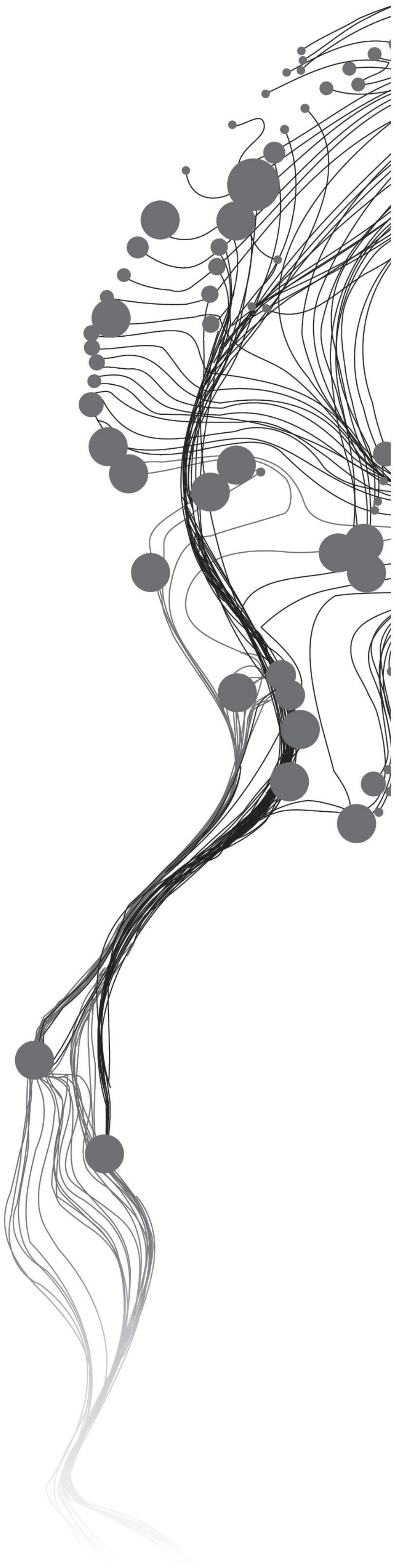
EMBEDDING DATA TYPES FOR CONTINUOUSLY EVOLVING OBJECTS IN A DBMS

BEZAYE TESFAYE
April, 2013

SUPERVISORS:

Dr.Ir. R.A. de By
Dr. Javier Morales



EMBEDDING DATA TYPES FOR CONTINUOUSLY EVOLVING OBJECTS IN A DBMS

BEZAYE TESFAYE
Enschede, The Netherlands, April, 2013

ABSTRACT

Recent researches in moving object databases are attempting to achieve the support for movement information in DBMS. But the lack of support for handling such information is still an issue in current DBMSs, which becomes a barrier in processing and analyzing movement information. With the advancement of telemetry technology, where different devices are producing vast amount of movement data, the need of fulfilling this limitation is becoming bigger and bigger. This makes the research area of moving objects to become more important to deals with representation of these huge amount of data coming from different devices. This thesis project aims to provide a refined abstract and discrete design for the representation of moving point in current DBMS. The available models are reviewed and used as a starting point. First the abstract design is done followed by the discrete design. The discrete design paves the way for the implementation of the data type and operations on it, which can be used as an extension for DBMSs.

Keywords

Spatio-temporal, Databases, Moving objects, PostgreSQL, PostGIS

ACKNOWLEDGEMENTS

Above all, I want to thank God for making me strong with his love whenever I feel weak and for everything he did to me.

I would like to thank my supervisors Dr.Ir. Rolf de By and Dr. Javier Morales. Specially my first supervisor for proposing this research title in the department. He was always giving me valuable feedback for every piece of work I did and it was a steeping stone for me to finish this thesis. So I would like to offer my utmost gratitude to him.

I can not pass with out thanking Ms. Connie Blok for her always kind support at the time I need help from the department.

I also want to thank my friends Loza, Effie, Sharmin and Tigist. Their caring and support through out the thesis time is unforgettable for me. I owe my deepest gratitude for Nuno Cesar de Sa for sharing all my good and bad times and supporting me every single day.

I would also like to thank my family, specially my mom whom I learned how to be a strong and confident woman. Thank you ETABEYE for everything you are to me.

At last, I would like to thank NUFFIC for offering me a fellowship to pursue this MSc.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1
Introduction

1.1 GENERAL

The need for storing and manipulating changes of real world objects in a DBMS has been a recent research area in spatiotemporal databases. The research area was first addressed in the second half of 1990s (Güting & Schneider, 2005). It was first started by carrying out researches in spatial and temporal databases separately. The reason for this separation was the high complexity of data structures that needed careful analysis so as to structure dimensions with the representation of data (Pelekis et al., 2005). In the next sections of this chapter, we review the literature and discuss the problems of current DBMS in handling spatiotemporal data, the main objectives of this thesis project towards solving such problems and the method followed to accomplish objectives.

1.2 LITERATURE REVIEW

Since the idea of integrating spatial and temporal aspects in a database emerged, many spatiotemporal data models have been proposed (Pelekis et al., 2005). However, as mentioned in (Güting, Behr, & Düntgen, 2010), most models are not implemented, and are rather left in paper only.

The first models proposed on spatiotemporal data focused on representation of discrete change of spatial objects. This was a big limitation for representing real world objects that change their geometries continuously. Later on, (Erwig et al., 1999) proposed a model for the first time to tackle this problem. The model considers time as integral part of spatial entities, while the other models use timestamps (Pelekis et al., 2005). Later Parent, Spaccapietra, and Zimányi (1999) proposed a spatiotemporal conceptual model and based on it, they design abstract and discrete design in Forlizzi, Güting, Nardelli, and Schneider (1999). Furthermore, orientation and direction of spatial objects during their movement and all spatial data handling mechanisms are only considered in the models proposed by Parent et al. (1999); Erwig et al. (1999) (Pelekis et al., 2005). Apart from this, when we consider representing continuous change, most of the approaches cannot cover all real world processes because of the limited operation set they provide. The only approach that resolves many kinds of spatiotemporal queries is the moving object data model proposed by Erwig et al. (1999) (Pelekis et al., 2005). This model is also implemented for testing the mathematical definitions it contains (Güting et al., 2010) in which they discussed the conceptual model in Tryfona and Hadzilacos (1998).

1.3 PROBLEM DEFINITION

Although most DBMS providers support spatial and temporal extensions so as to store geometries and time separately, analyzing the changes in spatiotemporal object is not possible by separated spatial and temporal aspects. Nowadays, information generated from GPS equipped devices like PDAs (Personal Digital Assistants), cell phones and sensors is creating a huge amount of movement and change information. Besides, the rapid development of such communication devices which made sharing of spatial information among people easy, it also made this data more complex to be handled by current databases. We may need to know the location at time t of a person cycling to one supermarket or we might be interested to know the time when there is a huge traffic jam in the Hengelosestraat in order to know the relationship of the accident that happened on that day with respect to the traffic jam. One can also ask, what is the minimum and maximum extent of Lake Hawassa during the year? To analyze such information, we have to store movement/change information of real world objects in database systems (Schneider, 2009), but the database and GIS technologies we are using currently are not able to perform this task. So, we need new query techniques for managing and processing such information and there must be a capability in current DBMS for handling objects with continuously time varying location data.

To handle movement of spatial objects over time in databases, many research projects have been carried out and different data models have been proposed. But most of them are not implemented. Those that are implemented are only suitable for specialized systems so as to experiment and measure (Güting et al., 2010) and there is no model implemented fully in a well-known DBMS until now. Furthermore, applying data mining techniques for analyzing information of spatio-temporal objects is difficult using the current DBMS. This can be solved when we start storing these objects in a database.

The change in geometries of spatiotemporal objects can be discrete or continuous. If we take spatiotemporal objects that move continuously with time, they are called moving objects (Güting et al., 2000; Pelekis et al., 2005). To model such objects, Güting et al. (2000) proposed two main abstractions: moving point and moving region. In the moving point abstraction, only the time-dependent position is of interest. If we want to know which satellite will be close to the route of some spacecraft in the next two hours, we are dealing with objects represented by moving points. In the case of moving region, not only the position is of interest but also the extent, and whether it can shrink or expand. To answer a historical question like “ what was the largest extent of Axumite empire in Ethiopia?” we need to have a moving region data type in our database that can store such kind of objects. Furthermore, if we think of a moving point which is projected onto a plane, yielding a curve and the boundary of a moving region where the linear extent can be lengthened and shortened with time, they can be considered as moving line (Schneider, 2009; Cotelo Lema, Forlizzi, Güting, Nardelli, & Schneider, 2003) (see Figure 1.1). We can think of the distance chord between two moving points over time as a moving line.

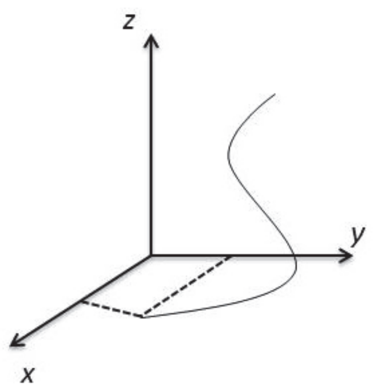


Figure 1.1: A moving point in xyz plane

Generally, time and space are important issues for representing phenomena of the real world and database applications should have the ability of representing real world objects. So, extending database technology for representation of such objects is needed. This research project aims to address this lack of support for storing and querying moving objects in one well-known DBMS.

1.4 RESEARCH OBJECTIVES

The aim of this research project is to implement a data type for spatiotemporal objects in a well-known DBMS. To achieve this, the main issues that are addressed are listed in the research objectives and research questions below.

1. To develop a refinement mechanism for the mathematical definitions proposed by Güting et al. (2000) for moving objects.
2. To develop abstract data type for a moving object, and realize these in a DBMS.
3. To implement operations on a moving object data type in a DBMS.

1.5 RESEARCH QUESTIONS

1. How can the mathematical design proposed by Güting et al. (2000) for moving objects be converted into implementable design?
2. How can we create abstract data types for moving objects by using standard data types?
3. How can we represent abstract data types of moving objects discretely?
4. What are the basic requirements of the data structure for abstract data types of moving objects?
5. How can operations of moving object types be defined using existing operations of non-temporal types?
6. What are the basic operations that can be feasibly implemented on spatiotemporal objects in a DBMS?

1.6 METHODOLOGY

The initial phase of this research project starts from problem definitions, and then specifications of the necessary features of the system follow (see Figure 1.2). This helps us as a stepping stone towards finding solutions for the problems.

The design phase is the vital stage that needs careful work. In this phase, suitable solutions are mentioned. To address the objectives of this project, a mathematical model that fits is selected. Based on the comparisons on spatiotemporal models made by Pelekis et al. (2005), apart from becoming a better choice in terms of temporal, spatial and spatiotemporal semantics, the moving object model proposed by Güting et al. (2000) is the only model with full set of operations for querying movement and change of real world objects. The operations are also tested through implementation (Cotelo Lema et al., 2003). Having all the necessary representations of spatiotemporal objects in its mathematical definitions made the work of Güting et al. (2000) as the best candidate model to start with for the implementation of spatiotemporal types in a DBMS.

Later in the design phase, the moving object model goes through refinement by converting the mathematical definitions into set of steps that are ready for implementation, and this is used as a base for developing the abstract data types.

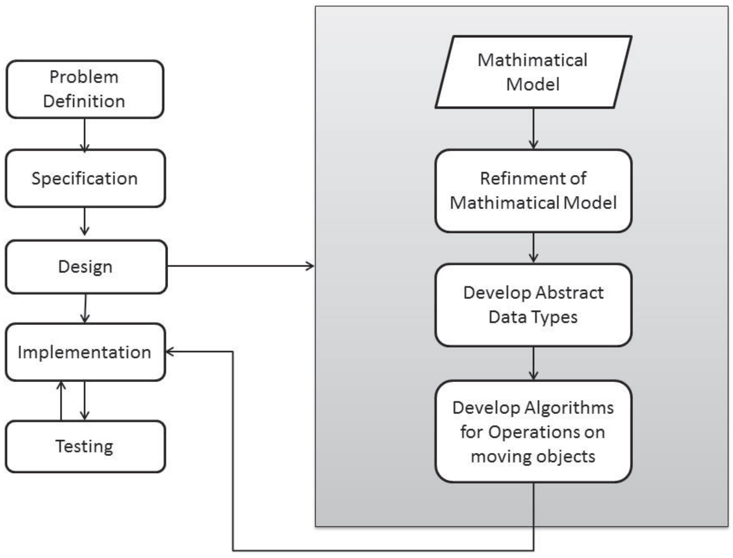

Problem Definition
Specification
Design
Implementation
Testing
Mathimatical Model
Refinment of Mathimatical Model
Develop Abstract Data Types
Develop Algorithms for Operations on moving objects

Figure 1.2: Development Methodology

While dealing with abstract data types, one of the most important issues is the consideration of data structures. Since these data types and operations are embedded in a DBMS, these values are stored in the memory space controlled by the DBMS. As a result of this, the data structure should use up a minimum of memory blocks (Cotelo Lema et al., 2003).

To develop executable algorithms for the basic operations on the abstract data types, the refined set of steps from the mathematical definitions and the general algorithms for operations on moving point and moving region, designed by Cotelo Lema et al. (2003), will be used.

In the implementation phase, the executable algorithms for the selected operations are coded in a high-level programming language. Throughout the coding process, continuous testing is done. This avoid misuse of incorrect blocks of code that is used as a template to develop another block.

