Supporting Movement in ORDBMS –

the ‘HERMES’ MOD Engine

Nikos Pelekis, Elias Frentzos, Nikos Giatrakos, Yannis Theodoridis

Information Systems Laboratory

Department of Informatics

University of Piraeus

Hellas

Technical Report Series

UNIPI-INFOLAB-TR-2010-01

July 2010

Supporting Movement in ORDBMS – the ‘HERMES’ MOD Engine

Nikos Pelekis, Elias Frentzos, Nikos Giatrakos, Yannis Theodoridis

Dept of Informatics,
University of Piraeus, Hellas
URL: http://infolab.cs.unipi.gr/
E-mail: {npelekis, efrentzo, ngiatrak, ytheod}@unipi.gr

Abstract

Composition of space and mobility in a unified data framework results into Moving Object Databases (MOD). MOD management systems support storage and query processing of non-static spatial objects and provide essential operations for higher level analysis of movement data. The goal of this paper is to present HERMES MOD engine that supports the afore-mentioned functionality through appropriate data types and methods in Object-Relational DBMS (ORDBMS) environments. In particular, HERMES exploits on the extensibility interface of ORDBMS that already have extensions for static spatial data types and methods that follow the Open Geospatial Consortium (OGC) standard, and extends the ORDBMS by supporting time-varying geometries that change their position and/or extent in space and time dimensions, either discretely or continuously. It further extends the data definition and manipulation language of the ORDBMS with spatio-temporal semantics and functionality based on advanced spatio-temporal indexing and query processing techniques. Its implementation over two ORDBMSs and its utilization in various domains proves the expressive power and applicability of HERMES in different application domains where knowledge regarding movement data of an organization is essential. As a proof-of-concept, in this paper HERMES is applied to a case study related with vehicle traffic analysis.

Keywords: spatio-temporal databases, data cartridge, Oracle

Table of Contents

1 Introduction

Due to the explosion of mobile devices, the positioning technologies and the low data storage cost, one of the most important assets of knowledge intensive organizations working with movement data, (i.e. Location-Based Services (LBS), traffic engineering, climatology, social anthropology and zoology, studying vehicle position data, hurricane track data, human and animal movement data, respectively etc.) is the data itself. Spatial database research has focused on supporting the modeling and querying of geometries associated with objects in a database 25. Regarding static spatial data, the major commercial as well as open source database management systems 11, 32, 34, 47, 51 already provide appropriate data management and querying mechanisms that conform to Open Geospatial Consortium (OGC) standards 35. On the other hand, temporal databases have focused on extending the knowledge kept in a database about the current state of the real world to include the past, in the two senses of “the past of the real world” (*valid time*) and “the past states of the database” (*transaction time*) 53. About a decade’s effort attempts to achieve an appropriate kind of interaction between both sub-areas of database research. Spatio-temporal databases are the outcome of the aggregation of time and space into a single framework 59, 47, 1, 37, 28 with up-to-date reviews of spatio-temporal models and systems proposed in the literature found in 44 and 20, respectively. As delineated in these papers, a serious weakness of existing approaches is that each of them deals with few common characteristics found across a number of specific applications. Thus the applicability of each approach to different cases, fails on spatio-temporal behaviors not anticipated by the application used for the initial model development. For the previous reasons, the field of the MOD has emerged 24, which has been shown 44 that it presents the most desirable properties among the proposals. However, although a lot of research has been carried out in the field of MOD, the efforts are independent trying to deal with specific problems and do not pay attention into embedding the proposed solutions (i.e. query processing algorithms) on top of existing DBMS where real world organizations base on. Towards this direction, the pioneer work of Guting et al. 24, 17 and 29 have proposed SECONDO system 2. However, SECONDO in contradiction to our approach is a stand-alone system, built from scratch, its design does not utilize the provided spatial extensions of existing ORDBMS, it does not conform to the OGC standards as it does not follow any predefined data model 12 and as such it is not embeddable into the DBMS infrastructure of an organization, where pure static spatial, as well as other types of data is stored.

