementation, GiST and SP-GiST store each tuple as a single entry in the index. When handling complex or composite

data types, much information is lost when compressing the object into a single index entry. This reduces the capabilities and efficiency of the constructed indices. The idea of storing multiple entries for a single tuple is already being used for one-dimensional data, e.g., as in the case of the text stream data type. The Generalized Inverted Index, GIN [67], indexes complex one-dimensional data types, e.g., text documents that are composed of simpler elements that one may want to search for, e.g., words and trigrams. While useful, GIN indexes only one-dimensional objects that exhibit total order, i.e., are sortable. Internally, and building on the total order property, GIN stores its entries in a B-tree but does not apply to the multi-dimensional case.

Consider a set, say S, of elements that are stored inside the index, and a set, say R, of indexable real-world complex objects. One can view the GiST and SP-GiST indices as forming a one-to-one relationship between sets S and R. In contrast, GIN provides an m-to-one relationship between S and R, e.g., that multiple words in the index correspond to one document object in the document database. Thus, the current GiST and SP-GiST can be viewed as special cases of the more general case, and they are 1-1 GiST and 1-1 SP-GiST. This paper highlights the importance of realizing the more general case of the m-1 GiST and m-1 SP-GiST indices, provides extensible designs for both indices and studies their performance.

This chapter presents the *multi-entry GiST* (MGiST) and *multi-entry SP-GiST* (MSP-GiST) frameworks as multi-entry m-1 generalizations of the 1-1 GiST and the 1-1 SP-GiST frameworks. Through the use of one additional external method, one can specify a splitting and decomposing mechanism that the framework will apply on each real-world object (the 1 side of the m-1 relationship) before the resulting sub-objects (the m-side of the m-1 relationship) get inserted inside the index. All the m index entries store the identifier of the object (the object's oid, for short) to point back from the index side to the same object/tuple in the relational table side. This allows for the creation of multi-entry variants of known indices, e.g., R-trees, quadtrees, and k-d trees. These multi-entry indices can be queried like the traditional GiST and SP-GiST indices. By exposing the object splitting mechanism as a pluggable module, MGiST and MSP-GiST can be tailored for many different complex and composite data types with minimal implementation effort from the user. Examples of the data types that can benefit from multi-entry m-1 indices are complex geometry types, e.g., paths and non-convex polygons; geometry collections, e.g., multipoints or multipolygons; and trajectory data. In this chapter, we focus on the case of point trajectories. Section 6.5 mentions how the MGiST and MSP-GiST frameworks can be used in an identical fashion for `tgeometry` data.

Trajectory data represents the movement of objects in space, e.g., vehicles or humans. This data can get large in two main aspects: the number of trajectories in a database, and the size of the individual trajectories. When observing the movement of objects over a long duration of time, the size of an individual trajectory can get large in terms of the number of observations,



(a) (b)

Fig. 6.1: Representing a trajectory using multiple bounding boxes can reduce false positives of spatio-temporal indices

duration, and distance covered. The purpose of moving-object databases is to store and analyze these large trajectory datasets.

When analyzing large datasets, queries are commonly applied to subsets of the data. For example, one might be interested in computing the average speed of all cars passing through a given street. Alternatively, one might want to know which 10 cars have passed closest to a given location. To speed up these queries, databases use indices to filter out a large part of the data. In the case of trajectories, this is achieved by computing the spatio-temporal extent of a trajectory, and then indexing the resulting 3D or 4D bounding boxes (2/3D Spatial + 1D Temporal extent). This method works relatively well for short trajectories (i.e., small-sized trajectories), but fails when the trajectories are long because the bounding box does not accurately represent the trajectory anymore. Refer to Fig. 6.1 for illustration, where the bounding box overlap (as tested by the index) would fail to filter out a trajectory that is not passing through the region of interest. When working with long trajectories, these situations are common, and significantly impact the performance of point, range, and nearest neighbor queries.

MGiST and MSP-GiST store a more detailed representation of the actual trajectory in the index. A trajectory is represented by a collection of smaller sub-trajectories that can each be indexed individually in MGiST and MSP-GiST. One extreme is to index each trajectory segment (pair of subsequent observations) individually. This method produces the best accuracy (in terms of false positives) but creates indices that are larger than the actual data, which is not desirable due to their large storage cost and long querying time. As a trade-off, one could split a trajectory into a set of smaller sub-trajectories, and index the bounding boxes of these sub-trajectories. Splitting

the trajectory into a set of smaller bounding boxes produces a better approximation of the actual trajectory than representing it with a single bounding box, which improves index efficiency by reducing the number of false positives (Fig. 6.1). By limiting the number of bounding boxes, one can bound the total size of the index in case of storage constraints.

In this chapter, we instantiate a multi-entry R-tree using MGiST, a multi-entry quadtree and a multi-entry k-d tree using MSP-GiST using several trajectory splitting strategies. We evaluate the performance of these instantiated indices for each of the splitting strategies and demonstrate that multi-entry indices can improve point, range and nearest neighbor query performance by up to an order of magnitude when compared to their traditional single-entry counterparts.

The rest of this paper proceeds as follows. We start by giving an overview of the GiST and SP-GiST frameworks in Sect. 6.1. Section 6.2 presents the MGiST and MSP-GiST frameworks and describes how they can realize instances of multi-entry indices. As a case study, Sect. 6.3 presents the implementation of multi-entry R-trees, quadtrees, and k-d trees for trajectories, together with their trajectory splitting algorithms. Section 6.4 presents the experimental evaluation of MGiST- and MSP-GiST-based indices on the BerlinMOD and maritime datasets. Finally, Sect. 6.5 explains how to adapt the algorithms to handle rigid temporal geometry data and Sect. 6.6 discusses additional use-cases of multi-entry indices.

6.1 Background: GiST and SP-GiST

Generalized search tree (GiST) [41] is a highly configurable generalized index structure that can instantiate balanced search trees supporting an extensible set of queries and data types. By specifying as little as six key methods, the GiST index can be used as, e.g., a B+-tree, an R-tree, or an RD-tree.

6.1.1 GiST Key Methods

One needs to specify six key methods to instantiate a GiST index. These methods each serve a different purpose.

Consistent(E, q): Method **Consistent** is used in the search algorithm of GiST. Given an index entry $E = (p, ptr)$, it determines if the entry predicate p is consistent with the query predicate q. This method only returns false if it is guaranteed that $p \wedge q$ is unsatisfiable.

Compress(E) and **Decompress($E\prime$)**: These methods are used to store compressed representations of the entry predicates. Given an entry $E = (p, ptr)$, **Compress(E)** returns a new entry $E\prime = (p\prime, ptr)$, whose predicate $p\prime$ is a

compressed representation of p. In contrast, given an entry $E\prime = (p\prime, ptr)$, formed by compressing entry $E = (p, ptr)$, **Decompress($E\prime$)** returns a new entry $E\prime\prime = (p\prime\prime, ptr)$, such that $p \rightarrow p\prime\prime$. The notation $p \rightarrow p\prime$ implies that if predicate p holds then $p\prime$ also holds. In the R-tree, **Compress** returns a bounding box given a complex spatial object, e.g., a polygon. **Decompress** is the identity and is thus lossy as it is not possible to recover the initial geometry from its bounding box.

The remaining three methods are used during tree construction or when inserting a new tuple into the index.

Union(P): Given a set $P = \{E_1, ..., E_n\}$ of n index entries $E_i = (p_i, ptr_i)$, determines a predicate $p\prime$ that holds for all entries in the set $(p_1 \vee ... \vee p_n \rightarrow p\prime)$.

Penalty(E_1, E_2): Given two entries E_1 and E_2, computes a penalty value for inserting entry E_2 into the sub-tree rooted at E_1.

PickSplit(P): Given a set $P = \{E_1, ..., E_{M+1}\}$ of $M + 1$ entries, splits P into two subsets P_1 and P_2, each containing at least $k \cdot M$ entries, where M is the maximum node capacity of the tree and k corresponds to the minimum fill factor.

An example of **Union** for spatial data computes the minimum bounding box enclosing all the bounding boxes of the set of entries P. **Penalty** would compute a value related to the increase in the area of the bounding box of E_1 caused by the insertion of E_2 into the sub-tree rooted at E_1. Lastly, a possible PickSplit method would split the set P of entries by trying to minimize either the total area of or the overlap between, the bounding boxes of P_1 and P_2.

Distance(E_1, E_2): GiST as realized in PostgreSQL has a seventh optional key method, **Distance**, that is used in answering nearest neighbor queries. Given two entries E_1 and E_2, **Distance** computes a distance value between the predicates of E_1 and E_2.

6.1.2 SP-GiST Key Methods and Parameters

Similar to GiST, *space-partioned generalized search tree* (SP-GiST) [4] presents a list of key methods that are to be implemented to instantiate a specific space partitioning index. Methods **Consistent** and **PickSplit** are similar to those of GiST.

Consistent(E,q,l): Given Index Entry $E = (p, ptr)$ at Level l, determines if Predicate $E.p$ is consistent with Query Predicate q.

PickSplit(P, l): Given a set $P = \{E_1, ..., E_{M+1}\}$ of $M + 1$ entries at Tree Level l, splits P into N partitions, where M (maximum bucket size) and N (number of partitions) are user-defined parameters.

SP-GiST can apply different partitioning methods based on the level. Thus, Tree Level l is passed as a parameter to Consistent and PickSplit,

e.g., the 2D k-d tree partitions on the x-axis at even levels and on the y-axis at odd levels. SP-GiST also has a **Cluster** method that clusters nodes into data pages. Although **Cluster** is described as a key method in [4], it is implemented using a default clustering algorithm in PostgreSQL. Lastly, SP-GiST supports nearest neighbor queries using a user-defined **Distance** method.

6.2 Multi-Entry Search Trees

A traditional index holds entries of the form: $E = (p, ptr)$, where p is a predicate that holds for a given object and is used to search the tree, while *ptr* is a pointer to said object. For example, in PostgreSQL's R-tree, p is a bounding box and *ptr* is the tuple ID (*TID*) that references the physical position of the tuple on disk. Commonly, each *ptr* only appears once in the index. That is, each tuple is indexed using at most one entry.

In this section, we introduce MGiST and MSP-GiST, multi-entry generalized search trees that allow objects to be split into multiple entries before insertion. Each entry contains a different predicate but references the same tuple, and thus has the same *ptr* value. This allows the predicates to be more restrictive and can improve index efficiency by decreasing the number of matches to a given query predicate. For example, representing one trajectory using a set of small bounding boxes can reduce the number of false predicate matches (Fig. 6.1). The application of multi-entry generalized search trees for indexing trajectories is discussed in Sect. 6.3.

MGiST and MSP-GiST contain all the key methods of GiST and SP-GiST as well as the additional method **ExtractValue**. This method splits each tuple into a set of index entries and is detailed in Sect. 6.2.1. This addition changes both the insertion and search methods. With multiple entries pointing to the same tuple, de-duplication mechanisms are necessary when querying the index. Query semantics are also different from the ones in a traditional single-entry index. These changes are detailed below.

