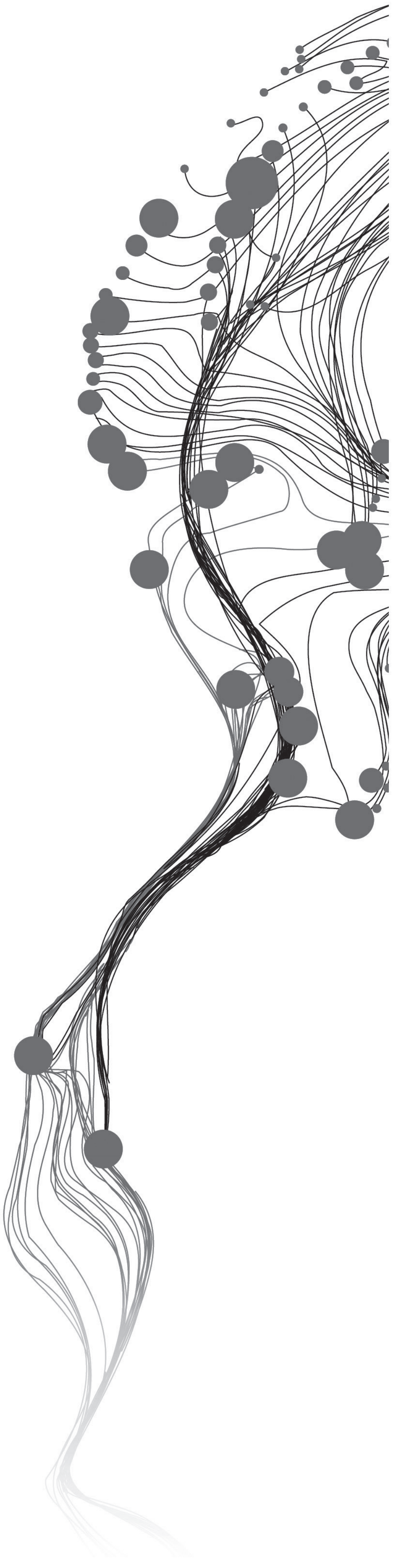
DEVELOPMENT OF A MOVING OBJECT DATA TYPE IN A DBMS

MULUGETA TADESE GUDECHA
February, 2014

SUPERVISORS:

Dr. Ir. R.A. de By
C. Piccinini MSc



DEVELOPMENT OF A MOVING OBJECT DATA TYPE IN A DBMS

MULUGETA TADESE GUDECHA
Enschede, The Netherlands, February, 2014

Thesis submitted to the Faculty of Geo-information Science and Earth Observation of the University of Twente in partial fulfilment of the requirements for the degree of Master of Science in Geo-information Science and Earth Observation.
Specialization: GFM

ABSTRACT

A spatiotemporal database allows its users to register space-time phenomena. The spatiotemporal database research area has been open and active for a long time. However, its theory has grown more rapidly than its implementation. Modeling of spatiotemporal objects, that change continuously with time requires abstractions in different categories, i.e. moving points, lines, regions, and etc. This research paper presents abstract and discrete designs used to represent moving points, basic abstraction if only the time-dependent change of position in the space of an object is important. In this study, the data models resulted from the discrete and abstract designs are mapped into a data structure that is implementable in a DBMS environment. This research succeeded to provide an implemented moving point data type and operations that provide support for spatiotemporal objects in PostgreSQL database management system. Not only moving point data type, the paper succeeded to provide design and implementation of other four auxiliary data types. On top of the spatiotemporal DBMS extension, this research offers alternative interpolation techniques for moving points and also includes innovative algorithms and implementations for aggregation functions that operates on sets of moving points.

Keywords

Spatiotemporal, DBMS, Data types, PostgreSQL, PostGIS, Moving Objects, Interpolation, Aggregation, BNF, MPOINT, MREAL, MPERIOD, INTIME_MPOINT, INTIME_MREAL

ACKNOWLEDGEMENTS

First and foremost, I wish to thank my parents. Your prayer for me was what sustained me to get this far. Words cannot express how grateful I am for all the sacrifices you have made on my behalf.

I would like to gratefully and sincerely thank my supervisor Dr. Ir. R.A. de By for his guidance, support, understanding, immense knowledge and especially for his confidence in me. You have been a tremendous mentor for me. I would like to thank you for sharing your vast knowledge of the field and well-rounded experience. I would also like to thank my second supervisor, C. Piccinini M.Sc. for his brilliant comments and suggestions throughout the project. I would especially like to thank Bas Retsios, who has always there to help me solve all my coding problems.

I would also like to thank my three younger sisters. They were always supporting me and encouraging me with their wholehearted best wishes.

Finally, and most importantly, I would like to thank Tigist Abraham for her support to help me achieve my goal and encouragement in my moments of crisis. Above all, I would like to thank God for his unlimited blessing and everything he has done for me.

LIST OF FIGURES

LIST OF TABLES

List of Algorithms

Listings

List of Acronyms

BNF	Backus Normal Form or BackusNaur Form
DBMS	Database Management Systems
GIS	Geographic Information Systems
LWGEOM	Light Weight Geometry)
GNU	GNU's Not Unix
LWPOINT	Light Weight Point
MON-tree	Moving Objects in Networks Tree
OGC	Open Geospatial Consortium
STDBMS	Spatiotemporal Database Management Systems
WKB	Well-Known Binary

TABLE OF CONTENTS

Chapter 1
Introduction

Spatiotemporal database management systems (STDBMSs) are becoming backbones of broad important applications such as Geographic Information Systems (GIS), multimedia, and environmental information systems.A spatiotemporal database allows its users to register space-time phenomena. Now and for the coming years, there is a need to store and manipulate changes of real world objects. In recent years, theoretical work has taken place on geometries changing with time to represent real world objects in a database. Different types of a spatiotemporal data model and database emerged. New concepts and research areas have been reviewed by Pelekis et al. [2004] . However, separate support for both spatial and temporal data in a database has remained one of the problems to analyze changes in spatial objects.

To analyze change/movement of spatial objects, current database and GIS technology does not suffice. Due to the complexity of spatial information caused by rapid growth of communication devices facilitating sharing of such information among users makes it difficult to use current database and GIS technology [Schneider, 2009]. In parallel, with the maturity of GPS and wireless technologies, large amount of data from various moving objects such as vehicles, mobile devices, and animals can be collected. Computations and analysis on such data has broad applications in movement pattern recognition, vehicle control, mobile communication control, and animal control. Therefore, there must be a mechanism to manage and manipulate, i.e., model, store, and query, continuous and discrete positions or location changes of spatial objects over time. This requires careful analysis due to the complexity introduced when the time dimension is added to the data structure.

Modeling of spatiotemporal objects that change continuously with time requires abstraction in different categories. This can be achieved by defining abstract data types for moving objects. In general, there exist abstractions for moving points, lines, regions, multipoints, multilines, multipolygons, and gometrycollections. However, abstractions describing objects with location change only (moving points), facilities with location and extent change for connection in space (moving lines), and objects with both location and size change (moving regions) are basics for describing the rest of moving objects [Güting et al., 2000]. Thus, spatiotemporal data models and query languages capable to include those abstract data types need to be implemented.

Finally, DBMS data models and query languages can be extended to manage and manipulate spatiotemporal objects (moving objects). Modeling and querying of spatiotemporal objects can be achieved by embedding spatiotemporal data types in DBMS. Particularly, the development of a function library for moving points requires incorporation of operations and procedures on the data types. These libraries can be used to register space-time phenomena in a database to understand their behavior with time and analyze their relations with other phenomena.

1.1 RESEARCH IDENTIFICATION

The aim of the research project is to develop a complete library package that extends a DBMS with support for spatiotemporal data type. To achieve this, the research questions and research objectives mentioned below are required to be addressed.

1.1.1 Research objectives

1. To improve moving point data type and operations implemented by Bezaye [2013], based on the mathematical model developed by Güting et al. [2000].
2. To implement operations on moving points omitted by Bezaye [2013], based on the mathematical model developed by Güting et al. [2000] and algorithms developed by Lema et al. [2003].
3. To extend moving point data type to support for alternative interpolation function other than linear interpolation proposed by Güting et al. [2000].
4. To include aggregate functions on the Güting et al. [2000] operations, that allow to perform computations on sets (cliques) of moving points.

1.1.2 Research questions

1. What are the weaknesses of the moving point data type and operations implementations by Bezaye [2013]? How can these be addressed?
2. How to implement operations on moving points omitted by Bezaye [2013]?
3. Which alternative interpolation functions can be used to support for moving point data type? And in which application contexts?
4. How to implement selected interpolation functions, based on the corresponding application context?
5. How can aggregate functions be designed to perform computations over sets of moving points?
6. How to implement the designed aggregate functions on moving points?

1.1.3 Innovation aimed at

The innovation of this research project is an implemented moving object data type for a major DBMS. This includes implementation of alternative interpolation and aggregate functions as well as improved and extended version of work done by Bezaye [2013]. The final implemented library package for moving object data types contains fully tested new operations in addition to those modeled by Güting et al. [2000].

1.1.4 Literature review

The spatiotemporal database research area has been open and active for a long time. It has grown theory more rapidly than implemented systems. The suitability of methods and resources for experimenting research in databases grant more advantages for publishing papers as mentioned by Güting et al. [2000]. Throughout this period of time, different theoretical and mathematical models were proposed to address spatiotemporal objects and databases.

Conceptual development and drawbacks on spatiotemporal databases and data models to represent objects that change discretely were reviewed by Pelekis et al. [2004]. Data models to represent spatiotemporal objects that change continuously in a database was first proposed by Erwig et al. [1999]. The proposed model considers moving objects as 2D+T (time as the third dimension) entities, while their behavior and structure is modeled through abstract data types. Later, Güting et al. [2000]introduced such abstract data types for moving points and moving regions together with operations on them. Again later, a discrete model for moving object data type was developed for the corresponding abstract data models by Forlizzi et al. [2000] and algorithms for operations on a moving object data type were developed by Lema et al. [2003]. All the authors from Güting et al. [2000] and Erwig et al. [1999] reflected that this approach to represent both changes and movements on spatiotemporal objects suits for most spatiotemporal queries. However, as mentioned in Güting et al. [2000], most of the models are not implemented, but rather were left in paper only.

1.1.5 Related work

A research network named CHOROCHRONOS, mainly putting its objectives on the design, implementation, and application of spatiotemporal DBMSs, was introduced in 1996 [Sellis, 1999]. CHOROCHRONOS allowed and supported researchers to work on spatiotemporal databases. Designs and partial implementations of STDBMSs were results from this project. In 1995, the University of Hagen started developing a prototype DBMS named SECONDO [Güting et al., 2010]. Their main design goal was to obtain a clean extension and support for spatial and spatiotemporal applications. Afterwards, SECONDO became an alternative for publishing research implementations on moving objects, even though PostGIS and OracleSpatial also remained interesting candidates. However, SECONDO was developed for high-level GIS and computer specialists to experiment with implementations of spatiotemporal research. As a consequence, it is not widely used by non-professional GIS users.

In recent years, trials were conducted to develop a function library package that can be embedded in PostgreSQL, to provide support for spatiotemporal moving data types. The first attempt was made by Eftekhar [2012] in designing and implementing a moving point data type in PostgreSQL. The research project by Eftekhar [2012], first reviewed theoretical foundations presented in [Güting et al., 2000] as a basis for further designing the models as an abstract data types and their follow up implementations. As a final result, the project succeeded to design and implement the data types for moving points. However, many operations mentioned in [Güting et al., 2000], were not fully implemented and refinement at design level also was one of the research project omissions. Later in 2012, attempts were made to refine abstract and discrete representation of moving points in PostgreSQL by [Bezaye, 2013] in her research project. And at the end, an implementation of a moving point data type was developed as an extension for PostgreSQL. This research project, however, did not provide a full implementation of operations mentioned in [Güting et al., 2000] for moving objects.

Currently, work on spatiotemporal databases can be extended in many directions. Let alone lack of completeness on moving point data types, extension of data types and operations on them to 3D and 4D spaces, boosting moving point data type for higher dimensional features like moving lines and regions, and incorporating new operations on moving features are possible ways of extensions.

1.1.6 Methodology

In general, the research project starts by reviewing previous mathematical models and implementations for moving points. During this learning period, mathematical models described by Güting et al. [2000] were studied so that they can be extended and refined to support for alternative interpolation function and include aggregate functions. Reviewing other mathematical models described by Pelekis et al. [2004] were also found to be necessary to compare and validate mathematical models with Güting et al. [2000]. Understanding implemented works from Bezaye [2013] and Eftekhar [2012] is also vital in quality improvement and functional extensibility for later implementations on the project.

After identifying the appropriate mathematical model and also identifying possible extension mechanism for previously implemented works, the design phase follows. In the design phase, mathematical definitions of moving objects and operations on them from [Güting et al., 2000], is used to design corresponding data types and functions in mathematical language which later be converted into implementable code. Afterward, algorithms developed by Lema et al. [2003] used as a stepping stone for implementation of those operations that were omitted by Bezaye [2013]. The overall work flow is illustrated in Figure 1.1.

The abstract model developed by Güting et al. [2000] offers a type system built up from basic types and type constructs. The semantics of types from the type system is defined by their carrier set. In addition, a set of operations on types from the type system is designed and their syntax is defined by signature. Güting et al. [2000] define operations separately for temporal types and non-temporal types. However, operations on non-temporal types can later become operations on temporal types, by a technique called operator lifting. From the set of operations, Bezaye [2013] has only implemented operations from simple set theory and first-order logic query languages. The rest of the operations are implemented in this research project, and their importance on different applications is also illustrated with examples in Chapter 4.

In this research project, discrete models (types) designed by Forlizzi et al. [2000] is used to represent values of their corresponding types of the abstract model. Unlike the abstract model, which domain is represented in terms of infinite representation, the discrete model restricts range of values of the abstract model that can be represented. This allows discrete representation of types to be simply translated to data structures. In this discrete representation, base types and instants can be directly represented in terms of corresponding programming language types. Spatial types also have their corresponding discrete representations. Most importantly, moving types are represented as “sliced representation”. This includes fragmenting a temporal development value over time called “slices” that can be described by a simple interpolation function [Lema et al., 2003]. In other words, object states are captured by a static geometry paired with a time stamp, and through which a moving object gets represented as a list of time-ordered snapshots. In the sliced representation, an interpolation function can be used to determine a location between two stored time stamps.

In order to interpolate a value for moving point at the time for which we do not have stored information, Bezaye [2013] used simple linear interpolation function in her work. In this case, to compute the in-between value between two stored points (x_0, y_0) and (x_1, y_1) paired with time stamps t_0 and t_1, a function $f(t) = (x_0 + x_1 t, y_0 + y_1 t)$ is used, in which $t(t_1 - t_0)$ represents the time interval between the two ordered states of the moving point. As stated in [Erwig et al., 1999], using higher-order polynomials in a piece-wise approximation of the curve between two consecutive states of a moving object, is a way to more precisely approximate the curve and fewer snapshots may be needed. The result obtained by computing derivatives of linear functions for operations like speed and velocity makes linear approximation a bit unnatural. If we take the derivative of such a result for operations like acceleration, it is either 0 or infinite. However, curve-based representation models represent most natural moving objects like ships, airplanes, boats, and vehicles that do not display sudden bends . A parametric cubic function $P(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3$, is used to introduce curve-based interpolation [Yu et al., 2004] assuming the acceleration changes linearly in one direction during the period. A parametric function of degree 5 $P(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3 + a_4 t^4 + a_5 t^5$ is used by Yu and Kim [2006] to

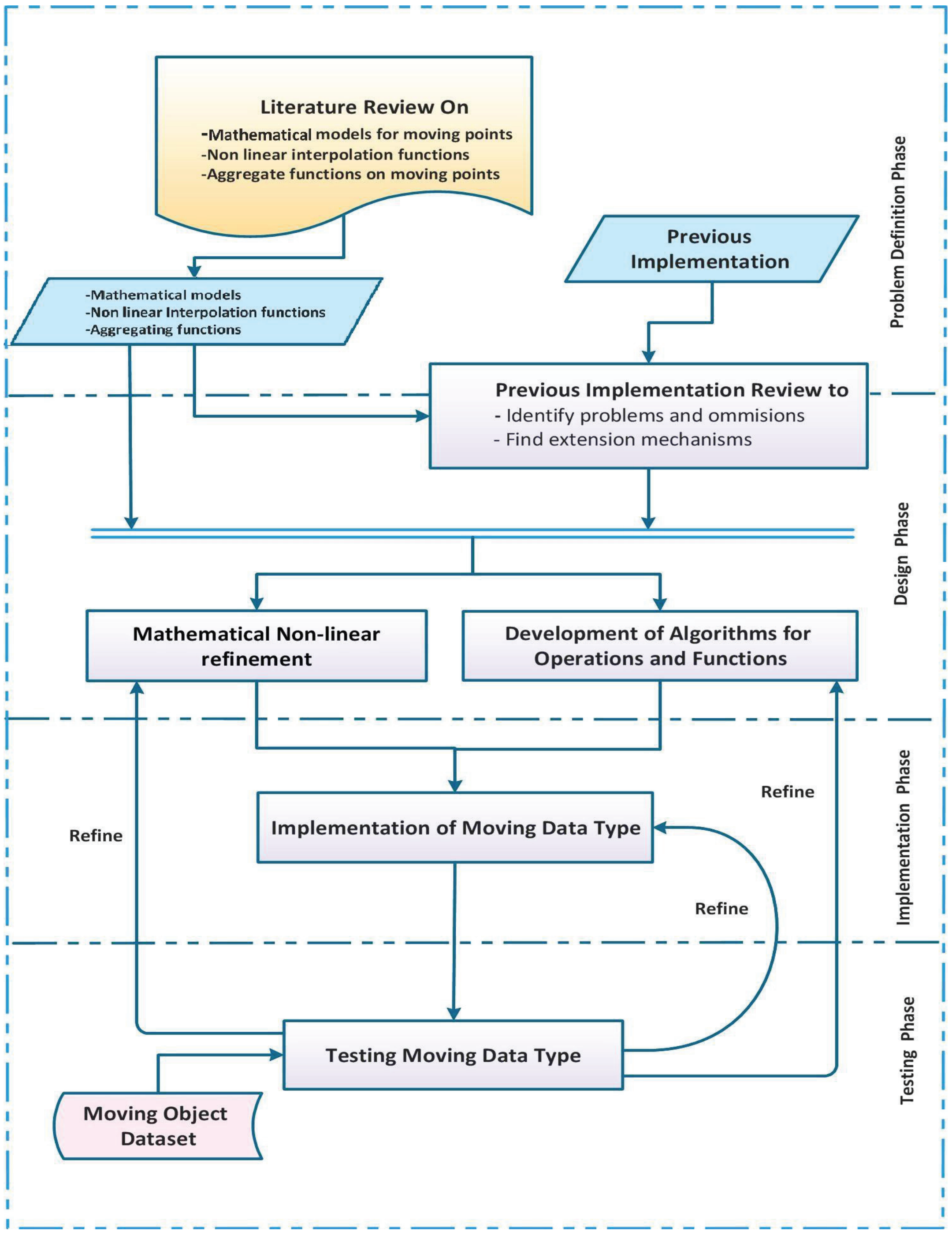

Literature Review On
-Mathematical models for moving points
-Non linear interpolation functions
-Aggregate functions on moving points
Previous Implementation
-Mathematical models
-Non linear Interpolation functions
-Aggregating functions
Previous Implementation Review to
- Identify problems and ommisions
- Find extension mechanisms
Mathematical Non-linear refinement
Development of Algorithms for Operations and Functions
Implementation of Moving Data Type
Refine
Refine
Refine
Testing Moving Data Type
Moving Object Dataset
Problem Definition Phase
Design Phase
Implementation Phase
Testing Phase

Figure 1.1: Method adopted

Table 1.1 List of Aggregate Functions on Sets of Moving Points

Method	Description
mpt_centroid	Returns the centroid ("center of gravity") as a geometry from the specified moving points within a given time interval.
mpt_mbb	Returns the minimum bounding rectangle of the specified Moving Points within a given time interval.
mpt_makeline	Return a line string from a set of moving points within a given time interval.
mpt_min_traj	Returns the shortest trajectory in space from a set of moving points. [Li et al., 2011]
mpt_max_traj	Returns the geometry line from the longest trajectory in space of a set of trajectories.[Li et al., 2011]
mpt_average	Returns an average speed of moving points within a given time interval.
mpt_array	Returns an array of points from sets of moving points for a given time instance.
mpt_outlier	Returns moving points from set of moving points that are not inside a given polygon for a given time instance.[Li et al., 2011]

consider speed, velocity, and acceleration of a moving object. However, the above-mentioned interpolation functions assume moving objects moving on a free two-dimensional space. Depending on the application scenario, movement of objects can be classified into unconstrained movement (e.g. vessels at sea), constrained movement (e.g. pedestrians), and movement in transportation networks (e.g. trains and cars) [De Almeida and Güting, 2005]. There exist features in space that hinder moving objects in their movement, and such feature "infrastructure" will have profound impact on their movement. Therefore, any interpolation function that takes such infrastructure into account requires a prior integration of background geographic information with the trajectory data. To store and retrieve objects moving in networks, the Moving Objects in Networks Tree (MON-tree) was proposed by Alvares et al. [2007]. In this research project, parametric cubic interpolation function of degree 3 is used to provide alternative support aside of linear interpolation function. Detailed explanation and also illustration with examples is provided in Chapter 5.

Non-temporal aggregate functions like min, max, avg, center, and count can be converted to their corresponding temporal aggregate function to perform computations over sets of moving objects [Lema et al., 2003]. There is no doubt about the importance of such operations, average speed as a time-dependent variable, center of a moving point sets as a time-dependent geometry are a few computations required for application having high interest on aggregate information. The list of aggregate functions realized in this research project is listed on Table 1.1. Detailed explanation and illustration with examples on aggregating functions is provided on chapter (unkown).

Chapter 2
Design for Representing Moving Points

2.1 INTRODUCTION

As it has been discussed on Chapter 1 of this research, geometries can display changes in both discrete and continuous steps. In addition, three basic abstractions were also mentioned to represent types of such geometries. To represent this change of value for spatial types with time, spatiotemporal models are required. Abstract and discrete models are used to represent objects that possess continuous position change in time. This includes defining algebras that are suitable for querying and storing spatiotemporal objects. The defined algebra consists of type system to introduce the basic types and type constructs. This type system is defined in Section 2.2.

To represent Spatiotemporal objects, the value for the types in the type system used by the model needs to be structured. From the type system, a data structure is designed for the moving point type. Moving real, moving period, intime real and intime point types are considered and their appropriated data structure is also designed. These temporal types are used to represent results from operations on moving point types or inputs to these operations. The next section explains detailed abstract models for moving points followed by the corresponding discrete representation and the data structure. This chapter also discusses implementation details and validation rules for types listed in this paragraph.

2.2 ABSTRACT REPRESENTATION OF MOVING POINT DATA TYPE

Abstract models are used to define moving points in terms of infinite set points. An abstract representation allows us to define moving point as a continuous curve in the 3-D space. A signature is used to generate an abstract representation for moving points. Kinds and type constructs are used to represent certain subsets of types and roles of operators, respectively. Table 2.1 shows the signature for defining the type system, from which important types like *int*, *real*, *instance*, *moving(int)*, *moving(real)*, *moving(region)*, *moving (point)* and so on are generated. The focuses of interest are spatiotemporal types representing moving objects as a moving point. *moving(point)*, *moving(real)*, spatial types and few base types are included in the design. **All the type definitions and philosophies used in this section are generated from Güting et al. [2000].**

Table 2.1 Signature Describing the Abstract Type System [Güting et al., 2000]

Type Constructor	Signature	
int,$real$,$string$,$bool$		$\rightarrow$ BASE
$point$,$points$,$line$,$region$		$\rightarrow$ SPATIAL
$instant$		$\rightarrow$ TIME
$moving$,$intime$	BASE $\cup$ SPATIAL	$\rightarrow$ TEMPORAL
$range$	BASE $\cup$ TIME	$\rightarrow$ RANGE

2.2.1 Base types

All the base types specified in Table 2.1 have formal interpretation, except that they are extended by the value $\perp$ (undefined).

Definition 1. *For a type α its carrier set is denoted by A_α. The carrier sets for the types int, $real$, $string$, and bool, are defined as*

$$A_{int} \triangleq \mathbb{Z} \cup \{\perp\}$$

$$A_{real} \triangleq \mathbb{R} \cup \{\perp\}$$

$$A_{string} \triangleq V^* \cup \{\perp\}, \text{ where } V \text{ is a finite alphabet,}$$

$$A_{bool} \triangleq \{FALSE,\ TRUE\} \cup \{\perp\}$$

2.2.2 Spatial types

Point, line, and regions are the basic abstraction of real world objects. In the design, two spatial types *point* and *points* are used. A value for a type *point* represents a point in the Euclidean plane or is undefined and a value for *points* represents a finite set of points. However, their mathematical definition is based on the point set paradigm which expresses the space as an infinite sets of points and spatial objects as distinguished subsets of space. In addition, point set topology provides concepts of continuity and closeness to express special topological structures of point set [Güting et al., 2000].

Definition 2. *The carrier sets for the types* point *and* points *are:*

$$A_{point} \triangleq \mathbb{R}^2 \cup \{\perp\}$$

$$A_{points} \triangleq \{P \subseteq \mathbb{R}^2 \mid P \text{ is finite }\}$$

2.2.3 Time type

In general, time can be considered as linear or continuous and it can be modeled as bounded or infinite. The type *instant* represents a point in time or undefined and it is isomorphic to the real numbers.

Definition 3. *The carrier set for* instant *is*

$$A_{instant} \triangleq \mathbb{R} \cup \{\perp\}.$$

2.2.4 Temporal types

To construct important temporal types from base and spatial types for this research project, the type constructor *moving* is used. Given a spatial type *point* to describe a point in a 3D plane and a base type *real*, temporal types *moving (point)* and *moving (real)* represents mapping from time to space and base type *real* (whose value comes from one dimensional domain), respectively.

Definition 4. *Let α be a data type to which the* moving *type constructor is applicable, with carrier set A_α. Then the carrier set for* moving(α), *is defined as follows:*

$$A_{moving(\alpha)} \triangleq \{f \,|\, f\colon A_{instant} \rightarrow A_\alpha \text{ is a partial function} \wedge \Gamma(f) \text{ is finite}\}$$

Hence, a value f from the carrier set of *moving (α)* is a function describing the development over time of a value from the carrier set of α. The condition " $\Gamma(f)$ is finite" says that f consists of only a finite number of continuous components [Güting et al., 2000]. As a result, it is possible to insert a point in time and ask for a position at a time between any two given times instances. The temporal type *moving (point)* derived using the *moving* constructor is used to represent a moving object that changes its point location with time. The other temporal type derived using the same constructor *moving(real)* is used to represent a sequence of continuously changing time varying real values. The importance of this temporal type is to support the *moving(point)* type to store supplementary information in addition to what it is intended for in the first place. For example, to store sensor reading associated with a particular moving object like speed and acceleration of moving objects. The importance of those temporal types is described in Section 2.4 in detail. Furthermore, to construct a type that associate instants of time with values of a given type α, the *intime* type constructor is used.

Definition 5. *Let* α *be a data type to which the* intime *type constructor is applicable with carrier set* A_α. *Then the carrier set for* intime(α), *is defined as follows:*

$$A_{intime(\alpha)} \triangleq A_{instant} \times A_\alpha.$$

based on the above definition, the *intime* type constructor is used to construct the new types *intime_mpoint* and *intime_mreal* to represent a single (instant, value)-pair of the temporal types *moving(point)* and *moving(real)*, respectively.

2.2.5 Range Types (Sets of Intervals)

In this project, operations to project the *moving(point)* and *moving(real)* types into their domain and range is implemented. Projections into the domain and range of the temporal types is represented as sets of intervals. For example, the temporal type *moving(real)* (whose values comes from the one-dimensional domain) are represented as sets of intervals over one-dimensional domain. The *range* type constructor is used to obtain types that represent these sets of intervals.

Definition 6. *Let* α *be a data type to which the* range *type constructor is applicable and there exists a total order. An* α*-interval is a set* $X \subseteq A_{(\alpha)}$ *such that* $\forall$ x, y $\in$ X, $\forall$ z $\in$ $A_{(\alpha)}$: $x < z < y \Rightarrow$ z $\in X$.