Chapter 2
Representation of moving points

2.1 INTRODUCTION

The concept of representing moving objects in a database system is important for many potential applications. Currently, famous DBMS softwares like Microsoft SQL server, Oracle and Postgresql don't have support for such kind of objects. Because of this lack of capabilities in current DBMSs, makes that potential applications are not developed.

In order to represent objects that change their location through time in the existing DBMS, (we call them 'moving objects') the design of abstract data types contains two levels of abstractions. In the first level, all the structures of the types which are used in the representation are defined. For this purpose, we define all these types in a type system. In the second level, the semantics of operations on moving types are defined. For example, how do we represent a trajectory of a moving point?

In this chapter, amongst the set of moving types, only moving point and moving real are considered and the representation of these types is done from abstract to discrete. The abstract representation has two levels of abstraction specified above and the discrete design follows the method of representing time varying information of real world object into a format that a computer can store. So in the next section of this chapter, the abstract design is first explained and the detail of discrete representation follows.

2.2 ABSTRACT REPRESENTATION OF A MOVING POINT

2.2.1 The approach

For a design of abstract data type for a moving point, there are different auxiliary types that are needed. For example, if we need to project a moving point onto the *xyz* plane, we need a *linestring* or *multilinestring*, or if we want to know the time-dependent speed of a moving point, we need a moving real. So, in the design of the abstract type for a moving point, first we defined types that are essential, then we define the abstract data type by using existing types that we already defined. The types that we use are in Table 2.1 applying the same approach as Güting et al. (2000) to define them.

Table 2.1: Type system

Type	Signature
$int, real, string, bool$	$\rightarrow BASE$
$point, multipoints, linestring, multilinestring$	$\rightarrow SPATIAL$
$instant$	$\rightarrow TIME$
$mpoint, mreal$	$BASE \cup SPATIAL \rightarrow TEMPORAL$

2.2.2 Base types and time type

Base types: All the base types have the same regular definition except that they all are augmented by a value which is undefined (Güting et al., 2000).

Definition 2.2.1. If α is any base type, then its carrier set is symbolized by A_α. Then the carrier sets for the base types: *int, real* and *string* are:
$A_{int} := \mathbb{Z} \cup \{\perp\}$,
$A_{real} := \mathbb{R} \cup \{\perp\}$,
$A_{string} := \Sigma^* \cup \{\perp\}$, where Σ is a finite set of alphabet,
$A_{bool} := \{FALSE, TRUE\} \cup \{\perp\}$,

Time type: To represent temporal values, we use time type *instant* which is for this purpose. For the definition of *instant*, we consider time as a linear continuous attribute. This is because our design is intended for existing types, which we can use to construct an abstract data type to represent an object with continuous time varying information.

Definition 2.2.2. The $real$ numbers are used to represent *instant* values, together with an undefined value. The carrier set is defined as:

$$A_{instant} := \mathbb{R} \cup \{\perp\}$$

2.2.3 Spatial types

We define spatial types based on the scheme used in Güting et al. (2000) and OGC (1999).

Point:

is a zero-dimensional geometry, used to represent a single location in space (OGC, 1999). The values of the coordinates are a real values.

Definition 2.2.3. The carrier set of point is defined as:

$$A_{point} := \mathbb{R}^3 \cup \{\perp\}$$

MultiPoint:

is a zero-dimensional geometry collection of which the elements are only *points* (OGC, 1999).

Definition 2.2.4. The carrier set of MultiPoint is :

$$A_{multipoint} := \{P \subseteq \mathbb{R}^3 | P \text{is finite}\}$$

Linestring:

is a one-dimensional geometry containing a sequence of points within a real closed interval (OGC, 1999).

Definition 2.2.5. A linestring is a continuous mapping of $f : [a, b] \to \mathbb{R}^3$ such that

$$\forall a, b \in \mathbb{R}, \forall x \in \mathbb{R} : a <= x <= b \Longrightarrow x \in [a, b]$$

It is called *simple* if it does n't pass a point more than once, which means: $\forall x, y \in [a, b] : \nexists z \in \mathbb{R}^3$ such that $f(x) = z = f(y)$

MultiLinestring

is a one-dimensional geometry collection of which elements are only linestrings.

2.2.4 Temporal types

To derive temporal types using the BASE and SPATIAL types, type constructors are used. The *moving* type constructor is mainly used for this purpose. For any given type α, the *moving* type constructor , applied to α, yields a mapping from time to α (Güting et al., 2000).

Definition 2.2.6. For any BASE or SPATIAL type α, the derived temporal type will be *moving(α)*. The semantics is defined by the carrier set as:

$$A_{moving} := \{f \mid f : \bar{A}_{instant} \rightarrow \bar{A}_{\alpha} \text{ is a partial function} \wedge \Gamma(f) \text{ is finite}\}$$

The condition "$\Gamma(f)$ is finite" indicates that f only contains a finite number of continuous components (Güting et al., 2000). A temporal type obtained through the moving constructor is described by infinite sets of pairs (instant,value) or a function.

Moving point

One of the temporal types that is derived by using the *moving* type constructor is $moving(point)$. It is used to represent an object that changes position through time. For example, a car, a pedestrian, a bird, etc. So for this temporal object of type *moving(point)*, or in short *mpoint*, the values can be partitioned along the time domain into continuous finite pieces as mentioned above. Each component is defined by a single time interval in which the object's value is described continuously. We call such a component a 'SEGMENT'.

Definition 2.2.7. The time interval, which we call '*period*' from here onwards, that any 'SEGMENT' defines, is described as a set such that:

$$period \subseteq \bar{A}_{instant} \text{ such that } \forall s, e \in period, \forall t \in \bar{A}_{instant} : s < t < e \Rightarrow t \in period$$

Definition 2.2.8. The temporal domain of an *mpoint* contains a set of time intervals for which its segments are defined as: let mp $\in A_{moving(\alpha)}$where $\alpha = point$, then

$$temp\_domain(mp) = \{period \subseteq \bar{A}_{instant}\}$$

Moving real

A moving real, in short *mreal* is a temporal type that is used to represent a sequence of continuously changing time varying real values. We use this type as a basic supplementary type for *mpoint*. For example, if we want to know the distance between any two *mpoints* or if we want to know the speed of an *mpoint*, the result is *mreal*. So the representation of such a type is important for the design.

2.3 DISCRETE REPRESENTATION OF MOVING POINTS

2.3.1 How do we represent a moving point discretely

Let us assume that a person with a GPS traveled from location A to location B for three hours. Figure 2.1 shows the path that the *moving point* followed.

The movement from A to B is considered as a continuous movement, in which the object traveled in a continuous change of location through time. To create a data structure for this segment,

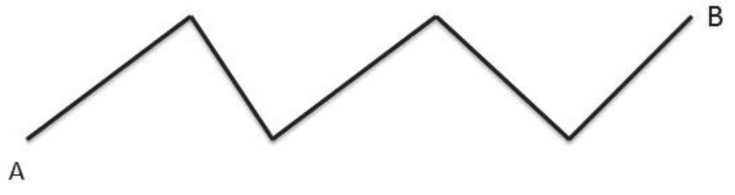


Figure 2.1: A moving object traveling from A to B

since we can't store continuous data in computer, we will create a discrete representation of the continuous movement for the moving object by using track points obtained from GPS equipped devices. Considering continuous movement of the object, to get the value of the moving point at the time that we don't have stored information, some form of linear interpolation is used. The proposed idea for the discrete representation is to take GPS tracks of the object within some time interval (for example setting the GPS to track the object every two seconds) and use a function that can do interpolation to get values that are not stored within that time interval during which the moving point is defined. At the end, we will have an array of records for the GPS tracks containing *x*, *y* coordinates and *time*.

For example, if a person wants to know the location of the point at time t_1 which is within the time interval of a segment, then a function will be defined on the moving point type to search from the first array record, check the time through the array until it gets the time that is greater than or equal to the time of the argument (t_1). While searching, if the current array index contains the time greater than t_1, then the function will take the *x* and *y* coordinates registered for that array index and for the previous array index, and then apply linear interpolation using the two coordinate pairs resulting in the location of the moving point at the required time. But if the current array index contains the same time value as the required time, then the function will automatically return the coordinates from the record at that array index.

While a moving point is traveling from *A* to *D*, as illustrated in Figure 2.2, we realize that a location for which the moving point is not defined may exist. For example, between *B* and *C*. For such case we will have multiple segments: one from *A* to *B* and another from *C* to *D*. But we have to keep in mind that the moving point travels from *A* to *D*. The data structure will then have two segments. Whenever there is a need of finding the value of the moving point at a specific time, a function will be available that searches for the correct value or correct pair of points for interpolation within the segments. Below, the data structure and further description of the data structure is discussed. We will use *mpoint* as a name for moving point type afterwards.

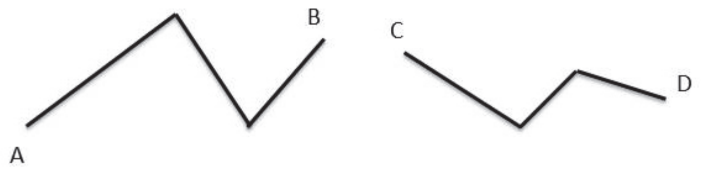
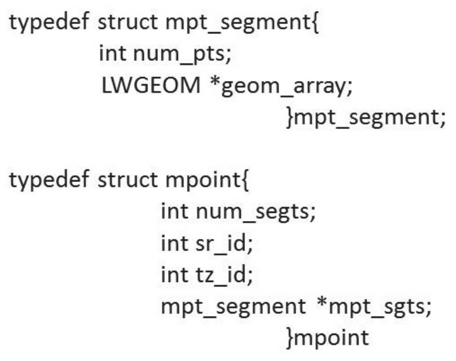


Figure 2.2: A moving object traveling from A to D

2.3.2 Data structure

To implement a data type that represents a moving point, the structure defined below in Figure 2.3 is used.



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

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

Figure 2.3: A data structure for mpoint

Each *mpoint* in the structure is represented by a record containing num_segts, sr_id, tz_id and mpt_sgts, which are defined as:

- num_segts: the number of segments the *mpoint* contains.
- sr_id: spatial reference identifier.
- tz_id: time zone identifier.
- mpt_sgts: a pointer to an array of mpt_segment which is represented as a record containing num_pts and geom_array

The main advantage of mpt_segment is to represent information of a single *mpoint* within a specific time interval. For example as in Figure 2.2, if the object is traveling from location *A* to *B*, and then from location *C* to *D*, the representation of this moving object will have two segments. The first segment contains information for the path from *A* to *B*, and the second containing information from *C* to *D*.

Inside each mpt_segment, num_pts represent the number of point geometries that are included in the segment and geom_array represent a pointer to the array of geometries. The geometry type for the array referenced by geom_array is called LWGEOM (Light weight geometry), defined by POSTGIS. It is the working structure of POSTGIS containing the type, flags, srid and bbox. Type indicate the type of the geometry (POINT, LINE, etc). So, if the geometry is POINT, then the type for the LWGEOM will have POINT type. Flags is used to indicate weather Z or M dimensions are present, these are considered as the third and the fourth dimension of the geometry, respectively as specified in OGC[add cite]. The bbox shows the extent of the geometry. Each LWGEOM also has a spatial reference identifier (srid) which is embedded with it.

For the data structure of *mpoint*, the array of LWGEOM, referenced by geom_array contains POINT type geometries. Each point is represented as a record containing: *x-coordinate, y-coordinate, z-coordinate and time instant* as the fourth dimension in which the M dimension specified in POSTGIS is used to represent time.

The spatial reference identifier is embedded with LWGEOM as well as in *mpoint* structure because each *mpoint* should contain one spatial reference identifier and each POSTGIS LWGEOM also needs the same, meaning that we can't change the structure, and the duplication of spatial reference identifier will remain.

2.3.3 Constraints on the moving point type

- c_mpt_segment:
    - c_num_pts: each num_pts should have the same value as the length of the geometry array which is referenced by geom_array. This is because since the advantage of the num_pts is to store how many points are included inside the segment, it has to match with the array size containing elements of LWGEOM type which is referenced by geom_array.
    - c_geom_array: the array of LWGEOM must be arranged in chronological order using the fourth dimension of each element (time instant as explained earlier).
- c_mpoint:
    - c_1_num_segts: num_segts must have the same value as the length of the mpt_segment array which is referenced by mpt_sgts.
    - c_2_num_segts: num_segts inside the mpoint is always an integer greater than or equal to zero.
    - c_1_sr_id: inside *mpoint*, the value of the spatial identifier (sr_id) is an existing primary key in spatial reference table inside PostgreSQL.
    - c_1_tz_id: the time zone identifier (tz_id) must be an integer which represents a known time zone identifier stated in tz database which contain the world's time zones.
    - c_1_mpt_sgts: elements of the array of mpt_segment pointed by mpt_sgts must be in chronological order, based on the time interval of each mpt_segment. The time interval can be realized in such a way that the start and end time of the segment are the fourth dimension of the first and last array element that is pointed by geom_array inside the segment respectively.
    - c_2_mpt_sgts: each consecutive mpt_segment inside the array pointed to by mpt_sgts must not overlap
    - c_3_mpt_sgts: in case of two consecutive segments where end time of the first segment and the start time of the second segment is the same, then the end time of the first segment will no longer be part of the time interval of the first segment and will be inclusive as the start of the second segment (see figure below). This will allow us to maintain disjointness between consecutive segments.
    - c_4_mpt_sgts: any element of the array of mpt_segment which is pointed by mpt_sgts cannot contain a pointer to an array of LWGEOM having zero length.

