ECOLE
POLYTECHNIQUE
DE BRUXELLES

ULB

UNIVERSITÉ LIBRE DE BRUXELLES

Creating Representations for Complex Moving Geometries in Databases

Fixed-shape Moving Regions in MobilityDB

Mémoire présenté en vue de l'obtention du diplôme
d'Ingénieur Civil en Informatique à finalité spécialisée

Maxime Schoemans

Promoteur
Professeur Esteban Zimanyi

Co-Superviseur
Mahmoud Sark

Service
Department of Computer and Decision Engineering [CoDE]

Année académique
2019 - 2020

Résumé

Résumé

Abstract

Abstract goes here

Acknowledgments

I want to thank...

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context

Temporal or spatio-temporal data exists in many types. It ranges from simple values evolving through time (e.g., temperature) to positions or even regions moving through time. Most current relational databases are not well-suited to handle this temporal data, and thus a lot of research has already been done to develop databases capable of storing and querying temporal data efficiently [1]. These databases are called *Moving Object Databases* [MOD].

MODs have to handle a large variety of (spatio-)temporal objects, also called *moving objects*. Moving reals can be used to represent the temperature of a room; moving points store the position 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. Still, previous research analyzed the construction of a moving region from snapshots [2, 3], or developed novel data models to represent moving regions efficiently [4, 5].

Despite all this previous work, most research ideas still have to be implemented into a large open-source database to be available for everyone to use. MobilityDB [6] aims to solve exactly that. It is an open-source extension on top of PostgreSQL [7] and PostGIS [8] that adds spatio-temporal types for moving points as well as temporal types for moving booleans, integers, reals and strings. The implementation of moving region concepts into MobilityDB still has to be done, and this is the topic of this Master thesis.

1.2 Structure and Objective of the Thesis

The first part of this thesis describes the current implementation of moving objects into MobilityDB. Section 2.3 details the structure of the system and the internal storage of the new types. The first goal of this thesis is to understand the system since the next part will involve making additions and modifications to it.

In a second step, Section 3 will discuss the different types of moving regions, with examples of real-world use cases. Depending on the type of moving region that needs to be handled, the implementation can vary considerably. For this reason, this master thesis will focus mainly on one type, fixed-shape moving regions, described in Section 3.2.

The next part concerns the implementation of fixed-shape moving regions into MobilityDB. The representation of moving regions will be discussed in Section 4.2, as well as a set of functions and operators that can be applied to this type in Sections 4.3, 4.4 and 4.5. The main goal of the thesis is to have a working implementation of a specific type of moving region in MobilityDB, with a large set of functions implemented that can be used

to process this data type. Implementation challenges and their solutions will be discussed for all implemented functions, and functions that remain unimplemented due to unsolved challenges will also be listed and described.

Finally, this thesis will expand on the previously implemented types and functions, by providing ideas and algorithms to later be able to extend the system to handle a wider set of moving regions. The objective of this part is thus to already think about possible improvements to the systems and define future work that could be done to be able to use the system with a wider set of moving regions, such as deforming (Section 5.4) or even three dimensional (Section 5.3) regions.

Section 6 will then summarize the contents and contributions of this thesis.

1.3 Contributions

The contribution of this master thesis can be listed as follows:

- Describe the state of the current research about moving regions in moving object databases, and discuss the advantages and disadvantages of the different proposed models.
- Develop a model for moving regions of fixed shape adapted from previous research and implement this model in practice as a new temporal type in MobiltiyDB.
- List and describe a set of SQL functions that can be applied to this new temporal type.
- Research, implement and describe multiple complex algorithms that are used in the current implementation of certain functions, or could be used in future work.

1.4 Terms and Notations

To make sure that there is no confusion or ambiguity, this section will define the terms and notations used in the rest of the thesis. Most of the terms will resemble the names already present in MobilityDB since this is the primary system that will be worked on.

- This master thesis will mainly work on moving regions. A static region will be represented by the PostGIS Polygon type. For this reason, the terms *region* and *polygon* will be used interchangeably throughout this thesis.
- Moving regions can be categorized into *deforming* regions and *non-deforming* regions. The first paper [5] making this distinction uses the term *fixed-shape* to denote these non-deforming regions. Based on this paper, other authors [4, 9] also started to use this terminology, and we will thus do the same.
- Most research papers use the notation `mbool`, `mfloat`, ... when talking about moving objects of respectively boolean or float base types. However, the notation in MobilityDB is different. There, moving objects are stored using *temporal types*, and these types use a notation starting with a `'t'`, followed by the base type. The two previous types thus become `tbool` and `tfloat`. This is the notation that will be used throughout this thesis.
- To denote moving points, again, most research papers use the notation `mpoint`. In MobilityDB, the notation again uses a `'t'` for temporal instead of an `'m'` for moving. A distinction is also made between the base types `geometry(Point)` and `geography(Point)` from PostGIS, so the resulting notations are `tgeompoint` and `tgeogpoint`.

- A moving region is usually written called `mregion` in previous research, but the type is not present in MobilityDB yet, so the notation still has to be defined. In [5], the distinction is made between a regular moving region (`mregion`) and a fixed-shape moving region (`fmregion`). Since we will not discuss the implementation of regular moving regions, this distinction will not be made here. The distinction between the base types `geometry(Polygon)` and `geography(Polygon)`, however, will be made, since the operations are not always the same in both cases. For the rest of this master thesis, moving regions will be called `tgeometry` and `tgeography`, depending on their respective base type.
- Based on the three previous points, we will also use the terms *moving* and *temporal* interchangeably, to denote a value that changes through time.

Chapter 2

Systems Used

2.1 PostgreSQL

PostgreSQL [7] is an open-source object-relational database management [ORDBMS] system 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 database management system [DBMS].

2.2 PostGIS

PostGIS [8], 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.

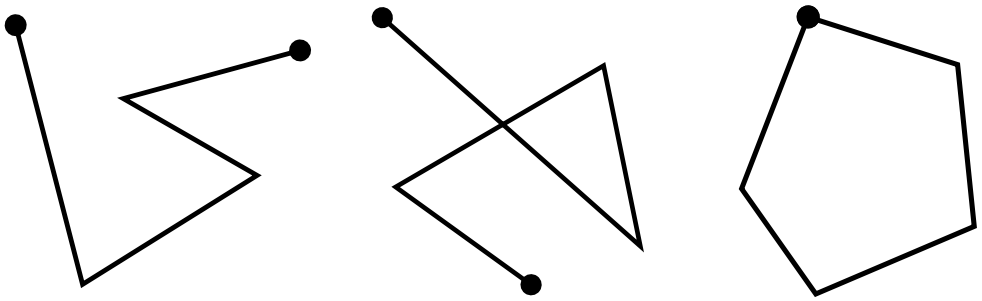
This extension allows *Geographical Information System* [GIS] objects, such as points, linestring or polygons, to be stored in the database. It also includes support for processing and analyzing GIS objects, by defining functions and operations that can be used to process them, such as distance and area 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.

2.2.1 PostGIS Polygon Type

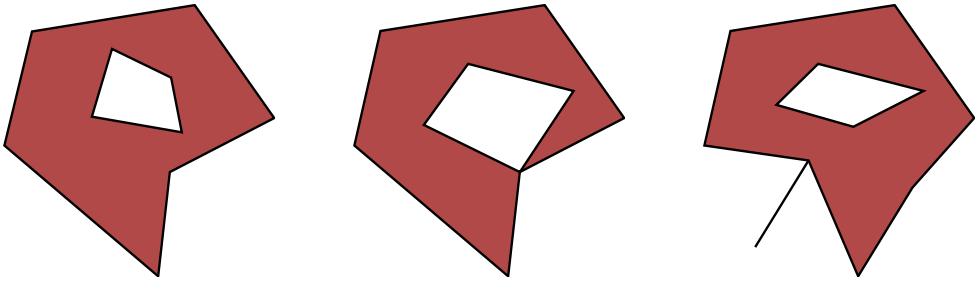
A PostGIS polygon object is defined as having an exterior ring, as well as zero or more interior rings that define holes in the polygon. The PostGIS *linestring* type is used to describe these rings. A linestring is a list of points representing a continuous line composed of multiple segments. For this linestring to be valid, it cannot intersect itself at any point (Figure 2.1a). 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* (Figure 2.1c). Both the exterior and interior rings of a polygon are described using closed linestrings.

A PostGIS polygon is valid if no two of its rings intersect (Figure 2.2a), except in a single point on the boundary of the polygon (Figure 2.2b). The polygon cannot have lines or spikes (2.2c), and the interior rings cannot be defined outside of the exterior ring.



(a) (b) (c)

Figure 2.1: Examples of valid (a, c) and invalid (b) PostGIS linestrings



(a) (b) (c)

Figure 2.2: Examples of valid (a, b) and invalid (c) PostGIS polygons

Contrary to most research papers about moving regions [4, 10, 11], 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.

2.3 MobilityDB

MobilityDB [6] is an extension to PostgreSQL and PostGIS that provides *temporal types*, used to represent the evolution on time of some base type. For example, the temperature of a room can be represented by a real value that changes through time. In this case, the temporal type is a *temporal float* (`tfloat`), with the base type being `float`. As another example, a *temporal integer* (`tint`) can be used to represent the number of people on a train. Similarly, a *temporal point* (`tgeompoint` or `tgeogpoint`) can be used to store the position of a taxicab, as reported by a GPS device.

MobilityDB makes use of the predefined operations on the base type, such as arithmetic operations and aggregations for integers and floats or spatial functions for geometries, to define new operations that can handle temporal types.

MobilityDB uses four time types to represent extents of time: `timestamptz`, `timestampset` , `period` and `periodset`. Timestamptz is a PostgreSQL type, while the three other types are new. Two new range types are also defined in MobilityDB: `intrange` and `floatrange`.

2.3.1 Temporal Types

There are six existing temporal types, `tbool`, `ttext`, `tint`, `tfloat`, `tgeompoint` and `tgeogpoint` , based on, respectively, the base types `bool`, `text`, `int`, `float`, `geometry(Point) and`

`geography(Point)`. The last two are the PostGIS types for points, restricted to 2D and 3D.

Temporal types can have four possible duration: `Instant`, `Instant Set`, `Sequence` and `Sequence Set`. These durations define the temporal extent at which the evolution of values is defined.

Temporal Instant

A temporal instant represents the value of an object at a particular timestamp using the notation

$$'v@t',$$

where v is the value and t is the timestamp.

For example, if we want to store the fact that the temperature of a room was 21 (degrees Celsius) on September 1st 2019 at midnight, we will store this as

```
tfloat '21@2019-01-01 00:00:00'
```

Temporal Instant Set

As the name hints, temporal instant sets are simply sets of temporal instants. Temporal objects of instant set duration are used to represent a value that is defined at multiple instants in time, without being defined in between these instants. For example, a temporal instant set composed of 3 instants will be written as

$$'\{v_0@t_0,\ v_1@t_1,\ v_2@t_2\}',$$

where v_i are the values and t_i are the timestamps in increasing order $t_i < t_{i+1}$.

The value of this object is thus defined only at the given timestamps, and no assumptions are made for intermediate timestamps.

Temporal Sequence

A temporal sequence is defined just as a temporal instant set, except that the values for intermediate timestamps can be obtained by interpolation. The value of the temporal object is thus assumed to be defined over the whole period of the sequence.

Depending on the base type of the object, two interpolation methods are possible, stepwise/discrete or linear/continuous. The base types bool, text, and integer are forced to use a discrete interpolation method, whereas the base types float, geometry and geography can use either one depending on the use case.

For example, when using a `tint` to represent the number of people in a bus, if a new temporal instant is added every time someone steps in or out of the bus, we can get the number of people in the bus at any time during its trip, by simply looking at the latest defined value before that instant. This is an example of a stepwise interpolation for temporal integers.

An example of a linear interpolation could be the case of a room thermometer. If we measure the temperature sufficiently often, we can assume that the temperature between two measures can be obtained by interpolating linearly between these two values. These measurements will thus be stored in a temporal sequence with float base type, and intermediate values will be retrieved using a linear interpolation method.

A temporal sequence composed of 3 instants is written as

$$'[v_0@t_0,\ v_1@t_1,\ v_2@t_2]',$$

with again $t_i < t_{i+1}$.

A temporal sequence has a lower and upper bound that can be either inclusive (represented by '[' and ']') or exclusive (represented by '(' and ')').

Temporal Sequence Set

A temporal value of sequence set duration represents the evolution of the value over a set of sequences, with the value between these sequences being unknown. A temporal sequence set with 2 sequences of 2 instants each will be written as

$$'\{[v_0@t_0,\ v_1@t_1],\ [v_2@t_2,\ v_3@t_3]\}',$$

where the same inclusion/exclusion rules as for sequences apply.

It would theoretically be enough to only implement the temporal sequence set type since all three previous types are essentially special cases of this one, but for efficiency reasons, all four types exist.

2.3.2 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

- `Period`: for the types `tbool` and `ttext`, where only the temporal extent is taken into account (1D box).
- `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 (2D box).
- `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 (4D box).

For example, a temporal int of instant set duration: `tint '{1@2001-01-01, 3@2001-01-02, 2@2001-01-03}'`, will have a corresponding Tbox: `Tbox((1, 2001-01-01), (3, 2001-01-03))`. The first tuple of the bounding box contains the minimum values for all dimensions (here only X and T), and the second tuple contains the maximum values for these same dimensions.

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

2.3.3 Functions and Operators for Temporal Types

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, and they can be grouped into the categories described below.

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 [12].

- Transformation Functions:

  These functions are used to transform a temporal value into another duration. If the transformation is not possible, an error will be raised.

```
SELECT tintseq(tint '1@2001-01-01');
-- "[1@2001-01-01]"
```

Appending a temporal instant to the temporal value, and merging two temporal values of the same duration, is also a part of the transformation functions.

```
SELECT appendInstant(tint '1@2001-01-01', tint '1@2001-01-02');
-- "{1@2001-01-01, 1@2001-01-02}"
```

- 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 Functions

  Spatial functions that can be applied to temporal points. Most of the functions support 3D points, and/or are available for geographies.