6.2.1 ExtractValue Method

ExtractValue is a key method that splits an object into multiple index entries before insertion. After splitting, all returned entries are inserted into the index using the existing GiST and SP-GiST insertion methods. Notice that nothing prevents **ExtractValue** from returning only a single entry. However, MGiST and MSP-GiST assume that at least one tuple is split into more than one index entry. If all tuples are split into only 1 entry each, the traditional GiST and SP-GiST indices can be used.

ExtractValue(E): Given an entry $E = (p, ptr)$, returns a set $P = \{E_1, ..., E_n\}$ of n index entries, with $n \geq 1$, $E_i = (p_i, ptr)$.

The insertion algorithm is given in Alg. 6.1. It combines the ExtractValue method with the existing insertion algorithms of GiST and SP-GiST.

Algorithm 6.1: Multi-Entry Insertion Algorithm

Input: An MGiST or MSP-GiST index rooted at R and a tuple pointed to by ptr.
Output: A new MGiST or MSP-GiST index after insertion of the tuple.

```
P = ExtractValue(ptr);
for Entry E_i of P do
    if MGiST then
        Call GiST Insert(R, E_i, level = 0);
    else if MSP-GiST then
        Call SP-GiST Insert(R, E_i, level = 0);
return R
```

ExtractValue offers a lot of flexibility. By deciding how the tuples are split and in how many parts, the created index can be tailored to specific use cases. Sect. 6.3 details the particular use-case of trajectory indexing and presents multiple different splitting algorithms that can be used with different purposes and effectiveness.

6.2.2 Multi-Entry Search

Searching an MGIST or MSP-GiST index consists of finding all entries that match the given query predicate. Each entry will contain a pointer to a tuple that is part of the result of the query. Multiple entries pointing to the same tuple may match the query predicate. Thus, it is necessary to apply a de-duplication step before returning the actual tuples. As no assumption is made on the relation between different entry predicates of the same tuples, existing de-duplication mechanisms [5, 21] cannot be used. Thus, duplicates are eliminated using a hashmap on tuple pointers (ptr). Before returning a tuple, the search algorithm checks that the pointer is not in the hashmap before returning the tuple. If the pointer is already in the hashmap, the tuple is skipped as it has already been returned before. Otherwise, the tuple is returned and its pointer is added to the hashmap.

The MGiST and MSP-GiST search algorithm is detailed in Alg. 6.2.

Thus, the returned tuples are ones with at least one entry predicate matching the query predicate. This has important implications. Let the entries of a tuple T be E_i^T $(i \in \{1, \cdots, n\})$. Based on the query predicate q, two cases exist.

Algorithm 6.2: Multi-Entry Search Algorithm

```
Input: An MGiST or MSP-GiST index rooted at R and a query predicate q.
Output: All the tuples matching q.
if MGiST then
    P = GiST Search(R, q);
else if MSP-GiST then
    P = SP-GiST Search(R, q);
T ;                                   /* Empty list of tuple pointers */
M ;                                   /* Empty hashmap */
for Entry E_i = (p_i, ptr) of P do
    if ptr ∈ M then
        skip;
    else
        T ← ptr ;                     /* Add to list */
        M ← ptr ;                     /* Insert in hashmap */
return T
```

$$Consistent(T, q) \Leftrightarrow \exists i : Consistent(E_i^T, q) \tag{6.1}$$

$$Consistent(T, q) \Leftrightarrow \forall i : Consistent(E_i^T, q) \tag{6.2}$$

Equation 6.1 is valid for all operators that require at least one entry to match the query predicate. An example for this case is the **spatial overlaps** operator. On the other hand, Eq. 6.2 corresponds to operators that need all entry predicates to match the query predicate, e.g., strictly left or right, contained by, and disjoint. The search algorithm returns all tuples that have at least one entry matching the query predicate, i.e., is equivalent to Eq. 6.1. These queries can be answered efficiently. To answer queries of the second type, it is still possible to use the index, but it will be less efficient. Indeed, if all entries need to match the query predicate, at least one will match. The index can thus be used to find all the tuples that have at least one matching entry. In a second refinement step, the actual operator can then be rechecked on the returned tuples to select only the ones that match the query predicate.

It is worth noting that many queries corresponding to Eq. 6.2 can already be answered efficiently with traditional GiST or SP-GiST indices. For example, the strictly left ($\ll$) spatial operator requires that all parts of the first geometry are left of the second geometry. In this case, even if the geometries are collections of many points or polygons, it is sufficient to know the extent of the collection as a single bounding box to know if one collection is strictly left of the other collection or not. The MGiST and MSP-GiST indices are thus not meant to completely replace the existing GiST and SP-GiST indices but rather complement them.

6.2.3 Nearest Neighbor Search

The search method in Sect. 6.2.2 scans the index for tuples qualifying a given predicate. Another type of query that can be answered using GiST and SP-GiST indices is the k-nearest neighbor search (k-NN). This query returns the k closest objects/tuples to a given query value based on a user-specified distance measure. Efficient algorithms exist to answer k-NN queries efficiently using GiST and SP-GiST indices, e.g., [48, 72]. Similarly, existing algorithms can be adapted to answer k-NN queries for MGiST and MSP-GiST.

To answer k-NN queries with MGiST or MSP-GiST indices, the user specifies a distance measure to be applied between an index entry and a query q. Using this distance measure, the index finds the k-closest entries to q, corresponding to the k closest tuples. However, in multi-entry indices, it is possible that multiple of these closest entries point to the same tuple. Thus, as in Sect. 6.2.2, a de-duplication step is necessary. De-duplication retains only the closest entry for each distinct tuple. Thus, a k-NN search in MGiST or MSP-GiST works only for distance measures that satisfy Eq. 6.3.

$$dist(T, q) = \min_{i \in \{1,\ldots,n\}} dist(E_i^T, q) \tag{6.3}$$

6.2.4 MGiST and MSP-GiST versus GIN

In this section, we discuss the similarities and differences between MGiST and MSP-GiST on one side and another generalized index structure: GIN [67] on the other side. We further explain the use cases of each. GIN is a generalized inverted index that can be used to index arrays and documents [67]. Its implementation also requires an **ExtractValue** method that also splits a tuple into multiple sub-elements, termed keys (e.g., values in an array, or words in a document). This is the main similarity. However, two main structural differences exist between MGiST/MSP-GiST and GIN indices. (1) GIN indexes the keys in a B-tree. This makes GIN not suitable for indexing complex spatial and spatio-temporal objects. (2) The data indexed by GIN is assumed to contain many duplicate keys. For example, a word can be present in many different documents. To handle this, GIN indexes the distinct keys in a B-tree and stores for each key at the leaf level a set of tuple pointers that contain this key. In MGiST and MSP-GiST, entries with the same keys are stored as separate entries in the index leaves. Adding this de-duplication of keys/predicates to MGiST and MSP-GiST is left as future work.

6.3 Multi-Entry Trajectory Indexing

In this section, we illustrate the use of MGiST and MSP-GiST for indexing trajectory data that can span over large spatio-temporal extents. Thus, indexing trajectories as a single bounding box in GiST or SP-GiST produces many false positives during query processing. However, if each trajectory is indexed using a combination of multiple smaller bounding boxes, this problem can be mitigated. Refer to Fig. 6.1, where a trajectory circles the city of Brussels. From this example, it is clear that trajectory indexing is a perfect use case for MGiST and MSP-GiST. Thus, we instantiate multi-entry implementations for the R-tree using MGIST, and a quadtree, and k-d tree indices using MSP-GiST. As in Sect. 6.2, these indices require a user-defined ExtractValue method that splits the tuples before indexing. Given a trajectory, the goal of these splitting algorithms is to produce a set of small bounding boxes that (accurately) represent the shape of the trajectory. Note that ExtractValue receives a single trajectory at a time. It does not have global information about the other trajectories.

Assuming the model used in MobilityDB, a trajectory is stored in a data type termed `tgeompoint` that can have four temporal durations: instant, instant set, sequence, or sequence set. A tgeompoint instant represents a spatial position (Point) at a specific instant in time. A tgeompoint instant set groups multiple instants with distinct timestamps, but assumes no interpolation between instants. A tgeompoint sequence is similar to an instant set but assumes linear interpolation between subsequent instants. This is the data type that is used to represent continuous trajectories. Finally, a tgeompoint sequence set groups non-overlapping tgeompoint sequences and can be used, e.g., to represent a trajectory containing stops or signal losses. We focus on the problem of splitting tgeompoint sequence data as it is the main data type for trajectory data. A trajectory is a tgeompoint sequence T, containing n instants defined between t_0 and t_{n-1}. An instant is a position-timestamp pair ($p@t$), with the position being either 2D or 3D. A trajectory segment is defined as two consecutive instants of the trajectory. Thus, a trajectory with n instants is made of $n-1$ consecutive segments.

$$T = [p_0@t_0, ..., p_{n-1}@t_{n-1}] \tag{6.4}$$

At time t between t_0 and t_{n-1}, Position $p(t)$ is computed using linear interpolation of Segment $[p_i@t_i, p_{i+1}@t_{i+1}]$ containing t by:

$$p(t) = \frac{p_i \cdot (t_{i+1} - t) + p_{i+1} \cdot (t - t_i)}{t_{i+1} - t_i}, t_i \leq t < t_{i+1} \tag{6.5}$$

There are many ways to split a trajectory into multiple bounding boxes. These methods vary in the way the splits are generated and in how many boxes the trajectory is split. One straightforward splitting algorithm is the following. Given a trajectory r and a constant parameter k (the same for all

tuples), split r into k 'equal' parts. This method produces a box for every $\lceil \frac{n}{k} \rceil$ consecutive segments. (The last box might contain less than $\lceil \frac{n}{k} \rceil$ segments.) Refer to this splitting algorithm by **EquiSplit**. Notice that **EquiSplit** splits all trajectories into the same number of k boxes. Given a dataset of N trajectories, **EquiSplit** produces in total $N \cdot k$ boxes, i.e., the size of the created index is directly proportional to k.

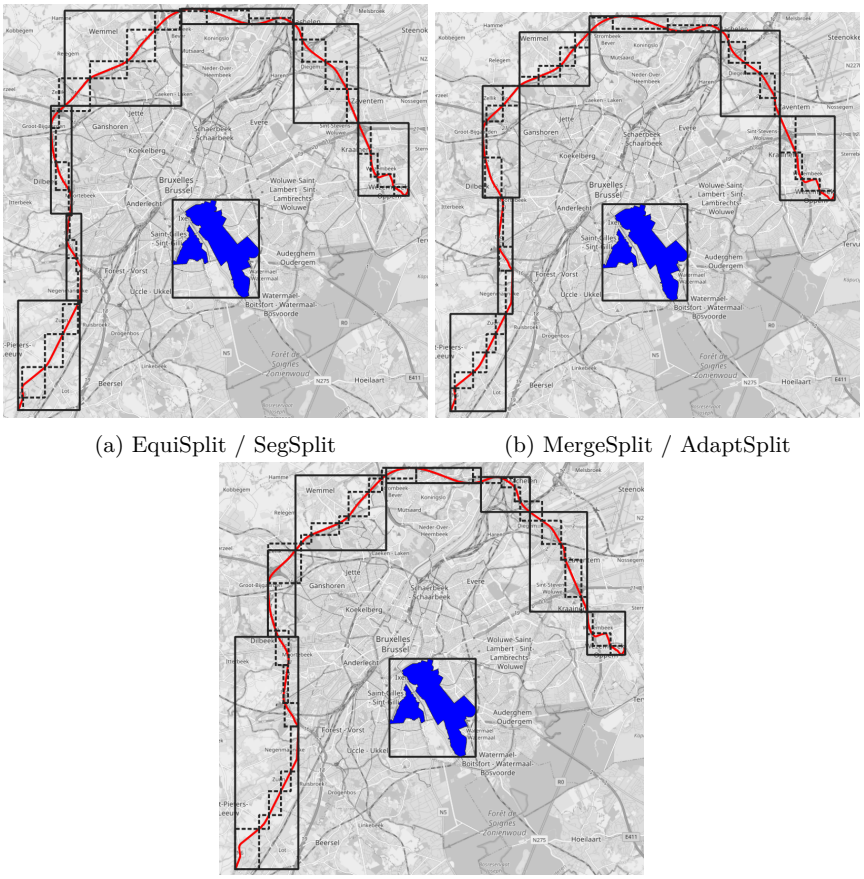
Another method to split trajectories into k boxes is by Hadjieleftheriou et al. [37], where it minimizes the total volume of the generated bounding boxes. Then, they propose an approximate version of the algorithm, termed **MergeSplit**, that produces near-optimal splits in $\mathcal{O}(nlogn)$ time (n is the number of instants in the trajectory). It starts by computing the bounding box of each trajectory segment. Then, it iteratively merges the two adjacent bounding boxes that produce the least increase in volume until the desired number of boxes is reached.