2.3.4 Language construct for a moving point type

The language construct for *mpoint* which is defined below is based on the notation of OGC Well Known Text representation of geometry.

```
<Mpoint Tagged Text> :=
                              <Srid Tagged Text> ;
                              <Tzid Tagged Text> ;
                              MPOINT <Mpoint Text>
<Srid Tagged Text> :=
                           SRID <Srid Text>
<Tzid Tagged Text> :=
```

TZID <Tzid Text>
<Srid Text> := unsigned integer
<Tzid Text> := unsigned integer
<Mpoint Text> := EMPTY |(<Mpt_Segment Text> , < Mpt_Segment Text > *)
<Mpt_Segment Text> := (<Point> , < Point > *)
<Point> := (<x> <y> <z> <m>
<x> := double precision literals
<y> := double precision literals
<z> := double precision literals
<m> := <YYYY> + '-'+ <MM> + '-'+ <DD> <hh> + ':'<mm> + ':'+ <ss>
<YYYY>:= unsigned integer
<MM>:= unsigned integer
<DD>:= unsigned integer
<hh>:= unsigned integer
<mm>:= unsigned integer
<ss>:= unsigned integer

2.3.5 Language Construct for a moving real

<Mreal Tagged Text> :=
Mreal <Mreal Text>
<Mreal Text> := EMPTY |(<r> <t> ,<r> < t> *)
<r> := double precision literals
<t> := <YYYY> + '-'+ <MM> + '-'+ <DD> <hh> + ':'<mm> + ':'+ <ss> <tzid>
<YYYY>:= unsigned integer
<MM>:= unsigned integer
<DD>:= unsigned integer
<hh>:= unsigned integer
<mm>:= unsigned integer
<ss>:= unsigned integer
<tzid>:= unsigned integer

Additionally, we define multi_mreal for multiple *mreal* values as:

<Multi_mreal Text> := EMPTY |(<Mreal>,<Mreal>*)

2.3.6 Language Construct for period

<Period Tagged Text> :=
<Initial_time Text> ;
<Final_time Text> ;
<Tzid Text> ;
<Period Text> := (<Initial_time>, < Final_time>, <Tzid>)
<Initial_time Text> := <YYYY> + '-'+ <MM> + '-'+ <DD> <hh> + ':'<mm> + ':'+ <ss>
<YYYY>:= unsigned integer
<MM>:= unsigned integer
<DD>:= unsigned integer
<hh>:= unsigned integer

<mm>:= unsigned integer
<ss>:= unsigned integer
<Final_time Text> := <YYYY> + '-'+ <MM> + '-'+ <DD> <hh> + ':'<mm> + ':'+ <ss>
<YYYY>:= unsigned integer
<MM>:= unsigned integer
<DD>:= unsigned integer
<hh>:= unsigned integer
<mm>:= unsigned integer
<ss>:= unsigned integer
<Tzid Text> := unsigned integer

Chapter 3

Operations on moving points

3.1 INTRODUCTION

The need for discrete representation of moving objects arises from the limited capacity of storage devices as well as the discrete nature of devices that are used for measurement of time varying information (Moreira, Ribeiro, & Abdessalem, 2000). To address this limitation, query operations must consider the behavior of such objects to give a better result.

In our design, even though the abstract and discrete design to represent a moving point are the crucial steps to be done first, it will be meaningless unless we have a design of query operations for the next step. This is because after representation, the next thing is naturally the need for manipulating the represented information. This mainly needs implementation of functions containing different operations. So in this chapter, we discuss the design of these functions on the moving point type.

The design of operations follows two principles as stated in Güting et al. (2000). First, a function should be as generic as possible. This is done by relating each type in the type system that is used for the design of the abstract data type of moving point to 1D, 2D or 3D and by considering all values as single elements or subsets of the respective space(Lema, Forlizzi, Güting, Nardelli, & Schneider, 2003). For example, we can take the *point* type, which represents a single object in 3D, whereas *line* describes subsets of the three-dimensional space containing 3D *points*.

The second principle is to include the most important operations in the design. Even though the type system contains different types, we only consider a few operations. Since there is no clear way of achieving the closure of the most useful phenomena, the selection of these operations is mainly from simple set theory and first-order logic. This is because they are the most basic components of query languages which are useful to get a valuable information.

Considering the above principles, we divide the main operations on the moving point type into three groups: operations on projection of domain, operations to deal with interaction with domain, and operations on rate of change.

In the next sections we discuss the full detail of operations that are listed in the Table 3.1.

Table 3.1: Operations on moving point

Type	Operations
Interaction with domain	mpt_atinstant, mpt_atperiod, mpt_initial, mpt_final, mpt_present_atinstant, mpt_present_atperiod
Projection to domain	mpt_location, mpt_trajectory
Rate of change	mpt_speed, mpt_velocity

3.2 OPERATIONS ON THE MOVING POINT TYPE

For the purpose of explanation in the next subsections, we define *empty_mpt* and *empty_mr*. *empty_mpt* is used to represent an empty *mpoint* geometry, that is a moving point with no registered location, which implies, the *mpoint* structure has no segment elements. We use *empty_mr* to represent an empty *mreal*.

3.2.1 Interaction with Domain

mpt_atinstant

Arguments :

m : mpoint and *i : instant*

Returns :

p' *: geometry(point)* and i' *: instant*

Description: By using mpt_atinstant function, we determine from *m* at what location it was at instant *i*. This function associates an instant of time with the location of a moving point. The signature of the function is defined as:

$$mpoint \times instant \rightarrow point \times instant$$

The function will take a moving point and time instant as its arguments and return the location of the moving point at the specified time instant as *(point, instant)* pair. The instant *i* is assumed to be in the time zone registered for the moving point *m*.

Preconditions :

- *i* must have a format of 'YYYY-MM-DD hh:mm:ss Tzid' following ISO standard (ISO, 2004).

Postcondition:

- a point geometry and instant pair, where the point is represented by its three dimensions (*x*, *y* and *z*) and an instant of the form of 'YY-MM-DD hh:mm:ss Tzid' which follows ISO (2004) rules will be returned upon calling the function if the moving point is defined at the given instance of time.
- *(NULL,* i'*)* will be returned if the given input instant *i* is not within the temporal domain of *m*.
- the input *i* and the output i' are always the same.

3.2.2 mpt_atperiod

Arguments :

m : mpoint and *ρ : period*

Returns :

m' *: mpoint*

Description: The function is used to restrict the given moving point *m* to the given time interval ρ. It returns an *mpoint* with known positions only within ρ. It will help us to answer a question like where is the moving object *m* within the time interval $\rho = (t_1, t_2)$.

We define the signature of the function as:

$$mpoint \times period \rightarrow mpoint$$

Precondition:

- ρ which represents a time interval must be of the form '(YYYY-MM-DD hh:mm:ss, YYYY-MM-DD hh:mm:ss, Tzid)' as defined in the language construct for *period*.
- each component of the format 'YYYY-MM-DD hh:mm:ss Tzid' in the input ρ must follow the rule stated in the precondition of mpt_atinstant for the input instant.
- if we represent the input ρ as (t_1, t_2), then $t_1 <= t_2$.

Postcondition:

- if the input *mpoint* is defined within *i*, i.e, if there is at least one known position at an instant of time within ρ, then $m^{'}$ has a temporal domain which is a subset of interval ρ.

  i.e

  $$temp\_dom(m') = \rho \cap temp\_dom(m)$$

  and

  $$\forall t \in temp\_dom(m') : mpt\_atinstant(m, t) = mpt\_atinstant(m^{'}, t)$$

- if the given *m* is not defined for any time instant within the time interval ρ, then an *empty_mpt* will be returned.

  $$\rho \cap temp\_dom(m) = \emptyset \Longrightarrow m' = empty\_mpt$$

3.2.3 mpt_initial

Arguments :

m : *mpoint*

Returns :

$p^{'}$: *point* and $i^{'}$: *instant*

Description: this function is used to retrieve the first registered location and the time associated with it for the moving point *m*. This will let us to know where and when *m* was initially. Its signature is:

$$mpoint \rightarrow point \times instant$$

For a moving point having registered location with the minimum time (*min_t*) associated with it:

$$mpt\_initial(m) = mpt\_atinstant(m, min\_t(temp\_dom(m)))$$

Precondition:

- None

Postcondition:

- at the time of the function call, a 3D *point* geometry and *instant* pair will be returned if the input moving point has at least one registered location.
- The i will be of the form 'YYYY-MM-DD hh:mm:ss Tzid' as in ISO (2004) to represent time.
- *(NULL, NULL)* will be returned if the moving point is not defined at any instant (if m is *empty_mpt*).

3.2.4 mpt_final

Arguments :

mpoint

Returns :

$p^{'}$ *: point* and $i^{'}$ *: instant*

Description: this function is used to retrieve the last defined location of the input moving point m. Its signature is defined as:

$$mpoint \rightarrow point \times instant$$

For a moving point having registered location with the maximum time (*max_t*) associated with it:

$$mpt\_initial(m) = mpt\_atinstant(m, max\_t(temp\_dom(m))$$

Precondition:

- None

Postcondition:

- if the moving point m is defined at least at a single time instant, then a (p', i') pair will be returned.
- $i^{'}$ is always in the form of 'YYYY-MM-DD hh:mm:ss Tzid' as in ISO (2004).
- if m is in stand-still, then $mpt\_initial = mpt\_final$.
- *(NULL, NULL)* will be returned if m is an *empty_mpt*.

3.2.5 mpt_present_atinstant

Arguments :

m *: mpoint* and i *: instant*

Returns :

$b^{'}$ *: boolean*

Description: this function is used to determine whether there is a known position for the given moving point *m* at the specified time instant.Its signature is:

$$mpoint \times instant \rightarrow boolean$$

Precondition:

- The *instant* value must be in the form of 'YYYY-MM-DD hh:mm:ss Tzid' as it is stated in ISO (2004) for date and time format.

Postcondition:

- *TRUE* is returned if *m*'s position is known at the given instant, *i*.
- *FALSE* is returned if *m*'s position is not known at the given instant *i*.

3.2.6 mpt_present_atperiod

Arguments :
m : mpoint and *ρ : period*
Returns :
b' : boolean

Description: this function is used to determine whether there is at least one known position for the input *m* within the specified time interval ρ.

$$mpt\_present\_atperiod(m, rho) = \exists t \in \rho : mpt\_present\_atinstant(m, t)$$

Its signature is:

$$mpoint \times period \rightarrow boolean$$

Precondition:

- ρ must be in the form of '(YYYY-MM-DD hh:mm:ss, 'YYYY-MM-DD hh:mm:ss Tzid)' and it must be based on the rule stated in mpt_atperiod precondition for the input time_interval above.

Postcondition:

- *TRUE* is returned if *m*'s position is known at least at a single instant of time within ρ.
- *FALSE* is returned if *m*'s position is not known for any instant within ρ.

3.3 PROJECTION TO DOMAIN

3.3.1 mpt_trajectories

Arguments :
m: mpoint
Returns :
g': multilinestring/linestring

Description: this function is used to display the path in the *xyz* plane over which the moving point traveled within the time interval that it is defined. Depending on the given moving point the returned geometry can be a *linestring* or *multilinestring*. A non-empty *mpoint* for which every segment produces a *linestring*, will itself produce a *multilinestring*. If there are one or more stand-stills in the segments in *m*, the function will ignore all the stand-stills. We should also be aware that if the function returns *multilinestring* the produced *linestrings* inside the *multilinestring* cannot be expected to be simple geometries under the OGC definitions (OGC, 1999). This is because if we see figure 3.1, the object can cross the same location it passed before. The signature is defined as:

$$mpoint \rightarrow multilinestring/linestring$$

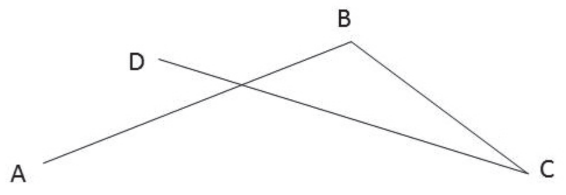


Figure 3.1: Segments inside *m* crossing each other

Precondition:

- None

Postcondition:

- if the given moving point *m* contains no registered location, then g' will be an empty *multilinestring*.
- if *m* contains a single segment with only one registered location of *m*, then the segment will be ignored and an empty *multilinestring* will be returned.
- if *m* contains a single segment with more than one registered location of *m*, then the segment will represent a *linestring* as a footprint.
- if *m* contains more than one segment in which all the segments represent stand-stills, then g' will be an empty *multilinestring*.
- if *m* contains more than one segment where each segment produces a footprint of *m* as a *linestring*, then g' will represent a *multilinestring* where all the *linestrings* produced are in chronological order. g' cannot be expected to be simple always based on the definition of 'simple' in OGC (1999).
- if *m* contains more than one segment where some of the segments produce *linestrings* while the others produce *point*, then g' will be a *multilinestring* if there are more than one *linestring* produced or it will be *linestring* if there is only one *linestring* produced from the segments. All stand-stills in one or more of the segments will be ignored.
- any segment inside the given moving point *m* which produces a geometry footprint as a *linestring* will have 3D points which are in chronological order.