  These functions range from the 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;
-- "{[0@2001-01-01, 0@2001-01-02], [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`, `atPeriod`, etc.

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

- Difference Functions:

  These functions act similarly to the restriction functions, except that they restrict the temporal value to the complement of a value or a time extent.

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

- Comparison Operators

  Implementation of the traditional comparison operators ('=', '<' and so on). Except for equality and inequality, these are not useful for real-world use cases, but they allow the construction of B-tree indexes on temporal types.

- Ever and Always Equals Operators

  These operators test whether a temporal value is ever equal ('`&=`') or always equal ('`@=`') to a value.

- Temporal Comparison Operators

  These comparison operators are a generalization of the traditional comparison operators and return a *tbool* value.

  For example, the temporal not equal operator can be applied applied using the symbol `#<>`.

```
SELECT tint '[1@2001-01-01, 4@2001-01-04]' #<> 2;
-- "{[true@2001-01-01, false@2001-01-02, true@2001-01-03, true@2001-01-04]}"
```

- Mathematical Function Operators

  The basic mathematical functions, such as addition, multiplication, rounding and more.

- Boolean Operators

  Boolean `and`, `or` and `not`.

- Text Functions and Operators

  Text concatenation, lowercase and uppercase.

- 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);
-- true
```

- 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 relationships 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, `intersects` and `relate`). The second version is defined with temporal semantics and returns a temporal value. For the two example functions, the second version is `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 return a temporal value.

2.3.4 Additional Notations

When talking specifically about a temporal type of instant duration, we will add a suffix `inst` to the name of the type. For example, a temporal float of instant duration will be denoted `tfloatinst`. Similarly, for temporal instants of instant set, sequence and sequence set duration, we will add respectively the suffix `i`, `seq` and `s`. For the base type float, this will thus become `tfloati`, `tfloatseq` and `tfloats`.

Chapter 3

Moving Regions

Real-world examples of moving regions are everywhere. The movement of a car can be represented by a moving rectangle, forest fires, pollution areas and glacier extents can be represented by regions whose boundaries change over time, etc. Difficulties arise when we want to store and query these moving regions efficiently using moving object databases. Multiple different models have already been developed [4, 5, 10], each one with positives and negatives. Most of these models make certain assumptions on the type of moving region or on the transformation that they undergo.

The literature makes a distinction between two major types of moving regions: deforming regions, analyzed and described by multiple authors [4, 10, 11], and fixed-shape regions, described in [5]. Another distinction that can be made is the difference between 2D and 3D regions.

We will first discuss the difference between deforming and fixed-shape moving regions in the 2D case, and then briefly discuss 3D regions.

3.1 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 many 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.

An example of a use case where a deforming moving region changes in discrete time steps is a cadastre. This type of moving regions is briefly discussed in Section 5.4.

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 to efficiently compute these interpolations has to be described as well. Since the start and end region might look very different, an additional difficulty arises when we want to construct 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 most common model used to represent a deforming moving region uses a *sliced representation* [10, 13] (in the time axis) of the moving region, which defines a transformation from a start to an end region. There exist also ways of creating this sliced representation starting from snapshots of the moving region [2].

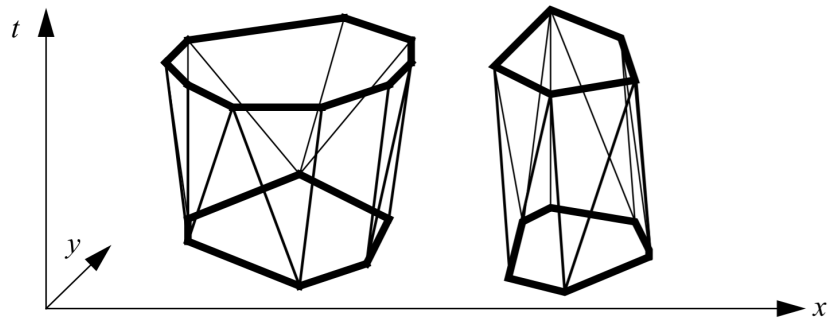


Figure 3.1: Sliced representation of moving regions [4]

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.

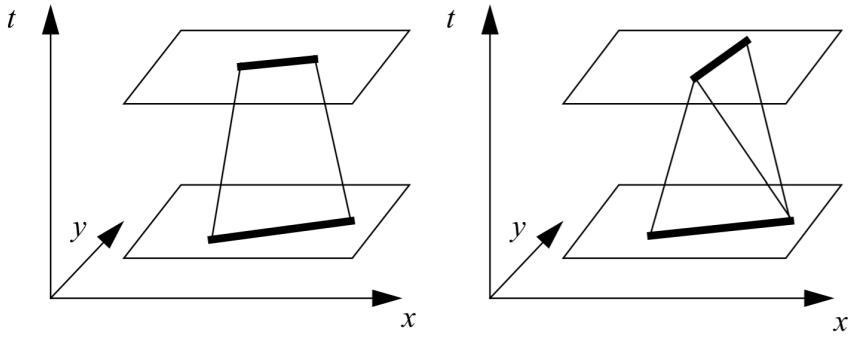


Figure 3.2: Sliced representation of moving segments, with rotating segments being represented using 2 distinct moving segments. [4]

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 [4], 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.

3.2 Fixed-Shape Moving Regions

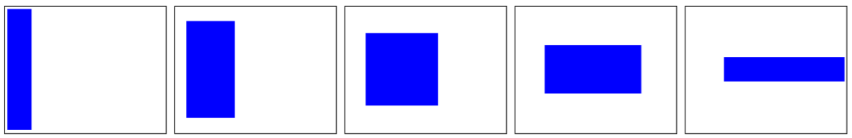
Fixed-shape moving regions form another subset of all moving regions and can be used to represent any moving rigid body, such as cars, boats, planes, etc. When the spatial extent of the object is negligible, and the orientation of no importance, the movement of such a body is usually represented by a 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 here 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.

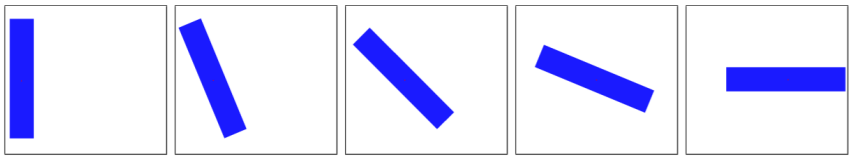
When we manage continuous moving regions, we again have to find a way to compute the interpolation efficiently. The previous model for deforming regions relies on moving segments, which are supposed to be non-rotating. This assumption is contradictory to our idea of rotating (and translating) regions, 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 Figure 3.3.



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

Figure 3.3: Interpolation of a moving region using linear interpolation between the vertices. [5]

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 [5], 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` in this paper, and an example of a desired interpolation for this type of region is seen in Figure 3.4.



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

Figure 3.4: Interpolation of a moving region using the fixed-shape assumptions. [5]

The model makes use of a *transformationunit*, which specifies a rotation centre 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.

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

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

$$\textit{fmregion} : (\mathcal{R}, \mathcal{T}^+)$$

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 centre for each transformation unit. This particularity allows the moving region to change rotation centre between two transformation units, which is useful to describe

a larger amount of trajectories than if the rotation centre was fixed for all transformations. On the other hand, this also requires the user to input the rotation centre for each transformation, since this cannot be uniquely computed by just seeing the start and end position of the region.

The same paper [5] 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.

3.3 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 fixed-shape will be discussed here. To even just discuss 3D deforming moving regions would require a much larger amount of research.

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 will define a 3D region as a volumetric geometry, thus representable by a 3D polyhedron.

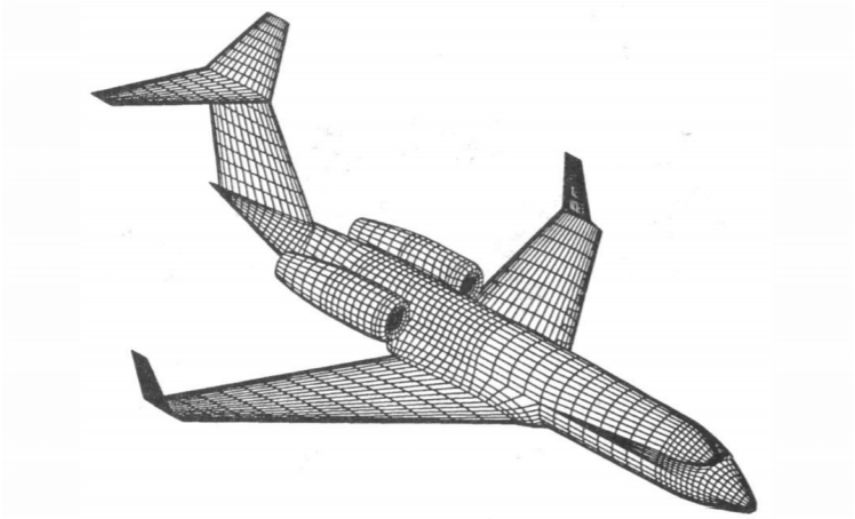


Figure 3.5: Representation of a plane using a 3D polyhedron [14]

Support for 3D geometries is currently extremely restricted in database management systems [15]. However, PostGIS does support two types of 3D geometries: *Polyhedral Surfaces* and *Triangulated Irregular Network Surfaces* [TIN], which are polyhedral surfaces where every polygon is a triangle. A few 3D functions are also implemented, such as `ST_3DClosestPoint` or `ST_3DDistance`.

First, let's look at the case where the regions move in discrete time steps. In this case, like previously, 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.

If we assume fixed-shape regions, we can again use the fact that the only possible transformations are translations and rotations. It is thus sufficient to store the region once as a 3D polyhedron, and then 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, we can apply the same storage idea, and retrieving the position of the region between two stored instants by computing a linear interpolation of the translation, just as was done in 3D, and computing a more advanced interpolation for the 3D rotation, based on *quaternions*.

This interpolation method will be discussed in a later section (Section 5.3), but the important part is that it can be done. Storing 3D moving regions of fixed-shape is thus possible, and retrieving the region at any arbitrary instant is too. Implementation of functions that can manipulate and analyze these regions is another issue and is left as future research.

Chapter 4

Implementation

4.1 Choice of Moving Region Type

The previous section describes two different approaches to handle moving regions, each using different assumption, and each having good and bad aspects. Implementing both is not a reasonable goal for this Master thesis, and a decision had to be made to decide which type of moving region to focus on.

The choice was made to focus on implementing a model for manipulating moving regions of fixed-shape, based on the model presented in [5] and described previously. This decision was mostly arbitrary, based on personal preference, but not only.

MobilityDB already has an underlying structure for the new data types, and this structure has to be preserved as much as possible when implementing a new feature, and this was also a factor in the final decision. The deforming region model uses a *sliced representation* of the movement, whereas MobilityDB uses a *sequence representation* described in [6], so the model would have to be heavily adapted to conform to the MobilityDB structure. Using a sequence representation for storing fixed-shape moving regions seems much more straight forward.

For the rest of the thesis, we can thus assume that we are always talking about regions of fixed-shape, except if explicitly mentioned that we talk about deforming regions.

4.2 Representation of Fixed-Shape Moving Regions in MobilityDB

Now that the choice of the type of region is done, the next challenge is finding a suitable representation for these moving regions in MobilityDB. As said previously, the structure of the system has to be preserved as much as possible, which means that a temporal region type will be required for all four durations: `Instant`, `Instant Set`, `Sequence` and `Sequence Set`. These four types will be called: `tgeometryinst` (for *temporal geometry of instant duration*), `tgeometryi`, `tgeometryseq` and `tgeometrys`.

4.2.1 Type `tgeometryinst`

A moving region of instant duration is a static region, with an associated timestamp. As said in Section 2.2, the PostGIS type `Geometry(Polygon)` will be used to describe a static region. A `tgeometryinst` can thus be represented similarly to any other temporal type of instant duration, that is: as a pair of value and timestamp

$$'v@t',$$

where the value is a PostGIS polygon object, and t is the timestamp. Since only one instant is present, there is no redundancy, and this is the most efficient way to store this type.

4.2.2 Type `tgeometryi`

As described in Section 2.3.1, a temporal value of instant set duration is simply a set of distinct timestamps stored in increasing order of their timestamp. In this case, an additional requirement on the values is present, namely the fact that all regions must be of the same shape.

A simple and naive representation for a `tgeometryi` is then to store all instants in the set using the previous representation. That is, store for each instant a pair of polygon and timestamp. However, since we have added the extra requirement on the polygons, namely that their shape has to be the same, we are introducing a lot of redundancy in this representation, and the storage is thus not very efficient.

Suppose that all the previously mentioned requirements are met, we can store the `tgeometryi` more efficiently, by storing the polygon only once and storing the subsequent instants using only the transformation with respect to the initial instant.

Using this representation, a `tgeometryi` with n instants is written as

$$'\{\mathcal{R}_0@t_0,\ \mathcal{T}_1@t_1,\ \mathcal{T}_2@t_2,\ ...,\ \mathcal{T}_{n-1}@t_{n-1}\}',$$

where $\mathcal{R}_0$ is the initial region, and $\mathcal{T}_i$ are the transformation units.

Section 3.2 describes a possible transformation unit representing a transformation from a start to an end instant. In our case, only the end instant has to be stored, since the start instant will always be represented by the reference region $\mathcal{R}_0$ at time t_0. Since the start and end timestamp are also already stored elsewhere, namely in the timestamps t_0 and t_i, we only need to store the rotation centre C, the translation vector $(v^x,\ v^y)$ and the rotation angle θ in the transformation unit.

$$\mathcal{T}_i : (C_i, v_i^x, v_i^y, \theta_i)$$