In both algorithms, the number of bounding boxes per trajectory is constant. This is a big restriction and is not optimal. In case some trajectories are much longer than others (have a bigger spatial extent), it could be beneficial to split them into more boxes. Thus, we propose an algorithm, termed **SegSplit**, that requires a parameter m; the number of segments that form a bounding box. A trajectory with n segments is split into $\lceil \frac{n}{m} \rceil$ boxes. **SegSplit** is the inverse of **EquiSplit**, which has a fixed number of boxes per trajectory but a variable number of segments per box.

Analogously, **MergeSplit** can be adapted to generate a different number of boxes per trajectory. Given a parameter m (same for all trajectories) and a trajectory with n segments, we compute the desired number of boxes $k = \lceil \frac{n}{m} \rceil$. Then, we apply **MergeSplit** given k. We refer to this adaptive splitting algorithm by **AdaptSplit**.

Rasetic et al. [70] present a cost model for splitting trajectories based on the expected number of I/O's of a given range query. They present an optimal splitting algorithm that minimizes the cost of a given range query. To avoid quadratic time complexity, they propose a heuristic linear-time algorithm, termed **LinearSplit**, that produces near-optimal results. **LinearSplit** starts by linearly accumulating the trajectory instants until a given criterion is met. A constant-slope approximation of the collected sub-sequence is used to compute the bounding boxes covering this part of the trajectory. This step takes linear time. **LinearSplit** continues with the remaining trajectory instants. For more detail, the readers are referred to [70]. Similarly to SegSplit and AdaptSplit, **LinearSplit** produces a variable number of boxes per trajectory.

Figure 6.2 illustrates the result of applying the different splitting algorithms on a single trajectory. Although the figure only illustrates the spatial projection of the bounding boxes, it's important to keep in mind that all splitting algorithms also consider the temporal dimension. Each image contains two splits, one in 7 bounding boxes (with solid border) and one in 28 boxes (dotted border). Note that for a single trajectory, EquiSplit and SegSplit are



(a) EquiSplit / SegSplit (b) MergeSplit / AdaptSplit

(c) LinearSplit

Fig. 6.2: Visualization of the bounding boxes resulting from using different splitting algorithms on a trajectory

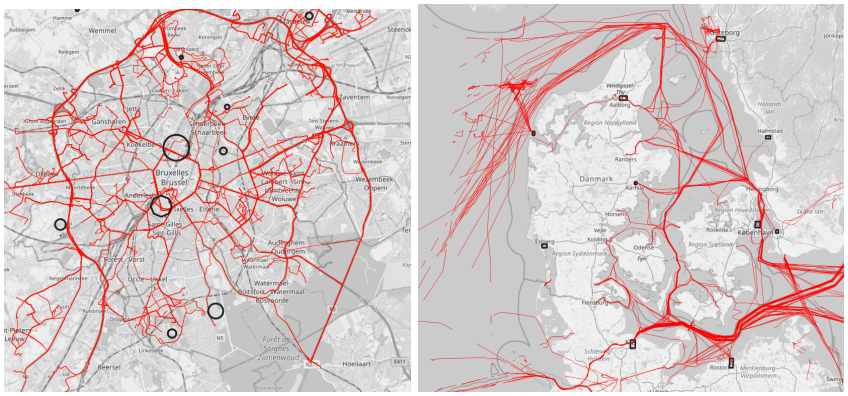
equivalent. Indeed, given a number of boxes k and a trajectory T of length n, one can compute the parameter $m = \lceil \frac{n}{k} \rceil$, such that EquiSplit($T$, k) and SegSplit(T, m) produce the same boxes. The same holds for MergeSplit and AdaptSplit. However, this is not true anymore when applying the algorithm with the same parameter on multiple trajectories of different lengths.

All the splitting algorithms presented in this section assume that the splitting can only occur at instants stored in the sequence. In theory, if two subsequent instants are far apart, the line formed by linear interpolation of these two instants could also be split to further reduce the size of the created bounding boxes. Taking this into account, however, would greatly complicate the splitting algorithms. In practice, such a situation is also very rare when working with moving objects data. This is thus left as future work.

The splitting algorithms are used as implementations of the ExtractValue method to produce four variants of the multi-entry R-tree, quadtree, and k-d tree implementations. These variants are evaluated in Sect. 6.4. The remaining user methods (Consistent, Picksplit, etc.) needed to implement the multi-entry indices are all identical to their traditional counterpart. The only exception is the **Compress** method. In GiST and SP-GiST, **Compress** receives a geometry or trajectory and produces a bounding box as a compressed version of the data. In MGiST and MSP-GiST, **Compress** applies to all entries returned by ExtractValue. Since these methods already return a list of bounding boxes, **Compress** is the identity.

6.4 Experiments

In this section, we evaluate the performance of the proposed multi-entry MGiST and MSP-GiST indices and their different splitting algorithms. In the first experiment, the four splitting algorithms are evaluated using the BerlinMOD benchmark for moving objects. We omit the evaluation of Equisplit, as Equisplit is consistently outperformed by the other four algorithms. We evaluate index construction in terms of both construction time and index size. The query speed of point, range and k-NN queries are reported and discussed. In a second experiment, the performance of the best index found is further evaluated on our maritime dataset.



(a) BerlinMOD Data (b) AIS Data

Fig. 6.3: Datasets used in the experiments

6.4.1 The BerlinMOD Benchmark

BerlinMOD (Sect. 3.2.1) is a benchmark for moving object databases that presents both a synthetic trajectory data generator and a set of benchmark queries. In these experiments, we use a version of the BerlinMOD data generator that is implemented in MobilityDB and thus generates the trajectories readily in MobilityDB moving point format. It is parameterized by a scale factor that determines the size of the generated dataset. We use a scale factor of 1 that produces trips for 2000 vehicles over 30 days, for a total of 157K trips containing on average 1.2 K instants each. The size of this trajectory dataset is 7.7 GB. Next to the trajectory data, the generator produces four other tables of interest: `QueryInstants`, `QueryPeriods`, `QueryPoints` and `QueryRegions`, which are used in the benchmark queries. The query regions are visible in Fig. 6.3a. The query regions (Fig. 6.3a, black) have an average size of 1 km^2.

A spatio-temporal index is constructed on the `trip` column of the `Trips` table, for different parameter values of the splitting algorithms. The parameter for MergeSplit is the number of boxes per trajectory. For SegSplit and AdaptSplit, the parameter is the (average) number of segments per box. Lastly, the parameter for LinearSplit corresponds to the average width, length and duration, in meters and minutes respectively, of the query box.

Figure 6.4 gives the index size and construction time for the four splitting algorithms. The experiments confirm linear proportionality between the number of splits and the index size in the case of MergeSplit, as mentioned in Sect. 6.3, and negative exponential in the case of SegSplit and AdaptSplit. For LinearSplit, a decrease in average query size results in an increase in the number of generated boxes and thus an increase in index size.

Similarly, the index construction times are proportional to the index size (and thus to the number of stored boxes), with MergeSplit and AdaptSplit being relatively slower due to the $O(nlogn)$ complexity of the MergeSplit algorithm.

Point and Range Queries

The BerlinMOD benchmark has a total of 17 point and range query types. However, not all 17 query types make use of a spatio-temporal index on the `Trips` table. We evaluate the following three point and range queries:

Query 6.1 (BerlinMOD 4/R) *Find the vehicles that passed by any of the query points*

```
SELECT DISTINCT vehicle_id, trip_id
FROM Trips, QueryPoints
WHERE eintersects(trip, point)
AND trip && stbox(point);
```

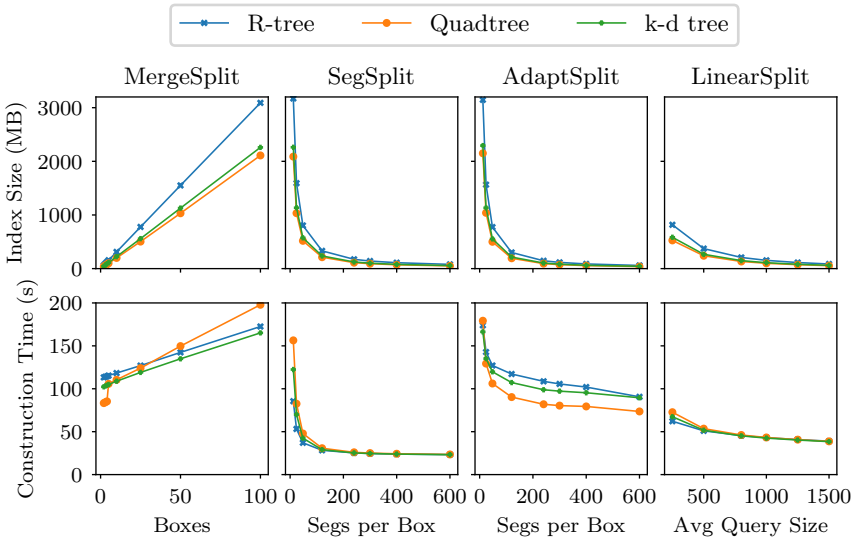


Fig. 6.4: Index size and construction time of the different splitting algorithms

Query 6.2 (BerlinMOD 11/R) *Find the vehicles that passed by any of the query points at one of the query instants*

```
SELECT DISTINCT vehicle_id, trip_id
FROM Trips, QueryPoints10, QueryInstants10
WHERE eintersects(atTime(trip, instant), point)
AND trip && stbox(point, instant);
```

Query 6.3 (BerlinMOD 13/R) *Find all the vehicles that passed through any of the query regions during one of the query periods*

```
SELECT DISTINCT vehicle_id, trip_id
FROM Trips, QueryRegions10, QueryPeriods10
WHERE eintersects(atTime(trip, period), region))
AND trip && stbox(region, period);
```

In the above queries, the `eintersects` predicate determines if a temporal point ever intersects a given geometric object. The function `atTime` is used to restrict a temporal point to a given instant or period. Table `QueryPoints10` contains only the first 10 rows of `QueryPoints` and the same holds for the three other `Query*10` tables. These tables are part of the BerlinMOD benchmark and are used to limit the size of the Cartesian product when multiple `Query*` tables are being joined. The last condition of every query tests the overlap between the trips and a spatio-temporal box (spatial only in the case of Query 4/R). This condition triggers the query optimizer to use the spatio-temporal index created on the trips.