Definition 7. *Let* α *be any data type to which the* range *type constructor is applicable. Then the carrier set for range(α) is*

$$A_{range(\alpha)} \triangleq \mathrm{X} \subseteq A_{(\alpha)} \mid \exists \text{ an } \alpha\text{-range } R : X = \mathit{points}(R).$$

2.3 DISCRETE REPRESENTATION OF MOVING POINT DATA TYPE

The associated problem with an abstract representation is that we cannot store and perform operations on them in computers. As a result, some corresponding discrete representations are needed. To realize a moving point data type in a computer system, its discrete representation is modeled. A discrete representation of moving point data type represents small finite set of values of the infinite point sets that are used to design the data structures and algorithms for the type [Erwig et al., 1999]. The moving type constructor does not automatically transform types into corresponding temporal types at the discrete level of representation. As a result, a few new type constructs were

Table 2.2 Signature Describing the Discrete Type System [Erwig et al., 1999]

Type Constructor	Signature	
int,$real$,$string$,$bool$		$\rightarrow$ BASE
$point$,$points$,$line$,$region$		$\rightarrow$ SPATIAL
$instant$		$\rightarrow$ TIME
$range$,	BASE $\cup$ TIME	$\rightarrow$RANGE
$intime$	BASE $\cup$ SPATIAL	$\rightarrow$TEMPORAL
$const$	BASE $\cup$ SPATIAL	$\rightarrow$UNIT
$ureal$,$upoint$,$upoints$,$uline$,$uregion$		$\rightarrow$UNIT
$mapping$	MAPPING	$\rightarrow$UNIT

introduced to implement the *moving* constructor. The rest of type system for discrete model looks similar to the abstract type. Table 2.2 shows the signature describing discrete type system.

The types that are available in a programming language determine the carrier sets of the discrete models for base types and type for time. The base types *int*, *real*, *string*, *bool* can be implemented using their corresponding representation in programming languages. Type *instant* is also represented using the programming languages real numbers. Spatial types *point* and *points* also have their own discrete representation as well. The discrete model to represent moving object uses "slice representation" technique to represent temporal ("moving") types as discussed in Section 1.1.6.

2.4 DATA STRUCTURE

To design and implement data structures that store spatiotemporal objects as moving points, we mainly set the focus on the movement of cars and birds. The movement of such objects is considered to be continuous; this indicates the continuous change of location through time. The choice of the application domain was based on the nature of their movement and requirements of different alternative interpolation functions. Different interpolation functions can be used to interpolate locations in which actual measurements was not taken. The movement of cars is highly restricted by the structure of the road infrastructure, which is mostly straight and do not show sudden change in velocity. In contrast, the movement of birds is not limited by any infrastructure; and their change in velocity through time is not linear. In this research project, "Linear interpolation" and " Cubic interpolation" are used to interpolate in between locations of moving birds. Detailed explanation is provided in Chapter 5 of the research.

Now, let us assume a bird is flying from location A to location B for some period of time as show in Figure 2.1. In the figure, continuous and discrete representation are provided. The continuous representation approximates to render the exact movement of the bird on a continuous basis. Since current GPS and telecommunication technologies only allow us to sample object position, a discrete representation is required to create a data structure and store the movement in computers. The list (12.38 48.90 25.0 2013-05-08 12:00:02 22.4) is used to represent the measurement taken from a device which is equipped with GPS. The device measures spatial location and speed of the bird at a time. Therefore, the first three floating point values of the list represents the spatial location of the bird. The date and time values represent the time in which the measurement was taken. The final floating value represents the speed of the moving bird. A sequential list of timely ordered states of the bird, that are captured by the device approximates the continuous movement to its corresponding discrete representation as shown in Figure 2.1.

The birds can have undefined positions in time throughout their movement. Birds are lightweight

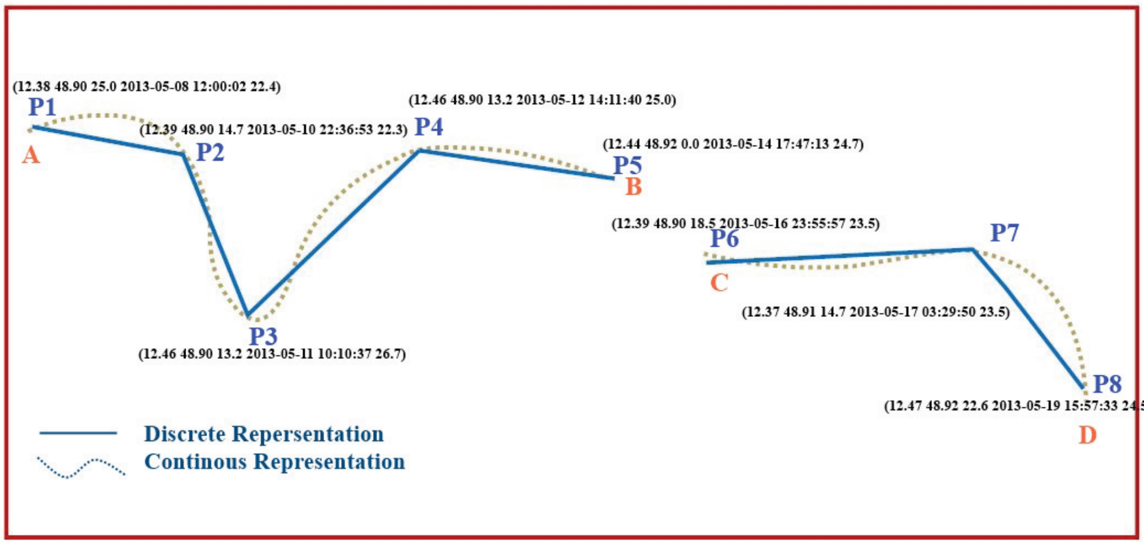


Figure 2.1: Movement of a Moving Car

animals that cannot carry around heavy equipments. As a consequence, the device installed acquires low number of positions or sometimes fails to obtain positions. This challenged bird-tracking with GPS, therefore, the data structure for moving point type is designed to consider moments when there is undefined positions of the moving point.

2.4.1 Data structure for the type moving point

After we create a discrete representation for the continuous movement of the bird, we can now easily define a data structure to represent a moving object (moving car and moving bird for our particular application domain) as a moving point. To implement a data type that represents a moving bird as a moving point, the structure defined in Listing 2.1 is used. The structure is highly generic and considers all the possible scenarios a moving object encounters. As show in Figure 2.1, a moving point is composed from an array of static geometries paired with a time stamp. In the data structure, it is represented using the LWPOINT (light weight point) data structure defined from LWGEOM (light weight geometry) of PostGIS. LWGEOM is much more like the original PostGIS geometries with few a differences. LWGEOMS are much smaller, support native 2d, 3d, and 4d points and they are internally very similar to OGC WKB representation.

The structure *mpoint* defined to represent moving points in Listing 2.1 consists of *num_segts*, *sr_id*, *tz_id*, and an array of reference **mpt_sgts*. *sr_id* represents spatial reference system identifier (SRID) of the point geometry and tz_id represent the time zone identifier of the time stamp. The pointer variable **mpt_sgts* stores an array of pointers to the structure *mpt_segment*. The reason to store segments of a moving point on a separate structure arises from the fact that, there might be times in which the movement of the moving object is not defined. For example, when the GPS device installed on the tracking device is switched off for several reasons. If the device acquires positions in a given interval, the GPS device being off causes moments where the position of the object is undefined. For example, as shown in Figure 2.1, to represent the movement of the point from location A to location D two segments are required. we need to store both the segments from location A to B and from location C to D separately as an array to define the movement of the point. *num_segts* stores the number of segments used to represent the moving point. The reason to count and store this information separately is that, the number of segments used to define the movement of the point is unknown. This cause the type *mpoint* be defined as variable-length type. PostgreSQL, requires all variable-length variables (types) begins with opaque length field of 4 bytes.This field contains the total length of the structure, which is calculated using the count of segments used to represent the moving point [Lockhart, 2013].

Listing 2.1: Data structure for moving point type

```
typedef struct mpt_segment
{
        int num_pts;
       LWPOINT *geom_array;
}mpt_segment;

typedef struct mpoint
{
        int num_segts;
        int sr_id;
        int tz_id;
        mpt_segment *mpt_sgts;
}mpoint;
```

The structure *mpt_segment* defined to represent segments of a moving point in Listing 2.1 contains *num_pts* and **geom_array*. The pointer variable **geom_array* stores references to LWPOINT point geometries of PostGIS. LWPOINT geometry consists the x, y, z coordinates and time t as the fourth dimension, in place of the M dimension specified by PostGIS. *num_pts* represents the number of positions acquired for a single segment of the moving object. The number of positions acquired for each segment of a moving point is different. The count information from this field is used to calculate the total size to be allocated for each segment of the moving point. Later, this size information is summed up to define the size of the moving point as a whole. Based on this structure, the moving bird rendered in Figure 2.1 in total contains five discrete positions in the first segment and three discrete positions for the second segment.

2.4.2 Data structure for the type moving real

The last floating point value (22.4), which represents the speed of the moving bird from the list (12.38 48.90 25.0 2013-05-08 12:00:02 22.4) is not included in the designed data structure of moving points. The reason behind is that, the speed of the moving bird may not be the only information required or measured from the devices used. In addition, the number of parameters measured from different devices and sensor readers is not always the same. In this situation, it is necessary to design a separate data structure to store those additional parameter values. The structure in Listing 2.2 is defined to store moving real values as *MMREAL*.

The structure *MMREAL* consists *num_mreal*, *tz_id* and an array of reference **mreal*. The pointer variable **mreal* stores an array of references to the structure *MREAL*. The structure *MREAL* in listing 2.2 contains *num_instance* and an array of reference **instance* to the structure *INTIME* which is composed of a variable *val* and time *t*. *num_instance* in the structure *MREAL* represent the number of value-time pairs used to defined the moving real type. *num_mreal* in the structure *MMREAL* represents the number of *MREAL* structures used to defined a collection of moving real values. The variables *size_mr* and *size_mmr* of the type size_t (long int) are used to store the total size allocated for the structures *MREAL* and *MMREAL*, respectively.

Listing 2.2: Data structure for moving real type

```
typedef struct INTIME{
        double val;
        time_t t;
} INTIME;

typedef struct MREAL{
        size_t size_mr;
        int num_instant;
        INTIME *instant;
}MREAL;

typedef struct MMREAL{
        size_t size_mmr;
        int num_mreal;
        int tzid;
        MREAL *mreal;
}MMREAL;
```

The above structures defined for moving point and moving real types in combination can enable us to store the list (12.38 48.90 25.0 2013-05-08 12:00:02 22.4). Now the list can be decomposed into parts MPOINT(12.38 48.90 25.0 2013-05-08) and MREAL(22.4 2013-05-08 12:00:02) to make them suitable for their corresponding structure. Finally, the decomposed components can be stored into a moving object database using two columns with the type *MPOINT* and *MREAL*. Therefore, decomposing and managing the data will be the responsibility of the user and this can be considers as one limitation of using moving reals to represent additional associated parameters of moving points.

2.4.3 Data structure for the type period

In this research project, operations that yields domain and range values of the function used to define moving point type are implemented. The types to represent those values is obtained through the *range* type constructor. The type *PERIOD* is used to represent ranges over the time domain, i.e., an interval. A time period is marked by an initial and a final timestamp. The type *MPERIOD* is defined for values that represent a collection of mutually non-overlapping *PERIOD*s. For example, the operation *mpt_deftime* in Section 4.2.1 returns time intervals in which the moving point is defined. This value is represented by a collection of interval values of the type *MPERIOD*. The structure *MPERIOD* consists *size_p*, *tz_id*, *num_period* and an array of pointer **period*. In this definition, *size_p* represents the memory size allocated for the structure *MPERIOD*, *tz_id* is the time zone and *num_period* represents the number of *PERIOD*s.

Listing 2.3: Data structure for period type

```
typedef struct PERIOD
{
        time_t  initial;
        time_t  final;
}PERIOD;

typedef struct MPERIOD
{
        size_t size_p;
        int     tzid;
        int num_period;
        PERIOD *period;
}MPERIOD;
```

2.4.4 Data structure for the type intime point

For all temporal types obtained from the *moving* type constructor, it's important to have a type that represents a single element from a set of value-instant pairs. To represent a single value of a moving point type, the *intime* type constructor defined in Section 2.2.4 is used. The type *INTIME_MPOINT* is the implementation of the theoretical notion of *intime(mpoint)*. As a result, functions that are interested in a single instance of a moving point can return or accept values of the type *INTIME_MPOINT*. The structure *INTIME_MPOINT* in Listing 2.4 consists of a variable *t* to represent a single instant of time, *tz_id* and a pointer **point* referencing a value of type LWGEOM.

Listing 2.4: Data structure for intime point type

```
typedef struct INTIME_MPOINT
{
        LWGEOM *point;
        time_t t;
        int tzid;
}INTIME_MPOINT;
```

2.4.5 Data structure for the type intime real

The type *INTIME_REAL* is implemented to represent a single element from the temporal type moving real. Similar definitions and concepts are used from the type *INTIME_MPOINT*. The structure *INTIME_REAL* in Listing 2.2 consists of a variable *t* to represent a single instant of time, *tz_id* and *val* to represent single value of the type *real*.

Listing 2.5: Data structure for intime real type

```
typedef struct INTIME_REAL
{
      double val;
      time_t t;
              int tzid;
}INTIME_REAL;
```

2.5 LANGUAGE CONSTRUCTS AND CONSTRAINTS

For the moving point and other supporting data type types, a valid textual representation is required to construct a single instance of each of the types selected in Section 2.2. This can be achieved in different ways: one is by specifying the context-free grammar (syntax) for each of the types and setting a number of constraints on the values (semantics) of each type. The context-free grammar used in this project is based on the OGC specification for a well-known text representation of simple features. Constraints defined for each type are be based on definitions from Güting et al. [2000] and PostgreSQL implementation requirements for extensibility.

BNF (Backus Normal Form or BackusNaur Form) notation technique is used to describe the syntax of languages used in computing. In this project, it is used to specify the context free grammar for each types. This consists, set of derivation rules, written as the form of '<symbol> ::= expression'. Symbols that appear on the left side are non-terminals and they are always enclosed between the pair <>. Symbols that never appear on the left side are also called terminals, sometimes written in italics. The following notations are used for BNF grammars listed below.

1: "::="- the notation denotes the right symbol is a possible substitution for the symbol on the left.

2: "()" - this notation denotes a sequence of symbols into a single symbol or token.

3: "*" - denotes the optional use of multiple instances of the symbol enclosed in braces or curly brackets.

4: "+" - denotes one or more instance of a symbol enclosed in braces or curly brackets.

5: "|" - denotes a choice between two symbols (used as OR conjunction) in a production.

6: "<>" - denotes a production defined elsewhere in the list or a basic type.

The General BNF specifications in Listing 2.6 are used and valid for all types in this project.

Listing 2.6: Common BNF specificaitons

```
                 <sign> ::= <plus_sign>|<minus_sign>
           <minus_sign> ::= -
            <plus_sign> ::= +
        <decimal_point> ::= <period>
               <period> ::= .
                <space> ::= <" ">
           <left_paren> ::= (
          <right_paren> ::= )
                <comma> ::= ,
                <colon> ::= :

     <unsigned_integer> ::= (<digit>)+
                <digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
 <signed_numeric_literal> ::= [<sign>]<unsigned_numeric_literal>
<unsigned_numeric_literal> ::= <unsigned_integer>
                            [<decimal_point>[<unsigned_integer>]]
                            |<decimal_point><unsigned_integer>

     <timestamp_string> ::= <formated_time_string>
 <formated_time_string> ::= <date_format><space><time_format>
          <date_format> ::= <unsigned_integer><minus_sign>
                            <unsigned_integer><minus_sign>
                            <unsigned_integer><space>
          <time_format> ::= <unsigned_integer><colon>
                            <unsigned_integer><colon>
                            <unsigned_integer>
       <timestamp_long> ::= <unsigned_integer>
```

2.5.1 Language construct and constraint for type MPOINT

Listing 2.7: Context free free grammer for *MPOINT* input

```
<MPOINT_tagged_text> ::= <SRID_text><TZID_text><MPOINT_text>
         <SRID_text> ::= SRID = <unsigned_integer>;
         <TZID_text> ::= TZID = <unsigned_integer>;

   <MPOINT_text> ::= MPOINT<left_paren>(<MPOINT_SEGMENT_text>)+<right_paren>|
                     MPOINTT<left_paren>(<MPOINTT_SEGMENT_text>)+<right_paren>|
                     MPOINTXY<left_paren>(<MPOINTXY_SEGMENT_text>)+<right_paren>|
                     MPOINTXYT<left_paren>(<MPOINTXYT_SEGMENT_text>)+<right_paren>

   <MPOINT_SEGMENT_text> ::= <left_paren>(<POINTXYZM_TIMESTAMP_text>)*<right_paren>
                             if(<MPOINT_SEGMENT_text>!=last)<comma>
  <MPOINTT_SEGMENT_text> ::= <left_paren>(<POINTXYZMT_TIMELONG_text>)*<right_paren>
                             if(<MPOINTT_SEGMENT_text>!=last)<comma>
 <MPOINTXY_SEGMENT_text> ::= <left_paren>(<POINTXYM_TIMESTAMP_text>)*<right_paren>
                             if(<MPOINTXY_SEGMENT_text>!=last)<comma>
<MPOINTXYT_SEGMENT_text> ::= <left_paren>(<POINTXYMT_TIMELONG_text>)*<right_paren>
```

```
                                        if(<MPOINTXYT_SEGMENT_text>!=last)<comma>

<POINTXYZM_TIMESTAMP_text>  ::=  <signed_numeric_literal><space>
                                 <signed_numeric_literal><space>
                                 <signed_numeric_literal><space>
                                 <timestamp_string>
                                  if(<POINT_TIME_text>!=last)<comma>
<POINTXYZMT_TIMELONG_text>  ::=  <signed_numeric_literal><space>
                                 <signed_numeric_literal><space>
                                 <signed_numeric_literal><space>
                                 <timestamp_long>
  <POINTXYM_TIMESTAMP_text>  ::=  <signed_numeric_literal><space>
                                 <signed_numeric_literal><space>
                                 <timestamp_string>
                                  if(<POINT_TIME_text>!=last)<comma>
                                  if(<POINT_TIME_text>!=last)<comma>
    <POINTXYM_TIMELONG_text>  ::=  <signed_numeric_literal><space>
                                 <signed_numeric_literal><space>
                                 <timestamp_long>
                                  if(<POINT_TIME_text>!=last)<comma>
```

Validation rules for moving points to express their semantics are as follows.

1: A moving point type must contain zero or more points in the same spatial reference system.

2: All time instants within a single moving point are registered against the same timezone.

3: A moving point must consist of zero or more segments. In case, the number of segments is zero, then the moving pointis considered to be *EMPTY*.

4: Segments of a moving point must contain zero or more 3D or 4D points. If the segment contains zero points and of consists only one segment, then the moving point is considered as *EMPTY*.

5: The point elements of a moving segment must be chronologically ordered using their time dimension represented as the third and fourth dimension of PostGIS LWPOINT for 3D or 4D points, respectively.

6: Segments of a moving point must be chronologically ordered.

7: Two different segments must not overlap in time.

Valid textual representation for the type *MPOINT* is provided in Listing 2.8.

Listing 2.8: Valid textual representation for moving points

```
"srid=4326;tzid=1;MPOINT((12.38 48.90 25.0 2013-05-08 12:00:02,12.39 48.90 14.7 2013-05-10
    22:36:53),(12.46 48.90 13.2 2013-05-12 14:11:40,12.44 48.92 0.0 2013-05-14 17:47:13)
    ,(12.39 48.90 18.5 2013-05-16 23:55:57,12.37 48.91 14.7 2013-05-17 03:29:50,12.47
    48.92 22.6 2013-05-19 15:57:33))"

"srid=4326;tzid=1;MPOINTT((12.38 48.90 25.0 1373773623,12.39 48.90 14.7 1373986714),(12.46
    48.90 13.2 1374184589,12.44 48.92 0.0 1374415352),(12.39 48.90 18.5 1374603004,12.37
    48.91 14.7 1374812624,12.47 48.92 22.6 1375228242))"

"srid=4326;tzid=1;MPOINTXY((12.38 48.90 2013-05-08 12:00:02,12.39 48.90 2013-05-10
    22:36:53),(12.46 48.90 2013-05-12 14:11:40,12.44 48.92 2013-05-14 17:47:13),(12.39
    48.90 2013-05-16 23:55:57,12.37 48.91 2013-05-17 03:29:50,12.47 48.92 2013-05-19
    15:57:33))"

"srid=4326;tzid=1;MPOINTTXYT((12.38 48.90 1373773623,12.39 48.90 1373986714),(12.46 48.90
    1374184589,12.44 48.92 1374415352),(12.39 48.90 1374603004,12.37 48.91
    1374812624,12.47 48.92 1375228242))"'
```

```

"srid=4326;tzid=1;MPOINT((12.38 48.90 25.0 2013-05-08 12:00:02,12.39 48.90 14.7 2013-05-10
    22:36:53))"

"srid=4326;tzid=1;MPOINT(())"

"srid=4326;tzid=1;MPOINT((NULL))"
```

2.5.2 Language construct and constraint for type MREAL

Listing 2.9: Context free free grammar for *MREAL* input

```
     <MREAL_tagged_text> ::= <TZID_text> <MREAL_text>
             <TZID_text> ::= TZID =<unsigned_integer>;

            <MREAL_text> ::= MREAL<left_paren>(<MREAL_SEGMENT_text>)*<right_paren>|
                             MREALT<left_paren>(<MREALT_SEGMENT_text>)*<right_paren>

    <MREAL_SEGMENT_text> ::= <left_paren>(<VALUE_TIME_text>)*<right_paren>
                             if(<MREAL_SEGMENT_text>!=last)<comma>
   <MREALT_SEGMENT_text> ::= <left_paren>(<VALUE_TIMET_text>)*<right_paren>
                             if(<MREAL_SEGMENT_text>!=last)<comma>

       <VALUE_TIME_text> ::= <signed_numeric_literal><space>
                             <timestamp_string>
                             if(<VALUE_TIME_text>!=last)<comma>
      <VALUE_TIMET_text> ::= <signed_numeric_literal><space>
                             <timestamp_long>
                             if(<VALUE_TIME_text>!=last)<comma>
```

Validation rules for moving reals to express their semantics are as follows.

1: A moving real must consist zero or more value-time pairs. If there exists no value-time pair, then the moving real type is *EMPTY*.

2: The value-time pairs must be chronologically ordered.

3: Collection of moving reals must not overlap in time.

4: If the moving real is used as a supporting for moving point, then the number of segment of the moving point must be equal to the number of moving real collection.

5: If the moving real is used as a supporting for moving point, then the number of points in a moving point segment must be equal to the number of value-time pairs of the corresponding moving real.

6: If the moving real is used as a supporting for moving point, then the timestamp of a point in a moving point segment must be equal to the timestamp from the value-time pair of the corresponding moving real.

Valid textual representation for the type *MREAL* is provided in Listing 2.10.

Listing 2.10: Valid textual representation for moving reals

```
"tzid=1;MREAL((0.000000 2013-05-08 12:00:02,0.009566 2013-05-10 22:36:53,0.076277
    2013-05-12 14:11:40),(0.142988 2013-05-14 17:47:13,0.209699 2013-05-16
    23:55:57,0.276410 2013-05-17 03:29:50),(0.379287 2013-05-19 15:57:33,0.482165
    2013-05-21 22:12:22,0.585042 2013-05-22 04:10:00,0.687920 2013-05-24
    07:21:08,40.693573 2013-05-26 17:53:27))"
```

```

"tzid=1;MREALT((0.000000 1374421723,0.009566 1378234514,0.076277 1378234995),(0.142988
    1378234514,0.209699 1378234995,0.276410 1378237908),(0.379287 1378240537,0.482165
    1378253353,0.585042 1378425103,0.687920 1378445656,40.693573 1378451654))"

"tzid=1;MREAL(())"

"srid=4326;tzid=1;MREAL((NULL))"
```

2.5.3 Language construct and constraint for type MPERIOD

Listing 2.11: Context free free grammar for *MPERIOD* input

```
<MPERIOD_tagged_text> ::= <TZID_text> <MPERIOD_text>
          <TZID_text> ::= TZID = <unsigned_integer>;

       <MPERIOD_text> ::= MPERIOD<left_paren>(<MPERIOD_SEGMENT_text>)*<right_paren>

<MPERIOD_SEGMENT_text> ::= <left_paren>(<PERIOD_text>)*<right_paren>
                           if(<MPERIOD_SEGMENT_text>!=last)<comma>

        <PERIOD_text> ::= <timestamp_string><minus_sign><timestamp_string>
```

Validation rules for periods to express their semantics areas follows.

1: The start and end of a period must be chronological.

2: Two different instances of a multiple period must not overlap with one another.

Valid textual representation for the periods as *MPERIOD* is provided in Listing 2.12.

Listing 2.12: Valid textual representation for multiple periods

```
"tzid=1;MPERIOD((2013-05-08 12:00:02 - 2013-09-19 02:53:58))"

"tzid=1;MPERIOD((2012-03-12 12:05:40 - 2012-03-12 12:09:40),(2012-03-12 12:12:45 -
    2012-03-12 12:17:55),(2012-03-12 12:19:00 - 2012-03-12 12:24:40))"
```

2.5.4 Language construct and constraint for type INTIME_MPOINT

Listing 2.13: Context free grammar for *INTIME_MPOINT* input

```
<INTIME_MREAL_tagged_text> ::= <SRID_text><TZID_text><INTIME_MPOINT_text>
               <SRID_text> ::= SRID = <unsigned_integer>;
               <TZID_text> ::= TZID = <unsigned_integer>;

      <INTIME_MPOINT_text> ::= INTIME_MPOINT<left_paren><POINT_TIME_text><right_paren>

         <POINT_TIME_text> ::= <signed_numeric_literal><space>
                               <signed_numeric_literal><space>
                               <signed_numeric_literal><space>
                               <timestamp_string>
```

Validation rules for intime points to express their semantics are as follows.

1: A single point-time pair of a moving point must be used to define the intime point type and it must be between the range and domain of a moving point.

2: In case the point-time is not provided then it is assumed to have *EMPTY* value.

Valid textual representations for the type *INTIME_POINT* is provided in Listing 2.14.

Listing 2.14: Valid textual representation for intime points

```
"srid=4326;tzid=1;INTIME_MPOINT(8.295159 45.188845 24.7 2013-07-12 12:07:40)"

"srid=4326;tzid=1;INTIME_MPOINTXY(8.295159 45.188845 2013-07-12 12:07:40)"
```

2.5.5 Language construct and constraint for type INTIME_MREAL

Listing 2.15: Context free free grammar for *INTIME_MREAL* input

```
<INTIME_MREAL_tagged_text> ::= <TZID_text> <INTIME_MREAL_text>
               <TZID_text> ::= TZID = <unsigned_integer>;

       <INTIME_MREAL_text> ::= INTIME_MREAL <left_paren><VALUE_TIME_text><right_paren>

         <VALUE_TIME_text> ::= <signed_numeric_literal><space>
                               <timestamp_string>
```

Validation rules for intime reals to express their semantics are as follows.

1: A single value-time pair from values of a moving real type must be used to define the intime real type and it must be between the range and domain of a moving real type.

2: In case the value-time is not provided then it is assumed to have *EMPTY* value.

Valid textual representation for the type *INTIME_REAL* is provided in Listing 2.16.

Listing 2.16: Valid textual representation for intime reals

```
"tzid=1;INTIME_MREAL(13965770.698342 2013-09-19 02:53:58)"
```

2.6 CONCLUSION

In this chapter the abstract and discrete representation of moving point and its supporting types are presented. In the abstract representation, the philosophy and mathematical basis for base, spatial, time, temporal and range types are discussed. The rationale behind the discrete representation of moving point type is also summarized in the chapter. Succeeding the theoretical basis for moving point type, data structures for moving point and its supporting types are designed. In addition, language constructs and constraints on moving point and supporting types are defined to provide a valid textual representation and semantics of the representation. Generally, the designs defined in this chapter is the base for the implementation of moving object data type and development of algorithms that are going to be used in the realizations of operations on moving points.

Chapter 3
Implementation and Setup

3.1 INTRODUCTION

This chapter discusses the setup, environment, implementation and demonstration of the new types defining the moving object library. This includes: a brief explanation about the development platform and the setup and build methods adopted on the research project. The motive for selecting PostgreSQL as a DBMS and PostGIS as a library as extension is also explained.