3.3.2 mpt_locations

Arguments :

m : mpoint

Returns :

multi_pt : multipoint/point

Description: the function *mpt_locations*, for a given input mpoint *m*, returns the set of isolated points at which m was known to be at a stand-still. By 'isolated point', we mean locations for which the footprint of a single segment is just a point, and we exclude from these stand-stills occurring in segments for which the footprint is a *linestring*.

$$mpoint \rightarrow multipoint/point$$

Precondition:

- No precondition

Postcondition:

- an empty *multipoint* will be returned if there is no registered position for *m*.
- if the moving point contains a segment with a single registered location, then a 3D *point* will be returned.
- a *multipoint* is returned if *m* contains a single segment which have more than one registered location. The *multipoint* will contain 3D *points*.
- if *m* contains more than one segments and each segment represents a geometry footprint as a *point*, then a *multipoint* will be returned having no repetitive locations. All stand-stills duplicated in one or more segments will appear only once in the output. So the output *multipoint* is always simple as it is defined in OGC (1999).
- if *m* contains more than one segment where some of the segments contain multiple registered locations while the other zero or more segments contains only stand-stills, then a *multipoint* will be returned by removing any duplicate location, if such exists.

3.4 RATE OF CHANGE

3.4.1 mpt_speed

Arguments :

m : mpoint

Returns :

$mr^{'}$ *: mreal*

Description: this function computes the speed of a moving point as a moving real. In each segment inside the moving point structure, the speed between each consecutive positions is calculated and the result is given as two real values: one for the displacement and the other for the difference between the time associated to the locations ($\Delta t1 = t_2 - t_3$, where t_1 is the time associated with the first location and t_2 is the time associated with the second location). So for a single moving point, the speed calculation returns a sequence of real values together with time like (d_1 Δt_1, d_2

Δt_2, d_3 Δt_3, d_4 Δt_4, ...), where $d_1, d_2, ...$ represent the speed between two consecutive positions and $\Delta t_1, \Delta t_2, ..$ represent the change in time between the positions. The signature is defined as :

$mpoint \rightarrow mreal$

Precondition :

- None

Postcondition:

- if *m* has no registered location, then *empty_mr* will be returned, i.e.
  $(m = empty\_mpt) \Longrightarrow empty\_mr$
- if *m* has one segment that represents stand-still, then *mreal* with zero value will be returned.
- if *m* has one segment with more than one registered location, then *mr* will be of the form (d_1 Δt_1, d_2 Δt_2, ..., d_n Δt_n), where n = total number of registered locations - 1.
- if *m* contain more than one segment where all represent stand-stills, then *mr* will be (0 Δt_1, Δt_2,...).
- if *m* contain more than one segments which are all containing more than one registered locations, then the speed calculation between any consecutive locations within each segment will have a value greater than zero.
- All real values inside mr' will have units in ms^{-1}.

3.4.2 mpt_velocity

Arguments:
m : mpoint
Returns :
m' *: mpoint*

Description: this function computes the velocity of a moving point as a sequence of vector values for each segment of the moving point. It calculates the velocity between two consecutive points in the x, y and z direction. This shows the rate of change of the location of *m* in each direction which we call a vector value within the time specified with both locations. So each segment inside the moving point is represented by a sequence of values of velocities in each direction together with the change in time, i.e.(V_{x1} V_{y1} V_{z1} Δt_1, V_{x2} V_{y2} V_{z2} Δt_2, ..., V_{xn} V_{yn} V_{zn} Δt_n). The result finally can be represented as a moving point by assigning default value for the Tzid and Srid. The output of this function tells what is the rate of change of *m* between two locations (within time interval) in each direction. But even though its an abuse to use, for simplicity we are using *mpoint* to represent the output by giving default value for Tzid and Srid to represent the output of *mpt_velocity*.

((V_{x1} V_{y1} V_{z1} Δt_1, V_{x2} V_{y2} V_{z2} Δt_2, ...), (V_{x3} V_{y3} V_{z3} Δt_3, V_{x4} V_{y4} V_{z4} Δt_4, ...), ...)

The signature is then defined as :

$mpoint \rightarrow mpoint$

Precondition :

- None

Postcondition:

- if the input *m* has no registered location then $m^{'}$ will be *empty_mpt* will be returned.
- if *m* contains a single segment with one registered location, then $m^{'}$ will be *empty_mpt*.
- if *m* contains a single segment having more than one registered location, then the velocity in each direction for any consecutive location will be greater than or equal to zero.
- if *m* has more than one segment where all of them represent stand-stills, then $m^{'}$ will contain a velocity of zero in each direction for all the segments.
- if *m* contains more than one segment where some of the segments represent stand-stills while the others do not, then the velocity computation in each direction for every consecutive locations which represent stand-stills will be zero.
- if *m* contains more than one segment which are all containing more than one registered location, then the velocity calculation in each direction between any consecutive locations in each segment will have a value greater than or equal to zero.
- the velocity in each direction inside a segment for any consecutive location in *m* always has a unit expressed in ms^{-1}.

Chapter 4
Implementation and Results

4.1 INTRODUCTION

In the previous chapters, we discussed the abstract and discrete design for the implementation of a moving point type and functions for it. The main concern of those chapters is in discussing how to represent a moving point: the abstract definition and the data structure in the discrete level, and descriptions of functions. The description of functions explains only what the functions do and what kind of outputs we obtain from them. In this chapter, we discuss how these functions and the moving point type are implemented as an extension to a DBMS as well as the choice of the working environment and the steps that are taken for setting up. Algorithms are described to show the implementation detail. Furthermore, we discuss the result of the implementation.

4.2 SETUP AND ENVIRONMENT

For the implementation, the first step was to choose the operating system and software needed. For the selection, the main resource was a previous thesis on the same subject that was carried out by Eftekhar (2012). On the basis of this, together with literature review the selection is made. Table 4.1 shows the overall list of software and languages that are used.

Table 4.1: Working environment

OS	Windows
DBMS	PostgreSQL 9.2
Compiler	Microsoft Visual Studio 2010
Geometry library	PostGIS 2.0
Programming language	C
Query language	SQL

As the main aim of this thesis project is to obtain an extension of the current DBMS softwares to handle moving objects, we choose one DBMS from the available DBMSs: PostgreSQL. The choice is made mainly because it is an open source software of which the source code is available for free and currently, it is also a powerful DBMS that many companies are using (Group, 2013a). It is compatible with all operating systems and comes with many GUI tools. So, for a developer who would like to extend, it is free to extend the software.

In the selection of a programming language, even though PostgreSQL allows the use of high level programming languages like C++, the language that fits totally with the implementation of new functions and types in PostgreSQL is C. But still PostgreSQL has its own C++ extensibility if one follows strictly the guidelines stated in Group (2013c). So we choose C, which is actually the language in which PostgreSQL is developed.

Since Microsoft Visual C++ is one of the PostgreSQL platforms (Group, 2013b), we used the 2010 version of it for the purpose of compiling and linking code. Furthermore, PostGIS is the spatial extension of PostgreSQL and contain many useful types and functions that we can use to develop the type and functions we designed, we used the PostGIS library. The scheme that PostGIS uses for the implementation of types and functions is based on OGC (1999) and this helped us as a framework to work with in the implementation.

4.3 IMPLEMENTATION OF A MOVING POINT DATA TYPE

As stated in PostgreSQL 9.2 documentation, to create a new type, there must be two functions that are used as a skeleton to create it: an input and an output function. They are mainly used to describe how the type appear as a text when the user insert and retrieve data and how it is organized internally. In other words, the input function is used for an internal representation (for both using it in memory and storing it to disk), while the output function is used to retrieve the data from the internal representation.

Generally, when we create a type in PostgreSQL, it should have textual representation, so that the value can be expressed by string in sql statement. so the input function is used to accept this text representation from the user and then transform it into internal representation. In contrary, when a user wants to retrieve the value of an attribute of the new type from a table using select statement, the output function will be implicitly invoked to retrieve the value from the internal representation and displays the textual representation (or binary format if the user asks).

To create input and output functions for *mpoint*, we followed the language constructs defined in the previous chapter. For the purpose of writing and reading data, we created functions to serialize and deserialize. Whenever the user wants to enter data, the serialization function is triggered by the input function. Figure 4.1 shows how the serialization of *mpoint* works.

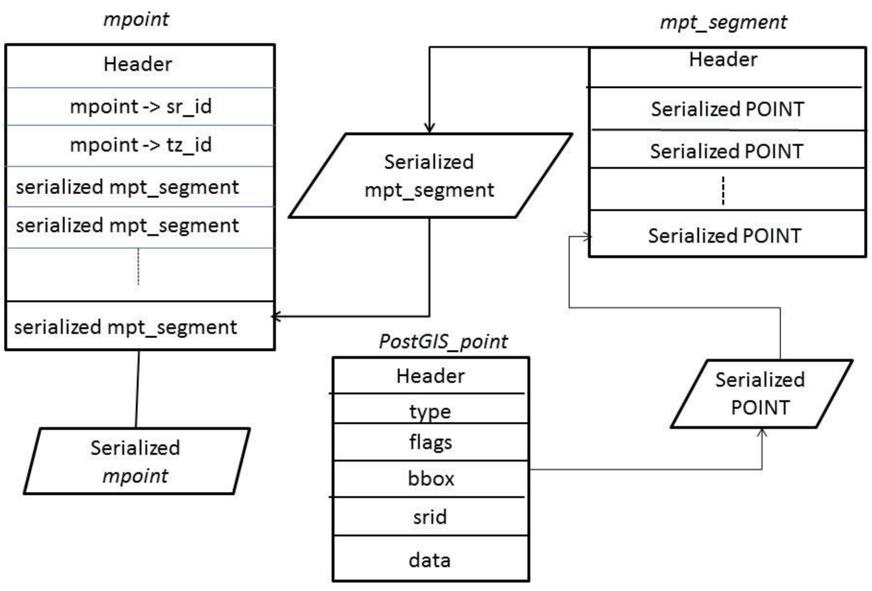


Figure 4.1: Serialization of mpoint

4.4 IMPLEMENTATION OF FUNCTIONS FOR MOVING POINT TYPE

In this section, we explain the details of how the implementation of functions on *mpoint* is done in an algorithmic way. To use terms in the algorithms, we use the definitions below:

- *i* represents an *instant*
- *mpt* represents an *mpoint*
- *time_interval* represents a *period*
- *num_sgts* represents number of segments in *mpt*
- *num_pts* represents number of registered locations.
- *mpt_sgts* represents an array of mpt_segment.

4.4.1 Algorithm for mpt_atinstant

The algorithm for mpt_atinstant works in three steps. In the first step, the timezone identifier in the input *mpoint* is checked for match with the timezone identifier of the input *instant*. In the second step, the search for the correct segment that is defined within a time interval containing *i* is retrieved. In the last step, within the segment that is found in step two, the location of the given *mpoint* is searched and returned if it is stored, otherwise, the algorithm searches two closest locations that can be used for interpolation to get the required location. The *instant* given as an input should be within the time interval of the two locations used for interpolation.

Step1: Validating time zone identifier

This step is used to check whether the time zone in the input time *instant*, *i* is the same as the time zone associated with the given *mpoint mpt*. It is described in pseudocode as follows:

```
if $(i.tzid = mpt.tzid)$ then
    continue
else
    display error message
end if
```

Step2: Searching for the correct segment

In this step, based on the time interval that each segment has in the given *mpt*, the algorithm searches for the correct segment that is defined within a time interval that contains the given *instant* i

```
for int $index = 0 \to num\_sgts - 1$ do
    if $i$ is within $mpt\_seg[index].time\_interval$ then
        return $mpt\_seg[index]$;
    end if
end for
```

Step3: Finding the required location

Once the segment is found in Step2, again a search is done to get the required position. If the location is not stored, then the closest two locations that can be used for interpolation will be taken and the result location will be returned. The algorithm is described using pesudocode below.

```
for int $index = 0 \rightarrow num\_sgts - 1$ do
    if $geom\_array[i].time >= i$ then
        if $geom\_array[i].time = i$ then
            return $geom\_array[i]$;
        else
            $new\_point \leftarrow interpolate(geom\_array[i-1], geom\_array[i])$
            return $new\_pont$;
        end if
    end if
end for
```

4.4.2 Algorithm for mpt_atperiod

The algorithm is used to restrict a given *mpoint* to the given time interval. The algorithm has three steps to return locations of the given *mpoint* within the given time interval. The first step is to validate the time zone specified in the input *period*. The second step searches for segments of which the time interval overlaps with the input time interval. The last step creates a new *mpoint* containing the returned segments of step2.