The cost of a point or range query utilizing an index combines the cost of the index scan and the cost of applying the actual predicate on the tuples returned by the index, i.e., the refine cost. The refine cost is also determined by two factors: The cost of the functions and operators in the query, and the number of tuples returned by the index. The main purpose of a multi-entry index is to reduce the number of tuples returned by the index, i.e., to improve the index filtering. However, this is at the cost of larger indices, and thus higher index scan cost. Based on this analysis, we can expect that the indices will behave better for queries with expensive functions and operators. Indeed, this is the case when reducing the number of tuples returned by the index becomes most beneficial.

Figure 6.5 gives the query duration of all tested indices for the three queries. Since the splitting algorithms are parameterized differently, we use the index size in the x-axis. The corresponding single-entry index is also shown.

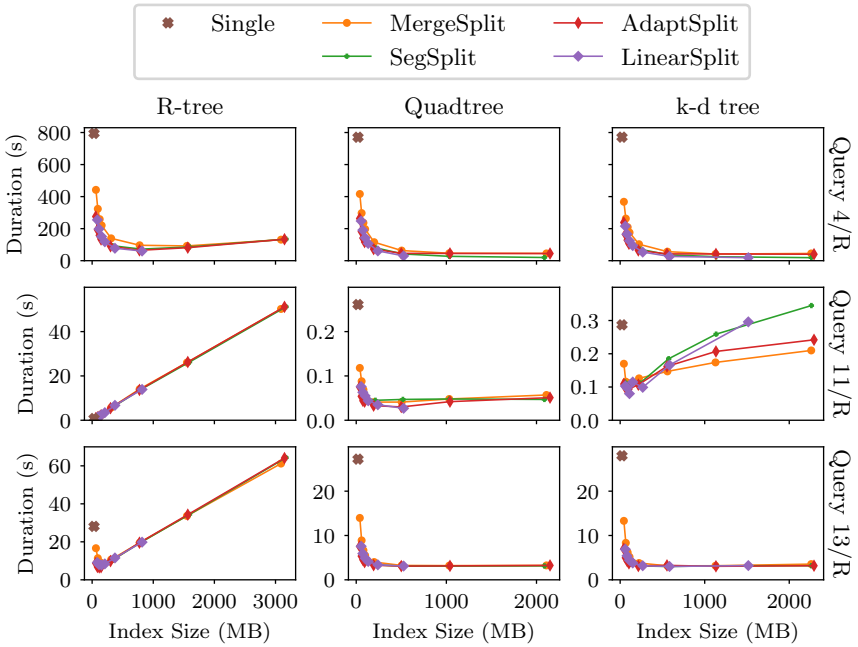


Fig. 6.5: Point and range query duration for the multi-entry R-tree, quadtree and k-d tree

The above analysis is coherent with the presented results. The most expensive query in terms of functions is Query 4/R which applies an intersection

test on the complete trajectories returned by the index. This index is also the one showing the most speedup when using multi-entry indices.

The least expensive query is Query 11/R. This query applies a point intersection test at a snapshot of the returned trajectories. For this query, the multi-entry R-tree implementations are slower than their traditional single-entry counterparts. In this case, the index scan is dominating the query duration and is thus slowing down the query. The same effect can be seen for the k-d tree at large index sizes but to a lesser extent. The multi-entry MSP-GiST indices thus seem to have a lower index scan cost, as they are still able to improve the duration of Query 11/R. The increased index scan cost of the R-tree indices is potentially due to the sequential insert method used by MGiST producing R-trees with high overlap between nodes.

Nearest-Neighbor Search Queries

Nearest-neighbor search in trajectories can have multiple semantics. A historical continuous k-NN query for moving objects is presented in [22]. This query searches for the k closest trajectories to a given spatio-temporal object at each instant that this object is defined. That is, the result of this query is also evolving in time and is returned as tuples of $(mobj, t_s, t_e)$, where *mobj* is the moving object and (t_s, t_e) is the start and end times during which *mobj* is part of the k closest trajectories. However, this continuous k-NN query cannot be expressed using traditional SQL syntax and is therefore not implemented in existing moving object databases. Thus, we restrict the experiments to static k-NN queries for moving objects. A static k-NN query searches for the trajectories with the smallest *nearestApproachDistance* to a given spatio-temporal object. We evaluate the following two k-NN queries:

Query 6.4 (BerlinMOD 1/NN) *For each query point, find the 5 vehicles that passed closest to it*

```
SELECT point, vehicle_id, trip_id
FROM QueryPoints10
CROSS JOIN LATERAL (
    SELECT vehicle_id, trip_id FROM Trips
    ORDER BY trip |=| stbox(point)
    LIMIT 5) ClosestTrips;
```

Query 6.5 (BerlinMOD 2/NN) *For each combination of query point and period, find the 5 vehicles that passed closest to the point during that period*

```
SELECT point, period, vehicle_id, trip_id
FROM QueryPoints10, QueryPeriods10
CROSS JOIN LATERAL (
    SELECT vehicle_id, trip_id FROM Trips
    WHERE trip && stbox(period)
    ORDER BY trip |=| stbox(point, period)
    LIMIT 5) ClosestTrips;
```

These queries find the 5 trajectories with the closest point of approach to the query point (Query 1/NN) or the spatio-temporal points constructed by joining the query points and the query periods (Query 2/NN). The first query is essentially a spatial-only k-NN query, while the second query is applied to temporal ranges.

Figure 6.6 gives the query duration of the two k-NN queries for the evaluated indices. As can be seen, the R-tree has poor query performance with increased index size, similar to its performance in Query 11/R. Both the multi-entry quadtree and k-d tree showcase a large speedup over their single-entry counterparts, with the quadtree performing the best overall. This is analogous to the results for the point and range queries.

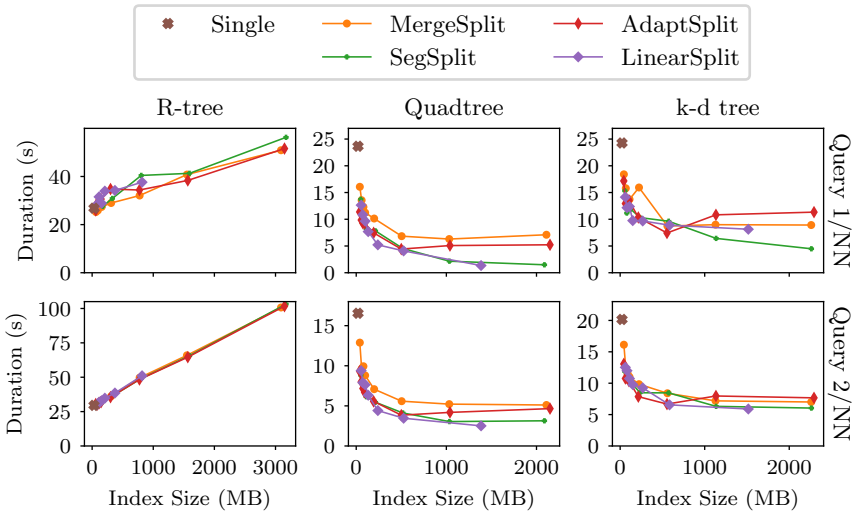


Fig. 6.6: k-NN query duration for the multi-entry R-tree, quadtree and k-d tree

Comparison with Single-Entry Indices

For all five queries, the speedup obtained using the best-performing index is summarized in Fig. 6.7. In this figure, the speed of each index is compared to the best-performing single-entry index, which for all queries is the single-entry quadtree (Note the log scale). From this figure, the multi-entry quadtree consistently provides equivalent or better query performance than the R-tree and k-d tree, with speedups up to 38 times for Query 4/R. Thus, the

remaining experiments on AIS data are applied to the multi-entry quadtree only.

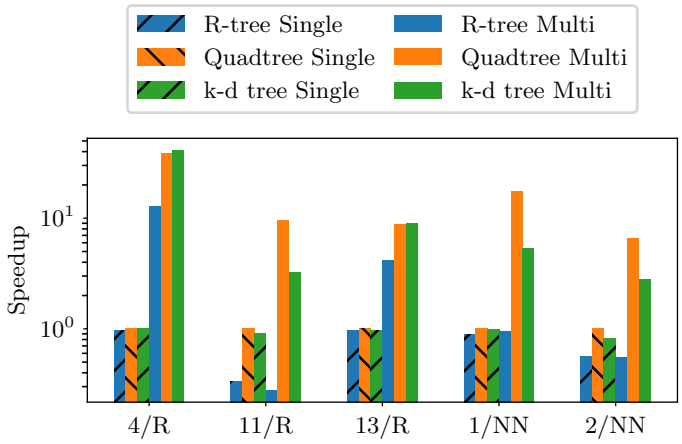


Fig. 6.7: Speedup of the multi-entry indices compared to traditional single-entry indices. The best-performing single-entry index (quadtree) is used as a reference (speedup of 1).

Concerning the splitting algorithms, MergeSplit is consistently performing the worst. This is because it splits each trajectory into an equal number of bounding boxes. This is not optimal when working with trajectories of various sizes. Secondly, the three remaining algorithms are approximately equivalent on Queries 4/R and 13/R, with SegSplit performing slightly worse on Query 11/R. We further evaluate these three algorithms on real-world trajectory data in Sect. 6.4.2.

6.4.2 Maritime Use Case

In this section, we further evaluate the performance of the multi-entry quadtree on the maritime dataset. In this experiment, we use the AIS data for January 2023. The raw dataset is 54.4 GB. After importing the data into MobilityDB and applying basic data cleaning, the remaining dataset contains 53 K trajectories having on average 2.9 K instants each. We also manually construct two additional tables: a table containing 10 ports around Denmark, and a table containing the periods from 2 pm to 4 pm for Week 1 of January 2023. Table 6.2 lists these tables.

Table 6.2: Tables used for the AIS experiment

Name	Columns	Count
Ships	⟨mmsi, trip_id, trip⟩	53,008
Ports	⟨port, region⟩	10
Periods	⟨day, period⟩	7

We construct multi-entry quadtree indices on the `trip` column of the `Ships` table, for the SegSplit, AdaptSplit and LinearSplit algorithms. We evaluate Queries 6.6 and 6.7. An overlaps predicate triggers the optimizer to invoke the index. Query 6.6 contains a spatial-only intersection test, while Query 6.7 is fully spatio-temporal. These two queries are equivalent to Queries 4/R and 13/R of the BerlinMOD benchmark, respectively.