3.2 ENVIRONMENT AND SETUP

The main objective of in this research project is the implementation of the moving objet data type. To achieve this objective, we selected a database management system software which supports high level extensibility. We choose to use PostgreSQl which is a catalog driven, powerful and open source database management system. PostgreSQL stores information about the tables, columns, data types, functions, and access methods on the catalogues. This allow users to access these informations and entertain the benefit of extensibility provided by the DBMS. In addition, PostgreSQL can incorporate user defined shared library into itself, through dynamic loading. In the research project C programming language was selected to build functions and types, arising for the fact that both PostgreSQL and PostGIS have been developed using the open source programming language GNU C.

After considering the implementation by Bezaye [2013], we decided to use the platforms and languages listed in Table 3.1.

Table 3.1 Platforms and languages used for building moving object library

Platform and Languages	Version
OS	Windows 7
DBMS	PostgreSQL 9.2
Compiler	Microsoft Visual Studio 2010
Geometry library	PostGIS 2.0.4
Programming language	C
Query language	SQL

PostGIS library is a spatial extension of PostgreSQL with spatial data types and enormous number of functions, that is used to support the implementation of the moving object library. We developed and compiled the moving object library with visual C++: the native development toolset for windows. PostGIS requires three dependencies for building and usage, namely GEOS, PROJ4, and GDAL. GEOS geometry library enhances topological expectations and improve geometry validation. PROJ4 library provides support for coordinate re-projection with in PostGIS. GDAL is extension to PostGIS for raster support. However, for this research project native geometry structures of PostGIS named LWGEOMs were used.

Extending PostgreSQL for Moving object library requires the development of user-defined data types and functions. As specified in PostgreSQL manual, user defined functions and types can be written in C. These functions and types are compiled into a dynamically loadable object (shared libraries) and are loaded by the server on demand [Lockhart, 2013]. Basically, PostgreSQL provides two different calling convention : 'version 0' and 'version 1'. We used the later by preceding every function implementation with the macro definition *PG_FUNCTION_INFOR_V1()*. PostgreSQL store data types as a "blob of memory" and for user defined types it requires functions to operate on them. This indicates that PostgreSQL requires user defined *input* and *output* functions to input and output the data. The *input* and *output* functions defined for types in the moving object library is explained in Section 3.3.1.

3.3 IMPLEMENTATION OF MOVING OBJECT TYPES

3.3.1 Input and Output functions

Any user defined types for PostgreSQL must always have input and output function to determine both the textual and in-memory type representation. Table 3.2 shows *Input* and *Output* functions implemented for each type in the moving object library.

Table 3.2 Input and Output functions

Types	Input and Output functions
MPOINT	mpoint_in()
	mpoint_out()
MMREAL	mreal_in()
	mreal_out()
MPERIOD	mperiod_in()
	mperiod_out()
INTIME_MPOINT	intime_mpoint_in()
	intime_mpoint_out()
INTIME_MREAL	intime_in()
	intime_out()

All of the input functions implemented for each type follow the same procedure. First, the function take the null-terminated string representation of the types as an argument. Afterward, Each input function has a complete and robust parser of the textual representation: This part of the function is responsible for validating the textual representation with the BNF provided in Section 2.5 of each type. If the null-terminated string violates any of the syntax provided as a grammar, the function passes a command to terminate the execution of the function and report an error message as a result. Otherwise the function stores the textual representation to the structure of each type. The parsing part of the function will not check for violation of any semantic based errors. To check for these semantic based errors the functions *mpt_validate* and *mpt_mreal_validate* are available in Chapter 4.

Having all the values stored in the structure, the next step is to serialize them into memory. The serialization process includes writing the content of the structures into memory and calculating the exact size of the values stored in the structure. The values of the structure are stored in a sequence of memory to easily de-serialize them back for the output function of the type. For this purpose, a special serialization function is written to be triggered by the input function after parsing and storing the string representation of the type to the structure. If the values of the types vary in size, PostgreSQL requires the structure of the types to contain a variable length of the type size_t, that comes before any of the other variables in the structure. This variable holds the computed size of the structure used to store the string representation of the type. This variable must be set using the function *SET_VARSIZE()* and retrieved using the function *VARSIZE()*. Finally the input function returns the initial memory address from the sequence of memories used to store values of the type to PostgreSQL.

The output functions defined for each type in Table 3.2 accept the pointer value that refers to the initial memory address of the type where the value is stored, de-serialize the information stored in a sequence of memory and store them back into the structures defined for the type. Afterward, from the values of the structures the textual representation of the types is constructed. And this textual representation is returned as a null-terminated string value to PostgreSQL; therefore we developed the input and output functions carefully to avoid several problems that might arise while we dump the data into a file and read it back.

3.3.2 Creating types in PostgreSQL

PostgreSQL provides three ways to create types owned by the user who defines them. User defined types compiled into a shared library can be defined in PostgreSQL using '*CREATE TYPE*' SQL. First, the '*CREATE TYPE*' creates a placeholder type that allows us to reference the type while defining input and output functions. After defining the Input/Output functions for the new user defined types, we can finally provide the full definition of the type as shown in Listing 3.1. The listing also shows the SQL statements used to initialize the type MPOINT in PostgreSQL. In general, to create an extension on PostgreSQL for moving point and its supporting types, a separate SQL file is written to contain all queries to initialize and register implemented types and functions associated with them. Refer Appendix A for Query statements used to create all the implemented types in PostgreSQL .

Listing 3.1: Query statments to create a moving point type in PostgreSQL

```
CREATE TYPE mpoint;

CREATE OR REPLACE FUNCTION mpoint_in(cstring)
  RETURNS mpoint AS
 'D:/postgis/postgisproject/Debug/PostgisRaster', 'mpoint_in'
LANGUAGE c IMMUTABLE STRICT;

CREATE OR REPLACE FUNCTION mpoint_out(mpoint)
  RETURNS cstring AS
 'D:/postgis/postgisproject/Debug/PostgisRaster', 'mpoint_out'
LANGUAGE c IMMUTABLE STRICT;

CREATE TYPE mpoint (
    input = mpoint_in,
    output = mpoint_out,
    alignment = double
);
```

3.4 DEMONSTRATION OF NEW TYPES

Having all the types and their associated input and output functions registered in PostgreSQL, we can now demonstrate the types by creating tables with fields of the types *mpoint*, *mmreal*, *mperiod*, *intime_mpoint*, and *intime_mreal*. Listing 3.2 shows how to use the types to create a table and insert values to them.

Listing 3.2: Query statements to demonstrate the use of the types MPOINT, MREAL, MPERIOD, INTIME_MPOINT, and INTIME_MREAL

```
/*creating a table containing the types MPOINT and MREAL*/

CREATE TABLE moving_birds
(
  ind_id varchar(80),
  ind_can_name varchar(80),
  locations mpoint,
  distance mmreal
);

/*feeding the data to the tables (Relations)*/

INSERT INTO moving_birds(ind_id,ind_can_name,locations,distance)
VALUES('Anita','Cuculus canorus','srid=4326;tzid=1;MPOINT((12.389940 48.906740 0.000000
2013-05-08 12:00:02,12.399370 48.905130 0.000000 2013-05-10 22:36:53,12.466070 48.903930
0.000000 2013-05-12 14:11:40, 7.731870 9.805520 0.000000 2013-09-19 02:53:58))',‘tzid=1;
MMREAL((0.000000 2013-05-08 12:00:02,0.009566 2013-05-10 22:36:53,0.076277 2013-05-12 14
:11:40,3561.189453 2013-09-16 19:48:54,3601.195313 2013-09-19 02:53:58))');

/*retrieving the data from tables (Relations)*/

SELECT * FROM moving_birds;

Result :

ind_id|ind_can_name   |locations                                |distance
------+---------------+-----------------------------------------+------------------------
Anita|Cuculus canorus|srid=4326;tzid=1;MPOINT((12.389940        |tzid=1;MMREAL((0.00
     |               |48.906740 0.00 2013-05-08 0.00            |2013-05-08 12:00:02,0.009566
     |               |2013-05-10 22:36:53,12.466070 48.903930|2013-05-10 22:36:53,0.076277
     |               |0.00 2013-05-12 14:11:40,7.731870         |2013-05-12 14:11:40,3561.189
     |               |9.805520  0.00 2013-09-19 02:53:58))"     |2013-09-16 19:48:54,3601.195
     |               |                                          |2013-09-19 02:53:58))

/*creating a  table containing the type MPERIOD*/

CREATE TABLE mperiod_table
(
 id int,
 period MPERIOD
);

/*feeding the data to the tables (Relations)*/

INSERT INTO mperiod_table(id,period)
VALUES(12,'tzid=1;MPERIOD((2012-03-12 12:05:40-2012-03-12 12:09:40),(2012-03-12 12:09:45-
2012-03-12 12:09:50),(2012-03-12 12:05:40-2012-03-12 12:09:40),(2012-03-12
     12:09:45-2012-03
-12 12:09:50))');

/*retrieving the data from tables (Relations)*/

SELECT * FROM mperiod_table;

Result :

id| period
--+-----------------------------------------------------------------------------------
12|tzid=1;MPERIOD((2012-03-12 12:05:40 - 2012-03-12 12:09:40),(2012-03-12 12:09:45 - 2012-
  |03-12 12:09:50),(2012-03-12 12:05:40 - 2012-03-12 12:09:40),(2012-03-12 12:09:45 -
  |2012-03-12 12:09:50))

/*creating a table containing the type INTIME_MPOINT*/

CREATE TABLE intime_table
(
 id int,
 intime INTIME_MPOINT
);

/*feeding the data to the tables (Relations)*/

INSERT INTO intime_table(id,intime)
VALUES(12,'srid=4324;POINTZ(8.295159 45.188845 0.000000),2013-07-12 12:07:40)');

/*retrieving the data from tables (Relations)*/

SELECT * FROM intimep_table;

Result :

id| intime
--+------------------------------------------------------------------------------------------
12|srid=4324;POINTZ(8.295159 45.188845 0.000000),2013-07-12 12:07:40)

/*creating a table containing the type INTIME_MREAL*/

CREATE TABLE intime_mreal_table
(
 id int,
 intime INTIME_MREAL
);

/*feeding the data to the tables (Relations)*/

INSERT INTO intime_mreaL_table(id,intime)
VALUES(12,'intime_mreal(1.8 2012-03-12 12:05:40)');

/*retrieving the data from tables (Relations)*/

SELECT * FROM intime_mreaL_table;

Result :

id| intime
--+------------------------------------------------------------------------------------------
12|INTIME_MREAL(1.799999 2012-03-12 12:05:40)
```

Chapter 4
Operations on Moving Points

4.1 INTRODUCTION

To represent entities that posses positional change in time like vehicles, aircraft's, ships, animals, peoples and etc., a data model, query language and corresponding DBMS implementation is required [Lema et al., 2003]. Based on the abstract and discrete representation of such entities, we introduced new types from the set of temporal system of types: moving point and moving real. In the next step, we designed and implemented the data structure for these types as an extension of PostGIS library. Storing the movement information on the database is not the only goal of the research project; rather it's the first step to achieve the objectives of the research.

A useful library should store both the moving entities and a set of operations to study the entities behavior. For example, we might be interested to know locations of a moving point at any time, within the moving objects defined time. We want to represent the moving object in different coordinate systems. In a car monitoring application, one might be interested to know if a moving car is defined for some period of time. This and other practical operations must be included in the moving object library. Therefore, we implemented such and other operations by writing functions on moving point types that would manipulate stored records from the database.

Table 4.1 Classes of operations on Nontemporal types

Class	Operations
Predicates	**isempty**
	$=, ! =$, **intersects, inside**
	$<, \leq, \geq, >$, **before**
	touches, attached, overlaps, on_border, in_interior
Set Operations	**intersection, union, minus**
	crossings, touch_points, common_border
Aggregation	**min, max, avg, center, single**
Numeric	**no_components, size, perimeter, duration, length, area**
Distance and Direction	**distance, direction**
Base Type Specific	**and, or, not**

The abstract model offers unlimited number of operations on moving objects. However, the design and implementation of operations for the types introduced in the research was based on classes of operations from Güting et al. [2000]. To include operations that are highly useful to study the behavior of a moving entity, simple set theory and first order logic queries on moving types are included. The operations are categorized into three classes: projection to domain/range, interaction with domain/range and rate of change. The operations included in each of these classes are listed in Table 4.2. In addition, operations that are not included in those classes are also designed and implemented for moving point type. Such as: operations based on order relationship (chronological order), topology and metric spaces are included, additionally, non-temporal operations are converted into temporal operations using the mechanism called operation lifting. The list of non-temporal operations are listed in Table 4.1. For example, distance operation on non-temporal types calculates the distance between two static geometries. This operation is lifted to calculate the distance between points on a moving point and as a result it return value of the type moving real. Non-temporal aggregating functions are also lifted (see Chapter 6).

Table 4.2 Classes of operations on Temporal types

Class	Operations
Projection to domain/range	**deftime,rangevalues, locations, trajectory, routes,traversed,inst,**
Interaction with domain/range	**atinstant,atperiods, initial,final,present, at,atmin,atmax,passed**
Rate of change	**derivative,speed, turn,velocity**
Lifting	(all new operations infered)

As mentioned in Güting et al. [2000], the design of operations on temporal types must adheres to three principles: the design must be as generic as possible; achieve between temporal and non-temporal types; and the operations must capture the interesting phenomena. The design and implementation of the algorithms used for the operations was based on a systematic study on algorithms from Lema et al. [2003]. Algorithms that are assumed to be straight forward are not included in this chapter; only their practical use and results are discussed.

When defining the algorithms for those operations, we made an assumption about the reader to have the preliminary knowledge on structures of the types MPOINT, MREAL, MPERIOD, INTIME_MREAL, and INTIME_MPOINT that are presented in Section 2.4. Terms *x, y, z, t, num_pts, geom_array, num_sgts, sr_id, tz_id, mpt_sgts, val, num_inst, instant,num_mreal, mrea_initial, final, num_period, period*, and *point* have similar meaning and definition with the fields of the structures defined for each type in Section 2.4. *mpoint, mreal, mperiod, intime_mreal*, and *intime_mpoint* represent a single instance of their corresponding type. *inst* represents time instance of type INSTANT. The algorithms does not cover how values of each type are written into and read from memory, as it has been discussed in Chapter 3.

This chapter provides a detailed explanation on algorithms, practical use, results and visualization for the operations realized. The operations are also tested by a dataset of movement of birds. Listing 4.1 shows the relations used to explain and test the types, and operations on them.

Listing 4.1: `Query statments to create relations used in the research project`

```
CREATE TABLE moving_birds(ind_id varchar(80),ind_can_name varchar(80),locations mpoint
    ,distance mmreal);
CREATE TABLE moving_birds_trajectoryp(ind_id varchar(80),ind_can_name varchar(80),
    locations geometry);
CREATE TABLE moving_birdsatinstant(ind_id varchar(80),ind_can_name varchar(80),
    locations geometry);
CREATE TABLE moving_birds_atperiod(ind_id varchar(80),ind_can_name varchar(80),
    locations mpoint);
CREATE TABLE moving_birds_atperiod_trajectory(ind_id varchar(80),ind_can_name varchar
    (80),locations geometry);
```

4.2 OPERATIONS FOR PROJECTION TO DOMAIN/RANGE

Operations explained in this section operate on values from moving point type. As a result, they perform computations to return projections with respect to the temporal value (the domain) or the function value (the range).

4.2.1 mpt_deftime

The domain function *mpt_deftime* returns all the times for which a moving object is defined. The function accepts a value of the type MPOINT as argument and returns the times the moving point is defined as a value of the type MPERIOD. The function returns *empty*, if a moving point is not defined. If a moving point have more than one segment, the function returns a collection of periods that is equal to the number of segments in the moving point. The collection of periods returned as a result are in the same time zone with the moving point, where the function is applied to. For non-empty values of the type MPERIOD, initial and final attributes of each period, holds the initial and final timestamp of the first and last value-time pairs of the segments in a moving point. The algorithm used to implement the function is presented in Algorithm 4.1.

Algorithm 4.1mpt_deftime operation

INPUT TYPE: MPOINT
OUTPUT TYPE: MPERIOD

```
1: $mperiod \leftarrow NULL$
2: $period \leftarrow NULL$
3: for $i = 0$to$mpoint.num\_sgts$ do
4:     for $j = 0$to$mpt\_sgts.num\_pts$ do
5:         if $j == 0$ then
6:             $period.initial \leftarrow mpt\_segts[i].geom\_array[j].m$
7:         else if $j == mpt\_segts[i].num\_pts - 1$ then
8:             $period.final \leftarrow mpt\_segts[i].geom\_array[j].m$
9:         end if
10:    end for
11:    $mperiod[i].period \leftarrow period$
12: end for
13: return $mperiod$
```

For example, from the dataset on moving birds, we might be interested to know the times for which the movement of a particular bird is defined. For this we can write a query that answers the question as follows in Listing 4.2.

Listing 4.2: What are the times for bird "Belarus 1" movement is defined?

```
Query 1.

SELECT ind_id, ind_can_name, mpt_deftime(locations) AS deftime
FROM moving_birds
WHERE ind_id= 'Belarus 1' ;

Result 1.

ind_id      |ind_can_name      |deftime
```

```
-----------+-----------------+--------------------------------------------------------
Belarus 1  |Cuculus canorus  |tzid=1;MPERIOD((2013-04-30 12:56:25 - 2013-09-17 22:38:52))
```

4.2.2 period_initial

The operation *period_initial* return the initial value of the type MPERIOD. The function accepts a value of the type MPERIOD as argument and return a timestamp as a result. In cases, where the type MPERIOD is composed of more than a single time interval(period), the function return the initial value of the first period. Moving points can be queried to request the time their movement has began. This can be achieved by combining the operations *mpt_deftime* and *period_initial*. The query statement in Listing 4.3 requests the time for the bird "Belarus1" starts its movement.

Listing 4.3: When did the bird "Belarus 1" start movement?

```
Query 2.

SELECT ind_id,ind_can_name,period_initial(mpt_deftime(locations)) AS initial
FROM moving_birds
WHERE ind_id= 'Belarus 1' ;

Result 2.

ind_id       |ind_can_name      |initial
-------------+------------------+---------------------
Belarus 1    |Cuculus canorus   | (2013-04-30 12:56:25)
```

4.2.3 period_final

The *period_final* function return the final value of the type MPERIOD. The function takes a value of the type MPERIOD as argument and return a timestamp as a result. For values of MPERIOD that have more than one non-overlapping time intervals (periods), the function return the final timestamp value of the last time interval (period). The query statement in Listing 4.4 return time of the bird "Belarus 1" ends its movement.

Listing 4.4: When did the bird "Belarus 1" stop its movement?

```
Query 3.

SELECT ind_id,ind_can_name,period_final(mpt_deftime(locations)) AS final
FROM moving_birds
WHERE ind_id= 'Belarus 1' ;

Result 3.

ind_id       |ind_can_name      |final
-------------+------------------+----------------------
Belarus 1    |Cuculus canorus   | (2013-09-17 22:38:52)
```

4.2.4 mpt_location

As mentioned in Güting et al. [2000], *points* and *lines* can be used to project moving points on the plane. These *points* and *lines* can be obtained from operations location and trajectory, respectively. Operation *locations* returns the projection as a multipoints value if the moving point changes its position in discrete steps only. For moving points that change their position continuously, operation *trajectory* returns the projection as a multilinestring value.

The operation *location* returns only the registered locations from the (location-time) pairs of a moving point as a type of *points* geometry value. The function implemented for this purpose, first initializes a lightweight multi-point geometry. The location coordinates used in the (location-time) pairs of each segment in the moving point are extracted to construct the newly initialized multi-point geometry. If the moving point contains no registered (location-time) pair, then the operation *mpt_locations* return the *null* geometry. This type of projection is useful when the moving point changes its position in discrete steps only, or never changes its position at all. The algorithm used for the implementation of the function is presented in Algorithm 4.2.

Algorithm 4.2mpt_location operation

INPUT TYPE: MPOINT
OUTPUT TYPE: MULTIPOINT

```
 1: $lwmultipoint \leftarrow NULL$
 2: for $i = 0 to mpoint.num\_sgts$ do
 3:     $LWLINE \leftarrow NULL$
 4:     for $j = 0 to mpt\_sgts.num\_pts$ do
 5:         $point.x \leftarrow mpt\_segts[i].geom\_array[j].x$
 6:         $point.y \leftarrow mpt\_segts[i].geom\_array[j].y$
 7:         $point.z \leftarrow mpt\_segts[i].geom\_array[j].z$
 8:         $lwmultipoint\_add\_point(lwmultipoint, point)$
 9:     end for
10: end for
11: return $lwmultipoint$
```

In practical terms, we want to know and perform different spatial operations on the locations extracted from a moving dataset. Let us say we planned to use this operation on service delivery buses, cars or individuals moving datasets. The operation returns the registered locations of these moving objects. This information in combination with few spatial queries can be used to decide which moving entity to assign for a new service request. The query in Listing 4.5 asks for the registered locations of a particular moving bird.

Listing 4.5: What are the locations registered for the bird "Nobert"?

```
Query 4.

SELECT ind_id,ind_can_name,mpt_location(locations) AS locations
FROM moving_birds
WHERE ind_id='Nobert';

 Result 4.

 ind_id   |ind_can_name      |locations
----------+------------------+------------------------------------------------------------
 Nobert   |Cuculus canorus   |SRID=4326;MULTIPOINT(12.285910 48.985900 0.000000,12.264980
          |                  | 48.986460 0.000000,12.263210 49.011830 0.000000,12.275840
          |                  | 48.985240 0.000000, ..... ,9.948900 10.455110 0.000000)

Query 5.

 SELECT ind_id,ind_can_name,mpt_location(locations) AS locations
   FROM moving_birds
```

The result of the query in Listing 4.5 is too large to be listed. However, Figure 4.1 visualizes it. All the locations registered for each bird in the moving bird dataset are included and separated by colors.

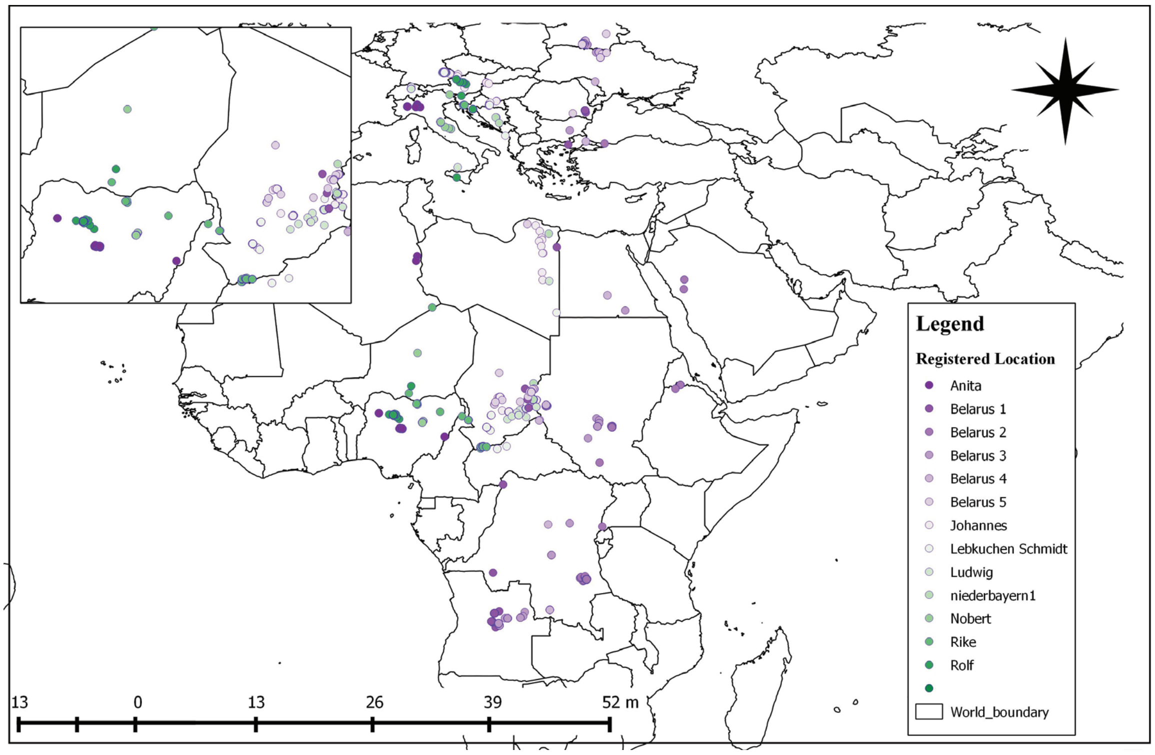


Figure 4.1: mpt_location

4.2.5 mpt_trajectories

As discussed in the previous section, a more natural projection of a continuously moving point on a plane is achieved with a *trajectory* operation. This operation returns the movement of the moving object as a multilinestring geometry which is termed as a *trajectory*. In other words, the trajectory refers to the trace of the moving point in time. In general, the function *mpt_trajectories* first initializes a lightweight line and a lightweight multiline geometries. All the registered locations of the segments in the moving point are used to construct the newly initialized line geometry. This line geometry constructs the multilinestring which is returned as a result of this operation. In case, there exists one or no point location registered for the moving point, the function returns *null* multilinestring geometry as a result. If the moving point only contains a single segment with two or more registered locations, a multilinestring with a single line segment is returned. The general algorithm used for the implementation of this function is presented in Algorithm 4.3.

The operation *trajectory* is a more preferable way to visualize the movement of the object. The trajectory that represent the path of the moving object in space is assumed to be linear, as a function of time in space. For example, we may be interested to know where the moving object might be within two registered locations. To answer this, we can use the interpolation methods discussed in Chapter 5. However, to make sure that the newly interpolated positions are accurately located on the trajectory of the moving object; we can use the result of this trajectory operation together with the interpolated position. The query to return the trajectory of a single and multiple moving birds is provided in Listing 4.6.

Algorithm 4.3`mpt_trajectories operation`

INPUT TYPE: MPOINT
OUTPUT TYPE: MULTILINESTRING

```
 1: $lwmline \leftarrow NULL$
 2: for $i = 0$ to $mpoint.num\_sgts$ do
 3:     $lwline \leftarrow NULL$
 4:     for $j = 0$ to $mpt\_sgts.num\_pts$ do
 5:         $point.x \leftarrow mpt\_segts[i].geom\_array[j].x$
 6:         $point.y \leftarrow mpt\_segts[i].geom\_array[j].y$
 7:         $point.z \leftarrow mpt\_segts[i].geom\_array[j].z$
 8:         $lwline\_add\_point(lwline, point)$
 9:     end for
10:     $lwmline\_add\_line(lwmline, LWLINE)$
11: end for
12: return $lwmline$
```

Listing 4.6: `Which route does the bird "Belarus 4" follows in the movement?`

```
Query 6.

SELECT ind_id ,ind_can_name ,mpt_trajectories(locations) AS trajectory
FROM moving_birds
WHERE ind_id='Belarus 4';

Result 5.

ind_id    |ind_can_name     | trajectory
----------+-----------------+----------------------------------------------------
Belarus 4|Cuculus canorus  |SRID=4326;MULTILINESTRING((27.780570 52.180970 0.000000
          |                 |,27.770520 52.182030 0.000000,27.766930 52.180850
          |                 | 0.000000, ..... ,23.957670 -10.141420 0.000000))
Query 7.

SELECT ind_id ,ind_can_name ,mpt_trajectories(locations) AS trajectory
FROM moving_birds
```

Figure 4.2 visualizes the results from the operation *mpt_trajectrory* on moving birds dataset, together with results from the operation *mpt_location*. This shows how the moving birds were moving from one location to another and where they have spent more time in their movement.

4.3 OPERATIONS FOR INTERACTION WITH DOMAIN/RANGE

This section lists and explains operations that interact with points and point sets in the domain and range of a moving point type. These types of operations relate the functional values of the moving point either with the time or the range; which is their registered location. For example, we can restrict a moving point for a given time interval. In addition, we can ask for a location of a moving point at a given time.