If we want to compute the transformation unit starting from the initial and final polygons, we need to also define a rotation centre for all the other parameters to be uniquely defined. For example, if we look at the start and end polygon (dark blue) shown in Figure 4.1, multiple transformation units can be defined, all having the same initial and final position, but they differ in their rotation centre and the path they take. The blue path has a fixed rotation centre at (3.5, 1), and there is no translation involved, while the red path uses the centroid of the polygon as the rotation centre, and involves a translation by 5 units along the x-axis.

We also know that a temporal instant set is defined only at its instants, and not in between. This means that we only care about the start and end position of a region, but we do not care about the path taken during the transformation.

Since the decision of the rotation centre only changes the path taken by the region during the transformation, the previous observation tells us that we can choose the rotation centre arbitrarily when computing the transformation unit.

The rotation centre takes a certain amount of storage space, and since we just realized that we can choose it arbitrarily, we will define the rotation centre of all transformation units as being the centroid of the region at instant t_0. With the rotation centre fixed, we

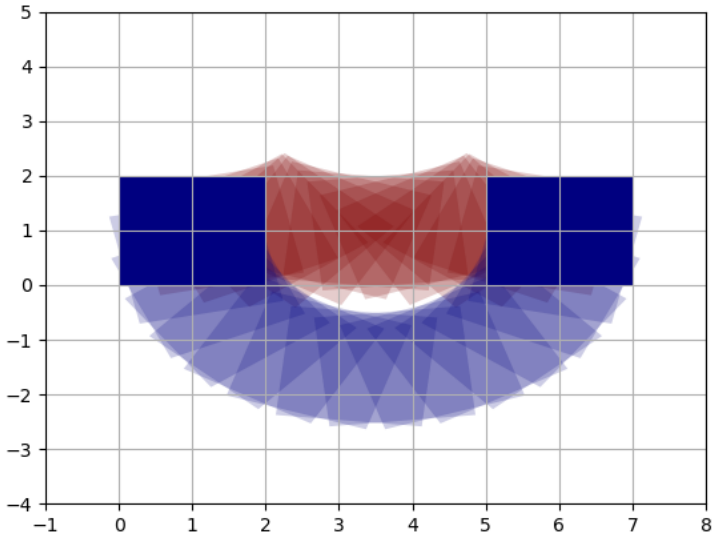


Figure 4.1: Importance of the centre of rotation for a transformation.

can save storage space by only storing the translation vector and the rotation angle for each transformation.

$$\mathcal{T}_i : (v_i^x, v_i^y, \theta_i)$$

Using this representation, we can store a `tgeometryi` using minimal storage space. The two remaining challenges are: how do we compute a transformation unit starting from an initial and final polygon value, and how do we recompute the position of the region, starting from an initial polygon, and a transformation unit? The solution to these two problems is described in Sections 4.3.1 and 4.3.2.

As a side note, the current implementation recomputes the centroid of the polygon (centre of rotation) using the PostGIS function `ST_Centroid` every time it is needed, but if we later realize that this computation is done often and is not very efficient, we can also compute it once during the input of the moving region, and then store it with the region in the first instant.

4.2.3 Type `tgeometryseq`

A temporal region of sequence duration represents the continuous movement of a region, where the position of the moving region can be retrieved at every instant between the initial and final instant. Here, we thus not only care about the position of the region at the input instants, but we also need to be able to interpolate the region between two defined instants.

As we realized previously, when computing a transformation unit starting from the initial and final position of the polygon, the choice of the rotation centre defines the path taken during the movement uniquely. This means that the rotation centre will have an impact on the interpolation.

One important assumption that was implicitly made for the `tgeometryi` type, is the fact that a `tgeometryseq` is input as a list of `tgeometryinst`. We are thus not getting any extra information other than the snapshots of the regions at specific instants, and,

in particular, we do not know what the rotation centre is for every transformation. This means that we need to define a rotation centre ourselves for the transformations to be well-defined.

Having to define a rotation centre has multiple consequences. First of all, since there is no way to know what the real centre of rotation was simply based on the snapshots of the region at the given instants, the only solution we have is to decide on an arbitrary centre of rotation. Secondly, since the arbitrary centre of rotation will most probably not be the real centre of rotation, there will be errors between the real path taken by the region and the interpolated path.

Since we know that we have to define an arbitrary centre of rotation, the most obvious solution is to take the centroid of the region as the rotation centre, since this point is well-defined for all regions and is easy to compute. As you might remember, this is the same solution as for `tgeometryi`, so we can represent a `tgeometryseq` as

$$'[\mathcal{R}_0@t_0,\ \mathcal{T}_1@t_1,\ \mathcal{T}_2@t_2,\ ...,\ \mathcal{T}_{n-1}@t_{n-1}]',$$

with $\mathcal{T}_i$ being the transformation defined with respect to $\mathcal{R}_0$.

$$\mathcal{T}_i : (v_i^x, v_i^y, \theta_i)$$

The algorithmic challenges for this data type are the same as for `tgeometryi`, with the addition of one problem. This problem is stated in the following way: given a polygon defined at time t_0, and two transformation units defined at times t_i and t_{i+1}, both defined with respect to the initial polygon, how do we compute the position of the region at any given time t, where $t_i < t < t_{i+1}$? The solution to this problem is given in Section 4.3.3.

One last unsolved problem is the fact that the interpolated path does not exactly follow the real path taken by the moving region when the real rotation centre of the moving region is not its centroid. Solving this issue algorithmically would require us to know the real path taken by the moving region and thus the real rotation centre, which, as we explained previously, is not possible when we only get snapshots of the region at distinct instants.

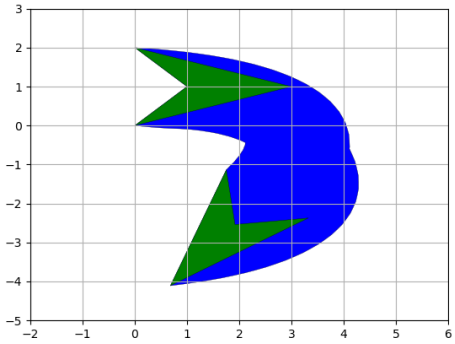
The only solution to this problem without changing the assumptions made is to give as many snapshots of the region in the input as needed to minimize the difference between the interpolated path and the real path. Indeed, we can achieve any arbitrary precision by increasing the number of instants given in the input between the initial and final position of the region. An example of how giving more instants can increase the precision of the interpolation is shown in Figure 4.2.

Figure 4.2a displays the real path (in blue) taken by a moving region (in green). Given only the start and end instants (in green) (Figure 4.2b), the computed path is quite far from reality. This computed path improves as more intermediate instants (in green) are given. The blue paths shown in Figure 4.2 are computed using the traversed area function described in Section 4.5.4.

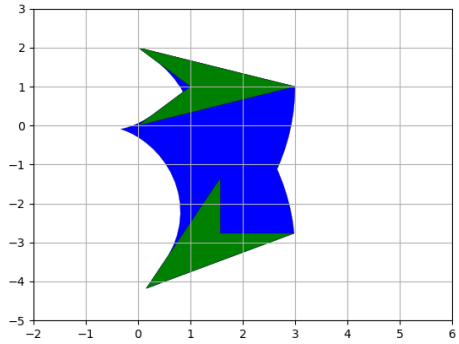
4.2.4 Type `tgeometrys`

The last type that we have to handle is `tgeometrys`, which is a temporal geometry of sequence set duration. The input to this type is a list of `tgeometryseq`. This means that we already receive some of the instants as transformations defined with respect to the first instant of their sequence.

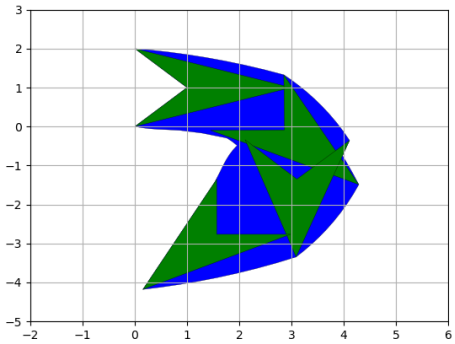
For this to be a valid `tgeometrys`, we first need to check the fact that all regions have the same shape. Since this check was already made independently for each sequence when



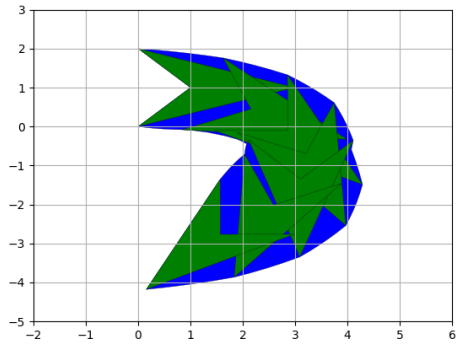
(a) Real path



(b) Only start and end instant given



(c) Two intermediate instants given



(d) Five intermediate instants given

Figure 4.2: Computing the path of a moving region defined using different number of instants

they were created, we thus only need to make sure that the polygons of the first instants of every sequence have the same shape.

Assuming that all sequences describe the movement of polygons of the same shape, we have two possibilities to store this set of sequences. The first possibility is to store every sequence as it was received

$$'\{[\mathcal{R}_0^0@t_0^0,\ \mathcal{T}_1^0@t_1^0,\ ...],\ [\mathcal{R}_0^1@t_0^1,\ \mathcal{T}_1^1@t_1^1,\ ...],\ ...\}',$$

with every transformation $\mathcal{T}_i^j$, being defined with respect to the region $\mathcal{R}_0^j$ at the start of the sequence.

This solution is easy since we do not have to do any more preprocessing, but it also has some redundancies, since we know that the first region of each sequence has the same shape.

The second possibility is thus to remove the redundancy by replacing the region values of the first instants of all sequences after the first by a transformation value. For all sequences after the first one, we have to compute the transformation of the first instant of the sequence $\mathcal{T}_0^j$ with respect to the first instant of the first sequence of the set $\mathcal{R}_0^0$. All other transformations of the sequence would have to be modified too, to have the start region being the first instant of the first sequence.

$$'\{[\mathcal{R}_0^0@t_0^0,\ \mathcal{T}_1^0@t_1^0,\ ...],\ [\mathcal{T}_0^1@t_0^1,\ \mathcal{T}_1^1@t_1^1,\ ...],\ ...\}'$$

In this representation, every transformation $\mathcal{T}_i^j$ is defined with respect to the region $\mathcal{R}_0^0$ at the start of the first sequence.

This option requires some more preprocessing but requires less storage space than the previous option. Another disadvantage of this method is the retrieval of a sequence in the set. This operation requires us to re-transform the whole sequence to its input representation if the required sequence is not the first one.

Depending on the use case, the average number of sequences in a set, the average number of instants in each sequence and a few more parameters, either the first option or the second option could prove to be more interesting.

The option that has been implemented in practice is the first one, as it seems to be the easiest one, and seems to require cleaner and less code than the second option. Implementing the second option and comparing their efficiency is left as future research.

4.2.5 Type `rtransform`

In three of the last four types, a transformation unit was used to represent an instant of a moving region more efficiently than by using a polygon. This transformation unit is stored in the database using a new type called `rtransform` (short for *region transformation*). As explained previously, this type only needs to store information about the translation vector $(v^x,\ v^y)$ and rotation angle θ of the final value of the region, since we assume that we know what the start instant is, and we also assume the centre of rotation to be the centroid of the region. Internally, all parameters of the `rtransform` type will be represented using the `double` type.

$$\text{rtransform } \mathcal{T} : (\text{double } v_x,\ \text{double } v_y,\ \text{double } \theta)$$

Since we know that rotating by θ or rotating by $\theta + 2\pi$ gives the same final result, we will add a constraint on the rotation angle stored in an `rtransform` object: $-\pi < \theta \leq \pi$. This constraint allows us to compute a unique transformation unit when starting from an initial and final polygon.

An `rtransform` like this always has to be related to a polygon object, usually the polygon stored in the value of the first instant of the sequence, which will be the initial polygon. We can then apply the `rtransform` as explained in 4.3.2 to the start polygon, to retrieve the final polygon. Retrieving the polygon at an instant that is in between two stored instants, will require us to interpolate between two saved transformations to retrieve the final polygon. This interpolation method is explained in Section 4.3.3.

4.3 Utility Functions for `tgeometry` and `rtransform` Types

Before diving into the implementation of the general MoblityDB functions that have to be adapted to handle moving regions as well as all the types already present, we will first talk about a set of functions specific to the `tgeometry` and `rtransform` types.

These functions can be called utility functions, since they are not presented to the user as an SQL function, but they are used in the backend a lot to simplify other functions. Table 4.1 shows a list of these functions with their signature (input and output types).

Functions	Signature
`compute`	`polygon` × `polygon` → `rtransform`
`apply`	`rtransform` × `polygon` → `polygon`
`interpolate`	`rtransform` × `rtransform` × `double` → `rtransform`
`interpolate_long`	`rtransform` × `rtransform` × `double` → `rtransform`
`combine`	`rtransform` × `rtransform` → `rtransform`
`diff`	`rtransform` × `rtransform` → `rtransform`

Table 4.1: Utility functions for the `rtransform` type

4.3.1 Compute

The `compute` function takes two polygon values as input and computes the transformation from the first to the second one. For example, taking the two polygons shown below as input, with the blue polygon being the first argument and the red polygon being the second, the resulting `rtransform` will be $\mathcal{T}: (1, \frac{1}{3}, \frac{-\pi}{2})$.

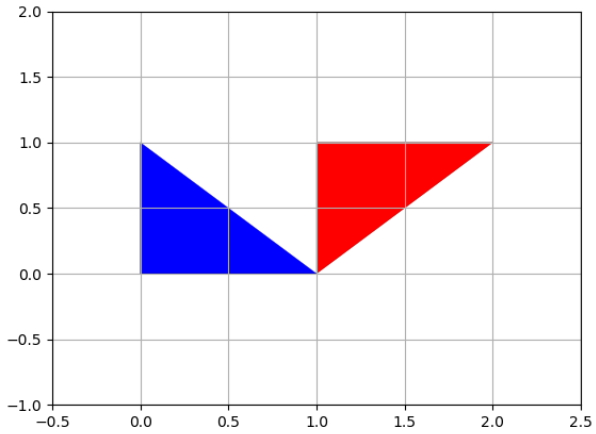


Figure 4.3: Example polygons used in the `compute` function

As said previously the type used to represent a region value is the PostGIS type `Geometry(Polygon)`. This polygon type is defined using rings: one outer ring, and zero or more inner rings defining holes in the polygon. One assumption that we make when receiving the polygons as input is that two polygons of the same shape have their corresponding points defined at the same position in the rings of said polygons.

The rings are defined as lists of points, with the first and the last points being equal to denote that the ring is closed. Our previous assumption thus says that the i'th point in the list defining the outer ring of the first polygon has to correspond to the i'th point in the list defining the outer ring of the second polygon, after application of the correct transformation.

Essentially this means that the points of the polygons are 'numbered' in the same way. This allows us, for example, to distinguish a square from its 90-degree rotations.