Query 6.6 *Find the vessels that entered a port*

```
SELECT mmsi, trip_id, port FROM Ships, Ports
WHERE eintersects(trip, region)
AND trip && stbox(region);
```

Query 6.7 *Find the vessels that entered a port between 2pm and 4pm on the first week of January 2023*

```
SELECT mmsi, trip_id, port, day
FROM Ships, Ports, Periods
WHERE eintersects(atTime(trip, period), region)
AND trip && stbox(region, period);
```

The duration of both queries is given in Fig. 6.8. The multi-entry quadtree provides a 6× and 3× speedup on Queries 6.6 and 6.7, respectively, with the three splitting algorithms producing similar results. Notice that in this experiment it is not beneficial to split trajectories into more than 5 boxes on average. In fact, past this point, the performance gain starts to plateau while the storage cost of the index continues to increase. A similar effect can also be seen in the BerlinMOD experiments. However, in this case, the cutoff is closer to 10 boxes per trajectory.

Thus, a good rule of thumb is to construct a multi-entry quadtree using one of the three best splitting algorithms, and using parameters such that the trajectories are split on average into 5 to 10 (or more) boxes. For MergeSplit and AdaptSplit, the parameter can be computed by dividing the average length of the trajectories $\tilde{n}$ by the desired number, e.g., $k = \frac{\tilde{n}}{10}$.

6.5 Indexing Rigid Temporal Geometries

Although this chapter details the use of MGiST and MSP-GiST for point trajectory data, these solutions can be used in an identical way to effi-

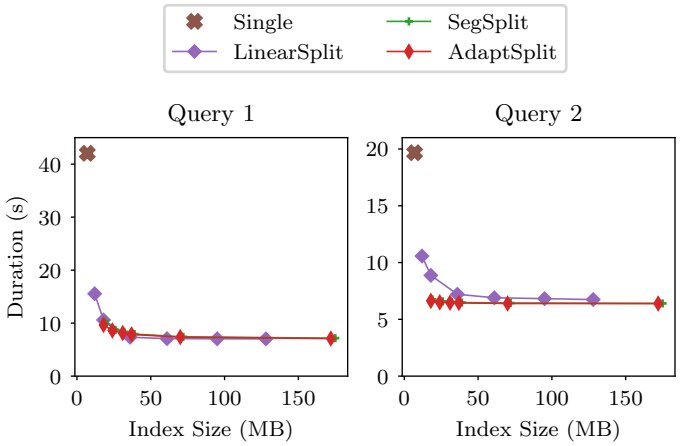


Fig. 6.8: Duration of range queries on the AIS dataset

ciently index rigid temporal geometries. Indeed, just like temporal points, rigid temporal geometries can travel for long durations and thus cover large spatio-temporal extents. Indexing them as a single bounding box as is done in Sect. 4.5.2 is thus not very efficient. In this section we discuss how the previously presented splitting algorithms for trajectory data can be reused for splitting temporal geometry data.

As mentioned in Section 4.2.4, computing an exact bounding box for temporal geometries is computationally expensive. For this reason, we presented a *rotating bounding box* algorithm that efficiently computes an approximate bounding box around a temporal geometry. This bounding box is slightly larger than the minimum bounding box but can still be used for filtering purposes as it completely contains the geometry at all times during its movement. The algorithm works by first computing the bounding box of the linearly moving rotation center of the geometry. Then it expands the resulting bounding box by a distance d, corresponding to the largest distance between any vertex of the geometry and its rotation center. The exact same technique can be used to adapt the previously mentioned splitting algorithms to work for rigid temporal geometries. Firstly, the splitting algorithm is applied on the trajectory of the rotation center of the geometry. Then, each resulting bounding box is expanded by the distance d to produce the resulting set of bounding boxes.

As the distances covered by temporal geometries are typically much larger than their own spatial extent, this expansion of the bounding boxes will not greatly affect the efficiency of the resulting indices. We can thus expect similar performance benefits as reported for trajectory data.

6.6 Discussion

We introduce MGiST and MSP-GiST as a solution to construct efficient indices for trajectory and rigid temporal geometry data. However, this solution is not limited to spatio-temporal data. Any type of composite data that can be decomposed into smaller parts can make use of MGiST and MSP-GiST. For example, checking for the intersection between a trajectory and a static point is almost identical to checking the intersection between a linestring formed by the spatial projection of the trajectory and this point. This is a spatial-only operation. Thus, MGiST and MSP-GiST can also provide benefits to spatial-only indices.

A specific use-case of MGiST and MSP-GiST is to index PostGIS line strings, polygons, and geometry collections. A splitting algorithm for line strings would be similar to the ones discussed in Sect. 6.3 but would have to take into account that line strings can have self-crossings. This is not the case in trajectories, as the time dimension is always increasing. Checking for overlap between complex polygonal shapes could be made more efficient by indexing the convex parts of the stored polygons. Lastly, if a database column stores geometry collections, each geometry can be indexed in a separate index entry. A typical example of this would be the indexing of island groups stored as multi-polygons.

Another data type that can benefit from multi-entry indexes is time-series data. Time-series data can be viewed as 1D equivalent to trajectories. Previous work [85] discusses ways to split time-series data into segments for better indexing. Using MGiST and MSP-GiST, these solutions can be implemented with little effort.

Chapter 7
Visualizing Moving Geometries

Mobility is omnipresent in our everyday lives, and understanding it is becoming increasingly important for numerous practical and financial reasons. Mobility data is best understood using visualizations, which explains the interest in common open-source tools that can handle such data and create rich visualizations of it. This chapter presents *Moving Objects Visual Exploration* (MOVE), a PostgreSQL- and QGIS-based visualization tool to interactively visualize and explore moving objects data.

The stack of PostgreSQL, PostGIS, and QGIS[1] is popular for spatial data visualization and exploration. When working with spatio-temporal data, in particular mobility data stored in MobilityDB, QGIS is not natively able to visualize it natively, as it does not understand the MobilityDB temporal types. With this work, we aim to empower this community with a tool for mobility data visualization, by connecting QGIS to MobilityDB. MobilityDB can handle large mobility datasets and offers a wide range of transformation and aggregation operations in SQL, which are indispensable to produce complex viualizations and animations of mobility data.

The work presented in this chapter thus contributes to proposing a new tool to visualize and explore MobilityDB data in QGIS. This is done through a QGIS plugin providing a simple and interactive interface for the user. The MOVE plugin is available as open-source (Sect. 1.3) and works with the latest versions of MobilityDB and QGIS.

The rest of the chapter is structured in two sections. Section 7.1 starts by describing the architecture of the MOVE plugin used to connect QGIS to MobilityDB, and Sect. 7.2 lists multiple visualization examples that can be created using simple queries through the MOVE interface.

[1] `https://www.qgis.org`

7.1 The MOVE Plugin

The architecture of the visualization tool is composed of three parts: MobilityDB, QGIS, and the MOVE plugin linking these two systems. QGIS is a geospatial visualization tool. It can connect to multiple spatial data stores, including PostGIS, and display static geometry objects, such as points, linestrings and polygons. Using the Temporal Controller, it has the capability of displaying animated maps. As such, QGIS has the capability of displaying both static and animated representations of moving object data.

MOVE allows users to write SQL spatio-temporal queries and to visualize the results in QGIS. These queries are executed by MobilityDB, and the user has thus access to the complete set of operations offered by PostgreSQL, PostGIS and MobilityDB. These include for instance projection, distance, speed, azimuth, temporal aggregations, and temporal topological predicates. With this rich API, users can be creative in expressing both exploratory as well as analytical queries and visualize their results. On the QGIS side, users are additionally offered a wide variety of map displays and symbology options.

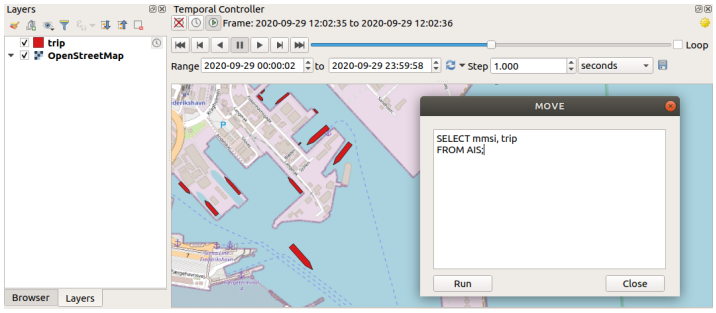


Fig. 7.1: MOVE User Interface

The MOVE plugin, displayed in Fig. 7.1, presents a simple but powerful interface to the user. When executing a `SELECT` query, the plugin inspects the types of the resulting columns and creates the appropriate layers in QGIS to visualize them. Columns storing PostGIS geometry types are displayed as they would have been by default. If the query returns MobilityDB spatio-temporal types, that is, `tgeompoint` or `tgeometry`, MOVE creates temporary database tables to approximate these spatio-temporal objects into a representation that uses native QGIS types, such as `LinestringM`. These tables are then added as layers in QGIS, marked as temporal, and can be animated using the QGIS built-in *temporal controller*. Additionally, indexes are built on the temporal and spatial columns of these tables. This allows users to scale

up their database, without compromising the responsiveness of the display in QGIS.

It is worth mentioning that MOVE chooses by design that all the layers it creates for visualization use native QGIS representations. As such, the user has access to the whole range of styling, annotation, etc. that is available in QGIS to enrich the visualization as needed. The symbology of the created layers is always set to default and can be modified by the user. Each layer also contains all the non-geometry columns returned by the initial query, and can thus also be used by the user as usual. Specifically, columns with scalar temporal properties are also handled by the plugin and can be displayed with the use of the DataPlotly[2] plugin. These columns are displayed as 2D line plots, with the time displayed on the x-axis and the scalar values on the y-axis.

7.2 Examples of Possible Visualizations

The following subsections present examples of possible visualizations, along with the queries that we used to generate them. These queries and visualizations use a maritime dataset published by the Danish Maritime Authority (Sect. 3.2.2). The data covers one day, September 29, 2020. The file size is 1.9 GB and contains 8 M AIS points of 1,788 different ships.

Data is loaded into MobilityDB in the table `AIS(mmsi integer, trip tgeometry, trippoint tgeompoint)`. The `mmsi` attribute is a unique ship identifier. The `trip` is a rigid temporal geometry representing the ship movement. The `trippoint` attribute represents the ship as a temporal point (`tgeompoint`) in case the shape and orientation information is of no interest to the user query. It has been computed using the operation `trajectory(trip)`. Since this operation is used in multiple queries of the following subsections, we precompute the result of this operation in column `trippoint` to simplify the queries. Next, we illustrate examples of possible visualizations on the AIS dataset.

7.2.1 Animated Points and Geometries

The main purpose of the MOVE plugin is to seamlessly produce animated visualizations of moving objects. Indeed, as mentioned in the previous section, producing these animations is not straightforward, as it requires transforming data from MobilityDB types to PostGIS types. This is handled by MOVE.

Query 7.1 `SELECT mmsi, trippoint AS tpoint FROM AIS;`

[2] `https://github.com/ghtmtt/DataPlotly`

Query 7.2 `SELECT mmsi, trip AS tgeom FROM AIS;`

Queries 7.1 and 7.2 can be used to display columns of temporal points and geometries, respectively. Running these queries in the plugin interface will automatically add a layer in QGIS with the temporal property activated. This displays the ships as animated polygons and points respectively, and it is possible to interact with the animation using the QGIS Temporal Controller.