Step1: Validating timezone identifier

In this step, the algorithm checks whether the time zone identifier specified in the input time interval is the same as the time zone identifier of the input *mpoint*.

```
if $(time\_interval.tzid == mpt.tzid)$ then
    Continue
else
    Display error message
end if
```

Step2: Search segments

In this step, the algorithm searches for segments inside the given *mpoint* having time interval overlapping with the input time interval.

```
$n = 0$;
for int $index = 0 \rightarrow num\_sgts - 1$ do
    $first\_location \leftarrow mpt\_sgts[index].geom\_array[0]$
    $last\_location \leftarrow mpt\_sgts[index].geom\_array[num\_pts - 1]$
    if $(first\_location.time >= time\_interval.initial\_time) and (last\_location.time <= time\_interval.final\_time)$ then
        $new\_mpt\_sgts[n] \leftarrow mpt\_sgts[index]$;
        $n++$;
    end if
end for
```

Step3: Create new mpoint

In the final step, by using the returned segments of the previous step, which is *new_mpt_sgts*. This step creates new *mpoint new_mpt* which has the same srid and tzid as the input *mpoint*.

```
$new\_mpt.tzid \leftarrow mpt.tzid$;
$new\_mpt.srid \leftarrow mpt.srid$;
```

```
$new\_mpt.mpt\_sgts \leftarrow new\_mpt\_sgts[n]$;
return $new\_mpt$;
```

4.4.3 Algorithm for mpt_initial

This algorithm is used to retrieve the initial registered location of the given *mpoint*. It first checks whether the given *mpoint* is an empty *mpoint* (with no registered location) or not. If it is not, then the algorithm retrieves the location stored in the first segment of *mpt* and return it together with the time associated with it.

```
if $(mpt.num\_sgts = 0)$ then
    return $(NULL, NULL)$
else
    $first\_segment \leftarrow mpt.mpt\_seg[0]$;
    $initial\_location \leftarrow first\_segment.geom\_array[0]$;
    $initial\_time \leftarrow initial\_location.time$;
    return $(initial\_location, initial\_time)$;
end if
```

4.4.4 Algorithm for mpt_final

The algorithm *mpt_final* is used to retrieve the final registered location of the given *mpoint*. It first checks weather the given *mpoint* is an empty *mpoint* or not. If it is, then it will return $(NULL, NULL)$ as a result to specify the given *mpoint* has no registered location. Otherwise, it returns the last location where the given moving point was together with the time instant associated with it

```
if $(mpt.num\_sgts = 0)$ then
    return $(NULL, NULL)$
else
    $last\_segment \leftarrow mpt.mpt\_seg[num\_sgts - 1]$;
    $final\_location \leftarrow last\_segment.geom\_array[num\_pts - 1]$;
    $final\_time \leftarrow final\_location.time$;
    return $(final\_location, final\_time)$;
end if
```

4.4.5 Algorithm for mpt_present_atinstant

The algorithm is used to check if the input *mpoint* was present at the given *instant* or not. To do this, it searches a segment that contains the input time instant. If it finds one, then it returns $TRUE$ to specify that the given *mpoint* was present at the the given *instant*, otherwise, $FALSE$.

```
for int $index = 0 \rightarrow num\_sgts - 1$ do
    $first\_location \leftarrow mpt\_sgts[index].geom\_array[0]$
    $last\_location \leftarrow mpt\_sgts[index].geom\_array[num\_pts - 1]$
    if $(first\_location.time <= i) and (last\_location.time >= i)$ then
        return $TRUE$
    else
        return $FALSE$
    end if
end for
```

4.4.6 Algorithm for mpt_present_atperiod

The algorithm is used to check if the input *mpoint* has registered locations within a specific time interval or not. The algorithm first accepts an *mpoint* and time_interval as inputs, then searches for a segment of which the input time interval is within the time_interval of the segment, which means the time interval that the segment is defined may or may not coincide to the input time interval. If it finds a segment that fulfill this, it returns $TRUE$, otherwise, returns $FALSE$. This is stated in an algorithmic way below.

```
$n = 0$;
for int $index = 0 \to num\_sgts - 1$ do
    $first\_location \leftarrow mpt\_sgts[index].geom\_array[0]$
    $last\_location \leftarrow mpt\_sgts[index].geom\_array[num\_pts - 1]$
    if $(first\_location.time >= time\_interval.initial\_time) and (last\_location.time <= time\_interval.final\_time)$ then
        $new\_mpt\_sgts[n] \leftarrow mpt\_sgts[index]$;
        $n++$;
    end if
end for
if $n = 0$ then
    return $FALSE$;
else
    return $TRUE$;
end if
```

4.4.7 Algorithm for mpt_trajectories

The algorithm is used to retrieve the path that a moving point traveled as a *multilinestring* or *linestring*. The algorithm works in two steps. First it checks if the given *mpoint* contains at least one registered location. If not, then it will return an empty *multilinestring*. If it contains registered locations, it will display either a *linestring* or *multilinesting* based on the cases stated in the postcondition description in Chapter three. The algorithm below shows how the function is implemented.

```
if $mpt.num\_sgts = 0$ then
    return empty $multilinestring$
else
    $count = 0$
    for $index = 0 \to num\_sgts - 1$ do
        if $mpt\_seg[index].num\_pts = 1$ then
            do nothing
        else
            for $j = 0 \to num\_pts - 1$ do
                $point\_coll[n \leftarrow mpt\_sgts[index].geom\_array[j]$
            end for
            create a $linestring$ from $point\_coll[n$ and set it as $new\_linestring$
            $new\_linestrings[count] \leftarrow new\_linestring$
            count++;
        end if
    end for
    if $count > 0$ then
```

```
        return $new\_linestrings$ as $linestring$;
    else
        return $new\_linestrings$ as $multilinestring$;
    end if
end if
```

4.4.8 Algorithm for mpt_locations

The algorithm for the implementation of mpt_locations only returns a *multipoint* or a *point*. Since the main aim of the function is to return at which locations was the given *mpoint*, the function removes any duplicate locations if these exists. The algorithm below describes how it is implemented.

```
if $mpt.num\_sgts == 0$ then
    return empty $multipoint$
else
    int $count = 0$
    for $index = 0 \to num\_sgts - 1$ do
        for $j = 0 \to num\_pts - 1$ do
            $new\_points[count] \leftarrow mpt\_sgts[index].geom\_array[j]$
            $count + +;$
        end for
    end for
    if $count = 1$ then
        return $new\_point$ as a $point$
    else
        remove all repeated locations in $new\_point$
        return $new\_point$ as a $multipoint$
    end if
end if
```

4.4.9 Algorithm for mpt_speed

The algorithm calculates the speed between any consecutive location for each segment of the given *mpoint* as a real value. The algorithm below explain how the function works.

```
int $count = 0;$
if then$mpt.num\_sgts == 0$
    return empty $mreal$
else
    for int $index = 0 \to num\_sgts - 1$ do
        for int $pos = 0 \to num\_pts - 1$ do
            $point1 \leftarrow mpt\_sgts[index].geom\_array[pos]$
            $point2 \leftarrow mpt\_sgts[index].geom\_array[pos + 1]$
            $x_1 \leftarrow point1.x$
            $y_1 \leftarrow point1.y$
            $z_1 \leftarrow point1.z$
            $x_2 \leftarrow point2.x$
            $y_2 \leftarrow point2.y$
            $z_2 \leftarrow point2.z$
            $t_1 \leftarrow point1.time$
            $t_2 \leftarrow point2.time$
            $distance \leftarrow \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$
            $speed \leftarrow \frac{distance}{t_2,t_1}$
            $mreal[count].real\_value \leftarrow speed$
            $mreal[count].time \leftarrow difference\_time(t_2, t_1)$
            $count + +;$
        end for
    end for
    return $mreal$
end if
```

4.4.10 Algorithm for mpt_velocity

The algorithm *mpt_velocity* calculates the velocity in x, y and z direction between any consecutive locations of the given *mpoint* and returns it as an *mpoint*.

```
int $count = 0$;
if thenmpt.num_sgts == 0
    return empty $mpoint$
elseint $n == 0$;
    create new mpoint as $new\_mpoint$
    for int $index = 0 \rightarrow num\_sgts - 1$ do
        for int $pos = 0 \rightarrow num\_pts - 1$ do
            $point1 \leftarrow mpt\_sgts[index].geom\_array[pos]$
            $point2 \leftarrow mpt\_sgts[index].geom\_array[pos + 1]$
            $x_1 \leftarrow point1.x$
            $y_1 \leftarrow point1.y$
            $z_1 \leftarrow point1.z$
            $x_2 \leftarrow point2.x$
            $y_2 \leftarrow point2.y$
            $z_2 \leftarrow point2.z$
            $t_1 \leftarrow point1.time$
            $t_2 \leftarrow point2.time$
            $velocity\_x \leftarrow \frac{x_2-x_1}{t_2-t_1};$
            $velocity\_y \leftarrow \frac{y_2-y_1}{t_2-t_1};$
            $velocity\_z \leftarrow \frac{(z_2-z_1)}{t_2-t_1}$
            $time\_diff \leftarrow difference\_time(t_2, t_1);$
            $new\_geom\_array[n].x \leftarrow velocity\_x$
            $new\_geom\_array[n].y \leftarrow velocity\_y$
            $new\_geom\_array[n].z \leftarrow velocity\_z$
            $new\_geom\_array[n].time \leftarrow time\_diff$
            $n + +;$
        end for
        $new\_sgts[count].geom\_array \leftarrow new\_geom\_array[n]$
        $count + +;$
    end for
    $new\_mpoint.mpt\_sgts \leftarrow new\_sgts$
    return $new\_mpoint$ which is an $mpoint$ with default tzid and srid
end if
```

Chapter 5
Results

5.1 RESULTS

In the previous chapter, we discussed the implementation of a *mpoint* data type and function that can be applied on them in algorithmic way. In this chapter, we demonstrate the output of the implementation with examples. For further detail the source code is attached on the appendix.

5.2 DEMONSTRATION OF *MPOINT*

Let us say that a company X is organizing a car competition and wants to register the information of the cars in a database to store the information during the competition. So, a GPS receiver is installed in all cars in the competition and and they are activated from the beginning of the race, to track the cars. For the demonstration, let us create a table called 'cars', which can be used to store the information of the cars. We make one of the attribute as *mpoint* type to represent the geometry of the car.

Create new table:

```
 CREATE TABLE car(car_model VARCHAR(20),
                  car_geom mpoint);
```

Insert data into table:

```
INSERT INTO car(car_model, car_geom)
VALUES('FIAT500','srid=4326;tzid=1;MPOINT((2.8 3.6 8.1 2012-03-12
    12:05:40),(2.8 3.6 8.1 2012-03-12 12:09:40),(2.5 3.5 8.0
   2012-03-12 12:09:45,2.0 3.1 8.7 2012-03-12 12:09:50,3.2 5.5
   9.0 2012-03-12 12:09:55),(4.2 4.5 9.3 2012-03-12 12:19:00))')
```

Retrieve data from the table:

```
SELECT car_geom
FROM car
```

output:

```
     car_geom
-----------------------------------------------------------------
"srid=4326;tzid=1;MPOINT((2.800000 3.600000 8.100000 2012-03-12
   12:05:40),(2.800000 3.600000 8.100000 2012-03-12 12:09:40)
   ,(2.500000 3.500000 8.000000 2012-03-12 12:09:45,2.000000
3.100000 8.700000 2012−03−12 12:09:50,3.200000 5.500000
9.000000 2012−03−12 12:09:55),(4.200000 4.500000 9.300000
2012−03−12 12:19:00))"
```

The above output shows the use of the input and out put functions discussed above section. The insert statement is used for the ‘INSERT INTO’ statement and the output function is used for ‘SELECT FROM’ statement.

So *mpont* data type is used once we have our data in the format of the textual representation of the type, like we did above.

5.3 FUNCTION DEMONSTRATIONS

To demonstrate the implemented functions, we use the same example we describe on the previous section by using possible questions that can be raised.

5.3.1 mpt_final

One of the possible question that can be raised if the car with a model number ‘FIAT500’ did not finish the competition is, where was its last stop? This can be find by using the function mpt_final. The SQL codes below show this usage.

```
SELECT mpt_final(car_geom)
FROM car
WHERE car_model = ‘FIAT500’
```

```
 mpt_final
------------------------------------------------------------------
 (POINT Z(4.200000 4.500000 9.300000),2012−03−12 12:19:00 4326)
```

This shows us the final registered location of the car.

5.3.2 mpt_initial

In the contrary, we might also want to know the initial location we can query:

```
SELECT mpt_initial(car_geom)
FROM car
WHERE car_model = ‘FIAT500’
```

```
 mpt_initial
------------------------------------------------------------------
 (POINT Z(2.800000 3.600000 8.100000),2012−03−12 12:05:40 426)
```

5.3.3 mpt_atinstant

In order to know the location of the car which is in competition at a specific time, we can apply this function over the car_geom attribute in the car table. For example if we want to know the position of a car with model number ‘FIAT500’ at time 2012-03-12 12:09:47, then we can query:

```
SELECT mpt_atinstant(car_geom,‘2012−03−12 12:09:50 4326’)
FROM car
WHERE car_model = ‘FIAT500’
```

```
mpt_atinstant
---------------------------------------------------------------
(POINT Z(2.300000 3.340000 8.280000),2012-03-12 12:09:47 426)
```

5.3.4 mpt_present_atinstant

To know if a given *mpoint* has registered location at a specific time instant, we can use the function mpt_present_atinstant. For example to check if a car was present at a given instant 2012-03-12 12:09:56:

```
SELECT mpt_present_atinstant(car_geom,'2012-03-12 12:09:56 4326')
FROM car
WHERE car_model = 'FIAT500'
```

```
mpt_atinstant
---------------------------------------------------------------
FALSE
```

5.3.5 mpt_atperiod

This function allows to know the path that the moving point traveled. For example if we want to know where the car with a model number 'FIAT500' was within specific time interval, we can write a query like this:

```
SELECT mpt_atperiod(car_geom,'(2012-03-12 12:09:45,2012-03-12
    12:09:55,4326)')
FROM car
WHERE car_model = 'FIAT500'
```

```
mpt_atinstant
---------------------------------------------------------------
"srid=4326;tzid=1;MPOINT((2.500000 3.500000 8.000000 2012-03-12
    12:09:45,2.000000 3.100000 8.700000 2012-03-12
    12:09:50,3.200000 5.500000 9.000000 2012-03-12 12:09:55))"
```