4.3.1 mpt_atinstant

The operation *mpt_atinstant* is similar to the timeslice operator found in different temporal relational algebras [Güting et al., 2000]. The operation restricts a moving point to a given instant to

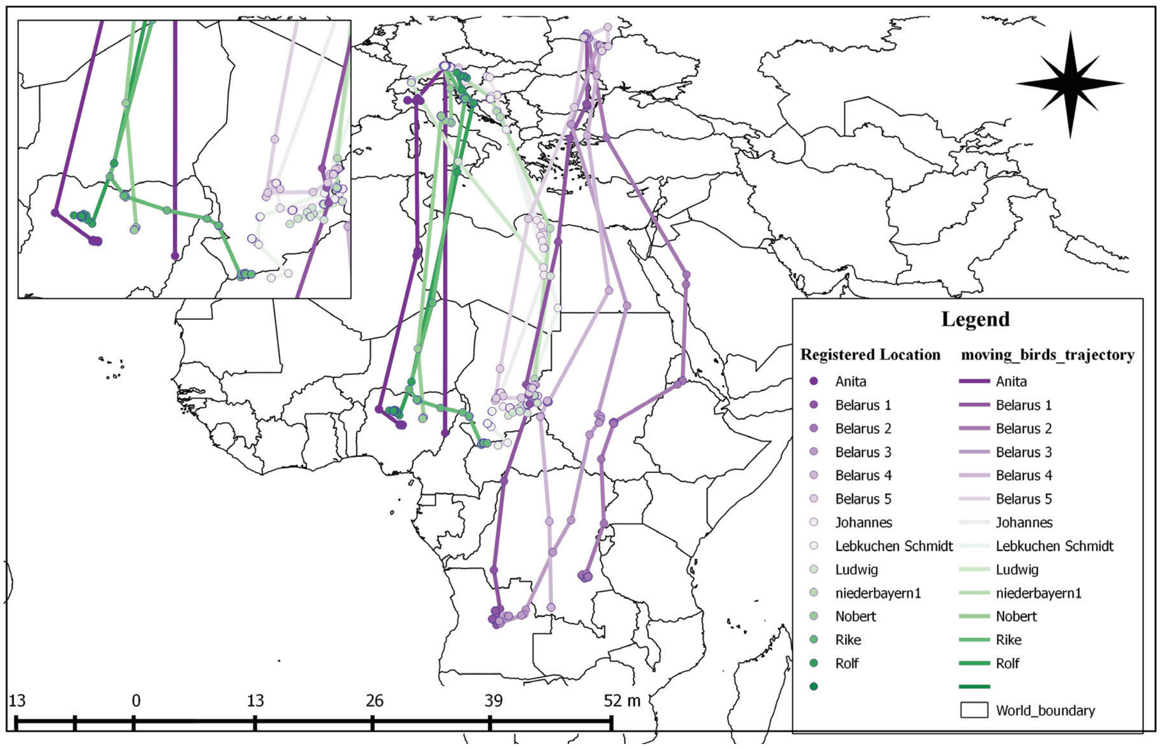


Figure 4.2: mpt_trajectory

return a value of the type INTIME_MPOINT. The operation takes a value of type *mpoint* and a timestamp as argument. The function checks the timestamps used for the moving point and the interpolation timestamp are in the same time zone. If the condition fails, it returns an error message. The function works under the assumption that the units in the moving point are chronologically ordered.

The function then searches for two consecutive registered locations, in which the timestamp provided as argument is within the interval constructed from the timestamps of the two consecutive locations. To do this, two lightweight point geometries are initialized. For each moving segment, two consecutive points are selected and assigned to the newly initialized point geometries to represent points on the left and right. If the timestamp provided is less than the timestamp used for point on the right and greater than point on the left, point on the left and right are used to interpolate the new location in between at a given time instant. Three different interpolation methods are implemented for this function. The selection of the interpolation function clearly depends on users' preference. Details on algorithms and mathematical formulas used to implement the interpolation functions are discussed in Chapter 5. The newly interpolated location together with the time instant provided as an argument, is returned as a value of the type INTIME_MPOINT. If the timestamp value provided as an argument is not within the moving object defined time, the operation return a *null* value. The algorithm used to implement this operation is presented in Algorithm 4.4.

In practice, this function is one of the most useful and practical operation of moving points because it satisfies the underlying principles for storing moving objects into the database. To store moving objects into the database, their movement must be sampled; and from that their location can be interpolated for those times we do not have a stored location. For example, in a taxi service application, the owner of the company may receive an accusation of a robbery from one of

Algorithm 4.4mpt_atinstant operation

INPUT TYPE: MPOINT,INSTANCE
OUTPUT TYPE: INTIME_MPOINT

```
1: $point\_left \leftarrow NULL$
2: $point\_right \leftarrow NULL$
3: $found \leftarrow$ **false**
4: $i \leftarrow 0$
5: **if** $inst.tz\_id == mpoint.tz\_id$ **then**
6:     $continue$
7: **else**
8:     $break$
9: **end if**
10: **while** $!finished \&\& i < mpoint.num\_sgts$ **do**
11:     **for** $j = 0$**to**$mpt\_sgts.num\_pts$ **do**
12:         $point\_left.x \leftarrow mpt\_segts[i].geom\_array[j].x$
13:         $point\_left.y \leftarrow mpt\_segts[i].geom\_array[j].y$
14:         $point\_left.z \leftarrow mpt\_segts[i].geom\_array[j].z$
15:         $point\_left.m \leftarrow mpt\_segts[i].geom\_array[j].m$
16:         $point\_right.x \leftarrow mpt\_segts[i].geom\_array[j+1].x$
17:         $point\_right.y \leftarrow mpt\_segts[i].geom\_array[j+1].y$
18:         $point\_right.z \leftarrow mpt\_segts[i].geom\_array[j+1].z$
19:         $point\_right.m \leftarrow mpt\_segts[i].geom\_array[j+1].m$
20:         **if** $point\_left.m < inst.val \&\& point\_right.m > inst.val$ **then**
21:             $intime\_mpoint.point \leftarrow INTERPOLATE(point\_left, point\_right)$
22:             $intime\_mpoint.t \leftarrow inst.t$
23:             **return** $intime\_mpoint$
24:         **else**
25:             $continue$
26:         **end if**
27:     **end for**
28:     $i \leftarrow i + 1$
29: **end while**
30: $intime\_mpoint.point \leftarrow NULL$
31: $intime\_mpoint.t \leftarrow NULL$
32: **return** $intime\_mpoint$
```

their customers, and the customers do not know which taxi was giving the service. However, the customer exactly knows where and when the robbery happened. Based on this information, it is possible to track the taxi which is responsible for the robbery. This can be done by performing the operation *mpt_atinstant* on the dataset of the moving taxis for the time the robbery took place. The result of the operation returns the interpolated locations, and this information can be used to identify the taxi which is closet to the location of the robbery. However, for birds, we can write a query to return the locations of all the moving birds at instant as it is presented in Listing 4.7.

Listing 4.7: `Where was the bird "Ludwig" on 2013-07-12 12:07:40?`

```
Query 8.

SELECT ind_id,ind_can_name,mpt_atinstant(locations,'2013-07-12 12:07:40 1') AS atinstant
FROM moving_birds
WHERE ind_id='Ludwig';

Result 6.

ind_id  |ind_can_name     | atinstant
--------+-----------------+----------------------------------------------------------------
Ludwig  |Cuculus canorus | INTIME_MPOINT(POINT Z (20.57992 11.409186 0),2013-07-12 12:07:40
     1)

Query 9.

SELECT ind_id,ind_can_name,mpt_atinstant(locations,'2013-07-12 12:07:40 1') AS atinstant
FROM moving_birds
```

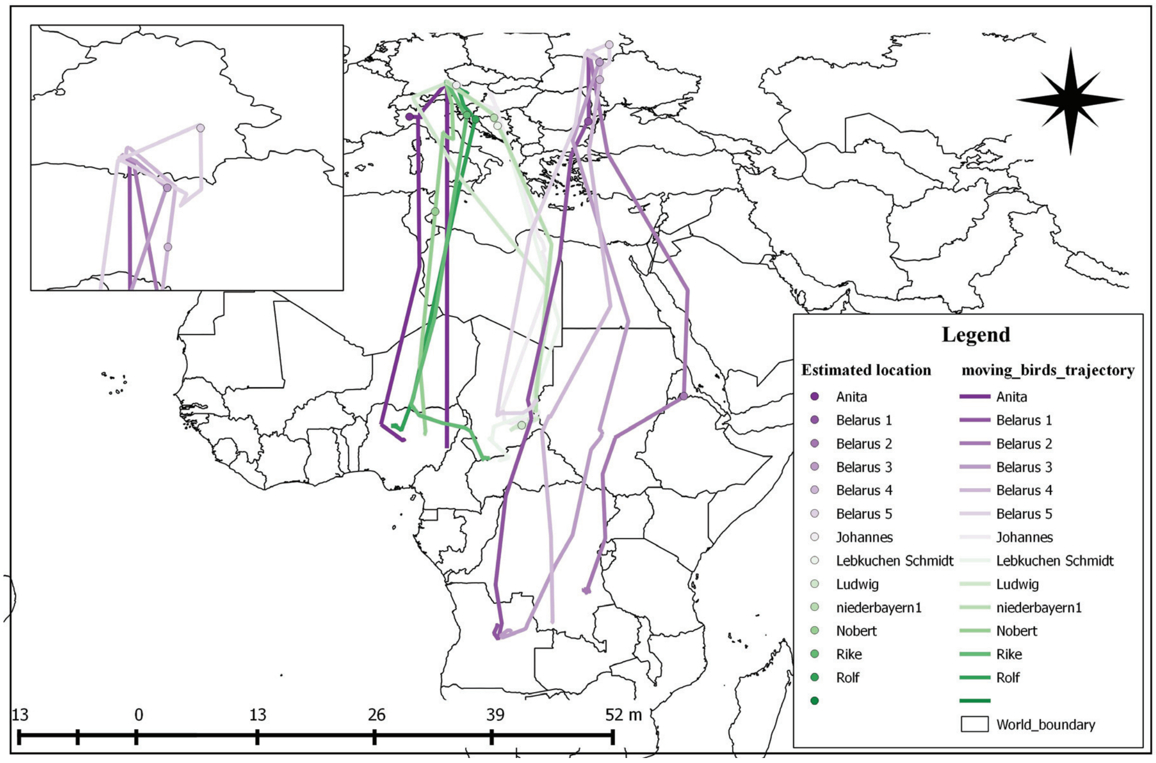


Figure 4.3: mpt_atinstant

The result of the query in Listing 4.7 is visualized in Figure 4.3 together with the trajectory of the moving birds. The location interpolated for each of the birds on the moving birds dataset is clearly located on the trajectory of the moving point. It is because that the operation *mpt_atinstant* uses linear interpolation in default. The visualization approves that the implemented operation returns correct interpolation result for interpolating locations between two points of a moving point. If cubic interpolation function is used, the locations returned from the operation *mpt_atinstant*, may not exactly be located on the trajectory of the moving bird. This and other related explanations are included in Chapter 5.

4.3.2 val_intime_mpoint

The operation *val_intime_mpoint* yields the point component of the type INTIME_MPOINT. The operation takes a values of the type INTIME_MPOINT and return a point geometry as a result. An *empty* geometry is returned to a *null* argument. This operation can be used to extract the newly interpolated location from the result of the operation *mpt_atinstant*. Therefore, the query in Listing 4.7 can be rewritten as in Listing 4.8, only to return the location of the bird "Ludwig" on 2013-07-12 12:07:40.

Listing 4.8: "Where was the bird "Ludwig" on 2013-07-12 12:07:40?

```
Query 10.

SELECT ind_id,ind_can_name,val_intime_mpoint(mpt_atinstant(locations,'2013-07-12 12:07:40
    1')) AS atinstant
FROM moving_birds
WHERE ind_id='Ludwig';

Result 7.

ind_id      |ind_can_name      | atinstant
------------+------------------+------------------------------------------------
Ludwig      |Cuculus canorus   | srid=4326;POINTZ(20.579920 11.409186 0.000000)
```

4.3.3 inst_intime_mpoint

The operation *inst_intime_mpoint* also yields the time component of the type INTIME_MPOINT. The operation returns a timestamp string value as a result. This can be used to extract the time component of the result from operations *mpt_atinstant*, *mpt_initial* and *mpt_final*. An emphnull timestring is returned to a *null* argument. The query statement in Listing 4.7 can be rewritten to return the time component of the final location for the moving bird 'Johannes' as it is presented in Listing 4.9.

Listing 4.9: When did the bird "Johannes" end the movement?

```
Query 11.

SELECT ind_id,ind_can_name,inst_intime_mpoint(mpt_final(locations)) AS final
FROM moving_birds
WHERE ind_id='Johannes';

Result 8.

ind_id          |ind_can_name       | final
----------------+-------------------+------------------------------------------
Johannes        |Cuculus canorus    |2013-09-19 05:46:31
```

4.3.4 mpt_present_atinstant

The function *mpt_atinstant* can be extended to check only if a moving point is defined for a specified time instant or not. For this purpose, the operation is extended to *mpt_present_instant*. The function accepts a value of the type *mpoint* and time instant as argument. The operation follows the same procedure as its parent operation *mpt_atinstant*, but instead of returning a value of the type INTIME_MPOINT, it returns a *boolean* value. If there exists two consecutive sampled positions of segments in a moving point and the timestamp provided as argument is within the time interval of the sampled postions, the operation returns *true* as a result. Otherwise, it returns *false*. The algorithm used to implement the function is provided in Algorithm 4.5.

Algorithm 4.5`mpt_present_atinstant operation`

INPUT TYPE: MPOINT,INSTANCE
OUTPUT TYPE: BOOL

```
1: $point\_left \leftarrow NULL$
2: $point\_right \leftarrow NULL$
3: $found \leftarrow$ false
4: $i \leftarrow 0$
5: if $inst.tz\_id == mpoint.tz\_id$ then
6:    $continue$
7: else
8:    $break$
9: end if
10: while $!finished\&\&i < mpoint.num\_sgts$ do
11:    for $j = 0$to$mpt\_sgts.num\_pts$ do
12:       $point\_left.x \leftarrow mpt\_segts[i].geom\_array[j].x$
13:       $point\_left.y \leftarrow mpt\_segts[i].geom\_array[j].y$
14:       $point\_left.z \leftarrow mpt\_segts[i].geom\_array[j].z$
15:       $point\_left.m \leftarrow mpt\_segts[i].geom\_array[j].m$
16:       $point\_right.x \leftarrow mpt\_segts[i].geom\_array[j+1].x$
17:       $point\_right.y \leftarrow mpt\_segts[i].geom\_array[j+1].y$
18:       $point\_right.z \leftarrow mpt\_segts[i].geom\_array[j+1].z$
19:       $point\_right.m \leftarrow mpt\_segts[i].geom\_array[j+1].m$
20:       if $point\_left.m < inst.val\&\&point\_right.m > inst.val$ then
21:          return true
22:       else
23:          $continue$
24:       end if
25:    end for
26:    $i \leftarrow i + 1$
27: end while
28: return false
```

To know if a moving object was in movement for the time of interest, the query statement from Listing 4.7 can be modified as of the query in Listing 4.10. The query allows us to check if a moving bird is defined for the given timestamp.

Listing 4.10: was "Ludwig" bird in movement on 2013-07-12 12:07:40?

```
Query 12.

SELECT ind_id,ind_can_name,mpt_presnt_atinstant(locations,'2013-07-12 12:07:40 1') AS
     present_atinstant
FROM moving_birds
WHERE ind_id='Ludwig';

 Result 9.

 ind_id |ind_can_name      | present_atinstant
--------+------------------+-----------------------------------------------
Ludwig  |Cuculus canorus   | t
```

4.3.5 mpt_atperiod

The operation *mpt_atperiod* restricts the moving point to a specified set of time interval. This operation accepts a value of type MPOINT and a period as an argument; and returns a new moving point as a result. The operation checks if the time zone identifier used for the input period and moving point is equal. Next to that, a new moving point is initialized. The registered timestamp from the (location-time) pairs of each segment of the moving point is used to restrict the moving point to the time interval provided as an argument. If the registered timestamp value for the (location-time) pair of a moving segment is comprised between the initial and the final timestamp value of the argument period, the (location-time) pair value is used to construct the segment of the new moving point. The same thing also happens if they have the same value. If the moving point has no registered location, the operation returns a *null* value of the type MPOINT. Algorithm 4.6 shows the the algorithm used to implement this operation.

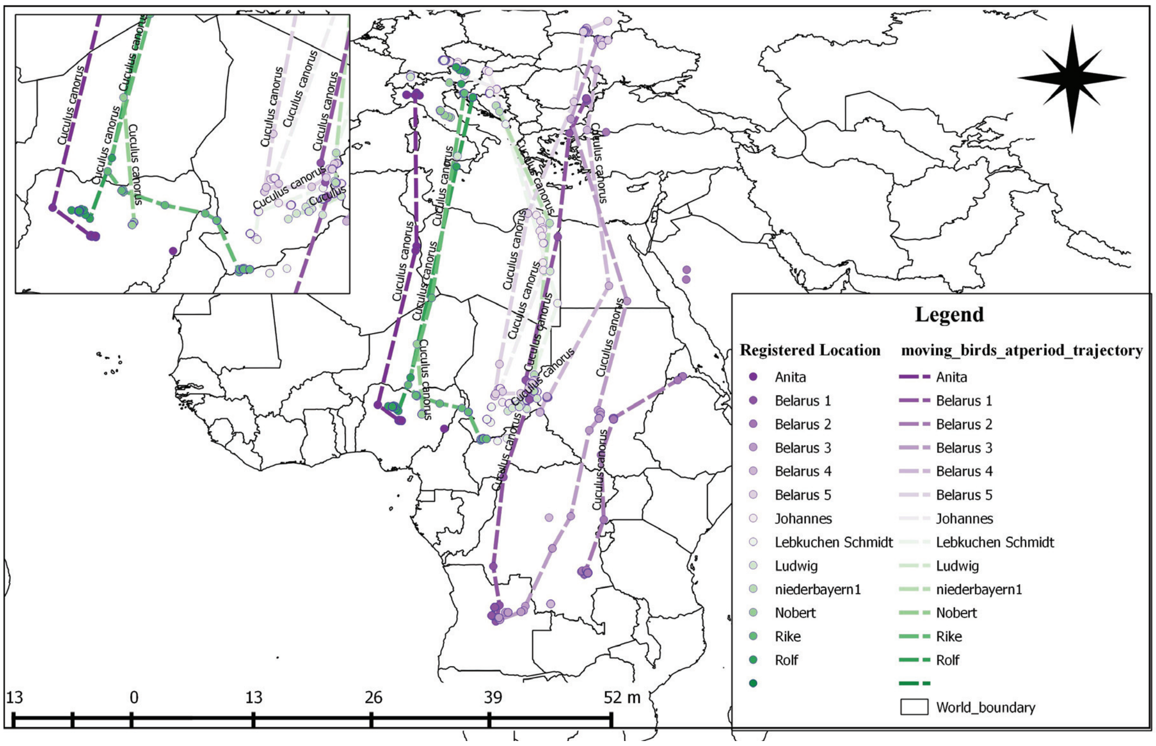


Figure 4.4: mpt_atperiod

Algorithm 4.6`mpt_atperiod operation`

INPUT TYPE: MPOINT,PERIOD
OUTPUT TYPE: MPOINT

```
1:  $new\_mpoint \leftarrow NULL$
2:  $point \leftarrow NULL$
3:  $new\_num\_sgts \leftarrow 0$
4:  $new\_num\_pts \leftarrow 0$
5:  $found \leftarrow$ false
6:  if $period.tz\_id == mpoint.tz\_id$ then
7:     $continue$
8:  else
9:     $break$
10: end if
11: for $i = 0$to$mpoint.num\_sgts$ do
12:    for $j = 0$to$mpt\_sgts.num\_pts$ do
13:       $point.x \leftarrow mpt\_segts[i].geom\_array[j].x$
14:       $point.y \leftarrow mpt\_segts[i].geom\_array[j].y$
15:       $point.z \leftarrow mpt\_segts[i].geom\_array[j].z$
16:       $point.m \leftarrow mpt\_segts[i].geom\_array[j].m$
17:       if $period.initial <= point.m <= period.final$ then
18:          $new\_mpoint.mpt\_segts[new\_num\_sgts].geom\_array[new\_num\_pts] \leftarrow point$
19:          $new\_num\_pts \leftarrow new\_num\_pts + 1$
20:          $found \leftarrow$ true
21:       else
22:          $continue$
23:       end if
24:    end for
25:    if $found ==$ true then
26:       $new\_mpoint.mpt\_segts[new\_num\_sgts].num\_pts \leftarrow new\_num\_pts$
27:       $new\_num\_pts \leftarrow 0$
28:       $new\_num\_sgts \leftarrow new\_num\_sgts + 1$
29:       $found \leftarrow$ false
30:    else
31:       $continue$
32:    end if
33: end for
34: $new\_mpoint.srid \leftarrow mpoint.srid$
35: $new\_mpoint.tzid \leftarrow mpoint.tzid$
36: $new\_mpoint.num\_sgts \leftarrow mpoint.num\_sgts$
37: return $new\_mpoint$
```

This operation can benefit different application in different ways. For example, we might be interested to study the effect of seasonal variations on moving birds. Total distance covered, the average speed and other questions can be asked for different seasons of the year. To do so, the moving point that represents a moving bird must be restricted to the four different calendar seasons of the year. And this can be achieved using the operation *mpt_atperiod*. When we apply this operation on a moving point that represent a moving bird, the result is four different moving points, each one representing a season of the year. The query statement in Listing 4.11 restricts the moving bird into a certain specified time interval (period).

Listing 4.11: Find bird "Anita" between 2013-07-12 12:07:40,2013-09-12 12:07:40?

```
Query 13.

SELECT ind_id ,ind_can_name ,mpt_atperiod(locations ,‘2013−07−12 12:07:40,2013−09−12
     12:07:40 ,1 ’) AS atperiod
FROM moving_birds
WHERE ind_id=’Anita ’;

Result 10.

ind_id  |ind_can_name      | atperiod
--------+------------------+---------------------------------------------------------
Anita   |Cuculus canorus   |srid=4326;tzid=1;MPOINT((8.332760 45.184050 0.000000
        |                  |2013−07−14 05:47:03 ,9.261420 45.134590 0.000000 2013−07−16
        |                  |16:58:34, .. ,7.727330 9.781080 0.000000 2013−09−12 02:31:44))
Query 14.

SELECT ind_id ,ind_can_name ,mpt_atperiod(locations ,‘2013−07−12 12:07:40,2013−09−12
     12:07:40 ,1 ’) AS atperiod
FROM moving_birds
```

The query statement in Listing 4.11 returns all the registered locations of the moving bird "Anita" for the time interval provided as argument. If there exists any undefined moment of the moving object between the time interval, the operation creates a new segment for the moving point which is returned as a result. The result of query 14 in Listing 4.11 is visualized together with the result from operation *mpt_locations* in Figure 4.4. From the visualization, we can observe that there are locations that are not placed on the trajectory of the result from the operation *mpt_atperiod*. This infers that there are (location-time) pairs that are not defined for the input time interval.

4.3.6 mpt_present_atperiod

The operation *mpt_present_atperiod* allows to check whether a moving point is ever present within the time interval provided as argument. The interval is a value of the type *MPERIOD*. The function follows similar principles with the operation *mpt_atperiod*. The major difference is that, if the operation *mpt_present_atperiod* find a single registered location within the time interval, it then automatically returns a *true* value of the type Boolean. If it fails to find a registered location, it returns a *false* value as a result. Algorithm 4.7 shows the algorithm used for the implementation of this function.

For example, we can query for birds that were present during the forest fire that has lasted for three months in 2012. This and other related questions can be queried using the operation *mpt_present_atperiod*. Listing 4.12 shows the query statement to check if a moving bird was ever present between the time interval ‘2013-07-12 12:07:40,2013-09-12 12:07:40,1’.

Algorithm 4.7`mpt_present_atperiod operation`

INPUT TYPE: MPOINT,PERIOD
OUTPUT TYPE: BOOL

```
$new\_mpoint \leftarrow NULL$
$point \leftarrow NULL$
$new\_num\_sgts \leftarrow 0$
$new\_num\_pts \leftarrow 0$
$found \leftarrow$ false
$present \leftarrow$ false
if $period.tz\_id == mpoint.tz\_id$ then
   $continue$
else
   $break$
end if
for $i = 0$to$mpoint.num\_sgts$ do
   for $j = 0$to$mpt\_sgts.num\_pts$ do
      $point.x \leftarrow mpt\_segts[i].geom\_array[j].x$
      $point.y \leftarrow mpt\_segts[i].geom\_array[j].y$
      $point.z \leftarrow mpt\_segts[i].geom\_array[j].z$
      $point.m \leftarrow mpt\_segts[i].geom\_array[j].m$
      if $period.initial <= point.m <= period.final$ then
         $new\_mpoint.mpt\_segts[new\_num\_sgts].geom\_array[new\_num\_pts] \leftarrow point$
         $new\_num\_pts \leftarrow new\_num\_pts + 1$
         $found \leftarrow$ true
         if $present! =$ true then
            $present \leftarrow$ true
         else
            $continue$
         end if
      else
         $continue$
      end if
   end for
   if $found ==$ true then
      $new\_mpoint.mpt\_segts[new\_num\_sgts].num\_pts \leftarrow new\_num\_pts$
      $new\_num\_pts \leftarrow 0$
      $new\_num\_sgts \leftarrow new\_num\_sgts + 1$
      $found \leftarrow$ false
   else
      $continue$
   end if
end for
if $present ==$ true then
   return true
else
   return false
end if
```

Listing 4.12: Is the bird "Anita" was ever present between the dates 2013-07-12 12:07:40,2013-09-12 12:07:40?

```
Query 15.

SELECT ind_id, ind_can_name, mpt_present_atperiod(locations,'2013-07-12 12:07:40,2013-09-12
    12:07:40,1') AS present_period
FROM moving_birds
WHERE ind_id=Anita;

Result 11.

ind_id |ind_can_name      | present_atperiod
-------+------------------+------------------------------------------------
Anita  |Cuculus canorus   | t
```

4.3.7 mpt_initial

The operation *mpt_initial* return the first registered location of a moving point together with the time value as a type of INTIME_MPOINT. If a given moving point has no registered location, *null* value of the type INTIME_MPOINT is returned. The algorithm used to implement this operation is shown in Algorithm 4.8.

Algorithm 4.8mpt_initial operation

INPUT TYPE: MPOINT
OUTPUT TYPE: INTIME_MPOINT

```
1: if $mpoint.num\_sgts == 0$ then
2:     $intime\_mpoint.point \leftarrow NULL$
3:     $intime\_mpoint.t \leftarrow NULL$
4:     return $intime\_mpoint$
5: else if $mpt\_sgts.num\_pts == 0$ then
6:     $intime\_mpoint.point \leftarrow NULL$
7:     $intime\_mpoint.t \leftarrow NULL$
8:     return $intime\_mpoint$
9: else
10:    $point.x \leftarrow mpt\_segts[0].geom\_array[0].x$
11:    $point.y \leftarrow mpt\_segts[0].geom\_array[0].y$
12:    $point.z \leftarrow mpt\_segts[0].geom\_array[0].z$
13:    $intime\_mpoint.point \leftarrow point$
14:    $intime\_mpoint.t \leftarrow mpt\_segts[0].geom\_array[0].m$
15:    return $intime\_mpoint$
16: end if
```

This operation can be used to ask questions that involve the moving objects beginning. For example, we might want to request where and when an object starts moving. The operation does not return a separate result for questions of the form 'where' and 'when'. However, the result of the operation *mpt_initial* can be used as an input for other operations like *val_intime_mpoint* and *inst_intime_mpoint*. The query statement in Listing 4.13 return the initial registered location of the bird "Johannes" together with its time value.

Listing 4.13: `When and where the bird "Johannes" start moving?`

```
Query 16.

SELECT ind_id,ind_can_name,mpt_initial(locations) AS initial
FROM moving_birds
WHERE ind_id='Johannes';

 Result 12.

 ind_id   |ind_can_name     | initial
----------+-----------------+------------------------------------------------
 Johannes|Cuculus canorus  |INTIME_MPOINT(POINTZ(12.36633 48.91536 0),2013-04-27 07:54:25
     4326)
```

4.3.8 mpt_final

The operation *mpt_final* returns the last registered location of a moving point together with the timestamp as a type of INTIME_MPOINT. In the beginning, the function checks whether a moving point is *empty* or not. If the moving point does not have any registered location, *null* value of the type INTIME_MPOINT is returned as a result. If a moving point only contains one registered location, the operation returns a similar value as the operation *mpt_initial*. The algorithm used to implement this operation is presented in Algorithm 4.9.