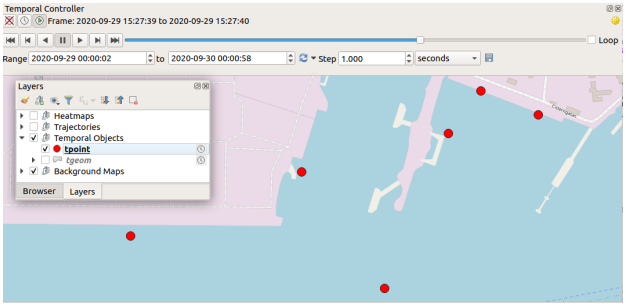


Fig. 7.2: Visualization of temporal points

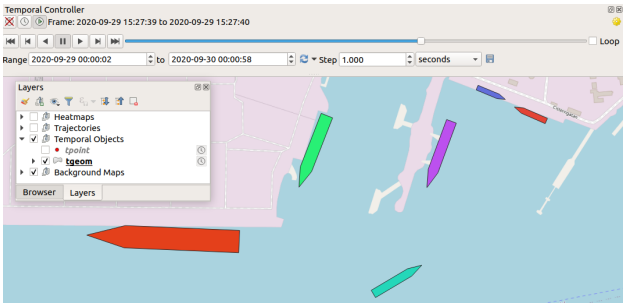


Fig. 7.3: Visualization of temporal geometries

Figures 7.2 and 7.3 display a snapshot of the layers produced by Queries 7.1 and 7.2, respectively. Notice that the added layer is already marked as temporal, as indicated with the clock symbol on the right of the layer names. Using the Temporal Controller visible at the top of the image, this data can be explored and animated.

7.2.2 Interactive Data Cleaning

When working with mobility data, errors can occur at multiple stages of the pipeline, and the data received by the database is thus not free of errors. Visualizing this data is essential to not only determine the nature of the errors but also verify that the cleaning process was successful. With the use of the plugin, this cleaning and verification process can be done interactively.

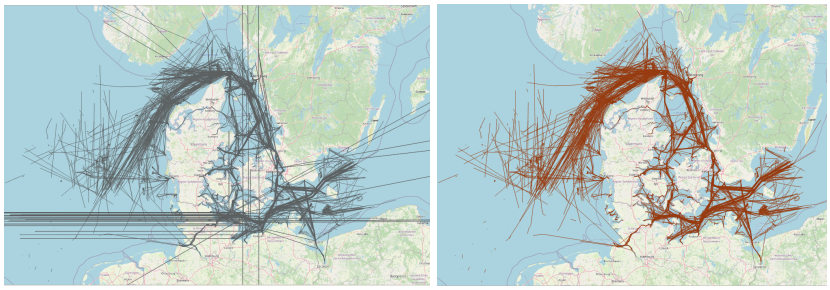
Query 7.3 *Trajectories before cleaning*

```
SELECT mmsi, trajectory(trippoint) FROM AIS;
```

Query 7.4 *Trajectories after cleaning*

```
SELECT mmsi,
    trajectory(atPeriodSet(trippoint,
        getTime(atValue(speed(trippoint) #< 30, True))
    )) AS trip
FROM AIS;
```

After loading the data in table `AIS`, we can display the trajectories of the ships using Query 7.3. Figure 7.4a displays the result of running this query in the plugin interface. In this figure, we can see long straight lines passing through solid terrain, which are errors in the data. These lines are present because some data points are not placed on the actual trajectory of the ship, but rather on an incorrect location far from the actual position of the ship.



(a) (b)

Fig. 7.4: Ship trajectories, before and after cleaning

A possible way to remove these errors is to compute the speed of the ship, and remove the segments of the trajectory having a speed higher than a certain threshold, such as 30 m/s. This can be done through the use of multiple MobilityDB functions on temporal points as displayed in Query 7.4. Figure 7.4b shows the result of Query 7.4, and the erroneous lines visible in Fig. 7.4a have indeed been removed. If this were not the case, a new query could have been run to continue this interactive cleaning process.

7.2.3 Heat Map of Trajectories

Query 7.5 *Heat map of the ship trajectories*

```
SELECT cell, count(*) as weight
FROM Grid, AIS
WHERE intersects(trippoint, cell)
GROUP BY cell;
```

Query 7.5 constructs a heat map using a second table `Grid`, storing the geometries of a regular square grid enclosing the vessel trajectories. Using the QGIS layer styling, we can then color the grid cells with a high weight. This creates the visualization shown in Fig. 7.5.

Fig. 7.5: Heat map of the ship trajectories

The lines shown in Fig. 7.5 display grid cells that were traversed by multiple ships on the same day, which could indicate navigation routes or zones with high traffic.

7.2.4 Traversed Area

Query 7.6 *Traversed area of a ship arriving at the docks*

```
SELECT mmsi, trip, traversedArea(trip)
FROM AIS WHERE mmsi = 538090508;
```

MobilityDB offers a wide range of operations to process temporal points and geometries, such as distance, intersection or restriction functions (Sect. 4.3).

An example of a complex operation on temporal geometries is `traversedArea`, which computes the area traversed by a temporal geometry as a PostGIS polygon and can be used, for example, to compute the closest point of approach of a ship to a fixed point on land. Query 7.6 computes the traversed area of a ship, and the result of this query is displayed in Fig. 7.6. This visualization shows that on arrival, the ship rotates and approaches the dock backwards. This is a detail that would not have been visible if the vessel was represented as a moving point, and thus shows the importance of representing certain moving objects as rigid temporal geometries.

Fig. 7.6: Traversed area of a ship arriving at the docks

7.2.5 Temporal Attributes

DataPlotly is a powerful plugin allowing QGIS to display attributes of a table using the Python Plotly API. This tool allows the creation of different plot types, such as scatter plots, histograms, bar plots and more.

Query 7.7 *Temporal count of ships at the port of Skage*

```
SELECT port, geom,
   tcount(atValue(tintersects(trippoint, geom), True)),
FROM Ports, AIS
WHERE port = 'Skagen' AND intersects(trippoint, geom)
GROUP BY port, geom;
```

Visualizing temporal attributes, such as temporal floats, integers or Booleans can also be of interest. The MOVE plugin thus allows DataPlotly to display these attributes as lines in a scatter plot, where the x and y-axes correspond to the time and value dimensions respectively. Figure 7.7 displays the result of Query 7.7, which counts the number of ships in the port of Skagen as a temporal integer. The figure shows the extent of the port on the map and the count of the ships as a stepwise function in a scatter plot.

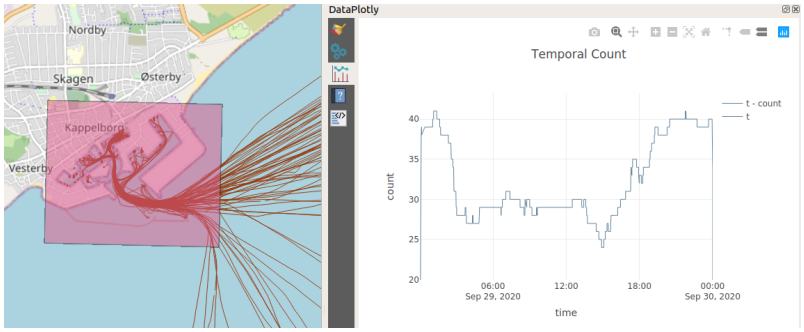


Fig. 7.7: Temporal count of ships at the port of Skage

7.2.6 Close Encounter

As the last example, the MobilityDB *tDwithin* operation can be used to find out when two temporal points are within a given distance of each other. Query 7.8 uses this operation to restrict the trajectories of ships to the times when they are within 100 meters of each other. This can for example be used to display close encounters of two ships at sea. Figure 7.8 shows a zoom of one close encounter returned by Query 7.8.

Query 7.8 *Return pairs of ships that passed within 100 meters of each other*

```
SELECT a.mmsi AS mmsi1, b.mmsi AS mmsi2,
    trajectory(atPeriodset(a.trip, getTime(
        atValue(tdwithin(a.trip, b.trip, 100), True)))) AS traj1,
    trajectory(atPeriodset(b.trip, getTime(
        atValue(tdwithin(a.trip, b.trip, 100), True)))) AS traj2,
    getTimestamp(NearestApproachInstant(a.trip, b.trip)) AS t,
    nearestApproachDistance(a.trip, b.trip) AS dist
FROM AIS AS a, AIS AS b
WHERE a.mmsi < b.mmsi;
```

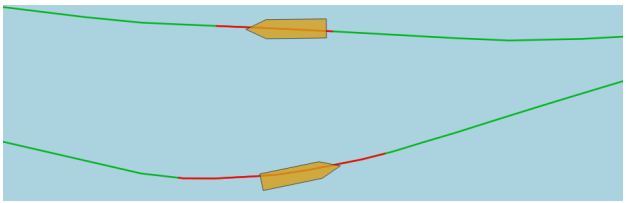
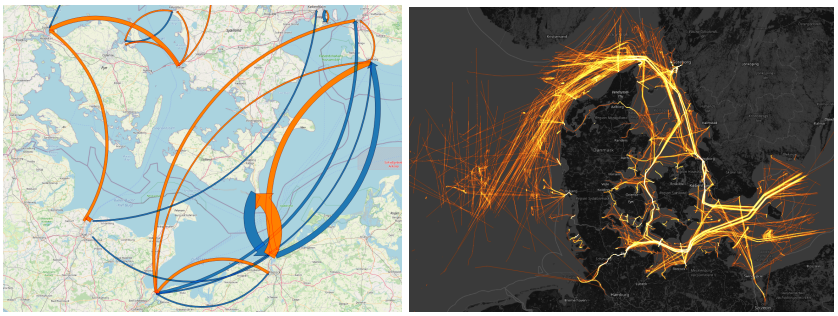


Fig. 7.8: Close encounter of two ships (< 100 m)

7.2.7 Advanced Visualizations

The described examples only form a subset of the visualizations possible using the plugin. More complex queries can be also executed to create more advanced visualizations, such as the flow map displayed in Fig. 7.9a. This can be achieved by combining both the wide range of operations in MobilityDB as well as the wide range of styling features in QGIS. Indeed, since the plugin adds QGIS layers to visualize the results of the queries, the users are then free to adapt the symbology of the data as they see fit. For example, Fig. 7.3 displays every ship in a random color, and Fig. 7.9b corresponds to Fig. 7.4b with different styling, giving a better impression of the density of the trajectories.



(a) Flow map between several ports (b) Density of ship trajectories

Fig. 7.9: Advanced visualizations using the styling features in QGIS

Chapter 8
Concluding Remarks

Ever since its emergence, the field of moving object databases has mainly been focused on providing solutions for moving points data. Another subset of moving objects are moving regions, which have a geometry that evolves over time. These moving regions can be categorized into two types: deforming regions and non-deforming regions, also called rigid temporal geometries. In this thesis, we extended the capabilities of moving object databases by proposing a new data model for rigid temporal geometries, complete with functions, operators, indexing, and visualization solutions.

Data Model

In a first step, Chap. 4 presented a new data model and implementation of rigid temporal geometries in the open-source moving object database MobilityDB. Delta encoding, through a new pose data type, is used to compactly represent these temporal geometries. We complement the data model with a large set of functions and operators to manipulate and analyze this data type. We also describe the International Standard ISO 19141—Schema for moving features, which specifies an abstract data representation of rigid temporal geometries. Our implementation conforms to this standard and also implements the operations for rigid temporal geometries mentioned in the standard. Experimental validation shows 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 indices. This is further utilized in the later chapters. 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.