5.3.6 mpt_present_atperiod

To know if a given *mpoint* has registered location within the specific time interval, we can use the function mpt_present_atperiod. If there is atleast one registered location within the given time interval, the function returns TRUE, otherwise FALSE is returned. For example to check if a car was present within a time interval (2012-03-12 12:09:45,2012-03-12 12:09:50,4326):

```
SELECT mpt_present_atperiod(car_geom,'(2012-03-12
    12:09:45,2012-03-12 12:09:55,4326)')
FROM car
WHERE car_model = 'FIAT500'
```

```
mpt_atinstant
---------------------------------------------------------------
TRUE
```

5.3.7 mpt_trajectories

This function is used to know the path a moving point is traveled. Following the description specified in chapter three, if we query the trajectory of the car with model number 'FIAT500', the out put is a *linestring*.

```
SELECT mpt_trajectories(car_geom)
FROM car
WHERE car_model = 'FIAT500'

 mpt_trajectories
------------------------------------------------------------------------
LINESTRING Z(2.500000 3.500000 8.000000,2.000000 3.100000
    8.700000,3.200000 5.500000 9.000000)
```

5.3.8 mpt_locations

This function is used to retrieve the registered locations of the given moving point. Based on the descriptions specified in Chapter three, if we query the function mpt_locations of the car with model number 'FIAT500', the out put is a *multipoint*.

```
SELECT mpt_locations(car_geom)
FROM car
WHERE car_model = 'FIAT500'

 mpt_locations
------------------------------------------------------------------------
MULTIPOINT Z(2.800000 3.600000 8.100000,2.500000 3.500000
    8.000000,2.000000 3.100000 8.700000,3.200000 5.500000
    9.000000,4.200000 4.500000 9.300000)
```

Chapter 6

Conclusion and Recommendation

6.1 CONCLUSION

In this research project, we reviewed the current available spatiotemporal models. Even though there is no perfect model in representing real world objects in a database system, the abstract design of Güting et al. (2000) is the better model that can be used to extend current DBMSs towards handling movement information in comparison to the other models (Pelekis et al., 2005). So, considering the design methods proposed by Güting et al. (2000) as a stepping stone in designing abstract and discrete representation can improve the current DBMSs. It proposes a foundation on the representation of moving objects. On this research, we did some refinement on the mathematical definitions proposed by Güting et al. (2000). This allowed us to define a clear abstract design for a moving point data type and operations. Then, we design the discrete representation which mainly focuses on how the proposed data type and functions containing operations can be implemented within the memory administered by the DBMS. We started by describing the use of each function and then proposed their algorithmic implementation.

As PostgreSQL extension we impelemented a moving point type, which we define it as a real world object that changes its position through time but not its extent. Functions that are used to manipulate and analyze information of this type are also implemented.

For the implementation of the moving point data type, we use the method of segmentation. The segmentation of information for a single moving point considers the natural possible cases, for example when a GPS receiver fails to obtain positions for a period of time or when the moving point stop at some point and continue moving. It assumes and follows the principle of a constant time step between any two consecutive locations within a segment.

In the design of the data structure, segments of a moving point type are stored in different memory locations and we used pointers to these segments from the data structure of *mpoint*. For example, let us say that an *mpoint* contains an array of pointers of size three. Each pointer in the array points to segments located to different memory locations. The toasting mechanism of PostgreSQL is automatically applied for segments with much data. If we store one million locations in each segment, PostgreSQL automatically compresses the data to minimize the memory consumption because we use variable size array in our implementation. But if the segments were in the same database array, the compression would be much easier as well as it will minimize much space. The reason that we could't make our implementation in a way that follows the array of segments i that when we serialize and de-serialize the data we have to identify and flag each segment in the data structure, otherwise it will be difficult to identify segments separately. This adds additional unwanted variables in *mpoint* structure. For simplicity on accessing data and design, we serialize segments separately and used a pointer array in the *mpoint* data structure to retrieve the information in each segment. PostgreSQL compresses each segment to minimize memory consumption when ever the data is large. This is done automatically by PostgreSQl because we are using variable size array for storing each segment. If our structure made the segments in the same database array, it would have been easier to compress all the segments together which can results in more reduction of memory consumption.

Generally, in this thesis project, the implementation is done for representation of a moving point, the research objectives are fulfilled and the discrete design is a clear guide to continue with the rest of unfinished moving types like moving region. Furthermore, the thesis project proves that using Güting et al. (2000) as a stone is the right choice.

6.2 RECOMMENDATION

The contribution of this thesis paper is a refined abstract and discrete design by taking the abstract design proposed by Güting et al. (2000) as a start.

The design and implementation is focused on a moving point type and some operations on it. The thesis does not cover all the possible implementations of functions since there is no limited number of functions that can be implemented on moving point type. Only functions that contain operations from set theory and first order logic and the ones we consider are useful to answer potential questions and feasible to implement within the time frame of the thesis are implemented. So its still open to implement more functions for this type and the source code for the moving point type found in the appendix can be used for this purpose as far as one follows the selection of softwares and programing language that is used in this thesis.

Since moving point is the base for the other moving types: moving region and moving line, this thesis can be used as a stepping stone for the design and implementation of such geometries. So as a future work, implementation of these types and functions associated with them is highly recommended as a follow up of this research project. Furthermore, Güting et al. (2000)'s work contains the abstract design of these geometries and by following the methods followed for the implementation of moving point on this thesis, one can implement more moving object types.

References

Cotelo Lema, J. A., Forlizzi, L., Güting, R. H., Nardelli, E., & Schneider, M. (2003). Algorithms for moving objects databases. , *46*(6), 680-712.

Eftekhar, A. (2012). *Development of moving feature data types in a major dbms*. Unpublished master's thesis, University of Twente, the Netherlands.

Erwig, M., Güting, R. H., Schneider, M., & Vazirgiannis, M. (1999). Spatio−temporal data types: An approach to modeling and querying moving objects in databases. *GeoInformatica*, *3*, 269-296.

Forlizzi, L., Güting, R. H., Nardelli, E., & Schneider, M. (1999). A data model and data structures for moving objects databases. In (pp. 319–330).

Group, P. G. D. (2013a). *Featured users.* Retrieved from `http://www.postgresql.org/about/users/`

Group, P. G. D. (2013b). *Feature matrix.* Retrieved from `http://www.postgresql.org/about/featurematrix/`

Group, P. G. D. (2013c). *Using c++ for extensibility.* Retrieved from `http://www.postgresql.org/about/featurematrix/`

Güting, R. H., Behr, T., & Düntgen, C. (2010). Secondo: A platform for moving objects database research and for publishing and integrating research implementations. *IEEE Data Eng. Bull.*, 56-63.

Güting, R. H., Böhlen, M. H., Erwig, M., Jensen, C. S., Lorentzos, N. A., Schneider, M., & Vazirgiannis, M. (2000, March). A foundation for representing and querying moving objects. *ACM Trans. Database Syst.*, *25*(1), 1-42.

Güting, R. H., & Schneider, M. (2005). *Moving objects databases*. Morgan Kaufmann Publishers.

ISO. (2004). *Iso 8601:2004. data elements and interchange formats — information interchange — representation of dates and times*.

Lema, J. A. C., Forlizzi, L., Güting, R. H., Nardelli, E., & Schneider, M. (2003). *Algorithms for moving objects databases.*

Moreira, J., Ribeiro, C., & Abdessalem, T. (2000). Query operations for moving objects database systems. In *Proceedings of the 8th acm international symposium on advances in geographic information systems.* New York, NY, USA: ACM.

OGC. (1999). *Opengis simple features specification for sql revision 1.1*.

Parent, C., Spaccapietra, S., & Zimányi, E. (1999). Spatio−temporal conceptual models: data structures + space + time. In *Proceedings of the 7th acm international symposium on advances in geographic information systems* (p. 26-33). New York, NY, USA: ACM.

Pelekis, N., Theodoulidis, B., Kopanakis, I., & Theodoridis, Y. (2005). Theodoridis. literature review of spatiotemporal database models. *The Knowledge Engineering Review*, *19*, 235-274.

Schneider, M. (2009). Moving objects in databases and gis: State-of-the-art and open problems. In G. Navratil, W. Cartwright, G. Gartner, L. Meng, & M. P. Peterson (Eds.), *Research trends in geographic information science* (p. 169-187). Springer Berlin Heidelberg.

Tryfona, N., & Hadzilacos, T. (1998). Logical data modeling of spatiotemporal applications: Definitions and a model. In *Ideas, international database engineering and applications symposium* (pp. 14–23). Press.

Appendix One

Header Files

```
#include <postgres.h>
#include <fmgr.h>
#include <lwgeom_pg.h>
#include <timestamp.h>
#include "deserial_mpoint.h"
#include "mpt_headers.h"
```

Input Function

```
PG_FUNCTION_INFO_V1(mpoint_in);
Datum
        mpoint_in (PG_FUNCTION_ARGS)
{
        char *str= PG_GETARG_CSTRING(0);
        int i;
        GSERIALIZED *result;
        int num_pts=0;
        int num_segts=0,srid ,tzid;
        bool finished = false;
        LWGEOM *geom_array;
        mpt_segment *pt_segts;
        double X,Y,Z,M;
        int year ,month ,day ,hour ,minute ,second ,pos ,count=0;
        time_t T;
        char mpt_as_4D[256];
        size_t size;
        LWGEOM_PARSER_RESULT lwg_parser_result;
        LWGEOM *lwgeom;
        sscanf_s(str ," srid=%d;%n",&srid ,&pos);
        str+=pos;
        sscanf_s(str ," tzid=%d;MPOINT((%n",&tzid ,&pos);

        str+=pos;
        while(!finished)
        {
```

```
sscanf_s(str,"%lf %lf %lf %d-%d-%d %d:%d:%d%n",&X
    ,&Y,&Z,&year,&month,&day,&hour,&minute,&second
    ,&pos);
T=tm_to_timet(&year,&month,&day,&hour,&minute,&
    second);
M=T;
sprintf(mpt_as_4D,"POINT(%lf %lf %lf %lf)",X,Y,Z,M
    );

if ( lwgeom_parse_wkt(&lwg_parser_result,
    mpt_as_4D, LW_PARSER_CHECK_ALL) == LW_FAILURE
    )
{
        PG_PARSER_ERROR(lwg_parser_result);
}
else {
        lwgeom = lwg_parser_result.geom;
        if ( srid )
                lwgeom_set_srid(lwgeom, srid);
        if ( lwgeom_needs_bbox(lwgeom) )
                lwgeom_add_bbox(lwgeom);
        if (num_pts == 0) {
                geom_array = (LWGEOM*)malloc(
                    sizeof(LWGEOM));
        } else {
                geom_array = (LWGEOM*)realloc(
                    geom_array , (1 + num_pts) *
                    sizeof(LWGEOM));
        }
        geom_array[num_pts] = *lwgeom_clone_deep(
            lwg_parser_result.geom); //make a copy
        ++num_pts;
        lwgeom_parser_result_free(&
            lwg_parser_result);
}

str += pos;
if(str[0]==')')
{
        if(num_segts == 0) {
                pt_segts = (mpt_segment*)malloc(
                    sizeof(mpt_segment));
        } else {
                pt_segts = (mpt_segment*)realloc(
                    pt_segts , (1 + num_segts) *
                    sizeof(mpt_segment));
        }
        pt_segts[num_segts].num_pts = num_pts;
```

```
                    pt_segts[num_segts].geom_array=geom_array;
                    num_pts=0;
                    ++num_segts;
                    str += 1;
                    if (str[0]==')')
                            finished = true;
                    else if (str[0] == ',')
                            str+=2;
            }
            else {
                    str += 1;
            }
    }

    result = gserialized_from_mpoint(pt_segts,srid,tzid,
       num_segts, &size);
    for (i = 0; i < num_segts; ++i)
            free(pt_segts[i].geom_array);
    free(pt_segts);
    PG_RETURN_POINTER(result);
}
```

———————————————————————————

Out Function

———————————————————————————

```
PG_FUNCTION_INFO_V1(mpoint_out);

Datum
        mpoint_out(PG_FUNCTION_ARGS)
{
        const GSERIALIZED *g = (GSERIALIZED*)PG_GETARG_POINTER(0);
        mpoint mpt;
        mpt_segment *mpt_sgts;
        int i;
        char * result =  (char *) palloc(1024);
        *result = 0;
        mpoint_from_gserialized(g, &mpt, &mpt_sgts);
        mpoint_as_text(mpt_sgts, result,mpt.num_segts,mpt.sr_id,
           mpt.tz_id);
        for (i = 0; i < mpt.num_segts; ++i)
                free(mpt_sgts[i].geom_array);
        free(mpt_sgts);
        PG_RETURN_CSTRING(result);
}
```

———————————————————————————

```
#include <postgres.h>
#include "deserial_mpoint.h"
```

```
Serialize SEGMENT
```

```
GSERIALIZED* gserialized_from_mpt_segment(LWGEOM *geom,int srid,
   int num_pts, size_t *size)
{
        int is_geodetic = 0;
        GSERIALIZED* gout;
        char * ptr;
        int i;
        *size = VARHDRSZ + sizeof(int);
        gout = (GSERIALIZED*)palloc(*size);
        ptr = ((char*)gout) + VARHDRSZ;
        *((int*)ptr) = num_pts;
        ptr += sizeof(int);
        for (i = 0; i < num_pts; ++i)
        {
                size_t gsize;
                GSERIALIZED* g;
                char* gout_before_realloc;
                {
                        LWPOINT * point = (LWPOINT*)geom;
                        if (point->type == POINTTYPE)
                        {
                                double X = lwpoint_get_x(point);
                                double Y = lwpoint_get_y(point);
                        }
                }
                if ( srid )
                        lwgeom_set_srid(geom, srid);
                g = gserialized_from_lwgeom(geom, is_geodetic, &
                   gsize);
                SET_VARSIZE(g, gsize);
                *size += sizeof(size_t) + gsize;
                gout_before_realloc = (char*)gout;
                gout = (GSERIALIZED*)repalloc(gout, *size);
                ptr = ptr + (long int)(((char*)gout) -
                   gout_before_realloc);
                *((size_t*)ptr) = gsize;
                ptr += sizeof(size_t);
                memcpy(ptr, g, gsize);
                ptr += gsize;
                ++geom;
        }
        SET_VARSIZE(gout, *size);
        return gout;
}
```