The aim of this paper is to describe a robust framework capable of aiding either an analyst working with movement data, or more technically, a MOD developer in modeling, constructing and querying a database with objects that change location, shape and size, either discretely or continuously in time. Objects that change location or extent continuously are much more difficult to accommodate in a database in contrast to discretely changing objects. Supporting both types of spatio-temporal objects (the so-called *moving objects*) is exactly the challenge adopted by this paper. In detail, we present an integrated and comprehensive design of moving object data types in the form of extensible modules that can be embedded in OGC-compliant Object-Relational Database Management Systems (ORDBMS) taking advantage of their extensibility interface. The proposed HERMES *MOD Engine* provides the functionality to construct a set of moving, expanding and/or shrinking geometries, which are just variables of simple continuous functions that obtain hypostasis when projected to the spatial domain (i.e. becoming OGC spatial data types) at a specific instance in time. Each one of these moving objects is supplied with a set of methods that facilitate the user to query and analyze spatio-temporal data. Embedding this functionality offered by HERMES in an ORDBMS data manipulation language, one obtains a flexible, expressive and easy to use query language for moving objects that was not available so far in real OGC-compliant ORDBMS.

The implementation of such a framework is based on a set of basic types including base data types (i.e. integer, real, string and boolean, available in all DBMS), together with spatial data types offered by spatial extensions of OGC-compliant ORDBMS and temporal data types introduced in a temporal extension, called *TAU Temporal Literal Library* (TAU-TLL) 38. Based on these temporal and spatial object data types and the ideas behind the abstract data types for moving objects that have been introduced in 24, this paper discusses the design principles and the implementation issues concerning HERMES. The values of such moving types are functions that associate each instant in time, with an OGC spatial type, in contradiction to 24 whose design does not follow the OGC standards. A rich palette of suitable operations is defined on these types to support querying and to make moving object data management easier and more natural and sensible to users and applications.

Moreover, given the ubiquitousness of location-aware devices, databases handling moving objects will, sooner or later, face enormous volumes of data. It consequently arises that performance in the presence of vast data sizes, is a significant problem for moving object databases and the only way to deal with such enormous sizes is the exploitation of specialized access methods used for spatio-temporal indexing purposes. The domain of spatio-temporal indexing is dominated by the presence of the R-tree, along with its variations and extensions, which include, among others, 3D R-trees 55, TB-trees and STR-trees 46, PA-trees 33, and MON-trees 4. As in the case of appropriate moving object data types and methods for extending the type system of ORDBMS, except the well-known R-trees, which are suitable only for static spatial data, none of the above proposals have been incorporated into existing ORDBMS. Among them, the Trajectory Bundle tree (TB-tree) 46, is adopted in this work and appropriately designed and implemented inside HERMES taking advantage of the indexing extensibility interface of ORDBMS. Being a member of the R-tree family, TB-tree is able to support traditional queries such as range and distance-based queries. At the same time, it supports objects moving on the unconstrained space (it is general-purpose and not network-based such as the MON-tree 4), and is the only one that fulfills the need for trajectory preservation so as to efficiently support trajectory-based operations.

Furthermore, apart from simple query operators (e.g. range queries) natively supported by R-trees, there is a variety of spatio-temporal operators that are essential higher level analysis and which require more sophisticated query processing techniques in order to be efficiently processed. Among them, an important class of queries is the so-called *k* nearest neighbor (*k*-NN) search, where one is interested in finding the *k* closest trajectories to a predefined query object *Q* (stationary or moving). Thus, one of the challenges being present in the domain of trajectory databases is to develop mechanisms to perform *k*-NN search on MODs exploiting spatio-temporal indexes storing historical information. Among the solutions proposed in the literature we adopt the one proposed by 19 which efficiently supports Nearest Neighbor (NN) queries over historical trajectory data.

Finally, as we aim at providing a powerful toolkit for analysts, HERMES provides qualitatively different techniques for trajectory similarity search, which is exploited to support trajectory clustering and classification mining tasks that imply a way to quantify the distance between two trajectories. More specifically, we adopt a novel set of trajectory distance functions 41, 39 based on primitive (space and time) as well as derived parameters of moving objects (speed, acceleration, and direction), which are also capable to support sub-trajectory similarity matching. The overall framework advances the contribution of our approach by two inter-related facts: firstly, the combination of the similarity operators in the extended with MOD semantics SQL-like query language (using AND/OR clauses) provides analysis functionality unmatched so far (e.g. “find objects that moved closely in space but with very dissimilar speed patterns”); secondly, the output of each of the supported operators defines similarity patterns that can be utilized to reveal local similarity features (e.g. "find the most similar portions between two, in general, dissimilar trajectories").