Algorithm 4.9`mpt_final final`

INPUT TYPE: MPOINT
OUTPUT TYPE: INTIME_MPOINT

```
 1: $last\_seg \leftarrow 0$
 2: $last\_pt \leftarrow 0$
 3: if $mpoint.num\_sgts == 0$ then
 4:     $intime\_mpoint.point \leftarrow NULL$
 5:     $intime\_mpoint.t \leftarrow NULL$
 6:     return $intime\_mpoint$
 7: else if $mpt\_sgts.num\_pts == 0$ then
 8:     $intime\_mpoint.point \leftarrow NULL$
 9:     $intime\_mpoint.t \leftarrow NULL$
10:     return $intime\_mpoint$
11: else
12:     $point.x \leftarrow mpt\_segts[mpoint.num\_sgts - 1].geom\_array[mpt\_sgts.num\_pts - 1].x$
13:     $point.y \leftarrow mpt\_segts[mpoint.num\_sgts - 1].geom\_array[mpt\_sgts.num\_pts - 1].y$
14:     $point.z \leftarrow mpt\_segts[mpoint.num\_sgts - 1].geom\_array[mpt\_sgts.num\_pts - 1].z$
15:     $last\_seg \leftarrow mpoint.num\_sgts - 1$
16:     $last\_pt \leftarrow mpt\_sgts.num\_pts - 1$
17:     $intime\_mpoint.point \leftarrow point$
18:     $intime\_mpoint.t \leftarrow mpt\_segts[last].geom\_array[last_pt].m$
19:     return $intime\_mpoint$
20: end if
```

The query statment in Listing 4.14 return the last known position of the bird “Johannes” together with its time value.

Listing 4.14: Where is the last know location of bird "Johannes"?

```
Query 17.

SELECT ind_id,ind_can_name,mpt_final(locations) AS final
FROM moving_birds
WHERE ind_id='Johannes';

Result 13.

ind_id   |ind_can_name     | final
---------+-----------------+--------------------------------------------------------------
Johannes|Cuculus canorus  |INTIME_MPOINT(POINTZ(22.47778 12.9396 0),2013-09-19 05:46:31
    4326)
```

4.3.9 mpt_srid

The operation *mpt_srid* return the spatial reference system identifier of the point geometries composing the moving point. For any moving point registered with invalid or no spatial reference system, the error message "UNKNOWN_SRID" is returned as a result. Listing 4.15 shows the query applied to get the spatial reference system used for the moving bird "Rolf".

Listing 4.15: Find the SRID for the bird "Rolf"?

```
Query 18.

SELECT ind_id,ind_can_name,mpt_srid(locations) AS srid
FROM moving_birds
WHERE ind_id='Rolf';

Result 14.

ind_id      |ind_can_name     | srid
------------+-----------------+-----------------
Rolf        |Cuculus canorus  | 4326
```

4.3.10 mpt_tzid

The operation *mpt_tzid* returns the time zone reference identifier used for the moving point. If no or invalid time zone reference identifier is provided, the function returns an error message of "UNKNOWN_TZID". Listing 4.16 shows the query to get the time zone identifier used for the moving bird "Rolf".

Listing 4.16: Find the TZID for the bird "Rolf"?

```
Query 19.

SELECT ind_id,ind_can_name,mpt_tzid(locations) AS tzid
FROM moving_birds
WHERE ind_id='Rolf';

Result 15.

ind_id      |ind_can_name     | tzid
------------+-----------------+----------------------------------------------------
Rolf        |Cuculus canorus  | 1
```

4.4 OPERATIONS FOR RATE OF CHANGE

Rate of change is one important concept for any time-dependent moving object. Any operation related to the rate of change can be applied to any of time dependent types. However, the range value of these types must support a difference operation and division by real values. For a moving point type, at least three operations can come in mind: the Euclidian distance, the direction between two points and the vector difference. This gives us functions *mpt_distance*, *mpt_direction*, and *mpt_speed*, respectively. The rate of change for moving point is represented by the velocity of the object as a vector value. This vector value provides the speed and direction of the movement. The velocity of a moving point is defined by the velocity in both the X and Y coordinates separately.

4.4.1 mpt_distance

The operation *mpt_distance* returns the distance covered between two consecutive registered locations of a moving point as moving real value. The operation accepts a value of the type MPOINT as an argument and returns the distance covered as a value of the type MREAL. The distance between two points is the euclidean distance in a two dimensional space, which is different from the distance between two geodetic points. The formula to compute the distance between two geodetic points is also quite different from the corresponding euclidean distance. The formula used for distance computation between Cartesian and geodetic points is shown in Equation 4.1. Since the time type is isomorphic to the domain of real numbers, it inherit their arithmetic operation.

$$distance(Cartesian) \leftarrow \sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2 + (z_1 - z_0)^2} \tag{4.1}$$

PostGIS provides two alternatives to compute the distance between two point locations. The first one is using Cartesian mathematics and straight line vector. The other one is using "geographic" coordinates. Geographic coordinates are spherical coordinates expressed in angular units (degrees). The major constraint for geographic coordinates is that they only works for WGS 84 long lat (SRID=4321). Over larger areas, the spheroidal calculation is more accurate than any calculation done on a projected plane.

The function first initializes a variable to hold for the total distance covered by the moving point. In the beginning, this value is initialized to zero, considering the first registered location as a starting position of the moving point. A new moving real value is also initialized. The function iteratively computes the distance between two consecutive points of each segment and the result is then summed with the total distance value covered by the moving object. For each iteration starting from the beginning of the moving point, the newly initialized moving real value is constructed from the value of the total distance covered and the time component of each points registered for the moving point. The function returns a multiple moving real value that is equal to the number of segments of the moving point. The algorithm used to implement this operation is provided in Algorithm 4.10.

In practice, the operation is useful to perform different analysis on the rate of change of the moving object. If we are interested to know the total traveled distance of the moving object, we can use this operation together with the operation *mreal_final* and *val_intime_mreal*. A query statement that requests for the distance covered by the moving bird "Belarus 3" throughout the movement is provided in Listing 4.17.

Algorithm 4.10`mpt_distance operation`

INPUT TYPE: MPOINT
OUTPUT TYPE: MREAL

```
1: $mreal \leftarrow NULL$
2: $distance\_initial \leftarrow 0.0$
3: $distance\_final \leftarrow 0.0$
4: $distance \leftarrow 0.0$
5: $spheroid \leftarrow NULL$
6: $t \leftarrow NULL$
7: for $i = 0$to$mpoint.num\_sgts$ do
8:     for $j = 0$to$mpt\_sgts.num\_pts$ do
9:         $point\_left \leftarrow NULL$
10:        $point\_right \leftarrow NULL$
11:        if $j == 0$ then
12:            $t \leftarrow mpt\_segts[i].geom\_array[j].m$
13:        else
14:            $t \leftarrow mpt\_segts[i].geom\_array[j].m$
15:            $point\_left.x \leftarrow mpt\_segts[i].geom\_array[j].x$
16:            $point\_left.y \leftarrow mpt\_segts[i].geom\_array[j].y$
17:            $point\_left.z \leftarrow mpt\_segts[i].geom\_array[j].z$
18:            $point\_right.x \leftarrow mpt\_segts[i].geom\_array[j+1].x$
19:            $point\_right.y \leftarrow mpt\_segts[i].geom\_array[j+1].y$
20:            $point\_right.z \leftarrow mpt\_segts[i].geom\_array[j+1].z$
21:            $t \leftarrow mpt\_segts[i].geom\_array[j+1].m$
22:            $initialize\_spheroid\_from\_sridmpt.sr\_id, \&s$
23:            $distance \leftarrow distance\_spheriod(point\_left, point\_right, \&s)$
24:            $dist\_final \leftarrow distance + dist\_intial;$
25:            $dist\_intial \leftarrow dist\_final;$
26:        end if
27:        $mreal[i].instant[j].val \leftarrow distance\_final$
28:        $mreal[i].instant[j].t \leftarrow t$
29:    end for
30:    $dist\_initial \leftarrow 0.0$
31:    $t \leftarrow NULL$
32: end for
33: return $mreal$
```

Listing 4.17: find the distance covered by the bird "Belarus 3"?

```
Query 20.

SELECT ind_id,ind_can_name,mpt_distance(locations) AS Distance
FROM moving_birds
WHERE ind_id='Belarus 3';

 Result 16.

 Ind_id     |ind_can_name     | Distance
------------+-----------------+------------------------------------------------------
 Belarus 3  |Cuculus canorus  |tzid=1;MMREAL((0.000000 2013-04-30 19:59:35,1865.612669
            |                 |2013-05-03 04:14:09,...,8025367.143174 2013-09-15 14:42:36))
```

4.4.2 mpt_speed

The operation *mpt_speed* takes a value of the type MPOINT and returns a moving real value as a result. This operation returns the numeric speed of the moving point from the discrete locations used to represent the moving point. The resulting moving real value consists of timely ordered (value, instant) pairs. The speed of the moving object between two measured locations is constant. The algorithm used to implement this operation is highly similar to the operation *mpt_distance* and its presented in Algorithm 4.11.

The result of the operation *mpt_distance* is a prerequisite to calculate the speed of the moving object as a moving real value. Since each point unit describes linear movement, distance between two consecutive point locations per their time difference gives us the speed of the moving point. The operation is much more meaningful and practically useful if they are operated on moving points with SRID of 4326 (geographic or geodetic coordinates). The operation *mpt_distance* returns the distance value in a unit of meter and the result of the operation *mpt_speed* is meters per second. Query statement in Listing 4.18 returns the speed of the bird “niederbayern1”.

Listing 4.18: What is the speed of the bird "niederbayern1" throughout the movement?

```
Query 21.

SELECT ind_id,ind_can_name,mpt_speed(locations) AS Speed
FROM moving_birds
WHERE ind_id='niederbayern1';

 Result 17.

 ind_id        |ind_can_name    | Speed
---------------+----------------+------------------------------------------------------
 niederbayern1 |Cuculus canorus |tzid=1;MMREAL((0.000000 2013-06-01 16:54:45,0.017209
               |                |2013-06-04 00:27:42,...,0.018222 2013-09-18 03:50:02,
               |                |0.129754 2013-09-18 09:33:15))
```

4.4.3 mreal_initial

The operation *mreal_initial* accepts a value of the type MREAL and return the first value-time pair as a type of INTIME_MREAL. A *null* value is returned for moving real values that have no value-time pairs. The algorithm used for this operation is highly similar as of the algorithm used for the operation *mpt_initial*. In practice, this operation can also be used to return the first value-time pair of the result from operation *mpt_speed* or *mpt_distance*. Listing 4.19 combines the operation *mpt_distance* and *mreal_initial* to return the initial registered distance value for the bird “Belarus 3”.

Algorithm 4.11mpt_speed operation

INPUT TYPE: MPOINT
OUTPUT TYPE: MREAL

```
1: $mreal \leftarrow NULL$
2: $distance\_initial \leftarrow 0.0$
3: $distance\_final \leftarrow 0.0$
4: $distance \leftarrow 0.0$
5: $spheroid \leftarrow NULL$
6: $time\_duration \leftarrow 0$
7: $t\_left \leftarrow NULL$
8: $t\_right \leftarrow NULL$
9: $speed \leftarrow 0.0$
10: for $i = 0$ to $mpoint.num\_sgts$ do
11:     for $j = 0$ to $mpt\_sgts.num\_pts$ do
12:         $point\_left \leftarrow NULL$
13:         $point\_right \leftarrow NULL$
14:         if $j == 0$ then
15:             $t\_right \leftarrow mpt\_segts[i].geom\_array[j].m$
16:         else
17:             $t \leftarrow mpt\_segts[i].geom\_array[j].m$
18:             $point\_left.x \leftarrow mpt\_segts[i].geom\_array[j].x$
19:             $point\_left.y \leftarrow mpt\_segts[i].geom\_array[j].y$
20:             $point\_left.z \leftarrow mpt\_segts[i].geom\_array[j].z$
21:             $t\_left \leftarrow mpt\_segts[i].geom\_array[j].m$
22:             $point\_right.x \leftarrow mpt\_segts[i].geom\_array[j+1].x$
23:             $point\_right.y \leftarrow mpt\_segts[i].geom\_array[j+1].y$
24:             $point\_right.z \leftarrow mpt\_segts[i].geom\_array[j+1].z$
25:             $t\_right \leftarrow mpt\_segts[i].geom\_array[j+1].m$
26:             $initialize\_spheroid\_from\_srid(mpt.sr\_id, \&s)$
27:             $distance \leftarrow distance\_spheriod(point\_left, point\_right, \&s)$
28:             $dist\_final \leftarrow distance + dist\_intial;$
29:             $dist\_intial \leftarrow dist\_final;$
30:             $time\_duration \leftarrow t\_right - t\_left$
31:             $speed \leftarrow distance/time\_duration$
32:         end if
33:         $mreal[i].instant[j].val \leftarrow speed$
34:         $mreal[i].instant[j].t \leftarrow t\_right$
35:     end for
36:     $dist\_initial \leftarrow 0.0$
37:     $t \leftarrow NULL$
38: end for
39: return $mreal$
```

Listing 4.19: find the initial distance value registered for the bird "Belarus 3"?

```
Query 22.

SELECT ind_id,ind_can_name,mreal_initial(mpt_distance(locations)) AS Distance
FROM moving_birds
WHERE ind_id='Belarus 3';

Result 18.

ind_id     |ind_can_name     | Distance
-----------+-----------------+------------------------------------------
Belarus 3  |Cuculus canorus  | INTIME_MREAL(0.000000 2013-04-30 19:59:35)
```

4.4.4 mreal_final

The operation *mreal_final* takes a value of the type MMREAL as argument and return the last (value-time) pair as a value of the type INTIME_MREAL. The algorithm used to implement the operation is similar to the algorithm used for the operation *mtp_initial*. The operation can be used to retrieve the last known value-time pair from a set of timely ordered value-time pairs of the type MMREAL. For example, the operation can be used to request the total distance covered by a moving object, from the results of operation *mpt_distance*. The query statement in Listing 4.20 returns the total distance covered by the bird “Belarus 3”.

Listing 4.20: what is the total distance covered by the bird "Belarus 3"?

```
Query 23.

SELECT ind_id,ind_can_name,mreal_final(mpt_distance(locations)) AS Distance
FROM moving_birds
WHERE ind_id='Belarus 3';

Result 19.

ind_id     |ind_can_name     | Distance
-----------+-----------------+------------------------------------------
 Belarus 3 |Cuculus canorus  | INTIME_MREAL(8025367.143174 2013-09-15 14:42:36)
```

4.4.5 val_intime_mreal

The *val_intime_mreal* operation takes a value of the type INTIME_MREAL as argument and return the value component separately. This operation can be used with operations that returns rate of change of a moving object. For example, query statement in Listing 4.21 returns the total distance covered by the bird “Belarus 3” without the time component.

Listing 4.21: what is the total distance covered by the bird "Belarus 3"?

```
Query 24.

SELECT ind_id,ind_can_name,val_intime_mreal(mreal_final(mpt_distance(locations))) AS
    Distance
FROM moving_birds
WHERE ind_id='Belarus 3';

Result 20.

ind_id     |ind_can_name     | Distance
-----------+-----------------+------------------------------------------
Belarus 3  |Cuculus canorus  | 8025367.143174
```

4.4.6 inst_intime_mreal

The operation *inst_intime_mreal* takes a value of the type INTIME_MREAL and return the time component as a timestamp string. This operation can be used to extract the time component of the results from operations *mreal_initial* and *mreal_final*. The query statement in Listing 4.20 is modified to return the time component for the moving bird Belarus finishs its movement only as shown in Listin 4.22.

Listing 4.22: When did the total distance for the "Belarus 3" was registered?

```
Query 25.

SELECT ind_id,ind_can_name,inst_intime_mreal(mreal_final(mpt_distance(locations))) AS
    Distance
FROM moving_birds
WHERE ind_id='Belarus 3';

Result 21.

ind_id      |ind_can_name     | Distance
------------+-----------------+------------------------------------------
Belarus 3   |Cuculus canorus  | 2013-09-15 14:42:36
```

4.5 VALIDATION AND TRANSFORMATION OPERATIONS

4.5.1 mpt_valid

The operation *mpt_valid* takes a pointer of the type MPOINT and return a Boolean value. The operation returns *true* if a moving point passes all the validation constraints set in Section 2.5.1. In general, this operation makes sure points of a segment and segments of a moving point are chronologically ordered. In addition, two consecutive segments should not overlap in time. This operation is significant because most operations returns invalid result if any of the conditions defined in Section 2.5.1 fails to be satisfied. For example, operation *mpt_atinstant* for cubic interpolation expects points of the moving point to be ordered chronologically. The query statement in Listing 4.23 returns *true* if the moving bird “Johannes” and “Rolf” are well formed (valid).

Listing 4.23: Are "Johannes" and "Rolf" moving birds are well formed?

```
Query 26.

SELECT ind_id,ind_can_name,mpt_valid(locations) AS mpt_valid
FROM moving_birds
WHERE ind_id='Johannes' or ind_id='Rolf';

Result 22.

ind_id       |ind_can_name     | mpt_valid
-------------+-----------------+------------------------------------------
Johannes     |Cuculus canorus  | t
Rolf         |Cuculus canorus  | f
```

4.5.2 mpoint_mreal_valid

Section 2.5.2 discusses that moving real type is basically designed to store additional supporting information of any moving point. The operation *mpoint_mreal_valid* takes a value of the type MPOINT and MREAL as argument and returns a Boolean value as a result. For this operation to return *true*, both the input arguments must be valid and symmetric. Symmetric in this case refers that the number of segments of the moving point and the number of moving reals are equal. Similarly the number of points in a segment and the number of (value-time) pairs of the corresponding moving real must be equal. In addition, the timestamp of a point in a segment are equal to the corresponding timestamp of the moving real type. A query statement in Listing 4.24 checks whether the representation of moving birds "Nobert" and "Rike" is valid or not. This includes to check if the associated distance value is well formed and symmetric with the location measured for the moving birds.

Listing 4.24: `Is the distance value measurement taken on different location for birds "Nobert"and "Rolf" is well formed?`

```
Query 27.

SELECT ind_id,ind_can_name,mpt_mreal_valid(locations,distance) AS mpt_mreal_valid
FROM moving_birds
WHERE ind_id='Rike' or ind_id='Nobert';

Result 23.

ind_id     |ind_can_name         | mpt_mreal_valid
-----------+---------------------+--------------------------------------------
Nobert     |Cuculus canorus      | f
Rike       |Cuculus canorus      | t
```

4.5.3 mpt_transform

The operation *mpt_transform* accepts a moving point and a spatial reference identifier (SRID) as argument and returns a moving point as a result. Selection of projection to transform the spherical world onto a Cartesian coordinate system depends on how we are planning to use the data. In this situation, the operation *mpt_transform* function allows us to transform between spatial references systems used by the moving point types. Each point coordinate in the moving point is transformed and used to instantiate a new moving point type with the transformed point coordinates. Finally this result is returned to the user. This operation does not cause any change on the input moving point. The query statements in Listing 4.25 returns the original moving point and the newly transformed moving point for the bird "Nobert".

Listing 4.25: `Transform the moving bird "Nobert" from 4326 to 3857`

```
Query 28.

SELECT ind_id,ind_can_name,locations AS original_mpoint
FROM moving_birds
WHERE ind_id='Nobert';

Result 24.

ind_id |ind_can_name      | original_mpoint
-------+------------------+-------------------------------------------------
Nobert |Cuculus canorus   | srid=4326;tzid=1;MPOINT((12.285910 48.985900 0.000000
       |                  | 2013-04-26 07:01:10 , ... ,9.948900 10.455110 0.000000 2013-
       |                  |08-16 06:38:48))

Query 29.

SELECT ind_id,ind_can_name,mpt_transform(locations,3857) AS transformed_mpoint
FROM moving_birds
WHERE ind_id='Nobert';

Result 25.

```

```
ind_id|ind_can_name      | transformed_mpoint
------+------------------+------------------------------------------------------
Nobert|Cuculus  canorus  |srid=3857;tzid=1;MPOINT((1367661.245132 6272469.257576
      |                  | 0.000000 2013-04-26 07:01:10,....,1103565.771979
      |                  | 1171040.886414 0.000000 2013-08-16 06:38:48))
```

4.6 CONCLUSION

In this chapter, operations for projection to domain/range, interaction with domain/range, rate of change, and validation and transformation are discussed in detail. All the operations are designed and implemented to study the behavior of moving points in a DBMS. Moving bird dataset is used to test the functionality of the operation. The result gained from the queries used in moving bird dataset was used to explain their practical application. The importance operations in combination with one another was also explained and visualized in this chapter. In general, in addition to the implementation of moving point type and its supporting types, the implementation of the operations completes the moving object library for moving objects that possess changes in position only.

Chapter 5
Interpolation Functions for Moving Points

5.1 INTRODUCTION

For reasons of limited bandwidth and server database update performance, the trajectory of moving objects cannot be updated continuously in the database. However, we can represent the moving objects by discretely reporting spatiotemporal attributes such as location, direction, and speed [Yu et al., 2004]. This in principle means that we cannot query moving object movement for moments in-between two discrete positions. In situations like this, using interpolation techniques give us the entire movement. Different application domain requires different types of interpolation methods. There are several types of interpolation method available; selection of the appropriate interpolation method clearly relies on pre-knowledge we have about the moving object, such as the speed, direction, acceleration and even higher-order derivatives for their rate of change.

The first and most commonly used approach is linear interpolation. Cubic and last known measurement interpolations are also used as an alternative for linear interpolation. The use of interpolation technique enhances the capability to represent moving objects with a lower number of sampled positions. The basic philosophy behind is that registered locations of the moving point are correct and are on some unknown curve. Estimating a value of the curve at any position between two registered locations gives the position of the moving object for timestamps at which the location is unknown. The result causes a significant difference between the interpolation techniques used for different application domains.

The implementation of mathematical formulas of each interpolation method is handled through the operation *mpt_atinstant* which has been described in Chapter 4. The operation implicitly makes a call to the corresponding interpolation function implementation that represents the mathematics behind each interpolation method. The functions *linear_interpolation*, *cubic_interpolation*, and *last_known_interpolation* together with their results are discussed in Section 5.5.

The rest of the chapter is organized as follows: Section 5.2 presents linear interpolation method with its drawbacks. Section 5.3 presents cubic interpolation with its benefits over linear interpolation. Section 5.4 briefly describes the last known interpolation method. Finally, results of each interpolation method is discussed and compared in Section 5.5.

5.2 LINEAR INTERPOLATION

A moving object trajectory can be represented using a sequence of connected line segments. Each of the line segments connects two reported sample positions of the moving point. To interpolate or estimate the trajectory of a moving object between two consecutive reported states, a linear interpolation method can be used. For two given sampled consecutive positions of a moving object, a linear interpolation method assumes the velocity of the moving object is constant (fixed) for the time interval of the segment. In general, linear interpolation is used to approximate a value of the function f for a moving point using two known results of the function. To interpolate a value of a function at $f(t)$, Equation 5.1 is used.

$$f(t) = f(t_0) + \frac{f(t_1) - f(t_0)}{t_1 - t_0} * (t - t_0) \tag{5.1}$$

Given two consecutive location of a moving point $P_0(x_0, y_0, z_0, t_0)$ and $P_1(x_1, y_1, z_1, t_1)$ as shown in Figure 5.1, to interpolate the location of the moving point at time t, using linear interpolation, Equation 5.2 is used to compute each coordinates of the interpolated point.

$$X = x_0 + \frac{x_1 - x_0}{t_1 - t_0} * (t - t_0) \tag{5.2}$$

$$Y = y_0 + \frac{y_1 - y_0}{t_1 - t_0} * (t - t_0) \tag{5.3}$$

$$Z = z_0 + \frac{z_1 - z_0}{t_1 - t_0} * (t - t_0) \tag{5.4}$$

The results of the above mathematical equation are later used to initialize the approximate location $P_{interpolate} = (X, Y, Z, t)$ of a moving point.

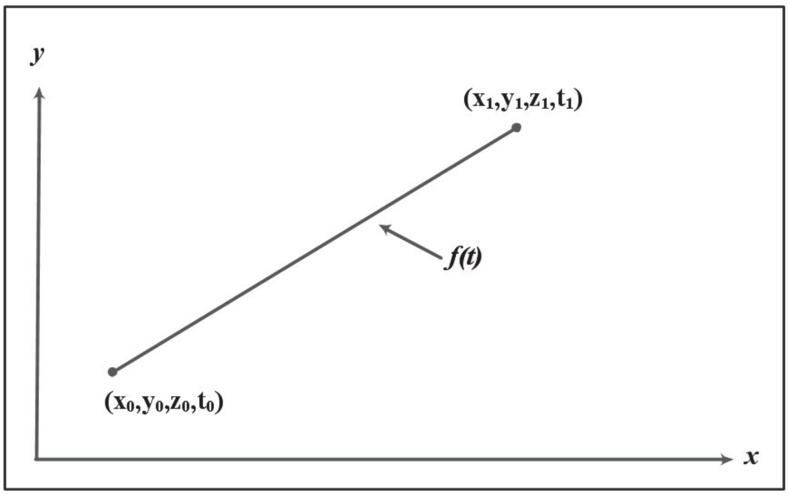


Figure 5.1: Linear Interpolation

Representing trajectories using line segments create unnatural turns and angles while representing the movement of a continuously moving point. The disadvantage of using linear interpolation is that the first-order derivative of the trajectory of a moving object only show changes in discrete positions [Becker et al., 2004]. In addition, the interpolation method is not much of use to capture non-zero acceleration in moving objects. The second-order derivative of the trajectory is always zero. For some application domains, this may be unwanted.

5.3 CUBIC INTERPOLATION

Curve-based interpolation is used to represent trajectories of real world moving objects using a sequence of curved segments. This approach requires fewer sample positions than the trajectory using line segments. Trajectories of this type are viewed as splines,[1], composed of a sequence of low degree curves [Yu et al., 2004]. Each sampled position from the continuous movement is used as a

[1]http://en.wikipedia.org/wiki/Spline_%28mathematics%29

joint between two adjacent curve segments. This provides a higher degree of continuity on every joint. Yu et al. [2004] proposed to use parametric cubic functions of degree three to obtain splines that pass through any given consecutive sampled points. For this research project, a polynomial function of degree three is used to represent a specialized parametric cubic function. In general, the idea of cubic interpolation is to draw a curve of the form $f(t) = a_0 + a_1t + a_2t^2 + a_3t^3$ that passes through four sampled positions of the moving point. The equation $f(t) = a_0 + a_1t + a_2t^2 + a_3t^3$ requires four consecutive sampled positions to estimate the coefficients a_0, a_1, a_2 and a_3. Having the values of the coefficients enables us to interpolate for moments in-between two sampled positions based on two before and after positions of a moving point. Figure 5.2 shows the difference between parametric cubic and linear interpolation. Cubic interpolation technique assumes the acceleration changes linearly in one direction for the time interval of the segment [Yu and Kim, 2006].

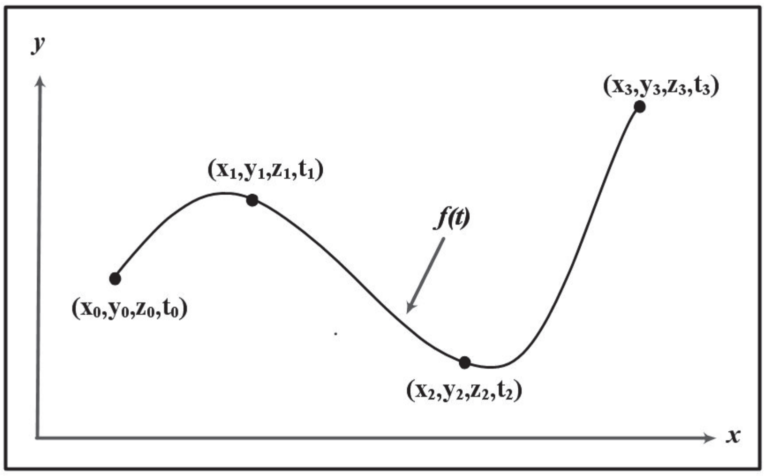


Figure 5.2: Cubic Interpolation

Given a sequence of chronologically ordered locations of a moving object $P_0(x_0, y_0, z_0, t_0)$, $P_1(x_1, y_1, z_1, t_1)$, $P_2(x_2, y_2, z_2, t_2)$, and $P_3(x_3, y_3, z_3, t_3)$, depicting a nonlinear change of velocity, one can use the cubic interpolation function to interpolate for $f(t)$, where $t_0 < t_1 < t < t_2 < t_3$. Therefore, the interpolated $P_{interpolate}(X, Y, Z, t)$ is computed as follows. To interpolate the X coordinate of the new location, we use x_i values (x_0, x_1, x_2, and x_3) from the four reported states of the moving object. In practice the function $f(t) = a_0 + a_1t + a_2t^2 + a_3t^3$ gives (X,Y,Z) and the coefficients a_0, a_1, a_2 and a_3 are vectors of length three. The function $f(t).x = a_0 + a_1t + a_2t^2 + a_3t^3$ results (x_0, x_1, x_2, and x_3) for timestamps t_0, t_1, t_2 and t_3, respectively. Then, we need to find a function of the form $f(t).x = a_0 + a_1t + a_2t^2 + a_3t^3$ that satisfies the condition. This can be rewritten as follows:

$$x_0 = a_0 + a_1t_0 + a_2t_0^2 + a_3t_0^3 \tag{5.5}$$

$$x_1 = a_0 + a_1t_1 + a_2t_1^2 + a_3t_1^3 \tag{5.6}$$

$$x_2 = a_0 + a_1t_2 + a_2t_2^2 + a_3t_2^3 \tag{5.7}$$

$$x_3 = a_0 + a_1t_3 + a_2t_3^2 + a_3t_3^3 \tag{5.8}$$

The above equation can be solved using substitution and matrix method. However, the later is selected and the four equations are rewritten in a matrix form as follow:

$$\begin{bmatrix} 1 & t_0 & t_0^2 & t_0^3 \\ 1 & t_1 & t_1^2 & t_1^3 \\ 1 & t_2 & t_2^2 & t_2^3 \\ 1 & t_3 & t_3^2 & t_3^3 \end{bmatrix} * \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \tag{5.9}$$

Solving the above equation results in the coefficient values a_0, a_1, a_2, and a_3. Hence, by substituting the computed coefficients into the cubic function that defines the curve that pass though coordinates $(x_0, x_1, x_2, and x_3)$, the X coordinate of the interpolated location can be computed for timestamp t. The Gaussian elimination method[2] is used to solve Equation 5.9.