The combination of rotating by θ around the origin, followed by a translation by (v^x, v^y), can be described using matrix multiplication.

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\theta) & \sin(\theta) & v^x \\ \sin(\theta) & -\cos(\theta) & v^y \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Using the notation

$$\begin{cases} a = \cos(\theta) \\ b = \sin(\theta) \\ c = v^x \\ d = v^y \end{cases},$$

and by taking the first two points of the outer rings of both input polygons, we arrive at the following list of equations

$$\begin{cases} a * x_1 - b * y_1 + c = x'_1 \\ a * y_1 + b * x_1 + d = y'_1 \\ a * x_2 - b * y_2 + c = x'_2 \\ a * y_2 + b * x_2 + d = y'_2 \end{cases},$$

with (x_1, y_1) and (x_2, y_2) being the first two points of the first polygon, and (x'_1, y'_1) and (x'_2, y'_2) being the corresponding points on the second polygon.

We can then solve these equations for a, b, c and d

$$\begin{cases} a = \frac{(x'_1 - x'_2)*(x_1 - x_2) + (y'_1 - y'_2)*(y_1 - y_2)}{(x_1 - x_2)^2 + (y_1 - y_2)^2} \\ b = \frac{(y'_1 - y'_2)*(x_1 - x_2) - (x'_1 - x'_2)*(y_1 - y_2)}{(x_1 - x_2)^2 + (y_1 - y_2)^2} \\ c = x'_1 - a * x_1 + b * y_1 \\ d = y'_1 - a * y_1 - b * x_1 \end{cases},$$

and finally we get the parameters of the transformation.

$$\begin{cases} \theta = atan2(b,\ a) \\ v^x = c \\ v^y = d \end{cases}$$

As said previously, this computes a transformation which is the combination of first a rotation θ applied around the origin, and then a translation $(v^x,\ v^y)$. To use these equations, we thus need the centroid of the first polygon (the centre of rotation for this transformation) to be at the origin. To solve this issue, we first compute the centroid of the first region, then translate both regions using the same translation, to have the new centroid of the first region being at the origin. Finally, we can use the above equations to compute the parameters of the transformations.

When we also need to verify that both input polygons have the same shape, we can apply this computed transformation to the start polygon, and compare all corresponding points with the end polygon after the transformation. If the distance between the position of two corresponding points differs by more than a given ϵ, we can assume that the two regions do not have the same shape.

4.3.2 Apply

This `apply` function does the opposite of the `compute` function. Given an `rtransform` and a start polygon, it applies the transformation to retrieve the end polygon. This is done

using a PostGIS function called `ST_Affine`, which applies an affine transformation to the given geometry.

Applying an affine transformation again means rotating around the origin, so we first have to translate the polygon to have its centroid at the origin, then we can apply the given transformation, and finally, we apply the inverse of the first translation to get the final polygon.

4.3.3 Interpolate

This function is used when we have a polygon defined at time t_1 and another one defined at time t_2, and we want to retrieve the position of the polygon at time t, where $t_1 < t < t_2$ using linear interpolation. Of course, when using stepwise interpolation this function is not needed, since if we are looking for the polygon at time t, where $t_1 < t < t_2$, the needed polygon is simply the one defined at time t_1.

If we wanted to interpolate two polygons using linear interpolation between their vertices, we would not keep the fixed-shape aspect of the polygon, as is in Figure 3.3. For this reason, the interpolation is done on the `rtransforms` defining the position of the polygon at times t_1 and t_2 with respect to a certain reference polygon.

For example, using a `tgeometryseq` with three instants defined at times t_0, t_1 and t_2

$$'[\mathcal{R}_0@t_0,\ \mathcal{T}_1@t_1,\ \mathcal{T}_2@t_2]',$$

suppose that we want to compute the position of the region at time $t_1 < t < t_2$. To do this, we first apply the `interpolate` function with $\mathcal{T}_1$ and $\mathcal{T}_2$ as arguments and then apply the resulting `rtransform` to $\mathcal{R}_0$ to get the result.

The `interpolate` function takes three input arguments. The first two arguments are the initial and final `rtransform` values. The third argument is a ratio value defined by the timestamp of the resulting `rtransform`. This ratio is always between 0 and 1 and is computed as follows.

$$ratio = \frac{t - t_1}{t_2 - t_1}$$

Since we assume linear interpolation between the `rtransform` values, we will simply use linear interpolation to compute the resulting translation value.

$$\begin{cases} v^x = v_1^x + (v_2^x - v_1^x) * ratio \\ v^y = v_1^y + (v_2^y - v_1^y) * ratio \end{cases}$$

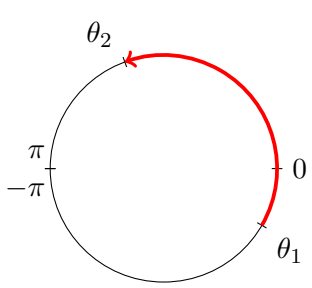
The interpolation of the angle of rotation, on the other hand, is slightly more complicated. We are still interpolating linearly, but we have to take into account that the angles $-\pi$ and π correspond to the same rotation. This means that there are always two possibilities for the interpolation. Either go directly from θ_1 to θ_2 or go from θ_1 to $-\pi$ (if $\theta_1 < \theta_2$) and then from π to θ_2. Both are valid interpolations, but one is shorter than the other. The shorter option is the option that will be taken for the interpolation of the angles.

(a) Go directly from θ_1 to θ_2

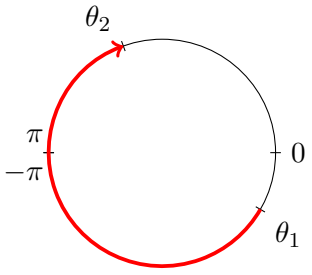
(b) Go from θ_1 to $-\pi$, then from π to θ_2

Figure 4.4: Interpolation of angles on a linear axis

Any angle of rotation of $-\pi < \theta \leq \pi$ can be represented by a point on the unit circle at $(\cos(\theta), \sin(\theta))$, which means that going from one angle to the other during the interpolation is the same as moving from one point to the other along the arc of the unit circle. The two possible ways of interpolating angles are thus represented by the two arcs connecting the points, one being longer than the other.



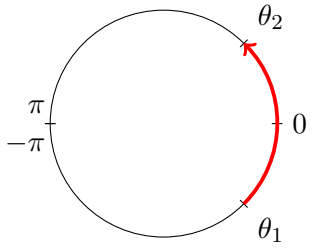
(a) Shorter arc connecting θ_1 and θ_2



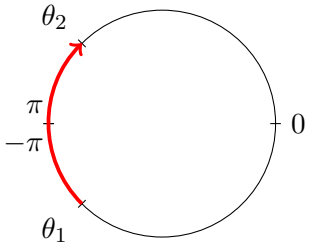
(b) Longer arc connecting θ_1 and θ_2

Figure 4.5: Interpolation of angles on the unit circle

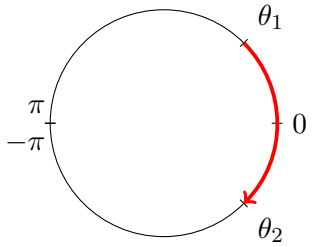
Using this unit circle representation, we can then visualize four different ways of interpolating two angles depending on the values of the two angles. These four cases are shown in Figure 4.6.



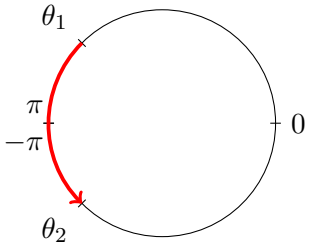
(a) $\theta_1 < \theta_2$ and $\theta_2 - \theta_1 <= \pi$



(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 $\theta_1 - \theta_2 >= \pi$

Figure 4.6: Four different situations when interpolating angles of rotation

Computing the interpolated angle is then done using one of the four equations below, depending on in which of the cases shown in Figure 4.6 we currently are.

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

Finally, if the resulting angle is larger than π, we subtract 2π from it to keep the angle 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 $\theta_1 - \theta_2 = \pi$

Figure 4.7: Interpolation of a 180 degree rotation

4.3.4 Interpolate Long

The `interpolate_long` function does the same as the `interpolate` function, except that during the interpolation of the angles, it always takes the longest arc instead of the smallest. This function will be useful in Section 4.5.2 when normalizing a `tgeometryseq`.

4.3.5 Combine

The `combine` function essentially does the addition of two `rtransform` value. With the input `rtransform` objects being written as $(v_1^x,\ v_1^y,\ \theta_1)$ and $(v_2^x,\ v_2^y,\ \theta_2)$, the resulting `rtransform` is computed using the following equations.

$$
\begin{aligned}
v^x &= v_1^x + v_2^x \\
v^y &= v_1^y + v_2^y \\
\theta &= \begin{cases}
\theta_1 + \theta_2 - 2\pi, & \text{if } \pi < \theta_1 + \theta_2 \\
\theta_1 + \theta_2 + 2\pi, & \text{if } \theta_1 + \theta_2 \leq -\pi \\
\theta_1 + \theta_2, & \text{otherwise}
\end{cases}
\end{aligned}
$$

This function is used when we need to change the start/reference region of the `rtransform` , which is used for example when merging multiple `tgeometry` objects.

4.3.6 Difference

The `diff` function is similar to the compute function, except that it does the difference between the first and the second input argument, instead of doing an addition.

$$
\begin{aligned}
v^x &= v_2^x - v_1^x \\
v^y &= v_2^y - v_1^y \\
\theta &= \begin{cases}
\theta_2 - \theta_1 - 2\pi, & \text{if } \pi < \theta_2 - \theta_1 \\
\theta_2 - \theta_1 + 2\pi, & \text{if } \theta_2 - \theta_1 \leq -\pi \\
\theta_2 - \theta_1, & \text{otherwise}
\end{cases}
\end{aligned}
$$

When we want to know what the transformation is between two instants, when both instants are saved as `rtransforms` relative to a third instant, we can use simply do the

difference between the `rtransform` of the final instant and that of the initial instant to get the result.

For example, if we have a `tgeometryi` with 3 instants $'\{\mathcal{R}_0@t_0,\ \mathcal{T}_1@t_1,\ \mathcal{T}_2@t_2\}'$, and we want to get the transformation to go from the second instant to the third, we can apply `diff`$(\mathcal{T}_2,\ \mathcal{T}_1)$.

4.4 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 Section 2.3.3, 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. The next section lists and describes most of these functions, and explains how they are used to manipulate the `tgeometry` type.

4.4.1 Type Declaration Functions and Parameters

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, two new types have been added to MobilityDB: `rtransform` and `tgeometry`. Four variations of the `tgeometry` type (`tgeometryinst`, `tgeometryi`, `tgeometryseq` and `tgeometrys`) are also present at the C-level, but only these two top-level types are exposed at the SQL level. Table 4.2 lists the declared parameters and functions for the `rtransform` type, and Table 4.3 described the `tgeometry` type.

Parameter	**Value**	**Description**
`internal length`	24	Internal storage length in bytes.
`alignment`	`double`	Storage alignment of the type.
Function		**Description**
`input`		Conversion from `cstring` to `rtransform` type
`output`		Conversion from `rtransform` to `cstring` type
`receive`		Conversion from `bytea` to `rtransform` type
`send`		Conversion from `rtransform` to `bytea` type

Table 4.2: Declared parameters and functions for the `rtransform` type

The `rtransform` type stores three double (8-byte) values, two for the translation and one for the rotation, and has thus a fixed length of 24 bytes. Since values of this type will never be store alone, but always as part of a `tgeometry` value, the storage strategy does not need to be declared. During the input from a string, we make sure that the value for the rotation is between $-\pi$ and π. The output as a string is formatted as

$$\text{"RTransform}(\theta,\ v_x,\ v_y)\text{"},$$

where all three values are written with up to 15 decimal digits. The binary representation is simply a list of three doubles in the same order as in the string representation.

Parameter	Value	Description
`internal length`	`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

Table 4.3: Declared parameters and functions for the `tgeometry` 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 Section 2.3 (instant, instant set, ...), and represents the values of the instants as PostGIS polygon types. These conversion functions from and to a string are defined together with the conversion functions from and to a byte list.

4.4.2 Constructors

Four different constructors exist for `tgeometry` types, one for each duration.

Function	Signature
`tgeometryinst`	`geometry(Polygon)` × `timestamptz` → `tgeometry(Instant)`
`tgeometryi`	`array tgeometry(Instant)` → `tgeometry(Instant Set)`
`tgeometryseq`	`array tgeometry(Instant)` × `lower inclusive` × `upper inclusive` × `linear` → `tgeometry(Sequence)`
`tgeometrys`	`array tgeometry(Sequence)` → `tgeometry(Sequence Set)`

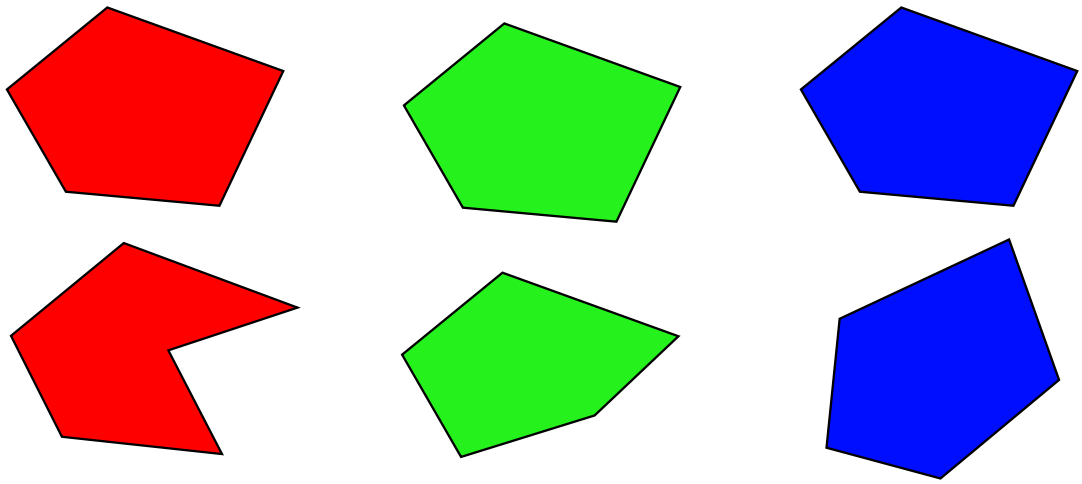
Table 4.4: Constructor functions

When creating a `tgeometryinst`, the input geometry is checked to make sure that it is a 2D polygon. Both the instant set and sequence constructors receive an array of `tgeometryinst` as input, while the sequence set constructor requires an array of `tgeometryseq` . Different checks are done to make sure that this input array corresponds to a correct temporal geometry.

First of all, when creating an instance set or a sequence, the usual timestamp checks are done, meaning that the instants must have strictly increasing timestamps. Lastly, the regions are compared with one another, 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 4.8 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).



(a) Different number of vertices

(b) Different shape

(c) Same shape, only transformed

Figure 4.8: Polygons failing or passing the different constructor checks

If all polygons are indeed fixed-shape, the input array is accepted and saved using the representation described in Sections 4.2.2 and 4.2.3.

The sequence constructor has three additional parameters. The first two specify if the start and end instant are inclusive or not and by default both are considered inclusive. The last parameter specifies the interpolation strategy for the sequence, either linear or stepwise. The sequence set constructor has checks similar to the two previous ones, but since the input durations are already assumed to be sequences, only the comparisons between distinct sequences have to be done. During these comparisons, checks are done to assure continuously increasing timestamps, and fixed-shape polygons across the whole sequence set.