Summarizing the previous discussion, the contributions of the paper are the following:

- We present a datatype-oriented model and an extension of SQL-like query language for supporting the modeling and querying of MOD on top of OGC-compliant ORDBMS.
- We describe the physical representation design decisions and the architectural aspects of our server-side MOD database engine, as well as the formulated interface for building advanced mobility-related applications.
- We demonstrate how novel, appropriate access methods and advanced, non-trivial query operators are embedded inside extensible ORDBMS providing efficiency and higher level analysis functionality.
- We investigate the expressive power and flexibility of the produced query language via a real-world application scenario.
- As a proof of concept, we have implemented the proposed framework on top of a commercial ORDBMS, namely Oracle, while our design has also been successfully applied and repeated in the open-source PostgreSQL with the PostGIS spatial extension 7.

To the best of our knowledge, HERMES is the first work that provides a complete framework for building MOD applications, which has been incorporated into two state-of-the-art OGC-compliant ORDBMS.

The outline of the paper is as follows: we first present the data type system for moving objects introduced in HERMES in an abstract way (Section 2) and then, we discuss implementation aspects (Section 3). An appropriate set of operations that extend the data definition and manipulation language of an ORDBMS with spatio-temporal semantics is discussed in Section 4. The overall architecture for implementing HERMES in a state-of-the-art ORDBMS, is presented in Section 5 together with a proof-of-concept case study related with vehicle traffic analysis. In Section 6 we assess the applicability of the proposed system in building other systems via presenting four tools and corresponding application domains that utilize HERMES as the platform for managing and analyzing their movement related data. An extensive discussion on the comparison of HERMES functionality with related work appears in Section 7. Finally, Section 8 concludes the paper, also pointing out some interesting future research directions.

2 A Data Type System for Moving Objects

The basic modeling primitives of the proposed moving object data type system are objects and literals. An *object* is a computational entity with a unique object identifier that encapsulates both state and behavior. The *state* of an object is defined by the values it carries for a set of properties. These *properties* can be attributes of the object itself or relationships between the object and one or more other objects. The *behavior* of an object is defined by a set of operations that can be executed on or by the object. On the other hand, a *literal* is a computational entity that has only state. Let *V* be a universe of all possible computational entities, containing objects and literals. A *type* is a set of elements of *V* that obey some technical properties. Each type is associated with a predicate function defined over the *V*. A value $v \in V$ satisfies a type iff the predicate is true for that value. A value that satisfies a type is called *member* of the type. A *type system* is a collection of types.

Types in the so-called *MOD Type System* are divided into *Base Types BT*, pure *Temporal Types TT*, pure OGC-compliant *Spatial Types ST* and *Moving Types MT*, i.e., the proposed *MOD Type System* is defined as:

$$MOD = BT \cup TT \cup ST \cup MT \qquad (1)$$

Figure 1 illustrates, in UML notation, all types in *MOD Type System*.

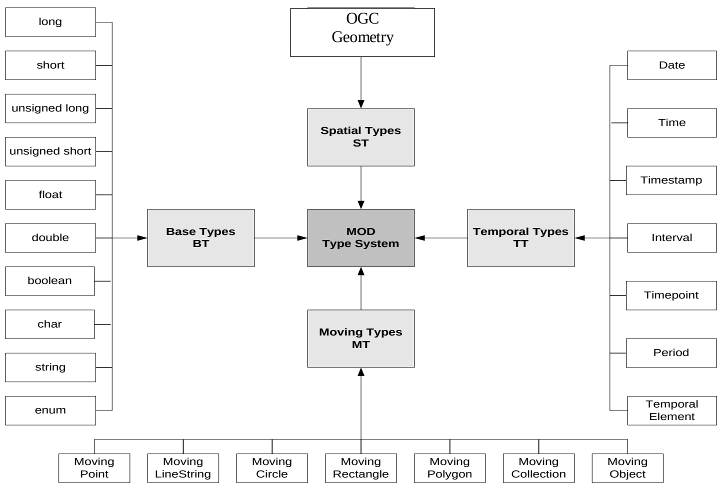


Figure 1 MOD Type System

2.1 Base, Temporal and Spatial Types

Base types are the standard database types built into any DBMS, such as integer and real (float) numbers, alphanumeric strings and booleans. These types form a subset of the Atomic Literal Types needed to define temporal types. The set *ALT* of *Atomic Literal Types* is defined as:

$$ALT = \Pi\,boolean\,T \cup \Pi\,char\,T \cup \Pi\,short\,T \cup \Pi\,ushort\,T \cup \Pi\,long\,T \cup \Pi\,ulong\,T \cup \Pi\,float\,T \cup \Pi\,double\,T \cup \Pi\,octet\,T \cup \Pi\,string\,T \cup \Pi\,enum\,T \quad (2)$$

where $\Pi * T$ denotes the domain of type *. For example, $\Pi\,boolean\,T = \{true, false\}$, $\Pi\,char\,T = \{x \mid x \in ASCII\}$, and so on.

Moving from base to temporal types, the set *TLT* of Temporal Literal Types is defined as 27, 38:

$$TLT = \Pi\,date\,T \cup \Pi\,time\,T \cup \Pi\,timestamp\,T \cup \Pi\,interval\,T \cup \Pi\,timepoint\langle g\rangle\,T \cup \Pi\,period\langle g\rangle\,T \cup \Pi\,temporalElement\langle g\rangle\,T \quad (3)$$

Basically, TLT augments the four temporal literal data types found in *ODMG* object model 8 (namely, *Date*, *Time*, *Timestamp* and *Interval*) with three new temporal object data types (namely, *Timepoint*, *Period* and *Temporal Element*). The widely used *Gregorian calendar* is implemented and the *discrete* model of time is adopted, where time is isomorphic to the integers because of its better representation and manipulation on databases. Time axis is partitioned into a finite number of discrete segments, called *granules* 58. The choice of a partitioning scheme is termed as *granularity*. The granularity of the timestamp that a fact is associated with denotes the precision to which the timestamp can be represented. Time order refers to whether the time axis can be always considered as linear or non-linear. In the linear model, time advances from past to future in a totally ordered form. The non-linearity of the time axis deals with aspects of the time such as *periodic time* and *branching time* 56. Formally:

$$\begin{aligned}
&date =_d \langle\!\langle year{:}\ GrYear,\ month{:}\ GrMonth,\ day{:}\ GrDay\rangle\!\rangle\\
&time =_d \langle\!\langle hour{:}\ GrHour,\ minute{:}\ GrMinute,\ second{:}\ GrSecond\rangle\!\rangle\\
&timestamp =_d date \parallel time\\
&interval =_d \langle\!\langle day{:}\ long,\ hour{:}\ GrHour,\ minute{:}\ GrMinute,\ second{:}\ GrSecond\rangle\!\rangle\\
&timepoint\langle g\rangle =_d tp\langle g\rangle \cup STV\\
&period\langle g\rangle =_d \{\langle\!\langle start{:}Timepoint\langle g\rangle,\ end{:}Timepoint\langle g\rangle\rangle\!\rangle \mid start \le end\},\ g \in granularity\\
&temporalElement\langle g\rangle =_d \{te{:}\ set\langle period\langle g\rangle\rangle \mid \forall i, j \cdot i \ne j \Rightarrow te_i \cap te_j = \varnothing\}
\end{aligned} \quad (4)$$

where the set *granularity* that contains elements that represent time accuracy according to the time divisions in the *Gregorian* calendar: $\Pi\ granularity\,T = \{YEAR, MONTH, DAY, HOUR, MINUTE, SECOND\}$, $tp\langle year\rangle =_d \langle\!\langle year{:}\ GrYear\rangle\!\rangle$, $tp\langle month\rangle =_d tp\langle year\rangle \parallel \langle\!\langle month{:}\ GrMonth\rangle\!\rangle$, ..., $tp\langle second\rangle =_d tp\langle minute\rangle \parallel \langle\!\langle second{:}\ GrSecond\rangle\!\rangle$ and $STV =_d \{beginning, forever, now\}$.