Temporal Distance

When analyzing temporal regions, one typical use-case is computing their distance with respect to other static or moving geometries. In Chap. 5, we described the problem of computing the time-varying distance between a continuously moving body and other static and moving objects in 2D and presented algorithms to solve this problem efficiently. When the moving bodies are convex, the algorithm computes the evolution of their closest features in $\mathcal{O}(log(n) + k)$ time, where n is the total number of vertices, and k is the number of times the closest features change (size of the result). This evolution is stored as a list of length k that, together with the initial moving bodies, completely determines the equation of the temporal distance between the moving objects. This list can thus be used to determine the distance at a given timestamp in $\mathcal{O}(log(k))$ time.

Around this distance algorithm, multiple extensions have been described. Firstly, the algorithm was generalized for non-convex polygons. Secondly, we developed an optimization for the case where the movement contains no rotation. This case appeared to be the most common one in the test we performed on a real dataset. Finally, we described how this algorithm, together with an additional linear approximation step, is used to implement multiple distance operators in MobilityDB.

The experiments confirm the linear complexity of the algorithm in terms of k and show a 5-8× speed-up when using the optimized algorithms for non-rotating moving bodies. Additionally, we show that the increased complexity of representing vessels using moving polygons can result in increased precision during distance computations with only a 1.4× increase in computation time.

Trajectory Indexing

Chapter 6 introduced two generalizable index structures MGiST and MSP-GiST. They allow the indexing of complex or composite multidimensional data types by decomposing them into smaller parts before being indexed in a traditional GiST or SP-GiST index. This splitting mechanism is exposed to the user as a pluggable module, similar to the existing GiST and SP-GiST modules. This allows the index to be tailored to specific data types and operators.

We then detailed how the MGiST and MSP-GiST index can be used to efficiently index trajectory data as one example of complex multidimensional data. Using the pluggable modules, we implemented two existing and two newly proposed splitting mechanisms to build multi-entry variants of three well-known indices, the R-tree, quadtree and k-d tree. Evaluations on synthetic and real-world datasets showcase significant speedups compared to the single-entry counterparts for point, range and k-NN queries. Although this indexing solution is presented mainly for the case of moving points, we also

discussed how this solution can be used for rigid temporal geometries with little implementation effort.

Moving Object Visualization

Lastly, Chap. 7 presented MOVE, a visualization tool that integrates with MobilityDB and QGIS. MOVE presents a simple interface to interactively query and visualize spatial and spatio-temporal data. Using the extensive MobilityDB API and the large number of visualization options available in QGIS, the users can easily build rich visualizations. Sepcifically, MOVE creates transformations of the MobilityDB temporal types to be able to build static and animated visualizations of moving objects in QGIS. MOVE delegates the data processing to MobilityDB, which can handle large datasets and offers a wide range of transformation and aggregation operations in SQL. This solution builds on the open-source stack of PostgreSQL, PostGIS, MobilityDB and QGIS, and is thus available and ready-to-use for its large community of users and developers.

Directions for Future Work

While the contributions of this thesis provide a strong foundation, there are several promising directions for future research and development.

- **Deforming Geometries:** Future work could focus on extending the current model to support additional types of movement, such as deformable geometries. Previous work [25, 40] already discussed different models to store and manipulate deforming polygons. However, it is not clear how these models can be adapted to the MobilityDB type system. Additionally, there does not exist any international standard that describes this type of moving region, so the requirements of such an implementation are not clear.
- **3D Temporal Distance:** This paper presents a solution for computing the temporal distance between moving geometries in 2D, but the idea of tracking the closest features between two objects can also be applied in 3D. In this case, the moving bodies are either 3D points or polyhedrons, and the features are vertices, edges or faces of the objects. Future work would thus be to generalize this algorithm to compute the time-varying distance between 3D moving bodies.
- **ExtractQuery:** Section 6.2.4 discusses two main structural differences between GIN and MGiST/MSP-GiST. Another behavioral difference is the fact that GIN also allows queries to split their query value into multiple query predicates before searching the index. This allows GIN to answer questions such as: Which documents contain this specific sentence?

The sentence is split into words before querying the GIN index. Splitting is done via a key method called **ExtractQuery**. This ExtractQuery method could also be useful for trajectory, geometry, or time-series data. For example, when looking for all trajectories crossing the border between two countries, the query value is a line string. Splitting this line string in bounding boxes before querying the trajectory index could further reduce false positives and decrease query times.

- **QGIS MobilityDB Connector:** The data types understood by QGIS are limited to scalar, time and geometry types. To create animated visualizations, QGIS relies on a combination of a geometry column and time columns. Unfortunately, QGIS does not understand the MobilityDB temporal types natively. For this reason, MOVE has to transform the MobilityDB types into combinations of geometry and time types. This is not ideal, as it limits the maximum amount of objects that can be smoothly animated. Future work could focus on extending the native capabilities of QGIS to understand the MobilityDB temporal types, which could allow for more optimizations during animated visualizations.

Summary

In conclusion, this thesis contributes to the field of moving object databases by addressing key challenges in data modeling, query performance, and data accessibility. Future research inspired by this thesis could continue to push the boundaries of moving object databases, unlocking new possibilities for real-time analysis and decision-making in various fields such as transportation, urban planning, and environmental monitoring.

References

[1] G. Andrienko, N. Andrienko, P. Bak, D. A. Keim, and S. Wrobel. *Visual Analytics of Movement*. Springer Berlin, Heidelberg, 2014.

[2] N. Andrienko and G. Andrienko. V-analytics, 2010. `http://geoanalytics.net/V-Analytics/`.

[3] P. Aoki. Generalizing "search" in generalized search trees. In *Proc. of the 14th Int. Conf. on Data Engineering, ICDE*, pages 380–389, 1998.

[4] W. G. Aref and I. F. Ilyas. SP-GiST: An extensible database index for supporting space partitioning trees. *Journal of Intelligent Information Systems*, 17:215–240, 2001.

[5] W. G. Aref and H. Samet. Hashing by proximity to process duplicates in spatial databases. In *Proc. of the third Int. Conf. on Information and Knowledge Management, CIKM*, pages 347–354, 1994.

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

[7] F. Beier, T. Kilias, and K.-U. Sattler. GiST scan acceleration using coprocessors. In *Proc. of the Eighth Inter. Workshop on Data Management on New Hardware, DaMoN*, page 63–69, 2012.

[8] A. Borodin, S. Mirvoda, S. Porshnev, and M. Bakhterev. Improving penalty function of R-tree over generalized index search tree possible way to advance performance of PostgreSQL cube extension. In *Proc. of the second Int. Conf. on Big Data Analysis, ICBDA*, pages 130–133, 2017.

[9] Y. Cai and R. Ng. Indexing spatio-temporal trajectories with chebyshev polynomials. In *Proc. of the Int. Conf. on Management of Data, SIGMOD*, pages 599–610, 2004.

[10] J.-L. Chabert. *Newton's Methods*, pages 169–197. Springer Berlin, Heidelberg, 1999.

[11] V. P. Chakka, A. Everspaugh, J. M. Patel, et al. Indexing large trajectory data sets with SETI. In *Proc. of the Conf. on Innovative Data Systems Research, CIDR*, page 76, 2003.

[12] B. M. Chazelle. *Computational Geometry and Convexity*. PhD thesis, Yale University, USA, 1980.

[13] F. Chin and C. A. Wang. Optimal algorithms for the intersection and the minimum distance problems between planar polygons. *IEEE Transactions on Computers*, 32:1203–1207, 1983.

[14] H.-P.-I. Computer Graphics Systems. GTX - geo temporal explorer, 2021. `https://www.gtx-vis.org/`.

[15] J. A. Cotelo Lema, L. Forlizzi, R. H. Güting, E. Nardelli, and M. Schneider. Algorithms for moving objects databases. *The Computer Journal*, 46:680–712, 2003.

[16] P. Cudre-Mauroux, E. Wu, and S. Madden. TrajStore: An adaptive storage system for very large trajectory data sets. In *Proc. of the 26th Int. Conf. on Data Engineering, ICDE*, pages 109–120, 2010.

[17] E. B. Dam, M. Koch, and M. Lillholm. Quaternions, interpolation and animation. Technical report, Department of Computer Science, University of Copenhagen, 1998.

[18] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry*. Springer Berlin, Heidelberg, 2008.

[19] A. P. Del Pobil, M. A. Serna, and J. Llovet. A new representation for collision avoidance and detection. In *Proc. of the Int. Conf. on Robotics and Automation, ICRA*, pages 246–247, 1992.

[20] 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.*, 11:787–799, 2018.

[21] J.-P. Dittrich and B. Seeger. Data redundancy and duplicate detection in spatial join processing. In *Proc. of the 16th Int. Conf. on Data Engineering, ICDE*, pages 535–546, 2000.

[22] C. Düntgen, T. Behr, and R. Güting. BerlinMOD: A benchmark for moving object databases. *The VLDB Journal*, 18:1335–1368, 2009.

[23] H. Edelsbrunner. Computing the extreme distances between two convex polygons. *Journal of Algorithms*, 6:213–224, 1985.

[24] M. Eltabakh, R. Eltarras, and W. Aref. Space-partitioning trees in PostgreSQL: Realization and performance. In *Proc. of the 22nd Int. Conf. on Data Engineering, ICDE*, pages 100–100, 2006.

[25] L. Forlizzi, R. H. Güting, E. Nardelli, and M. Schneider. A data model and data structures for moving objects databases. In *Proc. of the Int. Conf. on Management of Data, SIGMOD*, pages 319–330, 2000.

[26] L. Fornasier, W. Kraus, and H. Rieger. A CFD-based, application-oriented, integrated simulation environment for rapid prediction of aerodynamic sensitivities of 3-D configurations. *SAE Transactions*, 106:1754–1759, 1997.

[27] Y. Gao, C. Li, G. Chen, Q. Li, and C. Chen. Efficient algorithms for historical continuous kNN query processing over moving object trajectories. In *Proc. of the Joint 9th Asia-Pacific Web and 8th Int. Conf. on Web-Age Information Management Conference*, pages 188–199, 2007.

[28] T. Ghanem, R. Shah, M. Mokbel, W. Aref, and J. Vitter. Bulk operations for space-partitioning trees. In *Proc. of the 20th Int. Conf. on Data Engineering, ICDE*, pages 29–40, 2004.

[29] F. Goerlandt and P. Kujala. Traffic simulation based ship collision probability modeling. *Reliability Engineering & System Safety*, 96:91–107, 2011.

[30] A. Graser. Visualisierung raum-zeitlicher daten in geoinformationssystemen am beispiel von quantum GIS mit "time-manager"-plug-in. In *Proc. of FOSSGIS*, pages 73–75, 2011.

[31] A. Graser and M. Dragaschnig. Open geospatial tools for movement data exploration. *KN-Journal of Cartography and Geographic Information*, 70:1–8, 2020.

[32] L. J. Guibas, D. Hsu, and L. Zhang. A hierarchical method for real-time distance computation among moving convex bodies. *Computational Geometry*, 15:51–68, 2000.

[33] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. of the Int. Conf. on Management of data, SIGMOD*, pages 47–57, 1984.