Every `tgeometry` instance of either instant set, sequence or sequence set duration also computes and saves a bounding box used for optimization purposes. Section 4.4.10 lists a few bounding-box specific functions, and Section 4.5.1 explains how these bounding boxes can be computed. Temporal points of sequence and sequence set duration also save a *trajectory*, which is also used to optimize a few functions. The polygon equivalent of this trajectory is the *traverser area* of the polygon. The computation of the traversed area of a moving region is discussed in Section 4.5.4.

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 is further discussed in Section 4.5.2.

4.4.3 Transformation Functions

Table 4.5 lists the transformations functions that can be applied to `tgeometry` objects. The first four 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`. For example, for a temporal geometry to be transformed into a `tgeometryinst`, it must contain a single instant. Transforming

into an instant set from a sequence set is only possible if every sequence in the set contains a single instant. The six possible transformations are listed below.

- `Instant` $'\mathcal{R}@t' \leftrightarrow$ `Instant Set` $'\{\mathcal{R}@t\}'$
- `Instant` $'\mathcal{R}@t' \leftrightarrow$ `Sequence` $'[\mathcal{R}@t]'$
- `Instant` $'\mathcal{R}@t' \leftrightarrow$ `Sequence Set` $'\{[\mathcal{R}@t]\}'$
- `Instant Set` $'\{\mathcal{R}@t\}' \leftrightarrow$ `Sequence` $'[\mathcal{R}@t]'$
- `Instant Set` $'\{\mathcal{R}_0@t_0,\ ...,\ \mathcal{R}_n@t_n\}' \leftrightarrow$ `Sequence Set` $'\{[\mathcal{R}_0@t_0],\ ...,\ [\mathcal{R}_n@t_n]\}'$
- `Sequence` $'[\mathcal{R}_0@t_0,\ ...,\ \mathcal{R}_n@t_n]' \leftrightarrow$ `Sequence Set` $'\{[\mathcal{R}_0@t_0,\ ...,\ \mathcal{R}_n@t_n]\}'$

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

Function	**Signature**
`tgeometryinst`	`tgeometry` → `tgeometry(Instant)`
`tgeometryi`	`tgeometry` → `tgeometry(Instant Set)`
`tgeometryseq`	`tgeometry` → `tgeometry(Sequence)`
`tgeometrys`	`tgeometry` → `tgeometry(Sequence Set)`
`appendInstant`	`tgeometry` × `tgeometry(Instant)` → `tgeometry`
`merge`	`tgeometry` × `tgeometry` → `tgeometry`
`merge`	`array tgeometry` → `tgeometry`

Table 4.5: Transformation functions

4.4.4 Accessors

A large list of accessor functions exists to retrieve the duration, memory size, individual values, instants, timestamps and more. (Table 4.6)

`Duration` returns one of {`"Instant"`, `"Instant Set"`, `"Sequence"`, `"Sequence Set"`}, `interval` returns either `"Discrete"` for instants and instant set, or one of {`"Linear"`, `"Stepwise"`} for sequences and sequence sets, 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 4.6, 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 Nth element, and an array containing all elements respectively. This set is defined for the `Timestamp`, `Instant` and `Sequence` elements. A complete list of these functions can be found in Appendix A.

4.4.5 Always/Ever Comparison

These functions and operators compare a temporal geometry with a fixed region value (Polygon) 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.

Function	Signature
duration	tgeometry → text
interpolation	tgeometry → text
memSize	tgeometry → integer
getValue	tgeometry(Instant) → geometry(Polygon)
startValue	tgeometry → geometry(Polygon)
endValue	tgeometry → geometry(Polygon)
getTimestamp	tgeometry(Instant) → timestamptz
getTime	tgeometry → periodset
timespan	tgeometry → interval
shift	tgeometry → tgeometry
num*	tgeometry → integer
start*	tgeometry → *
end*	tgeometry → *
*N	tgeometry × integer → *
*s	tgeometry → array *

Table 4.6: Accessor functions

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

Table 4.7: Always/Ever comparison functions and operators

4.4.6 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) value to a timestamp, or even a set of period values. The full set of restriction functions is listed in Table 4.8, and Table 4.9 lists the difference functions.

Function	Signature
atValue	tgeometry × geometry → tgeometry
atValues	tgeometry × array geometry → tgeometry
atTimestamp	tgeometry × timestamptz → tgeometry
atTimestampSet	tgeometry × timestampset → tgeometry
atPeriod	tgeometry × period → tgeometry
atPeriodSet	tgeometry × periodset → tgeometry

Table 4.8: Restriction functions

For example, if we want to remove the data in a temporal geometry between to given

timestamps, we can use the `minusPeriod` function with as second argument a period starting at the first and ending at the second timestamp.

Function	Signature
minusValue	tgeometry × geometry → tgeometry
minusValues	tgeometry × array geometry → tgeometry
minusTimestamp	tgeometry × timestamptz → tgeometry
minusTimestampSet	tgeometry × timestampset → tgeometry
minusPeriod	tgeometry × period → tgeometry
minusPeriodSet	tgeometry × periodset → tgeometry

Table 4.9: Difference functions

4.4.7 Comparison Operators

A set of comparison operators is defined in PostgreSQL, and they have been extended to accept arguments of type temporal geometry. 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 on the `tgeometry` type.

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 4.10: Comparison functions and operators

4.4.8 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.

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

Table 4.11: Temporal comparison functions and operators

4.4.9 Spatial Functions

As has been done for the temporal points previously in MobilityDB (briefly described in Section 2.3.3), multiple spatial functions similar to those defined in PostGIS are exposed.

Table 4.12 shows a non-exhaustive list of spatial functions that can be applied to the `tgeometry` type.

Function	Signature
`traversed_area`	`tgeometry` → `geometry(Polygon)`
`centroid`	`tgeometry` → `tgeompoint`
`rotation_speed`	`tgeometry` → `tfloat`
`nearestApproachDistance`	{`tgeometry`, `geometry(Polygon)`} × {`tgeometry`, `geometry(Polygon)`} → `float`
`nearestApproachInstant`	{`tgeometry`, `geometry(Polygon)`} × {`tgeometry`, `geometry(Polygon)`} → `tgeometry(Instant)`
`shortestLine`	{`tgeometry`, `geometry(Polygon)`} × {`tgeometry`, `geometry(Polygon)`} → `geometry(LineString)`
`distance`	{`tgeometry`, `geometry(Polygon)`} × {`tgeometry`, `geometry(Polygon)`} → `tfloat`

Table 4.12: Spatial functions

The `traversed_area` function returns a polygon representing the area traversed by the moving region. The computation of this area is described in Section 4.5.4.

`Centroid` is a variant of the `ST_Centroid` PostGIS function, returning the evolution of the centroid of a temporal region. Since in our case the centroid is assumed to be the centre of rotation, it moves linearly between two defined instants, and a temporal point containing the position of the centroid of a temporal region can thus be easily computed and returned.

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 centroid of the moving region, only the rotation speed has to be handled. This is thus returned by the `rotation_speed` function, which returns a temporal float. Since we assume linear interpolation between instants, the rotation speed is assumed to be constants between to instants.

The four last 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 three first functions can be implemented easily if the `distance` 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 `distance` function, which, due to its complexity and time constraints, is left as future work.

4.4.10 Bounding Box Functions and Operators

Bounding boxes are precomputed for temporal geometries of instant set, 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 an empty value 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 4.5.1 explains 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`).

Function	Operator	Signature
`stbox`		`tgeometry` → `stbox`
`expandSpatial`		`tgeometry` × `float` → `stbox`
`expandTemporal`		`tgeometry` × `interval` → `stbox`
`contains_bbox`	`@>`	{`tgeometry`, `geometry`, `stbox`} × {`tgeometry`, `geometry`, `stbox`} → `stbox`
`contained_bbox`	`<@`	{`tgeometry`, `geometry`, `stbox`} × {`tgeometry`, `geometry`, `stbox`} → `stbox`
`overlaps_bbox`	`&&`	{`tgeometry`, `geometry`, `stbox`} × {`tgeometry`, `geometry`, `stbox`} → `stbox`
`same_bbox`	`~=`	{`tgeometry`, `geometry`, `stbox`} × {`tgeometry`, `geometry`, `stbox`} → `stbox`
`adjacent_bbox`	`-|-`	{`tgeometry`, `geometry`, `stbox`} × {`tgeometry`, `geometry`, `stbox`} → `stbox`

Table 4.13: Bounding box functions and operators

4.5 Interesting Internal Functions

MobilityDB is an extension to PostgreSQL and is thus exposed to the user using SQL types and functions, with a large portion of these functions being described in Section 4.4. Multiple internal functions were necessary to implement these SQL functions, and a few of them are worth mentioning here, due to their particularity and/or difficulty.

4.5.1 Rotating Bounding Boxes

One implementation issue that arises when handling temporal regions instead of points is the (pre-)computation of the bounding boxes of the different moving region objects. In MobilityDB, bounding boxes for temporal objects of instant set, sequence and sequence set duration are precomputed and stored together with the moving region.

When computing the bounding box of a moving region of instant set duration, the same method as for the other moving objects can be applied, i.e.: the bounding box of each instant is computed (for regions, this is done using a PostGIS function), and the resulting bounding box is simply the smallest box containing all the bounding boxes of the

instants. Since sequences of regions already have their bounding boxes precomputed, the same technique can be applied to moving regions of sequence set duration: the resulting bounding box is the smallest box containing all the bounding boxes of the sequences.

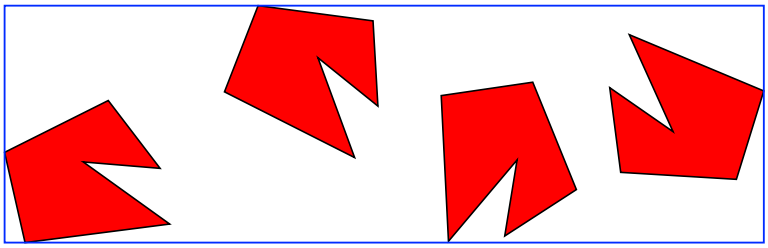


Figure 4.9: Bounding box of a tgeometryi

The last case that needs to be handled is the computation of the bounding box of a moving region of sequence duration. Applying the same method as for regions of instant set duration does not work anymore since the vertices of the regions do not move linearly when the region is rotating. This means that some vertices will go outside the smallest bounding box containing both the start and end regions. A trivial example is when a square rotates 90 degrees without translating. The bounding box containing its start and end vertices will just be the square itself, but the corners of the square exited this bounding box during the rotation.

A more advanced example can be seen in Figure 4.10, where the bounding box is shown in blue, and the path traversed by the moving region is shown in green. We can see that the moving region goes outside this naively computed bounding box.

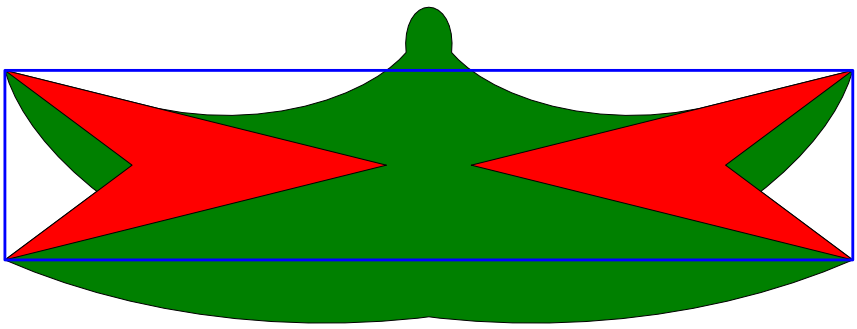


Figure 4.10: Error when computing the bounding box of a `tgeometryseq`

To solve this issue, we imagined two possible alternatives. First of all, if the traversed area (see Section 4.5.4) of the temporal region is known, we can just take the bounding box of this area, which will be indeed the minimal bounding box containing the moving region completely. However, since computing this traversed area is a computationally heavy operation, we propose another method that computed a bounding box more efficiently. This solution produces a correct but non-optimal bounding box. The bounding box is correct because the region will never exit this box during its movement, but non-optimal. Indeed, it will usually not be the smallest bounding box that can be found. These assumptions are still sufficient for the bounding box to be used for multiple purposes, so this solution has been implemented in MobilityDB.

To compute this bounding box for sequences, a similar procedure as for instance sets is used. The only difference is that when computing the bounding boxes of the instants, a *rotating bounding box* algorithm is used instead of the PostGIS bounding box functions. This rotating bounding box algorithm works as follows:

1. Find the vertex furthest from the rotation centre. In our case, the rotation centre is the centroid of the polygon representing the region. This step can be done with a linear search through the vertices.
2. Compute the distance d_{max} between the rotation centre and this vertex.
3. Create a square of side $2 * d_{max}$, centred around the rotation centre.
4. Return this square as the rotating bounding box.
5. (optional) Also return d_{max} to be used for the following instants, since this value does not change if the region and the rotation centre of the region stay the same. If this is done, we can skip step 1 and 2 when computing the next rotating bounding boxes.

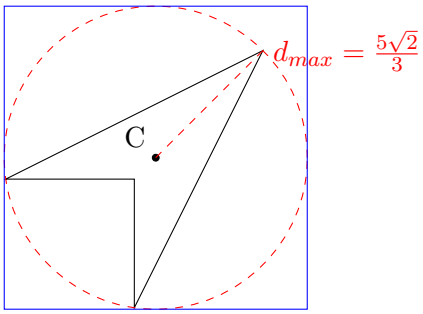


Figure 4.11: Rotating bounding box around a region

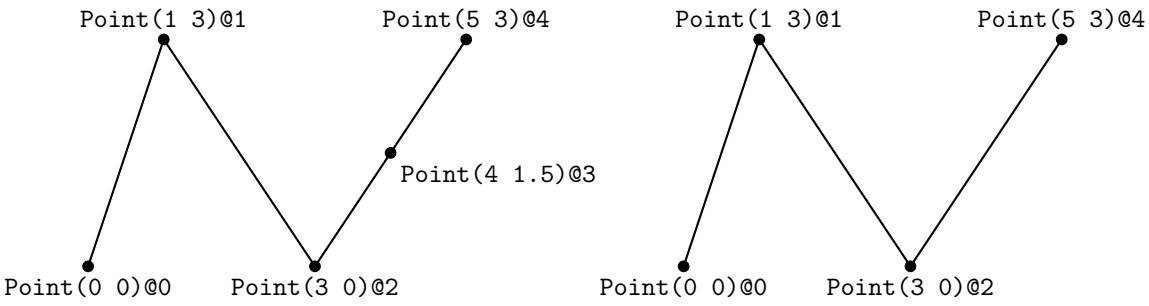
This algorithm will return a bounding box which completely contains the given region, independently of any rotation applied to it around its rotation centre. After computing the bounding boxes of all the instants, the resulting bounding box will simply be the smallest box containing all the bounding boxes of the instants. Depending on the temporal geometry, this bounding box will be close or far from the optimal bounding box. If the polygon representing the region is relatively 'round', and the region rotates quickly, this bounding box will be quite accurate. If, however, the region is long but thin and does not rotate a lot, the bounding box will be much bigger than the optimal bounding box.