The Y and Z coordinates can also be computed from (y_0, y_1, y_2, and y_3), and (z_0, z_1, z_2, and z_3), respectively. Using Equations 5.5 - 5.8 and 5.9, following the same procedure, we can calculate the coefficients a_0, a_1, a_2, and a_3 of the function $f(t) = a_0 + a_1t + a_2t^2 + a_3t^3$ for both coordinates. Later, by substituting the timestamp value into the functions formulated using the computed coefficients, the values for the coordinates X and Y are interpolated. The interpolation method is much more useful when we want to enforce a smooth change of speed and direction. Cubic interpolation method creates a smooth and balanced distribution on change of speed and direction between sampled positions of a moving point. Cubic interpolation requires registered positions to be ordered chronologically. If that fails to satisfy, the result of the interpolation may deviate significantly from the exact location of the moving point for a given timestamp.

5.4 LAST KNOWN INTERPOLATION

Unlike the two other interpolation techniques, this approach does not require any mathematical formula to interpolate positions of the moving point. It only takes temporally nearest sampled (registered) location between the start and end nodes of the segment. The main task in this interpolation method is to identify the two consecutive sampled locations, where the interpolation timestamp is within the time interval of the segment formed by the sampled locations. From the two sampled locations, the location with the timestamp temporally closest to the interpolating timestamp is returned as a result. This type of interpolation technique is important when we want to set the focus to the sampled locations of the moving point. For example, a transportation bus that only halts at bus stops is a good example to apply this interpolation method. The buses have sampled locations to represent the discrete positions they have stopped. If we are interested to know nearest bust-stop station in time, we can apply this interpolation method. This interpolation method is also essential to identify frequent patterns on moving points.

5.5 RESULT AND DISCUSSION

The functions *linear_interpolate*, *cubic_interpolate* and *last_known_interpolate* are only accessible through the operation *mpt_atinstant*. As discussed in Chaper 4, the operation comes with two interfaces that accept two different arguments. The first interface '*mpt_atinstant(mpoint,cstring)*' only accepts a value of the type *mpoint* and *timestamp* to interpolate the location. This implicitly indicates that the interpolation method to be applied is the default linear interpolation. The other interface '*mpt_atinstant(mpoint,cstring,bool)*' accepts a value of the type *mpoint*, *timestamp* and *boolean* value. This notifies that we are interested to apply the other two interpolation methods: namely cubic and last known interpolations methods. The choice between the two interpolations depends on the boolean value passed as an argument. If the value is *true*, the operation uses cubic interpolation method. Otherwise, last known interpolation method is used to interpolate

[2]http://en.wikipedia.org/wiki/Gaussian_elimination

the position of the moving point at a given timestamp. Query statements in Listing 5.1 exhibit the use of the three interpolation method based on the two interfaces provided for the function *mpt_atinstance*.

The implementation of the *linear_interpolate* function is simpler compared to that of the *cubic_interpolate* function. The function uses the operation *point_interpolate* operation as available in the LWGEOM library of PostGIS. The function accepts two point locations and an interpolation value which is used to define the ratio between the two locations. Therefore, the function *linear_interpolation* finds the two sampled locations and passes them as arguments to the function *point_interpolate* together with the interpolation timestamp. As a result the operation *point_interpolate* returns the interpolated location of the moving point. However, the implementation of the *cubic_interpolation* function is much more complicated and requires the implementation of every mathematical formula described in Section 5.3. This includes solving four simultaneous equations, that results with estimated value for the coefficients a_0, a_1, a_2, and a_3. The function uses linear interpolation instead, if the interpolation timestamp passed as an argument is between the time interval of the first or last segment for the trajectory of the moving point. The implementation of the *last_known_interpolate* function only requires identifying the right segment that contains the time instant value accepted as an argument. Subsequently, the node of the segment with a timestamp closest to interpolating timestamp is returned as a result of the function.

To compare linear interpolation and cubic interpolation methods, two testing mechanisms are applied on the moving bird dataset. First, to simply interpolate the location of the moving bird “Belarus 2” on a given timestamp; and observe that all interpolation methods applied return relatively similar result. This only infers that the mathematical basis and implementation of the functions for each interpolation method are correct. It neither perform performance analysis nor accuracy of the result. Second, to interpolate the moving point at a timestamp with an already know position of the moving bird. To achieve this, it is necessary to remove a single discrete location used to represent the movement of the moving bird and interpolate for the associated timestamp. The results gained from this approach tells the deviation of the interpolated location from the actual sampled position of the bird. In addition, it tells which interpolation method is suitable to use for the trajectory the moving bird. Query 1 from Listing 5.1 shows the result of the operation *mpt_atinstant* for the timestamp ‘2013-07-01 18:50:33’ using linear interpolation method. Query 2 shows the result using cubic interpolation for the same timestamp. Query 3 also shows the result of the operation *mpt_atinstnt* using *last_known_location* interpolation technique. The results of the queries surmise that all the interpolation techniques are showing relatively similar interpolation results with significant differences that can cause a potential problem in different applications. When we consider the the number of sampled positions per bird, it is obvious that we do not have sufficient amount of data that allows us to determine any type of higher-order speed/direction component. the sampling has been orders of magnitude too small for this. For now, while studying the behaviors of birds, linear interpolation has to be the norm. This may have been different when we monitor cars, which allows us to sample positions may times per second. In that application determining acceleration is a sensible and doable thing.

Listing 5.1: Where was the bird "Belarus 2" on 2013-07-01 18:50:33 ?

```
Query 1.

SELECT ind_id,ind_can_name,mpt_atinstant(locations,'2013-07-01 18:50:33 1') AS atinstant
FROM moving_birds
WHERE ind_id='Belarus 2';

Result 1.

ind_id      |ind_can_name      | atinstant
```

```
-----------+-----------------+-----------------------------------------------------------
Belarus 2 |Cuculus canorus | INTIME_MPOINT(POINT Z (33.804981 34.595561 0),2013-07-01
     18:50:33 1)

Query 2.

SELECT ind_id,ind_can_name,mpt_atinstant(locations,'2013-07-01 18:50:33  1',true) AS
     atinstant
FROM moving_birds
WHERE ind_id='Belarus 2';

Result 2.

ind_id     |ind_can_name    | atinstant
-----------+-----------------+-----------------------------------------------------------
Belarus 2 |Cuculus canorus | INTIME_MPOINT(POINT Z (44.31374 19.85662212 0),2013-07-01
     18:50:33 1)

         Query 3.

SELECT ind_id,ind_can_name,mpt_atinstant(locations,'2013-07-01 18:50:33  1',false) AS
     atinstant
FROM moving_birds
WHERE ind_id='Belarus 2';

Result 3.

ind_id     |ind_can_name    | atinstant
-----------+-----------------+-----------------------------------------------------------
Belarus 2 |Cuculus canorus | INTIME_MPOINT(POINT Z (29.97756 41.12452 0),2013-07-01
     18:50:33 1)
```

The other mechanism applied to test the interpolation methods is to remove a true sampled location of a moving bird. To progress on this, the location of the moving bird "Belarus 2" at a timestamp '2013-06-30 18:50:33' is removed from the sampled discrete locations. The bird "Belarus 2" was in the location '29.977560, 41.124520, 0.000000' on the given timestamp. Afterwards, the operation *mpt_atinstant* is used to interpolate the location of the moving bird at a time instant removed from the list. Query 4, 5, and 6 in Listing 5.2 are executed to interpolate the location of the bird using the three interpolation method.

Listing 5.2: Where was the bird "Belarus 2" on '2013-06-30 18:50:33 '?

```
Query 4.

SELECT ind_id,ind_can_name,mpt_atinstant(locations,'2013-06-30 18:50:33  1') AS atinstant
FROM moving_birds
WHERE ind_id='Belarus 2';

Result 4.

ind_id     |ind_can_name    | atinstant
-----------+-----------------+-----------------------------------------------------------
Belarus 2 |Cuculus canorus | INTIME_MPOINT(POINT Z (36.089725 32.552322 0),2013-06-30
     18:50:33 1)

Query 5.

SELECT ind_id,ind_can_name,mpt_atinstant(locations,'2013-06-30 18:50:33  1',true) AS
     atinstant
FROM moving_birds
WHERE ind_id='Belarus 2';

Result 5.

ind_id     |ind_can_name    | atinstant
-----------+-----------------+-----------------------------------------------------------
```

```
 Belarus 2 |Cuculus canorus | INTIME_MPOINT(POINT Z (38.5996 26.5190816 0),2013-06-30
     18:50:33 1)

 Query 6.

SELECT ind_id ,ind_can_name ,mpt_atinstant(locations ,'2013-06-30 18:50:33  1',false) AS
     atinstant
FROM moving_birds
WHERE ind_id='Belarus 2';

 Result 6.

 ind_id      |ind_can_name     | atinstant
-------------+-----------------+------------------------------------------------------------------
 Belarus 2 |Cuculus canorus | INTIME_MPOINT(POINT Z (38.73921 26.17857 0),2013-06-30
     18:50:33 1)
```

When we investigate the results of the queries in Listing 5.2, all location interpolation deviate from the sampled position of the moving bird. In general, the difference becomes smaller for linear and cubic interpolation if the moving object is moving along a linear infrastructure. For example, for a car moving in a straight road, the difference is smaller than for a bird that flies in the air. In all cases, the curved-based approach performs better than the linear interpolation approach as stated in [Yu et al., 2004]. Given similar sampled points of a moving point, curved based interpolation provides relatively more precise approximation than that of the linear interpolation and *last_known_interpolation* techniques.

5.6 CONCLUSION

In general, the preference for interpolation techniques strongly depend on the the domain of applications, and actually with that, also on the technology that has been used for positioning. In birds, hardware constraints are the cause of very low positioning frequency, which essentially means that higher-order interpolations are meaningless. Where the hardware constraints have less impact, think of planes, cars, your positioning frequency is much better, and then estimating higher-order movement parameters may be possible and actually also useful.

Chapter 6
Aggregate Functions on Moving Points

6.1 INTRODUCTION

It is common to use built-in aggregate functions (operators) in database queries. Aggregation functions on moving points are also important to understand the collective behavior of moving objects. This chapter includes detailed explanation on aggregation functions that are introduced briefly in Chapter 1. The design method and implementation detail for each aggregate function is presented Section 6.3. The functions are also tested and demonstrated with examples using subsets of moving bird data sets.

The data set used to experiment on the functions is structured and cropped to represent all possible states of moving points. Aggregation function execution over sets of moving points that have different number of segments and time segments causes such functions to return different result using similar algorithms. For example, the algorithms used for the aggregation function *mpt_centroid* return the moving point centroid that have different number of segments, and this depends on the number of segments and shared time interval of each of the moving points on the set.

6.2 IMPLEMENTATION REQUIREMENTS OF AGGREGATION FUNCTIONS

Implementation of aggregation functions in PostgreSQL requires state values and state transition functions [Lockhart, 2013]. A final function must also be implemented, if the data that needs to be returned as a result of the aggregation function is different from the data that is stored as an intermediate state value. The state transition function[1] is invoked every time a successive row is processed. If defined, a final function is executed when there is no other row left to be processed. The state value can have a data type that is similar to the base type and also it requires an initial value. Therefore, to implement the aggregate functions on moving points, a support type *mpoint_a* is developed to store the state value. Every time a successive row is processed, the state value is passed to the state transition function along with the base type value. The state transition function then performs the necessary aggregation operation and returns a modified state value for the next successive state transition function or final function.

The support type *mpoint_a* is designed and implemented to store intermediate state values that results from the centroid computation between the state and current moving point. The data structure in Listing 6.1 is used to represent the internal structure of the type. The structure consists of *num_segts*, *sr_id*, *tz_id*, and an array of reference **mpt_sgts*. The pointer variable **mpt_sgts* stores an array of pointers to the structure *mpt_segment*. The structure *mpt_segment* defined to represent segments of a moving point contains *num_pts* and **geom_array*. The pointer variable **geom_array* stores references to LWPOINT point geometries of PostGIS. All the varibales used to define the structure of the type *mpoint_a* have similar definitions to the structure of *mpoint* defined in Section 2.4. The only difference is the variable *count* which is defined to store additional

[1] http://www.postgresql.org/docs/8.3/static/xaggr.html

intermediate state value other than the computed centroid moving point. The value to be stored in the variable *count* becomes different depending on the aggregate function which is using the state type *mpoint_a*. For example, the *count* variable is used to store the number of moving points that are already participated on the computation of the centroid moving point for the operation *mpt_centroid*. For the operation *mpt_max_trajectory*, it stores the maximum traveling distance value of the moving point.

Listing 6.1: Data structure for support mpoint_a type

```

typedef struct mpt_segment
{
        int num_pts;
        LWPOINT *geom_array;
}mpt_segment;

typedef struct mpoint_a
{
  double count;
        int num_segts;
        int sr_id;
        int tz_id;
        mpt_segment *mpt_sgts;
}mpoint_a;
```

6.3 IMPLEMENTATION OF MOVING POINT AGGREGATION FUNCTIONS

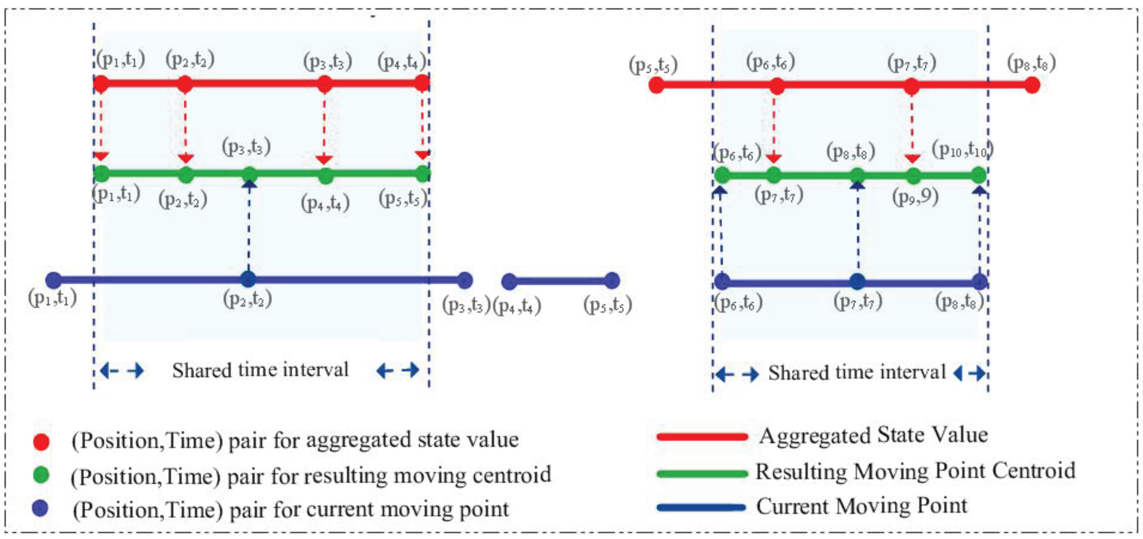
6.3.1 mpt_centroid

The aggregation operation *mpt_centroid* returns the center of gravity ("Centroid") for the set of moving points. The operation takes an *mpoint* value and an optional boolean value. The operation returns the centroid as a moving point representing a set of individual moving points. The centroid over a set of moving points can be computed using different algorithms. In this research project, three different algorithms are developed and implemented to compute the centroid. These are **"shared interval"**, **"closure"** and **"dormant point"** algorithms. To use these algorithms the operation *mpt_centroid* provides two interfaces. The first interface *mpt_centroid(mpoint)* uses the default **"shared interval"** algorithm. The second interface *mpt_centroid(mpoint,bool)* allows to select the other two algorithms. If the boolean argument value is *true*, the operation uses the **"closure"** algorithm; otherwise, it uses the **"dormant point"** algorithm. The operation uses the functions *mpt_agg* and *mpt_agg_final* as a state transition final function, respectively.

The support type *mpoint_a* is implemented to store the aggregated state value. The aggregated state value in the *mpt_centroid* operation includes the centroid moving point and the count value that represents the number of moving points that have already participated in the computation. The state transition function takes a moving point, the aggregated state value and a boolean value (if provided) as argument and returns the next aggregated state value of the type *mpoint_a*. In the first call of this function, the aggregated state value is passed as a *null* argument and the first non-empty moving point is returned as the next aggregated centroid moving point along with the count value. For the rest of successive functions calls, based on the choice of algorithm, the state transition functions passes the aggregated state moving point segments, current moving point segments, aggregated state moving point segment number, current moving point segment number and count value as an argument to the function implemented to realize the algorithms. To exactly represent the final centroid result, depending on the algorithm selected, every (location-time) pair of the aggregated and successive moving points have to contribute to the computation of the centroid result. The final function takes the aggregated state value and returns the centroid as a moving point. The philosophy used for the development of the three algorithms is explained below.

The “shared interval” algorithm

As the name indicates the algorithm **“shared interval”** focuses on the overlapping (shared) time intervals between the aggregated state moving point and current moving point. The algorithm computes the aggregation operation on all positions that are defined within the shared time intervals of the two input moving points. No (position-time) pair that is not defined within the shared time intervals participates in the computation of the centroid moving point. As illustrated in Figure 6.1, all (postion-time) pairs of both the aggregated state and current moving point that are defined within the shared time interval are participated on the construction of the (position-time) pairs of the resulting moving point centroid. The aggregated state value in this algorithm represents the centroid between N numbers of moving points that are bounded by their shared time intervals.



(p_1,t_1) (p_2,t_2) (p_3,t_3) (p_4,t_4)
(p_3,t_3)
(p_1,t_1) (p_2,t_2) (p_4,t_4) (p_5,t_5)
(p_1,t_1) (p_2,t_2) (p_3,t_3) (p_4,t_4) (p_5,t_5)
Shared time interval
(p_5,t_5) (p_6,t_6) (p_7,t_7) (p_8,t_8)
(p_6,t_6) (p_8,t_8) (p_{10},t_{10})
(p_7,t_7) $(p_9,9)$
(p_6,t_6) (p_7,t_7) (p_8,t_8)
Shared time interval
(Position,Time) pair for aggregated state value
(Position,Time) pair for resulting moving centroid
(Position,Time) pair for current moving point
Aggregated State Value
Resulting Moving Point Centroid
Current Moving Point

Figure 6.1: Resulting centroid based on the **“shared interval”** algorithm

For better explanation the algorithm is divided into three parts. Algorithms 6.1, 6.2 and 6.3 are used to to implement the three parts of the **“shared interval”** algorithm. The first part of the algorithm uses the timestamp of all (position-time) pairs of the aggregated state moving point that are defined within the shared time intervals of the two input moving points. The timestamps are used to interpolate positions of the current moving point. The interpolation happens when there is no (position-time) pair of the current moving point that have a timestamp equal to the interpolation time instant. This enforces the basic philosophy that every (position-time) pair of the two input moving points needs to contribute for the computation of the centroid. The resulting interpolated positions along with the original (position-time) pairs of the current moving point are used to construct a temporary moving point that is bounded to the shared time intervals. This temporary moving point representing the time bounded current moving point is used in the third part of the algorithm.

Algorithm 6.1The "shared interval" algorithm (Part_1)

INPUT TYPE: current_num_sgts,mpoint_current_sgt,state_num_sgts,mpoint_state_sgt,count
OUTPUT TYPE: mpoint_a

```
1:  $temp\_num\_points \leftarrow 0$
2:  $temp\_num\_segments \leftarrow 0$
3:  $current\_segment \leftarrow 0$
4:  $state\_segment \leftarrow 0$
5:  for $i = 0$ to $current\_num\_sgts$ do
6:    for $j = 0$ to $mpoint\_current\_sgt[i].num\_pts - 1$ do
7:      $temp\_point\_left \leftarrow mpoint\_current\_sgt[i].geom\_array[j]$
8:      $temp\_point\_right \leftarrow mpoint\_current\_sgt[i].geom\_array[j + 1]$
9:      $temp\_time\_left \leftarrow temp\_point\_left.m$
10:     $temp\_time\_right \leftarrow temp\_point\_right.m$
11:     if $j == 0$ then
12:       $temp\_geom\_current[temp\_num\_points] \leftarrow temp\_point\_left$
13:       $temp\_num\_points + +$
14:     end if
15:     for $l = 0$ to $state\_num\_sgts$ do
16:       for $k = 0$ to $mpoint\_state\_sgt[l].num\_pts$ do
17:         $temp\_point\_search \leftarrow mpoint\_state\_sgt[l].geom\_array[k]$
18:         $temp\_time\_search \leftarrow temp\_point\_search.m$
19:         if $(temp\_time\_search > temp\_time\_left)$ $AND$ $(temp\_time\_search <$
            $temp\_time\_right)$ then
20:           $temp\_geom\_current[temp\_num\_points] \leftarrow temp\_point\_left$
21:           $LWPOINT\_INTERPOLATED \leftarrow$ INTERPOLATE($temp\_point\_left$,
              $temp\_point\_right$,$temp\_time\_search$)
22:           $temp\_geom\_current[temp\_num\_points] \leftarrow LWPOINT\_INTERPOLATED$

23:           $temp\_num\_points + +$
24:         end if
25:       end for
26:     end for
27:     $temp\_geom\_current[temp\_num\_points] \leftarrow temp\_point\_right$
28:     $temp\_num\_points + +$
29:   end for
30:   $temp\_current\_segments[temp\_num\_segments] \leftarrow temp_geom_current$
31:   $temp\_num\_segments + +$
32: end for
```

The second part of the algorithm is similar to the first one. The only difference is that the time component of all (position-time) pairs of the current moving point are used to interpolate the positions of the aggregated state moving point. The resulting interpolated locations along with the original (position-time) pairs of the aggregated state moving point are used to construct a temporary moving point that is bounded by the shared time intervals. The next step in between is to chronologically order the resulting temporary moving points of the two algorithm parts.

Algorithm 6.2The "shared interval" algorithm (Part_2)

INPUT TYPE: current_num_sgts,mpoint_current_sgt,state_num_sgts,mpoint_state_sgt,count
OUTPUT TYPE: mpoint_a

```
1:  $current\_segment \leftarrow temp\_num\_segments$
2:  $temp\_num\_points \leftarrow 0$
3:  $temp\_num\_segments \leftarrow 0$
4:  for $i = 0$ to $state\_num\_sgts$ do
5:    for $j = 0$ to $mpoint\_state\_sgt[i].num\_pts - 1$ do
6:      $temp\_point\_left \leftarrow mpoint\_state\_sgt[i].geom\_array[j]$
7:      $temp\_point\_right \leftarrow mpoint\_state\_sgt[i].geom\_array[j+1]$
8:      $temp\_time\_left \leftarrow temp\_point\_left.m$
9:      $temp\_time\_right \leftarrow temp\_point\_right.m$
10:     if $j == 0$ then
11:       $temp\_geom\_state[temp\_num\_points] \leftarrow temp\_point\_left$
12:       $temp\_num\_points + +$
13:     end if
14:     for $l = 0$ to $current\_num\_sgts$ do
15:       for $k = 0$ to $mpoint\_current\_sgt[l].num\_pts$ do
16:         $temp\_point\_search \leftarrow mpoint\_current\_sgt[l].geom\_array[k]$
17:         $temp\_time\_search \leftarrow temp\_point\_search.m$
18:         if $(temp\_time\_search > temp\_time\_left)$ && $(temp_time_search < temp_time_right)$ then
19:           $temp\_geom\_current[temp\_num\_points] \leftarrow temp\_point\_left$
20:           $lwpoint\_interpolated \leftarrow$ INTERPOLATE($temp_point_left$, $temp_point_right$,$temp_time_search$)
21:           $temp\_geom\_state[temp\_num\_points] \leftarrow LWPOINT\_INTERPOLATED$
22:           $temp\_num\_points + +$
23:         end if
24:       end for
25:     end for
26:     $temp\_geom\_state[temp\_num\_points] \leftarrow temp\_point\_right$
27:     $temp\_num\_points + +$
28:   end for
29:   $temp\_state\_segments[temp\_num\_segments] \leftarrow temp_geom_current$
30:   $temp\_num\_segments + +$
31: end for
```

The final part of the algorithm takes the resulting chronologically ordered moving points and computes the spatial average between (position-time) pairs of the two temporary moving points. The spatial averaging is computed between two (position-time) pairs of the temporary moving points that have equal timestamps. One is from the temporary moving point that represent the aggregated state moving point and the other is from the temporary moving point that represent the current moving point. To compute the (position-time) pair of the resulting centroid moving point, the position from the aggregated temporary state moving point is multiplied by the count value that represent the number of moving points that have already participated on previous centroid computations. The result is then summed up to the position of the current temporary moving point. The result divided by the count plus one value represents the centroid position. This process is iterative and the resulting centroid positions are used to construct the centroid moving point between the input aggregated state moving point and current moving point. Finally the operation *mpt_centroid* returns the state value composed of the centroid moving point along with the incremented count value as a type of *mpoint_a*.

Algorithm 6.3The `"shared interval" algorithm (Part_3)`

INPUT TYPE: current_num_sgts,mpoint_current_sgt,state_num_sgts,mpoint_state_sgt,count
OUTPUT TYPE: mpoint_a

```
1:  $state\_segment \leftarrow temp\_num\_segments$
2:  $temp\_num\_points \leftarrow 0$
3:  $temp\_num\_segments \leftarrow 0$
4:  for $i = 0$ to $state\_segment$ do
5:      for $j = 0$ to $temp_state_segment[i].num\_pts$ do
6:          $temp\_point\_left \leftarrow temp\_state\_segment.geom\_array[j]$
7:          $temp_time_left \leftarrow temp\_point\_left.m$
8:          for $l = 0$ to $current\_segment$ do
9:              for $k = 0$ to $temp_current_segment[l].num\_pts$ do
10:                 $temp\_point\_right \leftarrow temp\_curernt\_segment.geom\_array[k]$
11:                 $temp\_time\_right \leftarrow temp\_point\_right.m$
12:                 if $temp\_time\_left = temp\_time\_right$ then
13:                     $aggregated\_x \leftarrow (count \times temp\_point\_left.x + temp\_point\_right.x) / (count + 1)$
14:                     $aggregated\_y \leftarrow (count \times temp\_point\_left.y + temp\_point\_right.y) / (count + 1)$
15:                     $aggregated\_z \leftarrow (count \times temp\_point\_left.z + temp\_point\_right.z) / (count + 1)$
16:                     $lwpoint4d \leftarrow$ make_new_4d_point($srid, aggregated\_x, aggregated\_y, aggregated\_z, temp\_time\_l$
17:                     $temp\_geom\_agg[temp\_num\_points] \leftarrow lwpoint4d$
18:                     $temp\_num\_points++$
19:                 end if
20:             end for
21:         end for
22:     end for
23:     $temp\_agg\_segments[temp\_num\_segments] \leftarrow temp_geom_agg$
24:     $temp\_num\_segments++$
25: end for
26: $mpoint\_return \leftarrow$ gserialized_from_mpointa($temp\_agg\_segments, temp\_num\_segments, count + 1$)
27: return $mpoint\_return$
```

The illustrative Figure 6.1 shows the result of the aggregation operation using the algorithm **"shared interval"**. In the computation of the centroid, the aggregated state moving point contributes five positions in total and the current moving point contributed four positions in total.