Serialize MPOINT

```
GSERIALIZED* gserialized_from_mpoint(mpt_segment *mpt_sgts, int
   srid, int tzid, int num_segts, size_t *size)
{
        int is_geodetic = 0;
        GSERIALIZED* gout;
        int i;
        char * ptr;
        mpoint mpt;
        mpt.sr_id = srid;
        mpt.tz_id = tzid;
        mpt.num_segts = num_segts;
        *size = VARHDRSZ + sizeof(mpoint);
        gout = (GSERIALIZED*)palloc(*size);
        ptr = ((char*)gout) + VARHDRSZ;
        *((mpoint*)ptr) = mpt;
        ptr += sizeof(mpoint);
        for (i = 0; i < num_segts; ++i)
        {
                size_t gsize;
                GSERIALIZED* g;
                char* gout_before_realloc;
                int num_pts = mpt_sgts[i].num_pts;
                LWGEOM * geom = mpt_sgts[i].geom_array;
                g = gserialized_from_mpt_segment(geom,srid,num_pts
                    , &gsize);
                *size += sizeof(size_t) + gsize;
                gout_before_realloc = (char*)gout;
                gout = (GSERIALIZED*)repalloc(gout, *size);
                ptr = ptr + (long int)(((char*)gout) -
                   gout_before_realloc);
                *((size_t*)ptr) = gsize;
                ptr += sizeof(size_t);
                memcpy(ptr, g, gsize);
                ptr += gsize;
        }

        SET_VARSIZE(gout, *size);
        return gout;
}
```

De-serialize MPOINT

```
void mpoint_from_gserialized(const GSERIALIZED *g, mpoint *mpt,
   mpt_segment **mpt_sgts)
{
        char * ptr = (char*)g;
        int j;
        ptr += VARHDRSZ;
        *mpt = *((mpoint*)ptr);
        ptr += sizeof(mpoint);
        *mpt_sgts = (mpt_segment*)malloc(mpt->num_segts * sizeof(
           mpt_segment));
        for( j=0;j<mpt->num_segts;++j)
        {
                int num_pts;
                int i;
                size_t gsize;
                ptr += VARHDRSZ;
                gsize = *((size_t*)ptr);
                ptr += sizeof(size_t);
                num_pts = *((int*)ptr);
                ptr += sizeof(int);
                (*mpt_sgts)[j].num_pts = num_pts;
                (*mpt_sgts)[j].geom_array = (LWGEOM*)malloc(
                   num_pts * sizeof(LWGEOM));
                for (i = 0; i < num_pts; ++i)
                {
                        size_t gsize = *((size_t*)ptr);
                        ptr += sizeof(size_t);
                        (*mpt_sgts)[j].geom_array[i] = *
                           lwgeom_from_gserialized((GSERIALIZED*)
                           ptr);
                        ptr += gsize;
                }
        }
}
```

MPOINT_AS_TEXT

```
void mpoint_as_text(mpt_segment *mpt_sgts,char * result,int
   num_sgts,int sr_id, int tz_id)
```

```
{
        char str [1024];
        int     i,j;

        sprintf(str,"srid=%d;tzid=%d;MPOINT((",sr_id,tz_id);
        strcat(result,str);
        for (i=0; i<num_sgts ; ++i)
        {
                for (j=0;j< mpt_sgts[i].num_pts; ++j)
                {
                        LWPOINT * point = (LWPOINT*) &(mpt_sgts[i
                            ].geom_array[j]);
                        if (point->type == POINTTYPE)
                        {
                                double X = lwpoint_get_x(point);
                                double Y = lwpoint_get_y(point);
                                double Z = lwpoint_get_z(point);
                                double M = lwpoint_get_m(point);
                                time_t T = M;
                                char utime[100];
                                strftime(utime, 100, "%Y-%m-%d %H
                                    :%M:%S", localtime(&T));
                                sprintf(str, "%lf %lf %lf %s", X,
                                    Y, Z, utime);
                                strcat(result,str);
                                if(j < mpt_sgts[i].num_pts - 1)
                                        strcat(result,",");
                        }

                }
                if(i < num_sgts - 1)
                        strcat(result,"),(");
        }
        strcat(result,"))");
}
```

atinstant

```
LWGEOM *atinstant(mpt_segment *mpt_sgts,int num_sgts,int sr_id,
    int tz_id,char *str)
{
        time_t T,T1,T2;
        int year,month,day,hour,minute,second,input_tzid,count=0,
            last_index;
        double X,Y,Z,t_p1,t_interpolate;
```

```
POINT4D point4D1,point4D2,point4D;
LWGEOM_PARSER_RESULT lwg_parser_result;
char coordinate='M',mpt_as_4D[200];
time_t start_t,end_t,time_point1;
LWGEOM *lwgeom;
bool finish=false,found=false;
int first=0,last,middle,current_num_sgts;
int first_loc,last_loc,middle_loc,total_num_pts,
    index_found_seg;
sscanf_s(str,"%d-%d-%d %d:%d:%d %d",&year,&month,&day,&
    hour,&minute,&second,&input_tzid);
if(((year<=9999 && year >= 0000)&&(month>=01 && month
    <=12)&&(day >=01 && day <=31)&&(hour>=00 && hour<=23)
    &&(minute>=00 && minute<=59)&&(second >=00 && second
    <=59))&&(input_tzid==tz_id))
{
         T= tm_to_timet(&year,&month,&day,&hour,&minute,&
             second);
         t_interpolate=T;
         current_num_sgts=num_sgts/2;
         last=num_sgts-1;
         middle=(first + last)/2;
         last_index=mpt_sgts[middle].num_pts-1;
         while((!finish) && (first<=last))
         {

                  LWPOINT *first_pt = (LWPOINT*) &(mpt_sgts[
                      middle].geom_array[0]);
                  LWPOINT *last_pt = (LWPOINT*) &(mpt_sgts[
                      middle].geom_array[last_index]);

                  if (first_pt->type == POINTTYPE)
                  {
                           start_t  = lwpoint_get_m(first_pt)
                               ;

                  }

                  if (last_pt->type == POINTTYPE)
                  {
                           end_t = lwpoint_get_m(last_pt);

                  }

                  if((t_interpolate <= end_t) && (
                      t_interpolate >=start_t))
                  {
                           finish = true;
```

```
                }
                else  if(t_interpolate <start_t)
                {
                        last=middle - 1;
                        middle=(first + last)/2;
                }
                else
                {
                        first = middle + 1;
                        last = num_sgts - 1;
                        middle=(first + last)/2;
                }
                last_index=mpt_sgts[middle].num_pts-1;

        }
        if(finish){
                index_found_seg = middle;
                total_num_pts = mpt_sgts[index_found_seg].
                   num_pts;
                first_loc=0;
                last_loc =total_num_pts-1;
                middle_loc=(first_loc + last_loc)/2;

                while((!found) && (first_loc<=last_loc))
                {

                        LWPOINT *point_middle = (LWPOINT*)
                             &(mpt_sgts[index_found_seg].
                           geom_array[middle_loc]);
                        if (point_middle->type ==
                           POINTTYPE)
                        {
                                X = lwpoint_get_x(
                                   point_middle);
                                Y = lwpoint_get_y(
                                   point_middle);
                                Z = lwpoint_get_z(
                                   point_middle);
                                t_p1 = lwpoint_get_m(
                                   point_middle);
                                time_point1 = t_p1;
                        }
                        if(t_p1 == t_interpolate)
                        {
                                found = true;
                        }
                        else  if(t_p1 < t_interpolate)
                        {
```

```
                first_loc = middle_loc +1;
                middle_loc=(first_loc +
                   last_loc)/2;

            }
            else
            {
                last_loc = middle_loc - 1;
                middle_loc=(first_loc +
                   last_loc)/2;
            }

        }
        if(found)
        {

            sprintf(mpt_as_4D,"POINT(%lf %lf %
               lf)",X,Y,Z/*,point4D.m*/);

            if ( lwgeom_parse_wkt(&
               lwg_parser_result, mpt_as_4D,
               LW_PARSER_CHECK_ALL) ==
               LW_FAILURE )
            {
                PG_PARSER_ERROR(
                   lwg_parser_result);
            }
            else {
                lwgeom = lwg_parser_result
                   .geom;
                if ( sr_id )
                        lwgeom_set_srid(
                           lwgeom, sr_id)
                           ;
                if ( lwgeom_needs_bbox(
                   lwgeom) )
                        lwgeom_add_bbox(
                           lwgeom);

            }

            return lwgeom;
        }
        else
        {
            LWPOINT *point_middle1 = (LWPOINT
               *) &(mpt_sgts[index_found_seg
               ].geom_array[middle_loc]);
```

```
if (point_middle1->type ==
   POINTTYPE)
{
         point4D1.x = lwpoint_get_x
            (point_middle1);
         point4D1.y = lwpoint_get_y
            (point_middle1);
         point4D1.z = lwpoint_get_z
            (point_middle1);
         point4D1.m = lwpoint_get_m
            (point_middle1);
         T1 = point4D1.m;

}
if(T1 < t_interpolate)
{
         LWPOINT *point2 = (LWPOINT
            *) &(mpt_sgts[
            index_found_seg].
            geom_array[middle_loc
            + 1]);
         if (point2->type ==
            POINTTYPE)
         {
                  point4D2.x =
                     lwpoint_get_x(
                     point2);
                  point4D2.y =
                     lwpoint_get_y(
                     point2);
                  point4D2.z =
                     lwpoint_get_z(
                     point2);
                  point4D2.m =
                     lwpoint_get_m(
                     point2);
                  T2 = point4D2.m;
         }
}
else
{
         LWPOINT *point2 = (LWPOINT
            *) &(mpt_sgts[
            index_found_seg].
            geom_array[middle_loc
            - 1]);
         if (point2->type ==
            POINTTYPE)
```

```
                    {
                            point4D2.x =
                                lwpoint_get_x(
                                point2);
                            point4D2.y =
                                lwpoint_get_y(
                                point2);
                            point4D2.z =
                                lwpoint_get_z(
                                point2);
                            point4D2.m =
                                lwpoint_get_m(
                                point2);
                            T2 = point4D2.m;
                    }
            }
            point_interpolate(&point4D1, &
                point4D2, &point4D, 1, 1,
                coordinate, t_interpolate);
            sprintf(mpt_as_4D,"POINT(%lf %lf %
                lf)",point4D.x,point4D.y,
                point4D.z/*,point4D.m*/);

            if ( lwgeom_parse_wkt(&
                lwg_parser_result, mpt_as_4D,
                LW_PARSER_CHECK_ALL) ==
                LW_FAILURE )
            {
                    PG_PARSER_ERROR(
                        lwg_parser_result);
            }
            else {
                    lwgeom = lwg_parser_result
                        .geom;
                    if ( sr_id )
                            lwgeom_set_srid(
                                lwgeom, sr_id)
                                ;
                    if ( lwgeom_needs_bbox(
                        lwgeom) )
                            lwgeom_add_bbox(
                                lwgeom);
            }

            return lwgeom;
    }
}
```

```
                else
                {
                        return NULL;


                }
        }
        else if( input_tzid!=tz_id)
        {
                ereport(ERROR,
                        (errcode(ERRCODE_FEATURE_NOT_SUPPORTED),
                        errmsg("Input tzid mismatch with the
                           timezone id of the mpoint "
                        )));
        }
        else{
                ereport(ERROR,
                        (errcode(ERRCODE_FEATURE_NOT_SUPPORTED),
                        errmsg("Invalid input "
                        )));
        }
}
```

---

atperiod

```
mpoint *atperiod(mpt_segment *mpt_sgts,int num_sgts,int sr_id, int
    tz_id, Period *period)
{
        bool finish=false;
        int i=0;
        mpoint *new_mpt;
        mpt_segment *new_mpt_sgts= (mpt_segment*)malloc(sizeof(
           mpt_segment));
        time_t start_t, end_t,t_start,t_end,prev_last_t,
           nxt_first_t,nxt_first,nxt_last;
        int first=0,last,middle,last_index,current_num_sgts;
        int num_segts=0;
        LWPOINT *first_pt,*last_pt,*start_point,*end_point,*
           prev_pt,*nxt_pt,*nxt_first_pt,*nxt_last_pt;
        new_mpt= (mpoint *) palloc(sizeof(mpoint));
        last=num_sgts-1;
        //last_index=mpt_sgts[middle].num_pts-1;
        new_mpt->num_segts = num_segts;

        while(!finish && i<num_sgts)
        {
                last_index = mpt_sgts[i].num_pts-1;
                first_pt = (LWPOINT*) &(mpt_sgts[i].geom_array[0])
                   ;
                last_pt = (LWPOINT*) &(mpt_sgts[i].geom_array[
                   last_index]);

                if (first_pt->type == POINTTYPE)
                {
                        start_t  = lwpoint_get_m(first_pt);

                }

                if (last_pt->type == POINTTYPE)
                {
                        end_t = lwpoint_get_m(last_pt);

                }
                if((start_t>= period->initial) && ( end_t <=period
                   ->final))
                {
                        if(num_segts > 0) {

                                new_mpt_sgts = (mpt_segment*)
                                   realloc(new_mpt_sgts , (1 +
                                   num_segts) * sizeof(
                                   mpt_segment));
                        }
                        new_mpt_sgts[num_segts]= mpt_sgts[i];
                        ++num_segts;
                }
                if(start_t > period->initial)
                {
                        finish = true;
                }
                i++;
        }
        new_mpt->sr_id=sr_id;
        new_mpt->tz_id=tz_id;
        new_mpt->num_segts=num_segts;
        new_mpt->mpt_sgts=new_mpt_sgts;

        return new_mpt;
}
```

tm_to_timet

```
time_t tm_to_timet (int * year, int * month, int * day, int * hour
    , int * minute,int * second)
{
        time_t result;
        struct tm timeinfo;
        timeinfo.tm_year = *year - 1900;
        timeinfo.tm_mon = *month -1;
        timeinfo.tm_mday = *day;
        timeinfo.tm_hour = *hour;
        timeinfo.tm_min = *minute;
        timeinfo.tm_sec = *second;
        timeinfo.tm_isdst = -1;
        result = mktime ( &timeinfo );
        return result;
}
```