The four temporal literal data types found in *ODMG* object model 8 are augmented with three new temporal object data types presented below:

- *Timepoint*: extends the *Timestamp* data type to include granularity. The new data type is a subtype of the *Timestamp* data type. It inherits all the properties and the operations that are defined for the *Timestamp* data type. It refines all the operations, which had as argument *Timestamp* to *Timepoint*. *Beginning* and *forever* are defined to be members of *timepoint* such as $\forall t \in timepoint\langle g \rangle \cdot beginning \leq t \leq forever$

- *Period*: is used to represent an anchored duration of time, that is, duration of time with starting and ending points. A period has an associated granularity. The period is the composition of two timepoints with the constraint that the timepoint that starts the period equals or precedes the timepoint that terminates it. Without loss of generality, it is assumed that both timepoints have the same granularity. There are four categories of periods depending on whether they include their starting and/or their ending timepoints or not: $[t_1, t_2]$ (closed-closed), $[t_1, t_2)$ (closed-open), $(t_1, t_2]$ (open-closed), and (t_1, t_2) (open-open). Without loss of generality, *TAU Model* supports only closed-open periods, with which it is possible to model any other category. For example, the period $[t_1, t_2]$ is equivalent to the period $[t_1, t_2$+1 "granule"). The meaning of "1 granule" depends on the granularity of the period. For instance, if the granularity is day then the period $[t_1, t_2]$ is equivalent to the period $[t_1, t_2$+1*DAY).

- *Temporal Element*: is used to represent a finite union of disjoint periods. Temporal elements are closed under the set theoretic operations of union, intersection and complementation.

On the other hand, spatial types (point, line segment, rectangle, etc.) are supported by another component of the *MOD* type system architecture, called *OGC Geometry*. Such a spatial extension is found in several state-of-the-art ORDBMS (e.g. 11, 32, 34, 47, 51) and provides an integrated set of functions and procedures that enable spatial data following the OGC standard to be efficiently stored in a spatial database, accessed and futher processed. Of course, the geometric operations forming the behavior of spatial types supported by these extensions, handle queries statically, meaning that there exists no notion of time associated to the spatial objects. This is exactly the target addressed in the *MOD* type system we propose in the sequel.

2.2 Preliminaries of Moving Object Data Types

As already mentioned, the authors in 24, 17 and 29 introduce the concept of *sliced representation*, the basic idea of which is to decompose the temporal development of a moving value into fragments called *"slices"* such that within the slice this development can be described by some kind of *"simple"* function. This is illustrated in Figure 2 for a time-varying point (moving point).

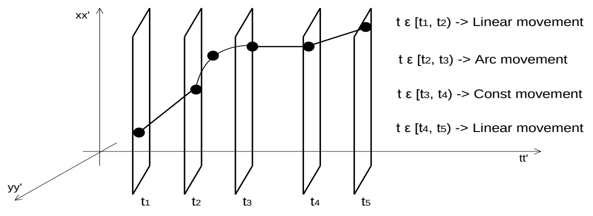

xx'
t ε [t1, t2) -> Linear movement
t ε [t2, t3) -> Arc movement
t ε [t3, t4) -> Const movement
t ε [t4, t5) -> Linear movement
tt'
yy'
t1
t2
t3
t4
t5

Figure 2 Moving Point with various types of movement

In this work, we adopt and extend the *sliced representation* concept and utilize it in the implementation of the MOD type system that results to HERMES. In order to use the sliced representation to define a moving type, one has to decompose the definition of each moving type into several definitions, one for each of the slices that corresponds to a simple function, and then compose these sub-definitions as a collection to define the moving type. Each one of the sub-definitions corresponds to a so-called *unit moving type*.

In order to define a unit moving type, we need to associate a period of time with the description of a simple function that models the behavior of the moving type in that specific time period. Based on this approach, two real world notions are directly mapped to our model as object types, namely *time period* and *simple function*. The first concept has been already introduced as one of the temporal literal types in *TLT* (closed-open *period* in formula (2)). The second concept is an object type, named *Unit_Function*, defined as a triplet of (x, y) coordinates together with some additional motion parameters. The first two coordinates represent the initial (x_i, y_i) and ending (x_e, y_e) coordinates of the sub-motion defined, while the third coordinate (x_c, y_c) corresponds to the centre of a circle upon which the object is moving. Whether we have constant, linear or arc motion between (x_i, y_i) and (x_e, y_e) is implied by a *flag* indicating the type of the simple function. Since we require that HERMES manages not only historical data, but also online and dynamic applications, we further let a *Unit_Function* to model the case where a user currently (i.e., at an initial timepoint) is located at (x_i, y_i) and moves with initial velocity v and acceleration a on a linear or circular arc route.