Therefore, the resulting centroid moving point have ten (postion-time) pairs distributed in two segments.

Listing 6.2 shows query results of the operation *mpoint_centroid* using the **“shared interval”** algorithm for four moving birds.

Listing 6.2: `Return the centroid of the moving birds?`

```
Query 8.

SELECT mpt_centroid(locations) from aggtest as mpts_centroid;

Result 6.

mpts_centroid
-------------------------+----------------------------------------------------------
"srid=4326;tzid=1;MPOINT((13.387776 19.763165 0.000000 2013-09-03 02:23:58,13.404665
19.702901 0.000000 2013-09-03 03:15:53,13.813233 18.510897 0.000000 ...............
2013-09-03 19:57:45, ...................., 13.876243 18.324567 0.000000 2013-09-03
22:34:46, ...... .......... , 13.890332 11.205585 0.000000 2013-09-14 22:06:06))"
```

The “closure” algorithm

For moving objects that have undefined moments during their movement, the closure operator on a moving point is used to fill the gap between the segments that defines their movement. For example, the aggregated state and current moving points in Figure 6.2 represents the closure of the aggregated state and current moving points from Figure 6.1. The **“closure”** algorithm computes the centroid for sets of moving points after the closure operator is applied on them. The **“closure”** algorithm works in combination with the function implemented for the algorithm **“shared interval”**. The closure operator is implemented for the state transition function *mpt_average* to join every segment of a moving point that is passed to the function as argument and it results a single segment moving point. To compute the actual centroid for sets of moving points, the function *mpt_average* passes the result gained from the closure operator to the previous **“shared interval”** algorithm function. As illustrated in Figure 6.2, the result of the operation *mpt_centroid* using the **“closure”** algorithm is a centroid moving point of single segment.

From the illustration in Figure 6.2, we can observe that for equal number of (position-time) pairs of the aggregated state and current moving points, the centroid moving point that is resulted from the operation *mpt_centroid* using the **“closure”** algorithm participates more locations than the **“shared interval”** algorithm. Listing 6.3 shows the results of the operation *mpt_centroid* based on the **“closure”** algorithm in four moving birds.

Listing 6.3: `Return the centroid of the moving birds?`

```
Query 8.

SELECT mpt_centroid(locations,true) from aggtest as mpts_centroid;

Result 6.

mpts_centroid
-------------------------+----------------------------------------------------------
"srid=4326;tzid=1;MPOINT((18.073690 45.782520 0.000000 2013-08-21 22:09:20,14.046055
34.762082 0.000000 2013-08-22 23:24:09,14.011606 34.825269 0.000000 2013-08-23 02:42:24,
................................ ,11.950405 11.078395 0.000000 2013-09-16 04:12:25))"
```

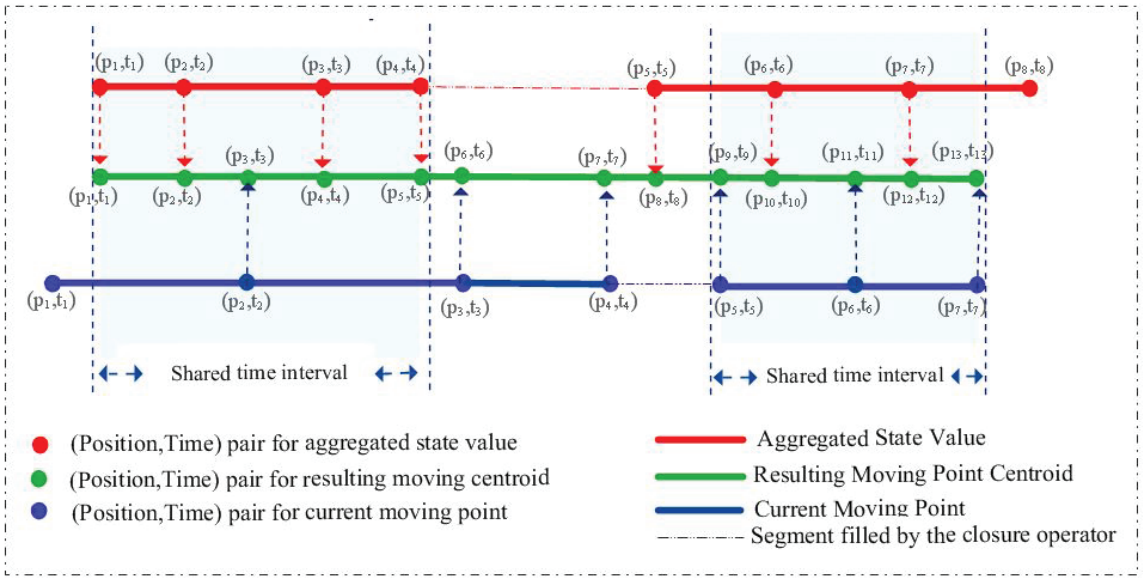

(p_1,t_1) (p_2,t_2) (p_3,t_3) (p_4,t_4) (p_5,t_5) (p_6,t_6) (p_7,t_7) (p_8,t_8)
(p_1,t_1) (p_2,t_2) (p_3,t_3) (p_4,t_4) (p_5,t_5) (p_6,t_6) (p_7,t_7) (p_8,t_8) (p_9,t_9) (p_{10},t_{10}) (p_{11},t_{11}) (p_{12},t_{12}) (p_{13},t_{13})
(p_1,t_1) (p_2,t_2) (p_3,t_3) (p_4,t_4) (p_5,t_5) (p_6,t_6) (p_7,t_7)
Shared time interval
Shared time interval
● (Position,Time) pair for aggregated state value
● (Position,Time) pair for resulting moving centroid
● (Position,Time) pair for current moving point
Aggregated State Value
Resulting Moving Point Centroid
Current Moving Point
Segment filled by the closure operator

Figure 6.2: Resulting centroid based on the **"closure"** algorithm

The "dormant point" algorithm

The **"dormant point"** algorithm computes the centroid (average) between the aggregated state and current moving point. The algorithm assumes that (position-time) pairs that are not defined within the shared time intervals of the aggregated state and current moving point are present and sleeping (not-changing position in time). This indicates that all the (position-time) pairs of the two input moving points participate on the computation of the centroid, unlike the **"shared interval"** algorithm which only participates (position-time) pairs that are defined within the shared time intervals of the two input moving points. As illustrated in Figure 6.3, the **"dormant point"** algorithm results similar moving point segments as of the **"shared interval"** algorithm.

The algorithm used to realize the function implementation of the **"dormant point"** algorithm is similar to Algorithms 6.1, 6.2 and 6.3 for (position-time) pairs that are defined within the shared time intervals of the two input moving points. The final result gained from the parts of the **"shared interval"** algorithm is merged with segments formed from (position-time) pairs that are not defined within the shared time intervals. The resulting collection of segments from this merging is then used to construct chronologically ordered segments of the centroid moving point.

As illustrated in Figure 6.3, the **"dormant point"** algorithm participates all (position-time) pairs of the aggregated state and current moving points. The resulting moving point centroid contains 15 (postion-time) pairs distributed over 5 segments of the moving point. Listing 6.4 shows the result of the operation mpt_centroid using **"dormant point"** algorithm on four moving birds.

Listing 6.4: Return the centroid of the moving birds?

```
Query 8.

SELECT mpt_centroid(locations, false) from aggtest as mpts_centroid;

 Result 6.

 mpts_centroid
---------------------------------+-------------------------------------------------------------------------
"srid=4326;tzid=1;MPOINT((18.107890 45.819990 0.000000 2013-08-19 11:06:00,18.073690
```

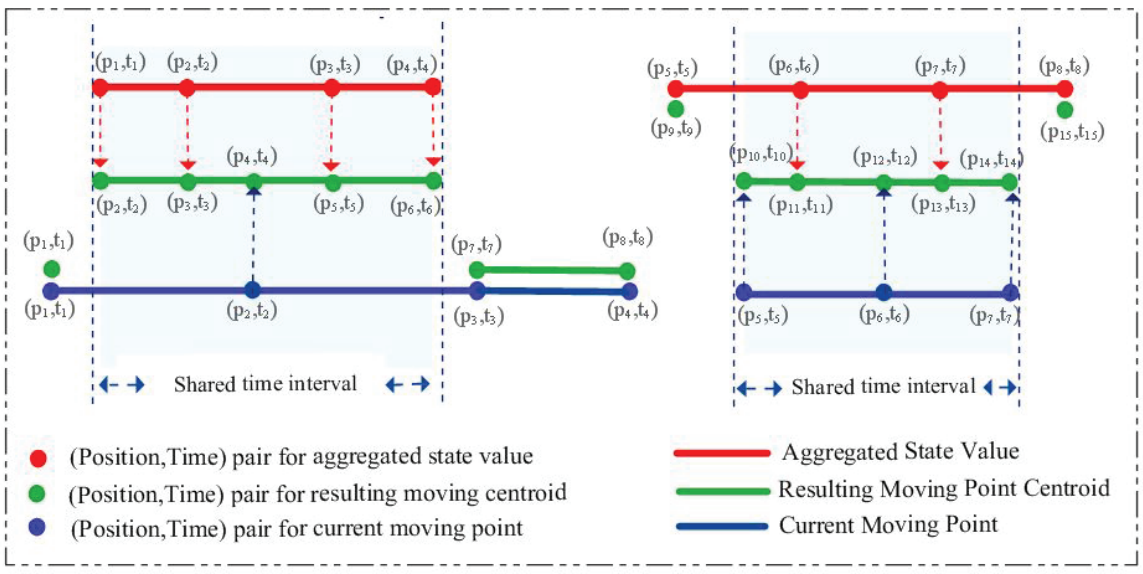

(p1,t1) (p2,t2) (p3,t3) (p4,t4)
(p4,t4)
(p2,t2) (p3,t3) (p5,t5) (p6,t6)
(p1,t1)
(p1,t1) (p2,t2) (p3,t3) (p4,t4)
(p7,t7) (p8,t8)
Shared time interval
(p5,t5) (p6,t6) (p7,t7) (p8,t8)
(p9,t9) (p15,t15)
(p10,t10) (p12,t12) (p14,t14)
(p11,t11) (p13,t13)
(p5,t5) (p6,t6) (p7,t7)
Shared time interval
(Position,Time) pair for aggregated state value
(Position,Time) pair for resulting moving centroid
(Position,Time) pair for current moving point
Aggregated State Value
Resulting Moving Point Centroid
Current Moving Point

Figure 6.3: Resulting centroid based on the **"dormant point"** algorithm

```
45.782520 0.000000 2013-08-21 22:09:20,14.046055 34.762082 0.000000 2013-08-22 23:24:09
,14.011606 34.825269 0.000000 2013-08-23 02:42:24, .......,13.570445 35.629771 0.000000
2013-08-24 22:46:46, .......... ,11.950405 11.078395 0.000000 2013-09-16 04:12:25))"
```

6.3.2 mpt_max_trajectory

The aggregation operation *mpt_max_trajectory* runs over a set of moving points, and returns the trajectory of the moving point that have a maximum distance covered. The result of the operation is returned as a multilinestring. The philosophy used to implement the operation is to calculate the distance covered by each moving point. Later, to return the moving point that have the maximum computed distance value. For example, in a racing sport, while monitoring the movement of the cars that have similar initial and final destination, we might be interested to know the path of the moving car that shows maximum driving distance. we can use the operation *mpt_max_trajectory* over moving cars that have already participated on the racing. The operation uses *mpt_max_trajectory* as a state transition function and the function *mpt_trajectory_final* as a final function. The state transition function takes the next moving point and a state transition value as an argument. The state transition value in this operation consists of the aggregated moving point with a maximum traveling (movement) distance and the computed distance value of the state moving point. When the function is called for the first time, the state transition value of *null* is passed as argument along with the first moving point. As a result, the first moving point with its calculated traveled distance is returned as the next state transition value. While the operation iterates on the moving points, the state transition function compares the calculated distance of the current moving point with the stored maximum distance of the state moving point. If the calculated distance is greater than the the store state maximum value, the state transition function returns the current moving point along with the calculated distance value as a state transition value. Otherwise, the previous state transition value is returned for the next state transition function. The final function takes the state transition value and returns the trajectory of the state moving point as a result of the operation. Listing 6.5 displays the query statement that returns the trajectory of a moving bird with the maximum distance covered.

Listing 6.5: Return the maximum trajectory of the moving birds?

```
Query 8.

SELECT  mpt_trajectory_max(locations)
FROM    aggtest as mpt_max_trajectory_result;

 Result 6.

 mpt_max_trajectory_result
----------------------------+-----------------------------------------------------------------------
"SRID=4326;MULTILINESTRING((18.107890 45.819990 0.000000,18.073690 45.782520 0.000000,
 16.973320 47.816490 0.000000, .................... ,18.481400 13.274580 0.000000),(
 18.718690 11.780730 0.000000, .... ,22.013210 12.328680 0.000000))"
```

The maximum trajectory of the moving bird returned from the operation *mpt_max_trajectory* in Listing 6.5 is visualized in Figure 6.4.

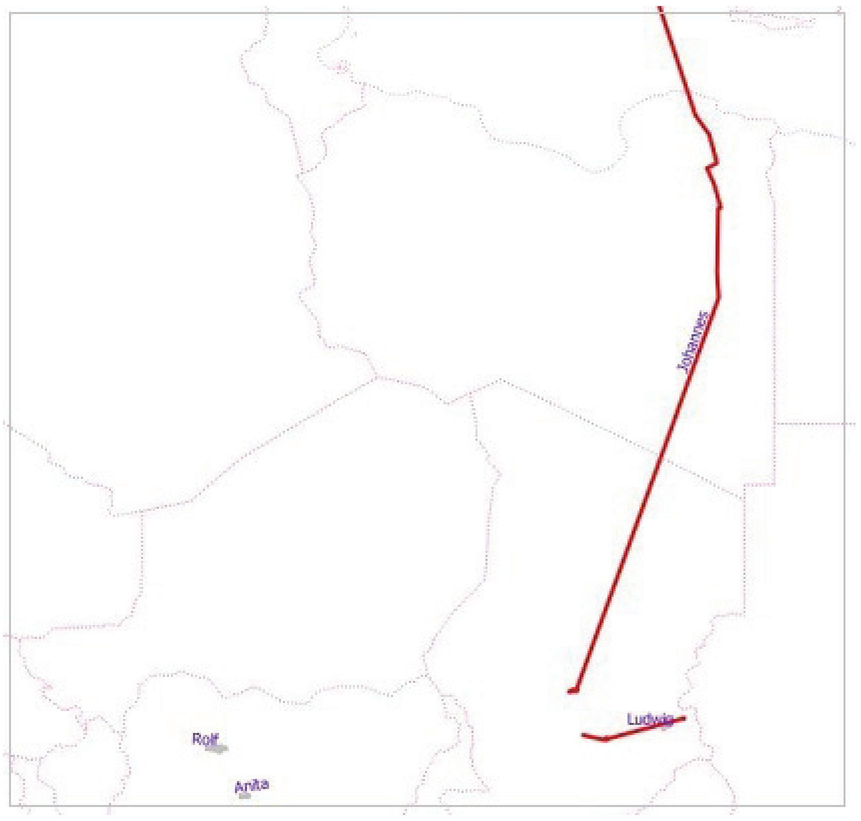


Figure 6.4: Maximum trajectory result from the operation mpt_max_trajectory

6.3.3 mpt_min_trajectory

The aggregation operation *mpt_min_trajectory* runs over a set of moving points, and returns the trajectory of the moving point that have a maximum distance covered. This operation follows a similar strategy as of the operation *mpt_max_trajectory*. It consists of the function *mpt_min_trajectory* as a state transition and *mpt_trajectory_final* as a final function. Listing 6.6 presents query statement that returns the trajectory of the moving bird with the minimum distance covered from the set of moving birds.

Listing 6.6: Return the minimum trajectory of the moving birds?

```
Query 8.

 SELECT mpt_trajectory_min(locations) from aggtest as mpt_min_trajectory_result;

Result 6.

mpt_min_trajectory_result
-----------------------------+---------------------------------------------------------------
"SRID=4326;MULTILINESTRING((7.507380 9.772630 0.000000,7.513720 9.775760 0.000000,7.420900
9.790020 0.000000, ........... ,7.563440 9.759180 0.000000,7.726490 9.789680 0.000000,
7.704990 9.802630 0.000000, ... ,7.694950 9.801070 0.000000),(7.727470 9.801400 0.000000,
........ 7.739270 9.808370 0.000000,7.735450 9.803660 0.000000))"
```

The minimum trajectory returned from the operation *mpt_min_trajectory* in Listing 6.6 is visualized in Figure 6.5.

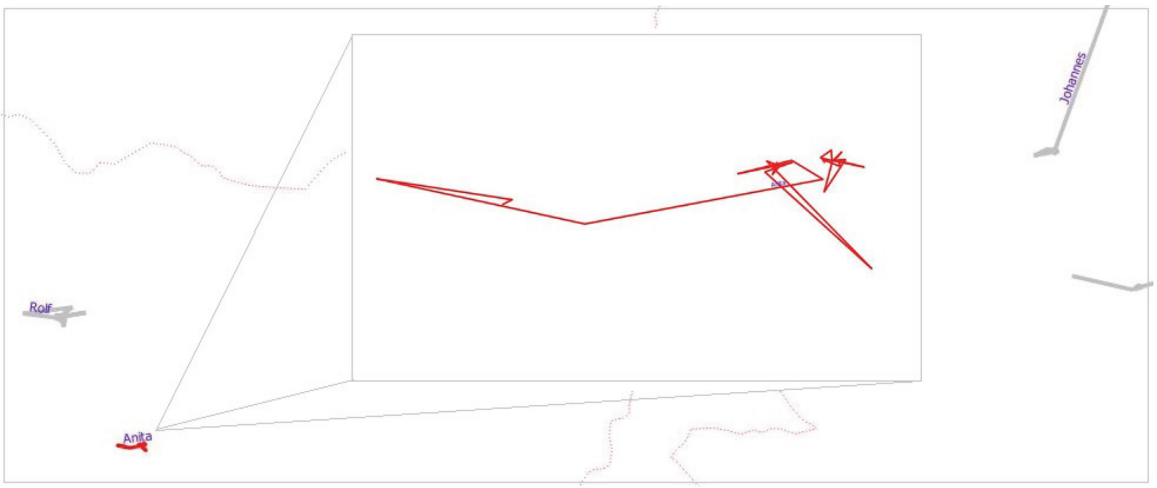


Figure 6.5: Minimum trajectory result from the operation mpt_max_trajectory

6.3.4 mpt_makeline

The aggregation operation *mpt_makeline* takes a set of moving points and returns the trajectory of every moving point as a single multilinestring. To implement the operation, every segment of each moving point is used to construct a single state moving point value. Each segment of the moving point on the set is represented by a line geometry which later is used to construct the multilinestring. The operation is composed of a state transition and final function. The state transition function *mpt_makeline* takes two arguments, a state value of the temporary type *mpoint_a* and the value of the type *mpoint* to be aggregated by the operation. A *null* value for the state type is passed as an argument for the first function call. For the rest of iterative function calls the constructed state value is used as argument. The function modifies the state value to include every segment of the input moving point and returns the modified state value for the next function call. The final function *mpt_trajectory_final* takes the final state value as an argument and returns the aggregated trajectory as a multilinestring. Listing 6.7 returns the trajectory of the moving birds using the operation *mpt_makeline*.

Listing 6.7: Return the total trajectory of the moving birds as a single geometry?

```
Query 8.

SELECT mpt_makeline(locations) from aggtest as mpt_makeline_result;

Result 6.

mpt_makeline_result
---------------------------------+---------------------------------------------------------------
"SRID=4326;MULTILINESTRING((7.507380 9.772630 0.000000,7.513720 9.775760 0.000000,7.420900
 9.790020 0.000000, .......... ,7.694950 9.801070 0.000000),(7.727470 9.801400 0.000000,
............ ,21.356290 12.010080 0.000000))"
```

The result of the query in Listing 6.7 is visualized in Figure 6.6.

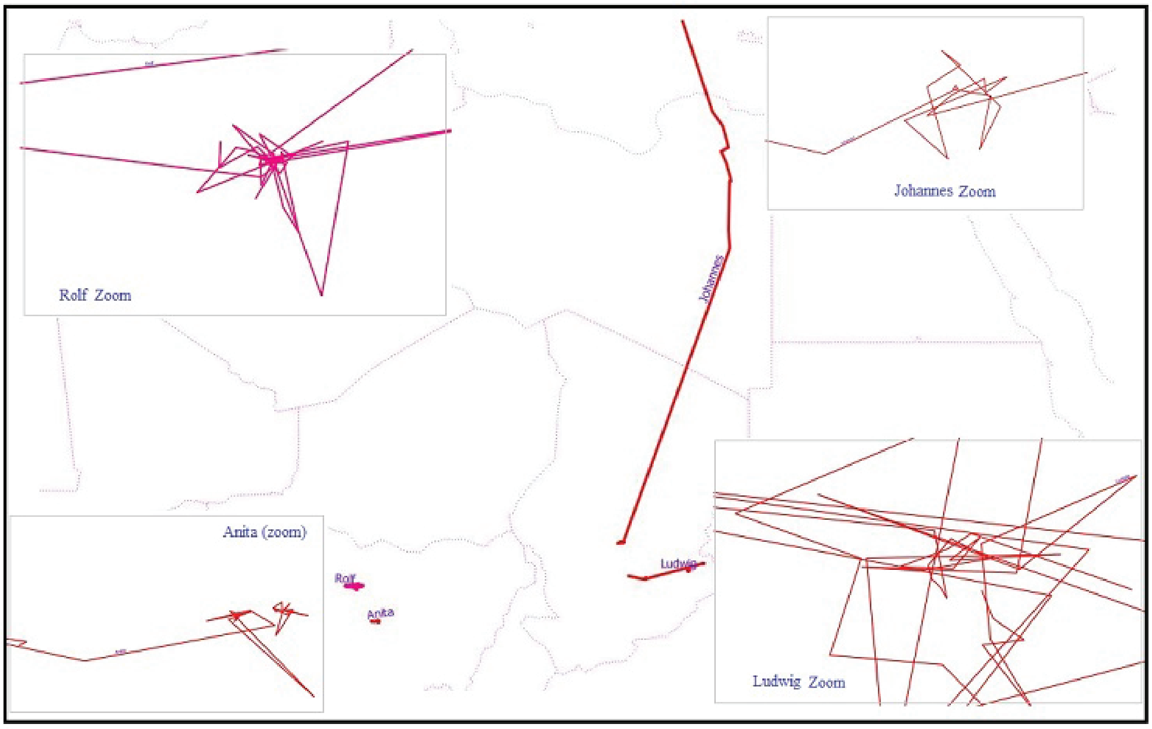


Figure 6.6: Trajectory result of the operation mpt_makeline

6.3.5 mpt_avg_speed

The operation *mpt_avg_speed* takes a moving point as an argument and returns the average speed as a result. The value returned is of type *double*. Therefore, any built-in aggregation function for base types can operate on the result. For example, to compute the average speed for a set of moving points, the built in *average* function of PostgreSQL can be used on top of the results from the operation *mpt_avg_speed*. The operations follows a similar strategy as the operation *mpt_speed*. The basic difference is that the speed of the moving point computed for every registered location is summed to the total speed variable, instead of constructing a moving real type. Simultaneously, the count value is also incremented by one. Finally, the operation returns the final speed by dividing the summed speed of the moving point by the count of the registered location. Listing 6.8 presents a query statement that returns the average speed of the moving birds "Johannes".

Listing 6.8: `What is the average speed of the bird "Johannes"?`

```
Query 8.

SELECT  ind_id ,mpt_avg_speed(locations) as average_speed
FROM aggtest
WHERE ind_id='Johannes ';

Result 6.

ind_id                  |average_speed
------------------------+----------------------------------------------------------
"Johannes"              |  2.29185
```

6.3.6 mpt_max_speed

The operation *mpt_max_speed* takes a moving point as an argument and returns the maximum speed as a result. The implementation of the operation follows a similar strategy with the operation *mpt_avg_speed*. The state variable max is initialized to zero and updated with the maximum speed, computed for each registered location of the moving point. Listing 6.9 presents a query statement that returns the maximum speed of the moving bird “Anita”.

Listing 6.9: What is the maximum speed of the bird "Anita"?

```
Query 8.

SELECT  ind_id ,mpt_max_speed(locations) as max_speed
FROM aggtest
WHERE ind_id='Anita ';

Result 6.

ind_id                  |max_speed
------------------------+----------------------------------------------------------
"Anita"                 |2.68961
```

6.3.7 mpt_min_speed

The operation *mpt_min_speed* takes a moving point as an argument and returns the minimum speed as a double value. Like the other operations, it initializes a variable minimum, to store the minimum speed of the moving point from the computed speed for each registered location. Listing 6.10 presents a query statement that returns the minimum speed of the moving bird “Anita”.

Listing 6.10: What is the minimum speed of the bird "Anita"?

```
Query 8.

SELECT  ind_id ,mpt_min_speed(locations) as min_speed
FROM aggtest
WHERE ind_id='Anita ';

Result 6.

ind_id                  |min_speed
------------------------+----------------------------------------------------------
"Anita"                 |0.00404072
```

6.4 CONCLUSION

Built-in aggregate functions perform a computation on a set of values to return a single value of the standard (base) type. In this chapter, aggregation functions that runs over a set of moving points are designed and implemented. All PostgreSQL aggregate functions require state transition functions and a state type. The optional final function is required for those aggregated functions that stores more than one state value or return a single value that does not have similar type with the base type. All aggregate functions other than the *mpt_centroid* designed and implemented in this chapter are deterministic. Inferring that the operation *mpt_centroid* could return different result based on the algorithm that has been selected. The operation returns a moving centroid with different number of (postion-time) pairs and segments for the three algorithms implemented. The "closure" algorithm always returns a single segment moving centroid independent of the input set of moving points. For the reset of the algorithms, the resulting number of segment and (position-time) pairs depends on the segment number and defined time intervals of the input moving points.

Chapter 7
Conclusion and Recommendation

7.1 INTRODUCTION

In this chapter, the achievements of this research which contributes in advance to the area of spatiotemporal database are reviewed. To achieve the predefined objectives defined in Chapter 1 several questions have been addressed. The research objectives and questions are reviewed in Section 7.2. In the final section, suggestions for further related works are presented.

7.2 CONCLUSION

The main objective of this research was to develop a moving object data type for a DBMS. This was achieved by developing a complete library package that extends PostgreSQL DBMS. To develop the moving object library the first objective was to critically examine the work of Bezaye [2013]. The study was mainly focused on identifying weakness of the research philosophy adopted and realization of the moving object library. Weakness of Bezaye [2013] and Eftekhar [2012] on the implementation of the moving object library has been depicted in Chapter 1. Similar to other related researches, motivated by current and upcoming needs of spatiotemporal databases, we mainly focused on the design and implementation of moving point data type as an extension to a DBMS.

The design part of moving point type was based on the theoretical foundations for moving objects presented by Güting et al. [2000]. From all the data models and abstractions provided by Güting et al. [2000], abstractions describing objects with time-dependent positional changes only have been used to represent moving objects. This allowed us to generate clearly defined abstract and discrete representations of moving point and four other supporting types such as: moving real, moving period, intime moving point, and intime moving real as presented in Chapter 2. Based on the theoretical and mathematical philosophies we studied, the data structure for moving point, moving real, moving period, intime moving point and intime moving real data types has been designed and realized. In addition, a valid textual representation and validation rule for all moving types realized in this research has been provided in Chapter 2 .

The implementation of the types was done as an extension to PostgreSQL DBMS and it is based on PostGIS library. To implement the types, platform and build methods have been selected in Chapter 3 taking previous and upcoming related researches into consideration. The first step in the implementation of the moving types was the development of the input and output functions. The development of the function used the textual representations and validation rules provided for moving point and its supporting types. The input and output functions are those responsible for storing the moving objects in memory and retrieve them. In total, 14 input and output functions have been developed and presented in Chapter 3. Only storing moving objects into the database was not sufficient enough to study the behavior of moving objects. The Second objective of the research was the implementation of operations on moving point data type. To achieve this objective, a set of useful operations are selected and designed from the unlimited number of operations available and feasible to undertake on spatiotemporal objects.