For this reason, this solution for bounding boxes might not be usable for some functions requiring an optimal (or almost optimal) bounding box and has to be handled carefully. It does, however, serve its purpose when it is used to prune the input set of some functions. For example, when we want to compute all temporal regions which are closer than a certain distance d from each other at some instant during their movement, these sub-optimal bounding boxes can still be used but will simply prune a smaller set than if the boxes were optimal.

4.5.2 Normalization

When comparing moving objects, it is important to have identical objects being represented in the same way to allow for an efficient comparison process. This will be done by applying a normalization process on the given moving objects. Every sequence or sequence set that represents the same moving object will thus have the same internal representation after normalization. This normalization process is applied when creating a new moving object from an input or when merging multiple objects (for example, when combining two sequences of the same region). During normalization, redundant instants that could be recomputed by interpolating between two other instants are removed. For instants and instant sets, there is no normalization process, since we do not assume any interpolation

process between instants, and thus we cannot have redundant instants. A simple example of a moving point is shown below to visualize the concept of normalization.



(a) Before normalization (b) After normalization

Figure 4.12: Normalization of a temporal point

Normalizing the representation of moving objects is important in multiple aspects. First, removing redundant instants in sequences and sequence sets makes the stored objects smaller. Secondly, having a unique representation for every moving object also allows to compare them for equality much more easily than if multiple representations of the same moving object are allowed.

As said previously, normalizing happens only in sequences and sequence sets. Normalization in sequence set can be seen as joining two sequences when the last instant of the first sequence is identical to the first instant of the second one and then normalizing the resulting sequence. In the following discussion, we will thus only focus on normalizing a single sequence.

To explain the normalization process of a moving region of sequence duration, we will first describe the normalization process of a moving point of sequence duration, since both processes are analogous. The interpolation technique used for moving points is a simple linear interpolation. In a sequence of points defined at increasing time instants, when three subsequent points are collinear and the ratio of their distance in the time dimension is the same as the ratio of their distance in 2d (or 3d) space, then the middle point is considered redundant since it could easily be recomputed using interpolation. This situation can be seen in Figure 4.12.

Normalizing a moving region is done similarly. The main difference here is that we do not check for collinearity of points, but we check for collinearity of transformation vectors (`rtransform` values). Three transformation vectors are collinear if their translation vectors are collinear in 2d space, and their rotation angles are also collinear with the same ratio as the translation vectors. To check collinearity of the rotation angles we have to handle edge cases since the angles are always between $-\pi$ and π. An example of collinear regions can be seen in Figure 4.13.

When handling moving points, collinear points are considered redundant, since they can always be retrieved using interpolation. On the other hand, when handling moving regions, all collinear regions are not necessarily redundant. Indeed, since the interpolation method always interpolates two regions using the smallest angle between them, removing a collinear region could cause the interpolation to have a different direction of rotation than expected. This issue can be seen in Figure 4.14.

This issue only comes up when the sum of the angles between three subsequent transformation vectors is larger than π. If this sum of angles is smaller than π, the collinear region is redundant and can be recomputed using interpolation. In this case, the region can simply be removed. However, if this is not the case, then the collinear region is not

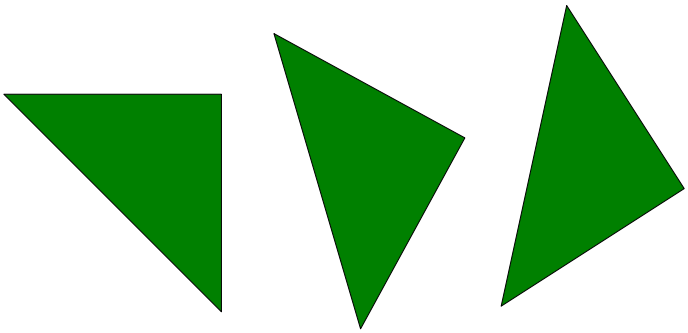


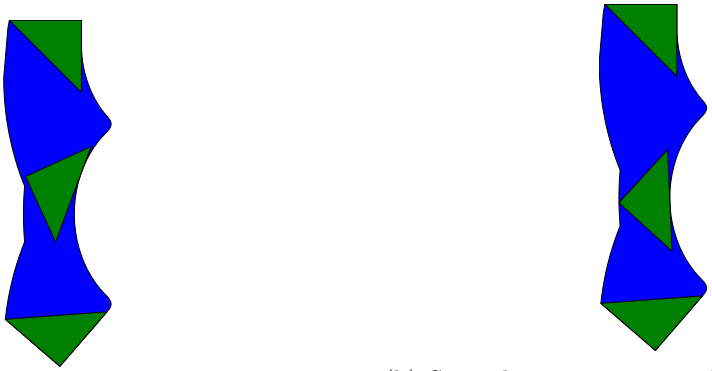
Figure 4.13: Example of collinear regions. The region is moving from left to right, and the corresponding rtransform values are: `Rtransform(0, 0, 0)`, `Rtransform(2, 0, -0.5)`, `Rtransform(4, 0, -1)`. If the time difference between two subsequent instants is constant, then the regions are indeed collinear.

(a) Path taken with the collinear region (counter clockwise rotation)

(b) Path taken without the collinear region (clockwise rotation)

Figure 4.14: Example of a collinear but not redundant instant in a moving region

completely redundant since it gives information about the direction of rotation that would not be present if the region was simply removed. Leaving the region present is also not a possibility since this could result in multiple moving regions being identical while having a different representation.



(a) First representation of the moving region

(b) Second representation of the same moving region

Figure 4.15: Two different representations of the same moving region. In the second representation, the second instant is defined slightly later than in the first representation.

To solve this issue multiple ideas are possible. A first idea is to add the direction of

rotation in the transformation vector. Since the interpolation of regions is done between two adjacent instants, this extra value would be defined with respect to the previous instant in the sequence. This contradicts with the other values in the stored transformation vector since previously the transformation vectors were defined with respect to the initial (first) region. This solution is thus feasible, but the values need to be handled with care.

Another solution is to add a *dummy* region that would be the same for all objects representing the same moving region. This dummy region would serve as a kind of guide for the interpolation to preserve the correct direction of rotation. Although this technique would not always reduce the storage space, it does take care of the unique representation of the moving regions, which is the main concern. It is, however, possible that in some cases a dummy region can take the place of two regions instead of just one, and thus still reduce slightly the storage space.

This dummy region has to be well-defined so that every possible input representing the same moving region could be transformed into a unique representation. The definition should also try to minimize the number of dummy regions needed in this unique representation. We chose to define the dummy region in the following way: When three collinear regions have a combined angle of larger than π, the middle region will be replaced by a region which has a rotation of π with respect to the first region if the rotation is counter-clockwise, or a rotation of $\pi - \epsilon$ with respect to the first region if the rotation is clockwise. The value of ϵ will be the same for all objects and has to be chosen so that the direction of rotation stay correct regardless of any rounding errors, while also being as small as possible. In practice, this value was chosen to be $\epsilon = 1e{-}5$.

This definition produces a minimal amount of dummy regions, while still being able to recreate the initial movement of the region correctly. This method also produces unique representations and is thus a good way to normalize moving regions. This is the solution that has been chosen to implement in MobilityDB.

Theoretically, since the transformation that is stored for this dummy region is partially redundant, we could replace it with a single value representing the direction of rotation, or the intermediate angle. This is in principle feasible to reduce storage space even more, but that would mean that we need to handle a third type of object, which is not the best idea when we want to implement clean code.

In practice, detecting if an instant is redundant or should be replaced by a dummy instant is done using the `interpolate` and `interpolate_long` functions described in Section 4.3. If the current instant can be computed with the `interpolate` function using the previous an next instants as input, then the instant is redundant and can be removed. However, if the instant can be computed with the `interpolate_long` function instead of `interpolate`, then it means that the instant has to be replaced by a dummy instant.

4.5.3 Standalone Instant

One particularity with temporal regions compared to all other types in MobilityDB is that some stored instants are not saved using the same schema depending on where they occur in the temporal value. For example, a `tgeometryinst` is stored as a pair of a polygon value and a timestamp. When an array of instants is given to the constructor `tgeometryi`, every instant except the first is transformed into an instant with an `rtransform` value. This means that to recompute one given instant, both this `rtransform` instant and the reference/initial instant (the first instant of the set) are needed, but the `rtransform` instant alone is not enough.

This is different from all other temporal types, where all the instants are stored as they are received and can be retrieved without any post-processing. In MobiliyDB, since the structure of all temporal types is identical, multiple functions can manipulate all temporal types without needing to know exactly what the base type of the object is. For example,

the `instantN` function described in Section 4.4.4 simply needs to retrieve the correct instant in the list, and return it. This is done using a function called `temporali_inst_n` for instance sets and `temporalseq_inst_n` for sequences. The returned instant can then be used as-is since it corresponds to a valid temporal instant.

When retrieving instants from a temporal region of instant set or sequence duration using the same functions, the returned instant cannot be used as is, since it does not represent a valid temporal region of instant duration (except if it is the first of the set). Instead, it needs to be combined with the region stored in the very first instant of the set. Sometimes retrieving a temporal instant with an `rtransform` value is what is needed, which is usually the case in region-specific functions, but in the most general functions (such as `instantN` used as an example before) this is not the case, and a *standalone* instant is required for the function to work.

To solve this issue, the new functions `temporali_standalone_inst_n` for instant sets and `temporalseq_standalone_inst_n` for sequences have been added, which returns a valid temporal instant. One thing to note is that the two initial functions only returned a pointer to the correct instant in the list since a copy is not necessarily needed. The new functions do the same, except when returning a `tgeometryinst` that was recomputed from an `rtransform` and a region instant. In this case, the returned instant is a newly created one, whose memory has to be correctly freed after it is used.

One might think that the same strategy has to be applied when retrieving a sequence from a temporal sequence set, but this is not the case. Indeed, as explained in Section 4.2.4, all sequences in the set are stored using the same schema, so as long as the first sequence can be handled correctly, which is the case with the `temporalseq_standalone_inst_n`, all subsequent sequences can be manipulated similarly.

However, if the solution chosen in Section 4.2.4 was the second one, where subsequent sequences are also preprocessed to only contain `rtransform` values and take up less storage space, then this would not be the case anymore, and a new function that can return a standalone sequence would also be needed.

4.5.4 Traversed Area

One function which is not yet explained but which is relatively important when handling moving regions is the computation of the *traversed area*. By traversed area, we mean the set of all points that were touched by or contained in the region at some point during its movement. This is analogous in its uses to computing the trajectory of moving points but is quite a bit harder to compute in practice since the edges of a traversed area are in general not straight edges because the region can rotate. Examples of the traversed area of multiple moving regions can be seen in Figure 4.16.

Having this traversed area precomputed for temporal regions simplifies the implementation of multiple functions. For example, computing the bounding box of a temporal sequence can be done by simply computing the bounding box of the traversed area of this sequence. Knowing if the temporal region will collide with a static point or region at some moment during its movement can also be done by simply checking if the traversed area intersects with the given point or region.