In the case of arc motions, following the categorization of realistic arc motions initially discussed in 62, we classify them according to the quadrant the motion takes place and motion heading (clockwise or counterclockwise). Figure 3 illustrates one of the possible eight cases (e.g. quadrant I - clockwise direction).

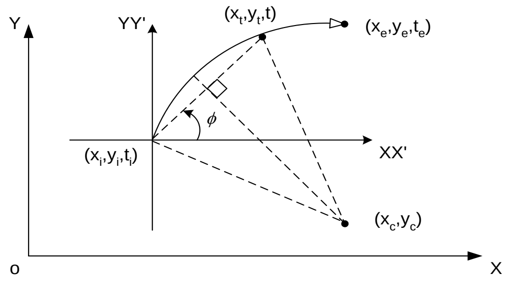


Figure 3 Motion on a circular arc

For constant and linear motions, the interpolation of a moving point's location in an intermediate timepoint *t* is straightforward. For arc motions, there is need of some trigonometric calculations. For the case of Figure 3 the necessary operations are illustrated in Eq. 5. Following a similar process, we develop all kinds of arc functions in each quadrant and direction.

$$\begin{aligned}
&ARC\_1(t) \Rightarrow (x_t, y_t) = (x_t + L_t \times Cos\phi, y_t + L_t \times Sin\phi)\\
&L_t = 2 \times R \times Sin(\frac{S_t}{2 \times R})\\
&S_t = v \times t + \frac{1}{2} \times a \times t^2\\
&\phi = \frac{\pi}{2} + Sin^{-1}(\frac{y_c - y_i}{R}) - \frac{S_t}{2 \times R}\\
&v \geq 0, t \in [t_i, t_e), \phi \in (-\infty, +\infty)
\end{aligned} \tag{5}$$

Consequently, in the general case the Unit_Function is defined as follows:

Unit_Function $=_d$ 《 *x_i:double, y_i:double, x_e:double, y_e:double, x_c:double, y_c:double, v:double, a:double, flag:TypeOfFunction*》 (6)

where *Π TypeOfFunctionT={ PLNML_1, ARC_<1..8>, CONST }*, meaning 1st order polynomial, one of the eight possible circular arcs, and constant function, respectively.

In the two sections that follow, we provide abstract definitions of the data types that compose the *MOD* type system that we propose as well as operations that exploit their functionality.

2.3 Abstract Definitions of Moving Object Data Types

Combining *time period* and *simple function* together, the most primitive and simplest unit object type is defined, namely *Unit_Moving_Point*. This is a fundamental type since all the successor unit types are defined based upon it. Formally:

Unit_Moving_Point $=_d$ 《*p: period*〈*SECOND*〉*, m: Unit_Function*》 (7)

Following this, we define two unit moving types directly based on *Unit_Moving_Point*, namely *Unit_Moving_Circle* and *Unit_Moving_Rectangle*. As it is easily inferred, these two object types model circle and rectangle geometry constructs that change their position and/or extent over time. Formally:

Unit_Moving_Rectangle= $_d${《 *ll:Unit_Moving_Point, ur: Unit_Moving_Point*》 | *equal (ll.p, ur.p)* } (8)

Unit_Moving_Circle=$_d$ { 《 *f: Unit_Moving_Point, s: Unit_Moving_Point, t: Unit_Moving_Point*》 | *equal (f.p, s.p, t.p)* } (9)

For modeling the subsequent object types (*Unit_Moving_Polygon* and *Unit_Moving_LineString*) an intermediate object type that represents the simplest built-in constituent of these types is needed. This requirement is met by the *Unit_Moving_Segment* object, which models a simple line or arc segment that changes its shape and size according to its starting and ending unit moving points. This is clarified in Figure 4 where a moving segment is mapped to a line segment at two different time instants t_1 and t_2. During the time period between t_1 and t_2, the starting moving point mp_1 follows a simple linear trajectory, while the ending moving point mp_2 follows an arc trajectory.