The selection was mainly based on set theory, first order logic and metric spaces. The theoretical foundation and design of the operation was based on Güting et al. [2000] which is depicted in Chapter 4. The algorithms used for the implementation of the operations were designed carefully based on algorithms on moving objects by Lema et al. [2003]. In total, 32 operations have been implemented to study the behaviors of moving objects. Those operations have implicit access to other 48 local function implementations that are not directly accessed through PostgreSQL SQL. The operations *mpt_atinstant* and *mpt_centroid* comes with three different calling interfaces representing different methods used for the same operation. Each of the interface provided for the operation *mpt_atinstant* allows to use the three different interpolation techniques that have been designed and implemented in this research. The interfaces for the operation *mpt_centroid* allows to use three different algorithms that are developed and realized to compute the centroid (average) for sets of moving points. Chapter 4 illustrated implementation of all the operations. This chapter also presented algorithms used for the implementation of selected operations. All the operations are tested on moving birds dataset and the results are depicted in the same chapter. Their practical use in real world application was also discussed and Visualization of the results was also provided for selected operations.

The third objective of the research was to provide alternative interpolation techniques. To achieve the objective, in addition to the linear interpolation which is common in most spatiotemporal databases; cubic and last known interpolation techniques are designed and implemented. The selection of interpolation techniques was motivated by several requirements of different real world application domains. The mathematical definitions and equations behind were discussed in Chapter 5. The implementation of the interpolation technique was achieved by implementing functions that are accessed through the operation *mpt_atinstant*. The results of the queries based on those interpolation techniques were also discussed and their drawback was presented in the same chapter.

Separate chapter was dedicated to discuss the design and implementation of aggregation operations that are used to query sets of moving points. The design and implementation of the operations in Chapter 6 allowed us to achieve the last objectives of the research. Selected aggregation functions from the list described briefly in Table 1.1 that are applied on a set of moving points are implemented. The algorithm and implementation detail used for the realization of the operations was discussed in the same chapter. The operation *mpt_centroid* which computes the centroid moving point between set of moving points was implemented to provide three different alternative calling interfaces. Those interfaces provides access to the three algorithms ("shared point", "closure" and "dormant point") used to compute the centroid. From the list in Table 1.1, operations *mpt_outlier* and *mpt_array* were not implemented since they can be obtained using few formal PostGIS operations. The aggregation operations were also tested and their result is presented in the same chapter. Visualization of results for selected aggregation operations was also provided.

In general, the research project was successful in the implementation of moving point type and operations on them. Furthermore, all initial expectations regarding the implementation of alternative interpolation functions and aggregating functions have been fulfilled entirely.

7.3 RECOMMENDATION

This research project was successful to contribute a library package that allowed to store moving objects in a DBMS and perform operations on them to the wider spatiotemporal database research area. However, our research still is the initial phase to the development of a complete library package that supports changes in both location and extent. In line with final products of the research, design and implementation of moving lines, moving regions and other abstract representations is highly advisable for further research work. Furthermore, all the design and implementation of the moving point data type and operations was based on moving points from three-dimensional spaces. We managed to include only the representation of the type in a two-dimensional space. One simple extension is to extend this research to support for operations on moving points from two dimensional spaces. In line with or independent of this research, infrastructure based interpolation method is potential extensions to the spatiotemporal databases. The operations were tested on real world moving object datasets. However, no performance analysis is done for the operations. Testing the performance of the operations and algorithms implemented for this research using larger datasets is one possible extension to this research.

Bibliography

Luis Otavio Alvares, Vania Bogorny, Bart Kuijpers, Jose Antonio Fernandes de Macedo, Bart Moelans, and Alejandro Vaisman. A model for enriching trajectories with semantic geographical information. In *Proceedings of the 15th annual ACM international symposium on Advances in geographic information systems*, page 22. ACM, 2007.

Ludger Becker, Henrik Blunck, Klaus Hinrichs, and Jan Vahrenhold. *A Framework for Representing Moving Objects*, volume 3180 of *Lecture Notes in Computer Science*, book section 82, pages 854–863. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-22936-0. doi: 10.1007/978-3-540-30075-5_82. URL `http://dx.doi.org/10.1007/978-3-540-30075-5_82`.

Tesfaye Bezaye. Embedding data types for continuously evolving objects in a dbms, 2013.

Victor Teixeira De Almeida and Ralf Hartmut Güting. Indexing the trajectories of moving objects in networks*. *GeoInformatica*, 9(1):33–60, 2005.

Afshin Eftekhar. Development of moving feature data types in a major dbms, 2012.

Martin Erwig, Ralf Hartmut Güting, Markus Schneider, Michalis Vazirgiannis, et al. Spatio-temporal data types: An approach to modeling and querying moving objects in databases. *GeoInformatica*, 3(3):269–296, 1999.

Luca Forlizzi, Ralf Hartmut Güting, Enrico Nardelli, and Markus Schneider. *A data model and data structures for moving objects databases*, volume 29. ACM, 2000.

Ralf Hartmut Güting, Michael H Böhlen, Martin Erwig, Christian S Jensen, Nikos A Lorentzos, Markus Schneider, and Michalis Vazirgiannis. A foundation for representing and querying moving objects. *ACM Transactions on Database Systems (TODS)*, 25(1):1–42, 2000.

Ralf Hartmut Güting, Thomas Behr, and Christian Düntgen. *SECONDO: A Platform for Moving Objects Database Research and for Publishing and Integrating Research Implementations.* Fernuniv., Fak. für Mathematik u. Informatik, 2010.

José Antonio Cotelo Lema, Luca Forlizzi, Ralf Hartmut Güting, Enrico Nardelli, and Markus Schneider. Algorithms for moving objects databases. *The Computer Journal*, 46(6):680–712, 2003.

Zhenhui Li, Jiawei Han, Ming Ji, Lu-An Tang, Yintao Yu, Bolin Ding, Jae-Gil Lee, and Roland Kays. Movemine: Mining moving object data for discovery of animal movement patterns. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 2(4):37, 2011.

Thomas Lockhart. Postgresql user's guide. *PostgreSQL Inc*, 2013.

Nikos Pelekis, Babis Theodoulidis, Ioannis Kopanakis, and Yannis Theodoridis. Literature review of spatio-temporal database models. *The Knowledge Engineering Review*, 19(03):235–274, 2004.

Markus Schneider. Moving objects in databases and gis: State-of-the-art and open problems. In *Research Trends in Geographic Information Science*, pages 169–187. Springer, 2009.

Timos Sellis. Research issues in spatio-temporal database systems. In *Advances in Spatial Databases*, pages 5–11. Springer, 1999.

Byunggu Yu and Seon Ho Kim. Interpolating and using most likely trajectories in moving-objects databases. In *Database and Expert Systems Applications*, pages 718–727. Springer, 2006.

Byunggu Yu, Seon Ho Kim, Thomas Bailey, and Ruben Gamboa. Curve-based representation of moving object trajectories. In *Database Engineering and Applications Symposium, 2004. IDEAS'04. Proceedings. International*, pages 419–425. IEEE, 2004.

Appendix A

Type and operation initialization queries

```
                    /* TYPE AND FUNCTION INITIALIZATION QUERY STATMENTS */

/* __________________________________________________________________________________ */
/* ____________________ MPOINT type for PostgreSQL STATMENTS _______________________ */
/* __________________________________________________________________________________ */

CREATE TYPE mpoint;

CREATE OR REPLACE FUNCTION mpoint_in(cstring)
  RETURNS mpoint AS
 'D:/postgis/postgisproject/Debug/PostgisRaster', 'mpoint_in'
LANGUAGE c IMMUTABLE STRICT;

CREATE OR REPLACE FUNCTION mpoint_out(mpoint)
  RETURNS cstring AS
 'D:/postgis/postgisproject/Debug/PostgisRaster', 'mpoint_out'
LANGUAGE c IMMUTABLE STRICT;

CREATE TYPE mpoint (
    input = mpoint_in,
    output = mpoint_out,
    alignment = double
);


/* __________________________________________________________________________________ */
/* _____________________ MREAL type for PostgreSQL STATMENTS _______________________ */
/* __________________________________________________________________________________ */

CREATE TYPE MMREAL;

CREATE OR REPLACE FUNCTION mreal_in(cstring)
  RETURNS MMREAL AS
 'D:/postgis/postgisproject/Debug/PostgisRaster', 'mreal_in'
LANGUAGE c IMMUTABLE STRICT;

CREATE OR REPLACE FUNCTION mreal_out(MMREAL)
  RETURNS cstring AS
 'D:/postgis/postgisproject/Debug/PostgisRaster', 'mreal_out'
LANGUAGE c IMMUTABLE STRICT;

CREATE TYPE MMREAL(
    input = mreal_in,
    output = mreal_out,
    alignment = double
);

/* __________________________________________________________________________________ */
/* ____________________ MPERIOD type for PostgreSQL STATMENTS ______________________ */
/* __________________________________________________________________________________ */

CREATE TYPE MPERIOD;

CREATE OR REPLACE FUNCTION mperiod_in(cstring)
  RETURNS MPERIOD AS
 'D:/postgis/postgisproject/Debug/PostgisRaster', 'mperiod_in'
```

```
LANGUAGE c IMMUTABLE STRICT;

CREATE OR REPLACE FUNCTION mperiod_out(MPERIOD)
  RETURNS cstring AS
'D:/postgis/postgisproject/Debug/PostgisRaster', 'mperiod_out'
LANGUAGE c IMMUTABLE STRICT;

CREATE TYPE MPERIOD(
    input = mperiod_in,
    output = mperiod_out,
    alignment = double
);

/* ______________________________________________________________________________ */
/* ____________Type INTIME_REAL type for PostgreSQL STATMENTS _____________________ */
/* ______________________________________________________________________________ */

CREATE TYPE INTIME_REAL;

CREATE OR REPLACE FUNCTION intime_in(cstring)
  RETURNS INTIME_REAL AS
'D:/postgis/postgisproject/Debug/PostgisRaster', 'intime_in'
LANGUAGE c IMMUTABLE STRICT;

CREATE OR REPLACE FUNCTION intime_out(INTIME_REAL)
  RETURNS cstring AS
'D:/postgis/postgisproject/Debug/PostgisRaster', 'intime_out'
LANGUAGE c IMMUTABLE STRICT;

CREATE TYPE INTIME_REAL(
    input = intime_in,
    output = intime_out,
    alignment = double
);

/* ______________________________________________________________________________ */
/* ___________Type INTIME_MPOINT type for PostgreSQL STATMENTS ____________________ */
/* ______________________________________________________________________________ */


CREATE TYPE INTIME_MPOINT;

CREATE OR REPLACE FUNCTION intime_mpoint_in(cstring)
  RETURNS INTIME_MPOINT AS
'D:/postgis/postgisproject/Debug/PostgisRaster', 'intime_mpoint_in'
LANGUAGE c IMMUTABLE STRICT;

CREATE OR REPLACE FUNCTION intime_mpoint_out(INTIME_MPOINT)
  RETURNS cstring AS
'D:/postgis/postgisproject/Debug/PostgisRaster', 'intime_mpoint_out'
LANGUAGE c IMMUTABLE STRICT;

CREATE TYPE INTIME_mpoint(
    input = intime_mpoint_in,
    output = intime_mpoint_out,
      alignment = double
);

/* ______________________________________________________________________________ */
/* ___________________mpt_deftime(mpoint) function SQL STATMENTS _________________ */
/* ______________________________________________________________________________ */

CREATE OR REPLACE FUNCTION mpt_deftime(mpoint)
  RETURNS cstring AS
'D:/postgis/postgisproject/Debug/PostgisRaster', 'mpt_deftime'
LANGUAGE c IMMUTABLE STRICT;

/* ______________________________________________________________________________ */
/* ___________________mpt_location(mpoint) function SQL STATMENTS ________________ */
/* ____________________________________________________________________________ */

CREATE OR REPLACE FUNCTION mpt_location(mpoint)
  RETURNS cstring AS
'D:/postgis/postgisproject/Debug/PostgisRaster', 'mpt_location'
LANGUAGE c IMMUTABLE STRICT;

/* ____________________________________________________________________________ */
/* ____________________mpt_srid(mpoint) function SQL STATMENTS ____________________ */
/* ____________________________________________________________________________ */

CREATE OR REPLACE FUNCTION mpt_srid(mpoint)
  RETURNS INT AS
'D:/postgis/postgisproject/Debug/PostgisRaster', 'mpt_srid'
LANGUAGE c IMMUTABLE STRICT;

/* ____________________________________________________________________________ */
/* ____________________mpt_tzid(mpoint) function SQL STATMENTS ____________________ */
/* ____________________________________________________________________________ */

CREATE OR REPLACE FUNCTION mpt_tzid(mpoint)
  RETURNS INT AS
'D:/postgis/postgisproject/Debug/PostgisRaster', 'mpt_tzid'
LANGUAGE c IMMUTABLE STRICT;

/* ____________________________________________________________________________ */
/* ___________function inst_intime_mpoint(INTIME_REAL) function SQL STATMENTS ______ */
/* ____________________________________________________________________________ */

CREATE OR REPLACE FUNCTION inst_intime_mpoint(CSTRING)
  RETURNS CSTRING AS
'D:/postgis/postgisproject/Debug/PostgisRaster', 'inst_intime_mpoint'
LANGUAGE c IMMUTABLE STRICT;

/* ____________________________________________________________________________ */
/* ___________function val_intime_mpoint(INTIME_REAL) function SQL STATMENTS _______ */
/* ____________________________________________________________________________ */

CREATE OR REPLACE FUNCTION val_intime_mpoint(CSTRING)
  RETURNS CSTRING AS
'D:/postgis/postgisproject/Debug/PostgisRaster', 'val_intime_mpoint'
LANGUAGE c IMMUTABLE STRICT;

/* ____________________________________________________________________________ */
/* ___________function val_intime_mreal(INTIME_REAL) function SQL STATMENTS ________ */
/* ____________________________________________________________________________ */

CREATE OR REPLACE FUNCTION val_intime_mreal(CSTRING)
  RETURNS CSTRING AS
'D:/postgis/postgisproject/Debug/PostgisRaster', 'val_intime_mreal'
LANGUAGE c IMMUTABLE STRICT;

/* ____________________________________________________________________________ */
/* ___________function inst_intime_mreal(INTIME_REAL) function SQL STATMENTS _______ */
/* ____________________________________________________________________________ */

CREATE OR REPLACE FUNCTION inst_intime_mreal(CSTRING)
  RETURNS CSTRING AS
'D:/postgis/postgisproject/Debug/PostgisRaster', 'inst_intime_mreal'
LANGUAGE c IMMUTABLE STRICT;

/* ____________________________________________________________________________ */
/* ___________function intime_inst(INTIME_REAL) function SQL STATMENTS ____________ */
/* ____________________________________________________________________________ */

CREATE OR REPLACE FUNCTION intime_inst(INTIME_REAL)
  RETURNS CSTRING AS
'D:/postgis/postgisproject/Debug/PostgisRaster', 'intime_inst'
LANGUAGE c IMMUTABLE STRICT;
```

```

/* ________________________________________________________________________________ */
/* ____________function intime_val(INTIME_REAL) function SQL STATMENTS ____________ */
/* ________________________________________________________________________________ */

CREATE OR REPLACE FUNCTION intime_val(INTIME_REAL)
RETURNS CSTRING AS
'D:/postgis/postgisproject/Debug/PostgisRaster', 'intime_val'
LANGUAGE c IMMUTABLE STRICT;

/* ________________________________________________________________________________ */
/* _________function MPT_ATINSTANT(mpoint) function SQL STATMENTS _________________ */
/* ________________________________________________________________________________ */

CREATE OR REPLACE FUNCTION mpt_atinstant(mpoint,cstring)
  RETURNS cstring AS
'D:/postgis/postgisproject/Debug/PostgisRaster', 'mpt_atinstant'
LANGUAGE c IMMUTABLE STRICT;

/* ________________________________________________________________________________ */
/* _________function MPT_ATINSTANT(mpoint) CUBIC function SQL STATMENTS ___________ */
/* ________________________________________________________________________________ */

CREATE OR REPLACE FUNCTION mpt_atinstant(mpoint,cstring,bool)
  RETURNS cstring AS
'D:/postgis/postgisproject/Debug/PostgisRaster', 'mpt_atinstant'
LANGUAGE c IMMUTABLE STRICT;

/* ________________________________________________________________________________ */
/* _________function MPT_INTIAL(mpoint) function SQL STATMENTS____________________ */
/* ________________________________________________________________________________ */

CREATE OR REPLACE FUNCTION mpt_initial(mpoint)
  RETURNS cstring AS
'D:/postgis/postgisproject/Debug/PostgisRaster', 'mpt_initial'
LANGUAGE c IMMUTABLE STRICT;

/* ________________________________________________________________________________ */
/* _________function MPT_FINAL(mpoint) function SQL STATMENTS _____________________ */
/* ________________________________________________________________________________ */

CREATE OR REPLACE FUNCTION mpt_final(mpoint)
  RETURNS cstring AS
'D:/postgis/postgisproject/Debug/PostgisRaster', 'mpt_final'
LANGUAGE c IMMUTABLE STRICT;

/* ________________________________________________________________________________ */
/* _________function PERIOD_FINAL(mpoint) function SQL STATMENTS __________________ */
/* ________________________________________________________________________________ */

CREATE OR REPLACE FUNCTION period_final(CSTRING)
  RETURNS cstring AS
'D:/postgis/postgisproject/Debug/PostgisRaster', 'period_final'
LANGUAGE c IMMUTABLE STRICT;

/* ________________________________________________________________________________ */
/* _________function PERIOD_INITIAL(mpoint) function SQL STATMENTS ________________ */
/* ________________________________________________________________________________ */

CREATE OR REPLACE FUNCTION period_initial(CSTRING)
  RETURNS cstring AS
'D:/postgis/postgisproject/Debug/PostgisRaster', 'period_initial'
LANGUAGE c IMMUTABLE STRICT;

/* ________________________________________________________________________________ */
/* _________function MREAL_FINAL(mpoint) function SQL STATMENTS ___________________ */
/* ________________________________________________________________________________ */

CREATE OR REPLACE FUNCTION mreal_final(CSTRING)
```

```
  RETURNS cstring AS
 'D:/postgis/postgisproject/Debug/PostgisRaster', 'mreal_final'
LANGUAGE c IMMUTABLE STRICT;

/* _____________________________________________________________________________ */
/* _________function MREAL_INITIAL(mpoint) function SQL STATMENTS __________________ */
/* _____________________________________________________________________________ */

CREATE OR REPLACE FUNCTION mreal_initial(CSTRING)
  RETURNS cstring AS
 'D:/postgis/postgisproject/Debug/PostgisRaster', 'mreal_initial'
LANGUAGE c IMMUTABLE STRICT;

/* _____________________________________________________________________________ */
/* _____function MPT_PRESENT_ATINSTANT(mpoint) function SQL STATMENTS ______________ */
/* _____________________________________________________________________________ */

CREATE OR REPLACE FUNCTION mpt_present_atinstant(mpoint, cstring)
  RETURNS bool AS
 'D:/postgis/postgisproject/Debug/PostgisRaster', 'mpt_present_atinstant'
LANGUAGE c IMMUTABLE STRICT;

/* _____________________________________________________________________________ */
/* _____function MPT_ATPERIOD(mpoint) function SQL STATMENTS ______________________ */
/* _____________________________________________________________________________ */

CREATE OR REPLACE FUNCTION mpt_atperiod(mpoint, cstring)
  RETURNS cstring AS
 'D:/postgis/postgisproject/Debug/PostgisRaster', 'mpt_atperiod'
LANGUAGE c IMMUTABLE STRICT;

/* _____________________________________________________________________________ */
/* _____function MPT_PRESENT_ATPERIOD(mpoint) function SQL STATMENTS _______________ */
/* _____________________________________________________________________________ */

CREATE OR REPLACE FUNCTION mpt_present_atperiod(mpoint, cstring)
  RETURNS bool AS
 'D:/postgis/postgisproject/Debug/PostgisRaster', 'mpt_present_atperiod'
LANGUAGE c IMMUTABLE STRICT;

/* _____________________________________________________________________________ */
/* _____function MPT_TRAJECTORIES(mpoint) function SQL STATMENTS __________________ */
/* _____________________________________________________________________________ */

CREATE OR REPLACE FUNCTION mpt_trajectories(mpoint)
  RETURNS CSTRING AS
 'D:/postgis/postgisproject/Debug/PostgisRaster', 'mpt_trajectories'
LANGUAGE c IMMUTABLE STRICT;

/* _____________________________________________________________________________ */
/* _________function MPT_DISTANCE(mpoint) function SQL STATMENTS __________________ */
/* _____________________________________________________________________________ */

CREATE OR REPLACE FUNCTION mpt_distance(mpoint)
  RETURNS cstring AS
 'D:/postgis/postgisproject/Debug/PostgisRaster', 'mpt_distance'
LANGUAGE c IMMUTABLE STRICT;

/* _____________________________________________________________________________ */
/* _________function MPT_SPEED(mpoint) function SQL STATMENTS ______________________ */
/* _____________________________________________________________________________ */

CREATE OR REPLACE FUNCTION mpt_speed(mpoint)
  RETURNS cstring AS
 'D:/postgis/postgisproject/Debug/PostgisRaster', 'mpt_speed'
LANGUAGE c IMMUTABLE STRICT;

/* _____________________________________________________________________________ */
/* _________function MPT_VALID(mpoint) function SQL STATMENTS ______________________ */
/* ___________________________________________________________________________ */

CREATE OR REPLACE FUNCTION mpt_valid(mpoint)
  RETURNS bool AS
'D:/postgis/postgisproject/Debug/PostgisRaster', 'mpt_valid'
LANGUAGE c IMMUTABLE STRICT;

/* ___________________________________________________________________________ */
/* _________function MPT_MREAL_VALID(mpoint) function SQL STATMENTS _______________ */
/* ___________________________________________________________________________ */

CREATE OR REPLACE FUNCTION mpt_mreal_valid(mpoint,mmreal)
  RETURNS bool AS
'D:/postgis/postgisproject/Debug/PostgisRaster', 'mpt_mreal_valid'
LANGUAGE c IMMUTABLE STRICT;

/* ___________________________________________________________________________ */
/* _________function MPT_TRANSFORM(mpoint) function SQL STATMENTS ________________ */
/* ___________________________________________________________________________ */

CREATE OR REPLACE FUNCTION mpt_transform(mpoint,integer)
  RETURNS CSTRING AS
'D:/postgis/postgisproject/Debug/PostgisRaster', 'mpt_transform'
LANGUAGE c IMMUTABLE STRICT;


/* ___________________________________________________________________________ */
/* ______ mpt_centroid AGGREGATE function for PostgreSQL STATMENTS _________________ */
/* ___________________________________________________________________________ */

/* Creation of the dummy mpoint type for mpt_centroid*/

CREATE TYPE mpointa;

CREATE OR REPLACE FUNCTION mpt_in(cstring)
  RETURNS mpointa AS
'D:/postgis/postgisproject/Debug/PostgisRaster', 'mpa_in'
LANGUAGE c IMMUTABLE STRICT;

CREATE OR REPLACE FUNCTION mpt_out(mpointa)
  RETURNS cstring AS
'D:/postgis/postgisproject/Debug/PostgisRaster', 'mpa_out'
LANGUAGE c IMMUTABLE STRICT;

CREATE TYPE mpointa (
    input = mpt_in,
    output = mpt_out,
    alignment = double
);


/* Initialization of state transition function and final function for aggregate function
     mpt_centroid */

CREATE OR REPLACE FUNCTION mpt_avg(mpointa,mpoint,bool)
  RETURNS mpointa AS
'D:/postgis/postgisproject/Debug/PostgisRaster', 'mpt_avg'
LANGUAGE c IMMUTABLE;

CREATE OR REPLACE FUNCTION mpt_avg(mpointa,mpoint)
  RETURNS mpointa AS
'D:/postgis/postgisproject/Debug/PostgisRaster', 'mpt_avg'
LANGUAGE c IMMUTABLE;

CREATE OR REPLACE FUNCTION mpt_avg_final(mpointa)
  RETURNS cstring AS
'D:/postgis/postgisproject/Debug/PostgisRaster', 'mpt_avg_final'
LANGUAGE c IMMUTABLE;

```

```

/* Creation of the Aggregate function mpt_centroid for moving points */

CREATE AGGREGATE mpt_centroid(mpoint,bool)(
SFUNC = mpt_avg,
STYPE = mpointa,
FINALFUNC = mpt_avg_final
);

CREATE AGGREGATE mpt_centroid(mpoint)(
SFUNC = mpt_avg,
STYPE = mpointa,
FINALFUNC = mpt_avg_final
);


/* ___________________________________________________________________________________ */
/* ______ mpt_makeline AGGREGATE function for PostgreSQL STATMENTS _________________ */
/* ___________________________________________________________________________________ */

/* Initialization of state transition function and final function for aggregate function
     mpt_centroid */

CREATE OR REPLACE FUNCTION mpt_makeline_final(mpointa)
  RETURNS cstring AS
 'D:/postgis/postgisproject/Debug/PostgisRaster', 'mpt_makeline_final'
LANGUAGE c IMMUTABLE;


CREATE OR REPLACE FUNCTION mpt_makeline(mpointa,mpoint)
  RETURNS mpointa AS
 'D:/postgis/postgisproject/Debug/PostgisRaster', 'mpt_makeline'
LANGUAGE c IMMUTABLE;

/* Creation of the Aggregate function mpt_centroid for moving points */



CREATE AGGREGATE mpt_makeline(mpoint)(
SFUNC = mpt_makeline,
STYPE = mpointa,
FINALFUNC = mpt_makeline_final
);

/* ___________________________________________________________________________________ */
/* ______ mpt_trajectory_min AGGREGATE function for PostgreSQL STATMENTS _________________
    */
/* ___________________________________________________________________________________ */


/* Initialization of state transition function and final function for aggregate function
     mpt_centroid */



CREATE OR REPLACE FUNCTION mpt_makeline_final(mpointa)
  RETURNS cstring AS
 'D:/postgis/postgisproject/Debug/PostgisRaster', 'mpt_makeline_final'
LANGUAGE c IMMUTABLE;


CREATE OR REPLACE FUNCTION mpt_min_tajectory(mpointa,mpoint)
  RETURNS mpointa AS
 'D:/postgis/postgisproject/Debug/PostgisRaster', 'mpt_min_tajectory'
LANGUAGE c IMMUTABLE;

/* Creation of the Aggregate function mpt_centroid for moving points */


```

```

CREATE AGGREGATE mpt_trajectory_min(
SFUNC = mpt_min_tajectory,
BASETYPE=mpoint,
STYPE = mpointa,
FINALFUNC = mpt_makeline_final
);


/* ____________________________________________________________________________ */
/* ______ mpt_max_tajectory AGGREGATE function for PostgreSQL STATMENTS ________________ */
/* ____________________________________________________________________________ */


/* Initialization of state transition function and final function for aggregate function mpt_centroid */



CREATE OR REPLACE FUNCTION mpt_makeline_final(mpointa)
  RETURNS cstring AS
'D:/postgis/postgisproject/Debug/PostgisRaster', 'mpt_makeline_final'
LANGUAGE c IMMUTABLE;


CREATE OR REPLACE FUNCTION mpt_max_tajectory(mpointa,mpoint)
  RETURNS mpointa AS
'D:/postgis/postgisproject/Debug/PostgisRaster', 'mpt_max_tajectory'
LANGUAGE c IMMUTABLE;

/* Creation of the Aggregate function mpt_centroid for moving points */



CREATE AGGREGATE mpt_trajectory_max(
SFUNC = mpt_max_tajectory,
BASETYPE=mpoint,
STYPE = mpointa,
FINALFUNC = mpt_makeline_final
);


/* ____________________________________________________________________________ */
/* _________function MPT_AVG_SPEED(mpoint) function SQL STATMENTS _____________________ */
/* ____________________________________________________________________________ */

CREATE OR REPLACE FUNCTION mpt_avg_speed(mpoint)
  RETURNS real AS
'D:/postgis/postgisproject/Debug/PostgisRaster', 'mpt_avg_speed'
LANGUAGE c IMMUTABLE STRICT;



/* ____________________________________________________________________________ */
/* _________function MPT_MAX_SPEED(mpoint) function SQL STATMENTS _____________________ */
/* ____________________________________________________________________________ */

CREATE OR REPLACE FUNCTION mpt_max_speed(mpoint)
  RETURNS real AS
'D:/postgis/postgisproject/Debug/PostgisRaster', 'mpt_max_speed'
LANGUAGE c IMMUTABLE STRICT;


/* ____________________________________________________________________________ */
/* _________function MAX_MAX_SPEED(mpoint) function SQL STATMENTS _____________________ */
/* ------------------------------------------------------------------------------- */

CREATE OR REPLACE FUNCTION mpt_min_speed(mpoint)
  RETURNS real AS
'D:/postgis/postgisproject/Debug/PostgisRaster', 'mpt_min_speed'
LANGUAGE c IMMUTABLE STRICT;
```