[34] R. H. Güting, T. Behr, and C. Düntgen. SECONDO: A platform for moving objects database research and for publishing and integrating research implementations. *Data Engineering Bulletin*, 33:56–63, 2010.

[35] R. H. Güting, T. Behr, and J. Xu. Efficient k-nearest neighbor search on moving object trajectories. *The VLDB Journal*, 19:687–714, 2010.

[36] R. H. Güting, M. H. Böhlen, M. Erwig, C. S. Jensen, N. A. Lorentzos, M. Schneider, and M. Vazirgiannis. A foundation for representing and querying moving objects. *ACM Transactions on Database Systems*, 25:1–42, 2000.

[37] M. Hadjieleftheriou, G. Kollios, V. J. Tsotras, and D. Gunopulos. Efficient indexing of spatiotemporal objects. In *Proc. of the 8th Int. Conf. on Extending Database Technology, EDBT*, page 251–268, 2002.

[38] F. Heinz and R. H. Güting. Robust high-quality interpolation of regions to moving regions. *GeoInformatica*, 20:385–413, 2016.

[39] F. Heinz and R. H. Güting. A data model for moving regions of fixed shape in databases. *Inter. Journal of Geographical Information Science*, 32:1737–1769, 2018.

[40] F. Heinz and R. H. Güting. A polyhedra-based model for moving regions in databases. *Inter. Journal of Geographical Information Science*, 34:41–73, 2019.

[41] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized search trees for database systems. In *Proc. of the 21th Int. Conf. on Very Large Data Bases, VLDB*, pages 562–573, 1995.

[42] S. Hertel and K. Mehlhorn. Fast triangulation of simple polygons. In *Proc. of the Foundations of Computation Theory*, pages 207–218, 1983.

[43] P. M. Hubbard. Interactive collision detection. In *Proc. of Research Properties in Virtual Reality Symposium*, pages 24–31, 1993.

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

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

[46] J. M. Keil. Decomposing a polygon into simpler components. *SIAM Journal on Computing*, 14:799–19, 1985.

[47] M. Kornacker, C. Mohan, and J. M. Hellerstein. Concurrency and recovery in generalized search trees. *SIGMOD Record*, 26:62–72, 1997.

[48] J. Kuan and P. Lewis. Fast k nearest neighbour search for R-tree family. In *Proc. of Int. Conf. on Information, Communications and Signal Processing, ICICS*, pages 924–928, 1997.

[49] 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 *Proc. of the 36th Int. Conf. on Data Engineering, ICDE*, pages 2002–2005, 2020.

[50] M. C. Lin and J. F. Canny. A fast algorithm for incremental distance calculation. In *Proc. of the Int. Conf. on Robotics and Automation, ICRA*, pages 1008–1014, 1991.

[51] M. C. Lin, D. Manocha, J. Cohen, and S. Gottschalk. Collision detection: Algorithms and applications. *Algorithms for robotic motion and manipulation*, pages 129–142, 1997.

[52] M. C. Lin, D. Manocha, and Y. J. Kim. *Collision and proximity queries*, pages 1029–1056. Chapman & Hall/CRC, 2017.

[53] A. R. Mahmood, S. Punni, and W. G. Aref. Spatio-temporal access methods: a survey (2010-2017). *GeoInformatica*, 23:1–36, 2019.

[54] M. Menze and A. Geiger. Object scene flow for autonomous vehicles. In *Proc. of the Conf. on Computer Vision and Pattern Recognition, CVPR*, pages 3061–3070, 2015.

[55] B. Mirtich. V-Clip: Fast and robust polyhedral collision detection. *ACM Transactions on Graphics*, 17:177–208, 1998.

[56] M. F. Mokbel, T. M. Ghanem, and W. G. Aref. Spatio-temporal access methods. *IEEE Data Engineering Bulletin*, 26:40–49, 2003.

[57] M. A. Nascimento and J. R. Silva. Towards historical R-trees. In *Proc. of the Symposium on Applied Computing, SAC*, pages 235–240, 1998.

[58] L.-V. Nguyen-Dinh, W. G. Aref, and M. Mokbel. Spatio-temporal access methods: Part 2 (2003-2010). *IEEE Data Engineering Bulletin*, 33:46–55, 2010.

[59] J. Ni and C. V. Ravishankar. PA-tree: A parametric indexing scheme for spatio-temporal trajectories. In *Proc. of the 9th Int. Conf. on Advances in Spatial and Temporal Databases, SSTD*, pages 254–272, 2005.

[60] OGC Open Geospatial Consortium. OGC Moving Features Access, 2016. `http://docs.opengeospatial.org/is/16-120r3/16-120r3.html`.

[61] I. F. D. Oliveira and R. H. C. Takahashi. An enhancement of the bisection method average performance preserving minmax optimality. *ACM Transactions on Mathematical Software*, 47:1–24, 2020.

[62] J. S. Park, C. Park, and D. Manocha. Efficient probabilistic collision detection for non-convex shapes. In *Proc. of the Int. Conf. on Robotics and Automation, ICRA*, pages 1944–1951, 2017.

[63] D. Pfoser, C. S. Jensen, Y. Theodoridis, et al. Novel approaches to the indexing of moving object trajectories. In *Proc. of 26th Int. Conf. on Very Large Data Bases, VLDB*, pages 395–406, 2000.

[64] M. Ponamgi, D. Manocha, and M. C. Lin. Incremental algorithms for collision detection between solid models. In *Proc. of the third ACM Symposium on Solid Modeling and Applications*, pages 293–304, 1995.

[65] PostGIS Project Steering Committee and Contributors. PostGIS – Spatial and Geographic objects for PostgreSQL, 2001-2024. `https://postgis.net/`.

[66] PostgreSQL Global Development Group. PostgreSQL: The World's Most Advanced Open Source Relational Database, 1996-2024. `https://www.postgresql.org/`.

[67] PostgreSQL Global Development Group. GIN Indexes, 2019-2024. `https://www.postgresql.org/docs/current/gin.html`.

[68] S. Quinlan. Efficient distance computation between non-convex objects. In *Proc. of the Int. Conf. on Robotics and Automation, ICRA*, pages 3324–3329, 1994.

[69] S. Ranu, P. Deepak, A. D. Telang, P. Deshpande, and S. Raghavan. Indexing and matching trajectories under inconsistent sampling rates. In *Proc. of the 31st Int. Conf. on Data Engineering, ICDE*, pages 999–1010, 2015.

[70] S. Rasetic, J. Sander, J. Elding, and M. A. Nascimento. A trajectory splitting model for efficient spatio-temporal indexing. In *Proc. of the 31st Int. Conf. on Very Large Data Bases, VLDB*, page 934–945, 2005.

[71] A. Rawson and E. Rogers. Assessing the impacts to vessel traffic from offshore wind farms in the thames estuary. *Scientific Journals of the Maritime University of Szczecin*, 43:99–107, 2015.

[72] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *Proc. of the Int. Conf. on Management of data, SIGMOD*, pages 71–79, 1995.

[73] Y. Sato, M. Hirata, T. Maruyama, and Y. Arita. Efficient collision detection using fast distance-calculation algorithms for convex and non-convex objects. In *Proc. of the Int. Conf. on Robotics and Automation, ICRA*, pages 771–778, 1996.

[74] M. Schoemans, W. G. Aref, E. Zimányi, and M. Sakr. Multi-entry generalized search trees for indexing trajectories. In *Proc. of the 32nd ACM Int. Conf. on Advances in Geographic Information Systems, SIGSPATIAL*, pages 1–5, 2024.

[75] M. Schoemans, M. Sakr, and E. Zimányi. On computing the time-varying distance between moving bodies. *ACM Transactions on Spatial Algorithms and Systems*, 9:1–28, 2023.

[76] M. Schoemans, M. Sakr, and E. Zimányi. Implementing rigid temporal geometries in moving object databases. In *Proc. of the 37th IEEE Int. Conf. on Data Engineering, ICDE*, pages 2547–2558, 2021.

[77] M. Schoemans, M. Sakr, and E. Zimányi. MOVE: interactive visual exploration of moving objects. In *Proc. of the Workshops of the 25th EDBT/ICDT Joint Conference*, pages 1–5, 2022.

[78] Z. Song and N. Roussopoulos. SEB-tree: An approach to index continuously moving objects. In *Proc. of the 4th Int. Conf. on Mobile Data Management, MDM*, pages 340–344, 2002.

[79] Y. Tao and D. Papadias. Efficient historical R-trees. In *Proc. of the 13th Int. Conf. on Scientific and Statistical Database Management, SSDBM*, pages 223–232, 2001.

[80] Y. Tao, D. Papadias, and Q. Shen. Continuous nearest neighbor search. In *Proc. of the 28th Int. Conf. on Very Large Databases, VLDB*, pages 287–298, 2002.

[81] Y. Theodoridis, M. Vazirgiannis, and T. Sellis. Spatio-temporal indexing for large multimedia applications. In *Proc. of the third Int. Conf. on Multimedia Computing and Systems, ICMCS*, pages 441–448, 1996.

[82] A. Tritsarolis, C. Doulkeridis, N. Pelekis, and Y. Theodoridis. ST_VISIONS: A python library for interactive visualization of spatio-temporal data. In *Proc. of the 22nd Int. Conf. on Mobile Data Management, MDM*, pages 244–247, 2021.

[83] E. Tøssebro and R. H. Güting. Creating representations for continuously moving regions from observations. In *Advances in Spatial and Temporal Databases*, pages 321–344, 2001.

[84] P. Valtr. Probability that n random points are in convex position. *Discrete & Computational Geometry*, 13:637–643, 1995.

[85] M. Vlachos, M. Hadjieleftheriou, D. Gunopulos, and E. Keogh. Indexing multidimensional time-series. *The VLDB Journal*, 15:1–20, 2006.

[86] L. Wang, Y. Zheng, X. Xie, and W.-Y. Ma. A flexible spatio-temporal indexing scheme for large-scale GPS track retrieval. In *Proc. of the 9th Int. Conf. on Mobile Data Management, MDM*, pages 1–8, 2008.

[87] X. Xu, J. Han, and L. Wei. RT-tree: An improved R-tree index structure for spatiotemporal databases. In *Proc. of the 4th Int. Symposium on Spatial Data Handling, SDH*, pages 1040–1049, 1990.

[88] C. L. Yang, M. Qi, X. X. Meng, X. Q. Li, and J. Y. Wang. New fast algorithm for computing the distance between two disjoint convex polygons based on voronoi diagram. *Journal of Zhejiang University Science*, 7:1522–1529, 2006.

[89] P. Zhou, D. Zhang, B. Salzberg, G. Cooperman, and G. Kollios. Close pair queries in moving object databases. In *Proc. of the 13th Annual Int. Workshop on Geographic information Systems, GIS*, pages 2–11, 2005.

[90] E. Zimányi. *MobilityDB Manual*. MobilityDB, 2020.
[91] E. Zimányi, M. Sakr, and A. Lesuisse. MobilityDB: A mobility database based on postgresql and postgis. *ACM Transactions on Database Systems*, 45:1–42, 2020.
[92] S. Zlatanova. 3D geometries in spatial DBMS. *Innovations in 3D Geo Information Systems*, pages 1–14, 2006.