(Pre-)computing the traversed area for temporal regions is thus interesting, but how can it be done? The traversed area of a `tgeometryinst` is simply the region itself and is thus already stored as a polygon value in the instant. For temporal instant sets, since the region is only defined at specific moments, the traversed area will be the union of the regions of all instants, which can be stored as a PostGIS multipolygon. The same is true for temporal sequences with stepwise interpolation, and for temporal sequence set, the traversed area is the union of the traversed areas of all of its sequences. This leaves us with computing the traversed area of a `tgeometryseq`.

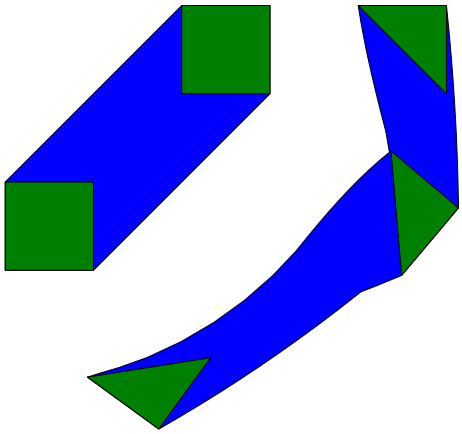


Figure 4.16: Traversed area (in blue) of two moving regions

An algorithm that can be used to compute the traversed area of a moving region is described in [5], 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 [5], 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.

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 [5]. 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 [5] 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.

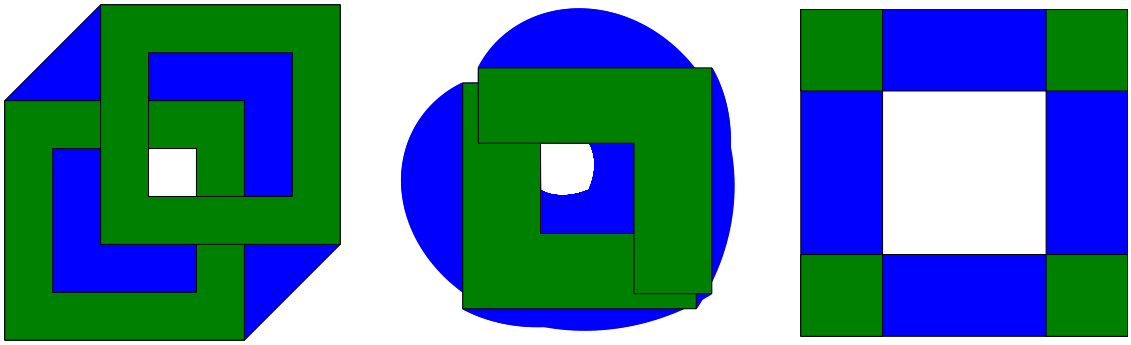
Algorithm

Given a `tgeometryseq` represented as $'[\mathcal{R}_0@t_0,\ \mathcal{T}_1@t_1,\ \mathcal{T}_2@t_2,\ ...,\ \mathcal{T}_{n-1}@t_{n-1}]'$, and assuming linear interpolation between the instants, we want to compute the area traversed by the region during its movement. The resulting area will be stored in a polygon object,

which can in all generality contain holes, even if the base region does not contain any holes. Three different types of holes can be present in a traversed area, and these different types are shown in Figure 4.17.

The first type is created from holes already present in the moving region (Figure 4.17a). 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 (Figure 4.17b). The last one can be created by any type of polygon and occurs when the moving region comes back at a previous position after multiple transformations (Figure 4.17c).

The described algorithm computes a polygon without taking into account any holes in the traversed area. Using a slight modification of the algorithm, the third type of hole can be taken into account, but the two remaining types are not yet handled.



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

Figure 4.17: Different types of holes created during the computation of the traversed area

The algorithm works in three steps.

As a first step, we create a list of segments that will be used in the second step. This list of segments consists of both the edges of the polygons at different instants and segments created by the movement of the vertices of the polygons between these instants. We previously explained that we cannot use linear interpolation between the vertices to compute an interpolation between two instants since the movement of the vertices is not linear. This is only true for regions that rotate, so if the region is only translating, linear interpolation can be used as well. Since we want to represent the movement of the vertices of any moving region using straight segments, we will have to approximate this curve by a set of straight segments.

To do this, we will assume that we can use linear interpolation between the vertices, and thus create a straight segment between these vertices to add to our list, for transformations which have a rotation which is smaller than a certain threshold (θ_{max}) in absolute value. When the rotation θ is larger than this θ_{max}, we will split the transformation into a set of n transformations, with each transformation having the same rotation angle θ^*. n is chosen as the smallest integer value such that $\theta^* < \theta_{max}$.

$$n = \left\lceil \frac{\theta}{\theta_{max}} \right\rceil$$

The precision or correctness of the resulting traversed area depends entirely on the angle θ_{max}, which has to be chosen small enough to keep the actual result sufficiently close from the desired result, but sufficiently large at the same time to not increase the computation time too much.

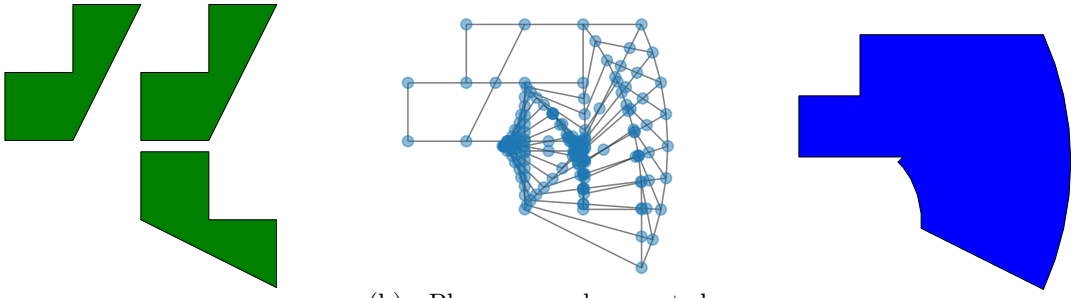
When receiving the temporal sequence as $'[\mathcal{R}_0@t_0,\ \mathcal{T}_1@t_1,\ \mathcal{T}_2@t_2,\ ...,\ \mathcal{T}_{n-1}@t_{n-1}]'$, we first add the needed amount of intermediate transformations to have the rotation between each subsequent transformation smaller than θ_{max}. When this is done, the edges of the

polygon defined by the first and last instant are put into a list. Then the segments created by the movement of the vertices during every transformation, as well as the segments corresponding to the edges of the polygon after each transformation, are also added to the list.

The second step 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 is internally stored as a mapping of vertices to a list of neighbour vertices. Doing this essentially consists of computing all the intersections in the received list of segments, and then computing for all vertices and intersections the list of neighbours. Two points are neighbours 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 Figure 4.18b.

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* [16], 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 neighbour of the current vertex when listing them in counterclockwise order starting from the previous vertex (which is one of the neighbours 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.



(a) Instants defining the moving regions

(b) Planar graph created from the computed segments

(c) Resulting traversed area

Figure 4.18: Process of computing the traversed area of a moving region starting from a sequence of instants.

As said previously, the polygon representing a traversed area can, in all generality, have an arbitrary amount 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 4.17 shows three kinds of holes, that are created in three different ways. One of them is created when the moving region comes back to a previous position after a certain amount of subsequent transformations. 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.

Computing a traversed area containing this third kind of hole is possible by adapting the previous algorithm slightly. Instead of computing the traversed area of the complete sequence in one go, we compute the traversed area between each pair of subsequent instants and store each of them in a list. The complete traversed area will then simply be the region created by the union of all polygons in this list. Computing this can be done using the PostGIS `ST_Union` function.

Chapter 5

Future Work

5.1 Temporal Geography Type

Until currently, every region was described using the PostGIS `geometry(Polygon)` type. However, PostGIS also exposes a `geography(Polygon)` type, which can be used to represent a polygon using *geographic coordinates*. These coordinates are spherical coordinates expressed in angular units (degrees), and these can be used to more accurately represent a position on a sphere (such as the earth), without having to project them on a sphere. Calculations done using these coordinates are more complicated since they must be calculated on a sphere. Implementing a temporal geography type to handle moving regions using geographic coordinates would thus require other functions than the ones used by the `tgeometry` type.

Moving points have been implemented in MobilityDB for both the `geometry(Point)` and `geography(Point)` types, so doing the same for moving regions would be consistent with the current implementation, but this is left as future work.

5.2 Unify `tgeompoint` and `tgeometry` Types

PostGIS uses a unified `geometry` type for all types of geometries: `Point`, `Polygon`, `LineString`, etc. Currently, the MobilityDB implementation uses different types to represent moving points (`tgeompoint`) and moving regions (`tgeometry`), which are not related in any way. To be as consistent with the PostGIS implementation as possible, it might be interesting to create a unified SQL type `tgeometry`, which can represent either a moving region (`tgeometry(Polygon)`) or a moving point (`tgeometry(Point)`, and would be handled differently by the SQL functions.

This is completely doable but requires a lot of code refactoring and modifications. Since moving points are the predominant us case, this unification should also not slow down any functions handling moving points. Currently, since the only two types that could be unified are `tgeompoint` and `tgeometry`, this does not seem necessary yet, but if new geometry types are added and handled in the future, it might become more interesting.

5.3 3D Regions

When we consider moving regions of fixed shape in three dimensions, some of the ideas and concepts used for 2D regions can be reused. For example, when we want to represent a moving region compactly, the technique of using the transformation between the instants can be reused. A 3D transformation is, just as in 2D, the combination of a translation and a rotation, but the translation now has three degrees of freedom instead of two, and the rotation also has three degrees of freedom instead of one.

The most important operation when implementing moving regions is the interpolation since this allows us to retrieve the value of the region at any given instant. Interpolation in 3D requires us to interpolate the translation vector linearly in 3D, which is relatively easy, and it requires us to interpolate the rotation vector 'linearly'. This interpolation of 3D rotations is quite a bit more complicated than in 2D and is done using *quaternions*. Quaternions and their use in computing the interpolation of 3D rotations are described in Section 5.3.1.

Representation and interpolation are the only aspects of 3D moving regions that have been researched for this master thesis, and the implementation of other functions is left as future work.

5.3.1 Quaternions for the Interpolation of Rotations

Section 4.3.3 describes a basic way to compute the interpolation of a rotation in 2D defined by a single angle. In 3D, however, the angles needed to describe a rotation are the Euler angles, and this method is not well-suited for these angles. Another method for describing rotations in 2D and 3D are rotation matrices. Lastly, rotations in 3D (and 2D) can also be described using quaternions, and this technique allows for an easy interpolation method, which will be described later. A small section is added in the end showing how complex numbers can be used in a similar way to interpolate 2D rotations by using less parameters than quaternions.

The equations in the following section are presented here without proof. For a more formal definition of these concepts, refer to [17].

Representing 3D Rotations Using Quaternions

Quaternions were first described by William Rowan Hamilton in 1843 and are an extension to complex numbers. Quaternions are represented in the following way.

$$q = a + bi + cj + dk, \text{ with } a, b, c, d \in \mathbb{R} \tag{5.1}$$

The symbols i, j and k can be interpreted as unit vectors in a three-dimensional space, and are analogous to the symbol i in complex numbers. Multiplication of complex numbers uses the fact that $i^2 = -1$. Similarly, quaternion multiplication is defined using the multiplication rules of i, j and k with each other.

$$i^2 = j^2 = k^2 = ijk = -1 \tag{5.2}$$

The equations above define the complete behaviour of the three symbols, and the equations below can be found using them.

$$\begin{aligned} ij &= k & ji &= -k \\ jk &= i & kj &= -i \\ ki &= j & ik &= -j \end{aligned}$$

Quaternion multiplication can then be done similarly to the multiplication of complex numbers by taking into account the multiplication rules for these three symbols. An important thing to note is that quaternion multiplication is not commutative, as can be seen when multiplying quaternions $q_1 = i$ and $q_2 = j$: $q_1 * q_2 = ij = k$ and $q_2 * q_1 = ji = -k = -q_1 * q_2$.

A unit quaternion is a quaternion of unit length, meaning its norm is equal to 1.

$$\|q\| = \sqrt{a^2 + b^2 + c^2 + d^2} \tag{5.3}$$

Unit quaternions can be used to efficiently represent orientations or rotations in three dimensions. When a unit quaternion is used to represent a rotation, it is called a rotation quaternion. For example, a rotation of an angle θ around the axis $\vec{v} = (x, y, z)$ can be represented by the rotation quaternion

$$q = \cos(\frac{\theta}{2}) + \frac{(xi + yj + zk)}{\sqrt{x^2 + y^2 + z^2}} * \sin(\frac{\theta}{2}) \tag{5.4}$$

Rotating a point/vector $p = (p_x, p_y, pz_) = p_x i + p_y j + p_z k$ by θ around $\vec{v} = (x, y, z)$ involves evaluating the conjugation of p by q

$$p' = q * p * q^{-1}, \tag{5.5}$$

where q^{-1} is the reciprocal of q, which for unit quaternions is the same as the conjugate of q since q is already normalized.

$$\begin{aligned} q^{-1} &= \frac{q^*}{\|q\|^2} \\ &= \frac{\cos(\frac{\theta}{2}) - \frac{(xi+yj+zk)}{\sqrt{x^2+y^2+z^2}} * \sin(\frac{\theta}{2})}{\|q\|^2} \\ &= \cos(\frac{\theta}{2}) - \frac{(xi + yj + zk)}{\sqrt{x^2 + y^2 + z^2}} * \sin(\frac{\theta}{2}) \end{aligned} \tag{5.6}$$

When rotating a point or polygon around an axis that does not go through the origin, we first have to translate the point/polygon to make the rotation axis go through the origin. For example, if the axis goes through the centroid of a polygon, we first have to translate this polygon so that its centroid and the origin are coincident. Then, we apply the rotation to each vertex of the polygon. Finally, the polygon is translated back to return its centroid at the initial position.