Input function for period

```
#include "period.h"
#include "mpt_headers.h"


PG_FUNCTION_INFO_V1(period_in);

Datum
        period_in (PG_FUNCTION_ARGS)
{
        char *str = PG_GETARG_CSTRING(0);
        int  year1, month1 ,day1, hour1, minute1, second1;
        int  year2, month2 ,day2, hour2, minute2, second2;
        int     tzid;
        Period * result;
        if (sscanf_s(str, "(%d-%d-%d %d:%d:%d,  %d-%d-%d %d:%d:%d
            , %d)", &year1, &month1, &day1, &hour1 ,&minute1, &
            second1, &year2, &month2, &day2, &hour2, &minute2, &
            second2, &tzid) != 13)
                ereport(ERROR,
                (errcode(ERRCODE_INVALID_TEXT_REPRESENTATION),
                errmsg("invalid input syntax for period: \"%s\"",
                str)));
        result = (Period *) palloc(sizeof(Period));
        result->initial = tm_to_timet(&year1, &month1, &day1, &
            hour1 ,&minute1, &second1);
        result->final = tm_to_timet(&year2, &month2, &day2, &hour2
            , &minute2, &second2);
        result->tzid = tzid;
        if (difftime(result->final,result->initial)< 0)
                ereport(ERROR,
                (errcode(ERRCODE_INVALID_TEXT_REPRESENTATION),
                errmsg("final smaller than initial.")));
        PG_RETURN_POINTER(result);
}
```

---

output function for period

```
PG_FUNCTION_INFO_V1(period_out);

Datum
        period_out(PG_FUNCTION_ARGS)
{
        Period   *period = (Period *) PG_GETARG_POINTER(0);
        char *result;
        char initial[20];
        char final[20];
        result = (char *) palloc(90);
        strftime(initial, 20, "%Y-%m-%d %H:%M:%S", localtime(&
            period->initial));
        strftime(final, 20, "%Y-%m-%d %H:%M:%S", localtime(&period
            ->final));
        snprintf(result, 90, "(%s, %s, %d)",initial , final,
            period->tzid);
        PG_RETURN_CSTRING(result);
}
```

```
PG_FUNCTION_INFO_V1(mpt_atinstant);

Datum
        mpt_atinstant(PG_FUNCTION_ARGS)
{

        const GSERIALIZED *g = (GSERIALIZED*)PG_GETARG_POINTER(0);
        char *str= PG_GETARG_CSTRING(1);
        mpoint mpt;
        mpt_segment *mpt_sgts;
        LWGEOM *geom;
        GSERIALIZED *result;
        mpoint_from_gserialized(g, &mpt, &mpt_sgts);
        geom = atinstant(mpt_sgts,mpt.num_segts,mpt.sr_id, mpt.
            tz_id, str);
        if(geom == NULL){
                char *res = (char *) palloc(1024);
                char str1 [1024];
                *res=0;
                sprintf(str1,"(NULL,%s)",str);
                strcat(res,str1);
                PG_RETURN_CSTRING(res);
        }
        else
        {
                LWGEOM *lwgeom;
                char *wkt;
                char * final_result =  (char *) palloc(1024);
                char str2 [1024];
                size_t wkt_size;
                *final_result = 0;
                POSTGIS_DEBUG(2, "LWGEOM_asEWKT called.");
                result = geometry_serialize(geom);
                lwgeom = lwgeom_from_gserialized(result);
                wkt = lwgeom_to_wkt(lwgeom, WKT_EXTENDED, DBL_DIG,
                    &wkt_size);
                lwgeom_free(lwgeom);
                sprintf(str2,"(%s,%s)",wkt,str);
                strcat(final_result,str2);
                PG_RETURN_CSTRING(final_result);
        }
}
```

```
PG_FUNCTION_INFO_V1(mpt_atperiod);
Datum
        mpt_atperiod(PG_FUNCTION_ARGS)
{

        const GSERIALIZED *g = (GSERIALIZED*)PG_GETARG_POINTER(0);
        char *str= PG_GETARG_CSTRING(1);
        int year1,month1,day1,hour1,minute1,second1,input_tzid,
            tz_id,pos;
        int year2,month2,day2,hour2,minute2,second2;
        time_t initial,final;
        int i;
```

```
size_t size;
mpoint mpt,*new_mpt;
mpt_segment *mpt_sgts;
Period *period;
char * result ;
period = (Period *) palloc(sizeof(Period));
result = (char *) palloc(1024);
*result = 0;
mpoint_from_gserialized(g, &mpt, &mpt_sgts);
tz_id=mpt.tz_id;
sscanf_s(str,"(%d-%d-%d %d:%d:%d,%d-%d-%d %d:%d:%d,%d%n",&
    year1,&month1,&day1,&hour1,&minute1,&second1,&year2,&
    month2,&day2,&hour2,&minute2,&second2,&input_tzid,&pos
    );
if (((year1<=9999 && year1 >= 0000)&&(month1>=01 && month1
    <=12)&&(day1 >=01 && day1 <=31)&&(hour1>=00 && hour1
    <=23)&&(minute1>=00 && minute1<=59)&&(second1 >=00 &&
    second1 <=59))&&((year2<=9999 && year2 >= 0000)&&(
    month2>=01 && month2 <=12)&&(day2 >=01 && day2 <=31)
    &&(hour2>=00 && hour2<=23)&&(minute2>=00 && minute2
    <=59)&&(second2 >=00 && second2 <=59))&&(input_tzid==
    tz_id))
{
        initial= tm_to_timet(&year1,&month1,&day1,&hour1,&
            minute1,&second1);
        final = tm_to_timet(&year2,&month2,&day2,&hour2,&
            minute2,&second2);

        if(initial <= final)
        {
                period->initial= initial;
                period->final= final;
                period->tzid= input_tzid;

                new_mpt = atperiod(mpt_sgts,mpt.num_segts,
                    mpt.sr_id, mpt.tz_id, period);
                gserialized_from_mpoint(new_mpt->mpt_sgts,
                    new_mpt->sr_id, new_mpt->tz_id,new_mpt
                    ->num_segts, &size);
                mpoint_as_text(new_mpt->mpt_sgts,result,
                    new_mpt->num_segts,new_mpt->sr_id,
                    new_mpt->tz_id);
                for (i = 0; i < mpt.num_segts; ++i)
                        free(mpt_sgts[i].geom_array);
                free(mpt_sgts);
                PG_RETURN_CSTRING(result);
        }
        else
```

```
                    ereport(ERROR,
                    (errcode(ERRCODE_FEATURE_NOT_SUPPORTED),
                    errmsg("The input time interval is not
                       correct "
                    )));

        }

        else
                ereport(ERROR,
                (errcode(ERRCODE_FEATURE_NOT_SUPPORTED),
                errmsg("Invalid input"
                )));


}
```

---

mpt_present_atinstant

```
bool atinstant(mpt_segment *mpt_sgts,int num_sgts,int sr_id, int
   tz_id,char *str)
{
        time_t T,T1,T2;
        int year,month,day,hour,minute,second,input_tzid,count=0,
           last_index;
        double X,Y,Z,t_p1,t_interpolate;
        POINT4D point4D1,point4D2,point4D;
        LWGEOM_PARSER_RESULT lwg_parser_result;
        char coordinate='M',mpt_as_4D[200];
        time_t start_t,end_t,time_point1;
        LWGEOM *lwgeom;
        bool finish=false,found=false;
        int first=0,last,middle,current_num_sgts;
        int first_loc,last_loc,middle_loc,total_num_pts,
           index_found_seg;
        sscanf_s(str,"%d-%d-%d %d:%d:%d %d",&year,&month,&day,&
           hour,&minute,&second,&input_tzid);
        if(((year<=9999 && year >= 0000)&&(month>=01 && month
           <=12)&&(day >=01 && day <=31)&&(hour>=00 && hour<=23)
           &&(minute>=00 && minute<=59)&&(second >=00 && second
           <=59))&&(input_tzid==tz_id))
        {
                T= tm_to_timet(&year,&month,&day,&hour,&minute,&
                   second);
                t_interpolate=T;
```

```
current_num_sgts=num_sgts/2;
last=num_sgts-1;
middle=(first + last)/2;
last_index=mpt_sgts[middle].num_pts-1;
while((!finish) && (first<=last))
{

         LWPOINT *first_pt = (LWPOINT*) &(mpt_sgts[
            middle].geom_array[0]);
         LWPOINT *last_pt = (LWPOINT*) &(mpt_sgts[
            middle].geom_array[last_index]);

         if (first_pt->type == POINTTYPE)
         {
                  start_t  = lwpoint_get_m(first_pt)
                     ;

         }

         if (last_pt->type == POINTTYPE)
         {
                  end_t = lwpoint_get_m(last_pt);

         }

         if((t_interpolate <= end_t) && (
            t_interpolate >=start_t))
         {
                  finish = true;
         }
         else if(t_interpolate <start_t)
         {
                  last=middle - 1;
                  middle=(first + last)/2;
         }
         else
         {
                  first = middle + 1;
                  last = num_sgts - 1;
                  middle=(first + last)/2;
         }
         last_index=mpt_sgts[middle].num_pts-1;

}
if(finish){
         index_found_seg = middle;
         total_num_pts = mpt_sgts[index_found_seg].
            num_pts;
```

```
first_loc=0;
last_loc =total_num_pts -1;
middle_loc=(first_loc + last_loc)/2;

while((!found) && (first_loc<=last_loc))
{

        LWPOINT *point_middle = (LWPOINT*)
             &(mpt_sgts[index_found_seg].
            geom_array[middle_loc]);
        if (point_middle->type ==
            POINTTYPE)
        {
                X = lwpoint_get_x(
                    point_middle);
                Y = lwpoint_get_y(
                    point_middle);
                Z = lwpoint_get_z(
                    point_middle);
                t_p1 = lwpoint_get_m(
                    point_middle);
                time_point1 = t_p1;
        }
        if(t_p1 == t_interpolate)
        {
                found = true;
        }
        else if(t_p1 < t_interpolate)
        {
                first_loc = middle_loc +1;
                middle_loc=(first_loc +
                    last_loc)/2;

        }
        else
        {
                last_loc = middle_loc - 1;
                middle_loc=(first_loc +
                    last_loc)/2;
        }

}
if(found)
{
        return true;
}
else
{
```

```
                                        return false;
                                }

                        }
                        else
                        {
                                return false;
                        }
                }
```

mpt_present_atperiod

```
bool atperiod(mpt_segment *mpt_sgts,int num_sgts,int sr_id, int
    tz_id, Period *period)
{
        bool finish=false;
        int i=0;
        mpoint *new_mpt;
        mpt_segment *new_mpt_sgts= (mpt_segment*)malloc(sizeof(
            mpt_segment));
        time_t start_t, end_t,t_start,t_end,prev_last_t,
            nxt_first_t,nxt_first,nxt_last;
        int first=0,last,middle,last_index,current_num_sgts;
        int num_segts=0;
        LWPOINT *first_pt,*last_pt,*start_point,*end_point,*
            prev_pt,*nxt_pt,*nxt_first_pt,*nxt_last_pt;
        new_mpt= (mpoint *) palloc(sizeof(mpoint));
        last=num_sgts-1;
        new_mpt->num_segts = num_segts;
        while(!finish && i<num_sgts)
        {
                last_index = mpt_sgts[i].num_pts-1;
                first_pt = (LWPOINT*) &(mpt_sgts[i].geom_array[0])
                    ;
                last_pt = (LWPOINT*) &(mpt_sgts[i].geom_array[
                    last_index]);

                if (first_pt->type == POINTTYPE)
                {
                        start_t  = lwpoint_get_m(first_pt);

                }

                if (last_pt->type == POINTTYPE)
                {
                        end_t = lwpoint_get_m(last_pt);
                }
                if((start_t>= period->initial) && ( end_t <=period
                   ->final))
                {
                        if(num_segts > 0) {

                                new_mpt_sgts = (mpt_segment*)
                                    realloc(new_mpt_sgts , (1 +
                                    num_segts) * sizeof(
                                    mpt_segment));
                        }
                        new_mpt_sgts[num_segts]= mpt_sgts[i];
                        ++num_segts;
                }
                if(start_t > period->initial)
                {
                        finish = true;
                }
                i++;
        }
        if(finish)
        {
                return true;
        }
        else{
                return false;
        }

}
```