Multiple rotations can be combined into a single one by combining the corresponding quaternions. For example, the combined rotation of first applying q_1 and then q_2 can be computed as

$$\begin{aligned} q &= q_2 * q_1 \\ p' &= q * p * q^{-1} \\ &= (q_2 * q_1) * p * (q_2 * q_1)^{-1}, \\ &= (q_2 * q_1) * p * (q_1^{-1} * q_2^{-1}) \\ &= q_2 * (q_1 * p * q_1^{-1}) * q_2^{-1} \end{aligned} \tag{5.7}$$

which indeed correspond to first applying q_1 to p and then applying q_2 to the previous result.

The efficiency of using quaternions instead of rotation matrices or axis-angle representations can be debated, but the main advantage of using these rotation quaternions is that we can easily apply a spherical linear interpolation (`slerp`) to them. This is necessary to correctly interpolate between two 3D rotations, which is needed for the 3D version of the `interpolate` function described in Section 4.3.3. To use the resulting quaternion to

effectively rotate a polygon can be done by either first transforming all polygon vertices to a quaternion representation and then using quaternion multiplication, or by representing the resulting rotation quaternion in another way, for example using a rotation matrix, and then applying this rotation matrix to the polygon.

Interpolating 2D and 3D Rotations Using Quaternions

Now that rotation quaternions have been presented as a method used to represent 3D rotations, let's look at how we can compute an interpolation between two rotations. An example of the interpolation of two 3D rotations can be seen in Figure 5.1. The computation can be done relatively easily using quaternions and corresponds to a spherical linear interpolation (`slerp`) in 3D.

(a) Start and end rotation

(b) Interpolated rotation

Figure 5.1: Interpolation of rotations in 3D [18]

The interpolation process can be represented mathematically as follows. To interpolate between two rotations represented by quaternions q_0 and q_1, we first compute an angle θ between them and then apply the `slerp` process to find the interpolated rotation. r is a ratio value which depends on the instant of the resulting rotation, just as in Section 4.3.3.

$$\begin{aligned} \cos(\theta) &= q_0 \bullet q_1 \\ q(r) &= Slerp(q_0, q_1, r) \\ &= \frac{q_0 * \sin((1-r)*\theta) + q_1 * \sin(r*\theta)}{\sin(\theta)} \end{aligned} \tag{5.8}$$

Interpolating 2D Rotations Using Complex Numbers

We have shown how quaternions can be used to represent 3D rotations and to compute interpolation between 3D rotations. These techniques can thus also be used for 2D rotations since they are a special type of 3D rotations and the interpolation between two 2D rotations also results in a 2D rotation. Nevertheless, using a quaternion rotation for 2D rotations might be overkill if 3D rotations do not need to be handled. A method has previously been described that handles only 2D rotations but as said previously, it has to handle edge cases because rotation angles have to be between $-\pi$ and π. Another method uses complex numbers to, similarly to the previously mentioned quaternions, efficiently compute interpolation of 2D rotations. This is done using the same technique as with quaternions, but using fewer parameters. The equations are shown below, and the similarity with the previously defined equations for quaternions can be seen clearly.

A rotation of θ around the origin can be represented using the complex number $q = \cos(\frac{\theta}{2}) + i * \sin(\frac{\theta}{2})$ and a 2D point $p = (x, y)$ can be represented using the complex number

$p = x + i * y$. Computing the interpolation between two rotations q_0 and q_1 can be done similarly as quaternion interpolation

$$\begin{aligned}
\cos(\frac{\theta}{2}) &= q_0 \bullet q_1 \\
&= \cos(\frac{\theta_0}{2}) * \cos(\frac{\theta_1}{2}) + \sin(\frac{\theta_0}{2}) * \sin(\frac{\theta_1}{2}) \\
&= \cos(\frac{\theta_1 - \theta_0}{2}) \\
& \qquad , \\
q(r) &= Slerp(q_0, q_1, r) \\
&= \frac{q_0 * \sin((1-r) * (\frac{\theta}{2})) + q_1 * \sin(r * (\frac{\theta}{2}))}{\sin((\frac{\theta}{2}))} \\
&= \cos((1-r) * \frac{\theta_0}{2} + r * \frac{\theta_1}{2}) + i * \sin((1-r) * \frac{\theta_0}{2} + r * \frac{\theta_1}{2})
\end{aligned} \tag{5.9}$$

and the same can be said for applying the rotation to an arbitrary point.

$$\begin{aligned}
p' &= q * p * q \\
&= q^2 * p \\
&= (\cos(\frac{\theta}{2})^2 + \sin(\frac{\theta}{2})^2 + i * 2\cos(\frac{\theta}{2})\sin(\frac{\theta}{2})) * p \\
&= (\cos(\theta) + i * \sin(\theta)) * (x + i * y) \\
&= (x * \cos(\theta) - y * \sin(\theta)) + i * (x * \sin(\theta) + y * \cos(\theta))
\end{aligned} \tag{5.10}$$

5.4 Deforming Regions

The last aspect of moving regions not handled in this thesis deformation. Section 3.1 describes some of the research made on deforming moving regions, but as explained in that section, the most common representation of this type of region is not suitable for MobilityDB and has to be adapted quite a bit. This special sliced representation is used to allow for an efficient interpolation. Of course, if we are talking about regions moving in discrete time steps, this interpolation is not necessary, and the sliced representation does not offer any advantage over the MobilityDB representation. We can thus split the set of deforming moving regions in two.

A deforming region subject to continuous transformations is the most difficult type to handle but is also the type most present in real-life: boundaries of forests, ponds, oil leaks in the ocean, clouds, etc. However, as said previously, the current solutions for this type of region is quite far from the MobilityDB implementation, so it will not be discussed further. Regions changing in discrete time steps, however, can be implemented without too much trouble in MobilityDB, and this is discussed below.

Note that, contrary to all previous section, these ideas have not been tested in practice, so this is only a discussion and not a presentation of results.

Discrete movement

The main issue when handling deforming regions is that the value of the region between two defined instants is hard to interpolate. For regions that move in discrete time steps, this is not the case anymore, since this value is equal to the one stored in the last instant. Since interpolation is not an issue anymore, we do not have to adapt the representation and

this type of moving regions can thus be represented easily using the internal MobilityDB representation

$$'\{[\mathcal{R}_0^0@t_0^0,\ \mathcal{R}_1^0@t_1^0,\ ...],\ [\mathcal{R}_0^1@t_0^1,\ \mathcal{R}_1^1@t_1^1,\ ...],\ ...\}',$$

where all region values $\mathcal{R}_i^i$ are different from each other because we handle deforming regions.

This representation would be acceptable and all the needed functions could be implemented to handle this new type. The only true issue that I can see with this representation is that the memory size might become large when handling large sequences/set. One possible solution would be to store a delta/difference between the current region and the previous one if this is smaller than the current region, but depending on the changes that the region undergoes, this might not be beneficial.

One good example of a use case for this type of regions is a *cadastre*, which records the ownership of land parcels in countries. Cadastral updates are by nature discrete, and the different parcels in a cadastre could thus be represented like shown above.

Implementing this new region type, and testing if and how it could be used for real-world applications such as cadastres is left as future work.

Chapter 6

Summary

This master thesis discusses the concepts of moving regions in moving object databases and focuses on the implementation of fixed-shape moving regions in MobilityDB, an open-source moving object database built on top of PostgreSQL and PostGIS.

In Chapters 2 and 3, the systems used and the current state of the research on moving regions are both described to give enough context for a clear understanding of the following sections. We here make a distinction between deformable moving regions, fixed-shape moving regions, and between 2D and 3D moving regions. We then decide to focus on the practical implementation of fixed-shape moving regions in 2D.

In the next part (Chapter 4), we dive into the main contribution of this master thesis, which is developing a model for fixed-shape moving regions in MOD's, and implementing this model in practice in MobilityDB. For this, multiple types are added into MobilityDB: the `tgeometry` type, which has four variants (`tgeometryinst, tgeometryi, tgeometryseq` and `tgeometrys`) depending on its duration, and the `rtransform` type, which is used to represent an instant of a region more concisely than by using a polygon.

In Section 4.4, a large set of functions that can be applied to the new `tgeometry` type is listed and described. Most of these functions are similar to existing MobilityDB functions but are simply extended to allow arguments with the `tgeometry` type.

During the implementation of these functions, different algorithms have been developed, and the most interesting ones are described in the last section of Chapter 4, Section 4.5. The first algorithm is used to compute the bounding box of a moving region efficiently, although not optimally. The second one describes the normalization process that is applied to all MobilityDB sequences when constructed, and the algorithm to normalize sequences of regions is also explained in detail. Section 4.5.3 talks about a particularity of temporal geometries compared to all other temporal types of MobilityDB and talks about the addition of a small utility function that is needed to emulate the usual behaviour of temporal types when working with temporal geometries. Lastly, the algorithm used to compute the traversed area of a moving region, as well as possible improvements, is described thoroughly, since this the result of this algorithm is used in various functions.

Due to their complexity, multiple aspects of this project have been researched without being effectively implemented into MobilityDB yet. One particular algorithm for the interpolation of rotations in 3D is already developed and tested outside MobilityDB and is thus also thoroughly described in Section 5.3.1. Other aspects and extensions of this project are also discussed in Chapter 5, and their implementation is left as future work.

Bibliography

[1] O. Wolfson, B. Xu, S. Chamberlain, and L. Jiang, "Moving objects databases: Issues and solutions," in *Proceedings on the 10th International Conference on Scientific and Statistical Database Management*, pp. 111–122, 1998.

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

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

[4] F. Heinz and R. H. Güting, "A polyhedra-based model for moving regions in databases," *International Journal of Geographical Information Science*, pp. 41–73, 2020.

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

[6] E. Zimányi, M. Sakr, A. Lesuisse, and M. Bakli, "Mobilitydb: A mainstream moving object database system," in *Proceedings of the 16th International Symposium on Spatial and Temporal Databases*, p. 206–209, 2019.

[7] "Postgresql: The world's most advanced open source relational database," 1996-2020.

[8] "Postgis – spatial and geographic objects for postgresql," 2001-2020.

[9] M. José, D. José, and D. Paulo, "Modeling and representing real-world spatio-temporal data in databases," in *Proceedings of the 14th International Conference on Spatial Information Theory*, pp. 6:1–6:14, 2019.

[10] L. Forlizzi, R. H. Güting, E. Nardelli, and M. Schneider, "A data model and data structures for moving objects databases," *SIGMOD Rec.*, p. 319–330, 2000.

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

[12] E. Zimányi, *MobilityDB Manual*, 2020.

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

[14] 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*, pp. 1754–1759, 1997.

[15] S. Zlatanova, "3d geometries in spatial dbms," *Innovations in 3D Geo Information Systems*, pp. 1–14, 2006.

[16] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars, *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 2008.

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

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

Appendix A

Complete List of Transformation Functions

Function	Signature
duration	tgeometry → text
interpolation	tgeometry → text
memSize	tgeometry → integer
getValue	tgeometry(Instant) → geometry(Polygon)
startValue	tgeometry → geometry(Polygon)
endValue	tgeometry → geometry(Polygon)
getTimestamp	tgeometry(Instant) → timestamptz
getTime	tgeometry → periodset
timespan	tgeometry → interval
shift	tgeometry → tgeometry
numInstants	tgeometry → integer
startInstant	tgeometry → tgeometry(Instant)
endInstant	tgeometry → tgeometry(Instant)
instantN	tgeometry × integer → tgeometry(Instant)
instants	tgeometry → array tgeometry(Instant)
numTimestamps	tgeometry → integer
startTimestamp	tgeometry → timestamptz
endTimestamp	tgeometry → timestamptz
timestampN	tgeometry × integer → timestamptz
timestamps	tgeometry → array timestamptz

Table A.1: Transformation functions (complete)

Function	**Signature**
numSequences	tgeometry → integer
startSequence	tgeometry → tgeometry(Sequence)
endSequence	tgeometry → tgeometry(Sequence)
SequenceN	geometry × integer → tgeometry(Sequence)
Sequences	tgeometry → array tgeometry(Sequence)

Table A.1: Transformation functions (complete), continued