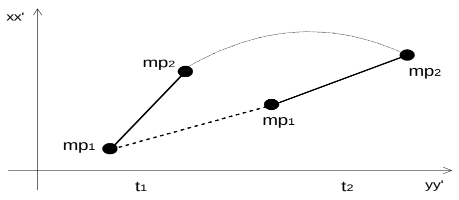


Figure 4 Linear Unit_Moving_Segment with its first Unit_Moving_Point mp_1 moving linearly and the second mp_2 moving on a circular arc

Formally:

Unit_Moving_Segment= $_d${《 *b:Unit_Moving_Point, e: Unit_Moving_Point, m: Unit_Moving_Point, kind:TypeOfSegment*》 | *(kind=SEG* $\Rightarrow$ *equal (b.p, e.p))* $\wedge$ *(kind =ARC* $\Rightarrow$ *equal (b.p, e.p, m.p))* } (10)

Unit_Moving_Linestring= $_d${*l:set*〈*Unit_Moving_Segment*〉 | $\forall i, j \in$ *ulong:* $i \neq j \Rightarrow$ *equal* $(l_i.b.p, l_j.e.p)$ } (11)

Unit_Moving_Polygon =$_d$ {《 *l: set*〈*Unit_Moving_Segment*〉*, hole:boolean* 》| $\forall i, j \in$ *ulong:* $i \neq j \Rightarrow$ *equal* $(l_i.b.p, l_j.e.p)$ } (12)

where Π *TypeOfSegment* T = {*SEG, ARC*} and SEG, ARC denote the two alternative modes of interpolation in between two end points (line segment vs. arc, respectively).

Having defined the fundamental unit moving types, we now introduce the moving types that play the dominant role in our spatio-temporal data type system. The process that we followed to define the moving types is to introduce a moving type as a collection of the corresponding unit moving type, which means, in terms of object orientation, that there exists a composition relationship between the unit moving type and the moving type. As such, the *Moving_Point*, *Moving_Circle*, *Moving_Rectangle*, *Moving_LineString* and *Moving_Polygon* object types are introduced as a collection of *Unit_Moving_Point*, *Unit_Moving_Circle*, *Unit_Moving_Rectangle*, *Unit_Moving_LineString*, *Unit_Moving_Polygon* object types, respectively. Formally:

$$\mathit{Moving\_Point} =_d \{\, p: \mathit{set}\langle \mathit{Unit\_Moving\_Point}\rangle \mid \forall i, j \in \mathit{ulong},\ 1 \le i, j \le |\mathit{set}\langle \mathit{Unit\_Moving\_Point}\rangle| : j = i+1 \Rightarrow p_i.p < p_j.p \wedge \neg \mathit{overlaps}(p_i.p, p_j.p) \wedge \forall t \in \mathit{double}: \mathit{inside}(t, p_i.p) \Rightarrow \mathit{at\_instant}(p, t) \in \mathit{OGC\text{-}GEOMETRY}_{GTYPE}{=}\mathit{point} \,\} \tag{13}$$

$$\mathit{Moving\_Rectangle} =_d \{\, r: \mathit{set}\langle \mathit{Unit\_Moving\_Rectangle}\rangle \mid \forall i, j \in \mathit{ulong},\ 1 \le i, j \le |\mathit{set}\langle \mathit{Unit\_Moving\_Rectangle}\rangle| : j = i+1 \Rightarrow r_i.ll.p < r_j.ur.p \wedge \neg \mathit{overlaps}(r_i.ll.p, r_j.ur.p) \wedge \forall t \in \mathit{double}: \mathit{inside}(t, r_i.ll.p) \Rightarrow \mathit{at\_instant}(r, t) \in \mathit{OGC\text{-}GEOMETRY}_{GTYPE}{=}\mathit{rectangle} \,\} \tag{14}$$

$$\mathit{Moving\_Circle} =_d \{\, c: \mathit{set}\langle \mathit{Unit\_Moving\_Circle}\rangle \mid \forall i, j \in \mathit{ulong},\ 1 \le i, j \le |\mathit{set}\langle \mathit{Unit\_Moving\_Circle}\rangle| : j = i+1 \Rightarrow c_i.f.p < c_j.s.p \wedge \neg \mathit{overlaps}(c_i.f.p, c_j.s.p) \wedge \forall t \in \mathit{double}: \mathit{inside}(t, c_i.f.p) \Rightarrow \mathit{at\_instant}(c, t) \in \mathit{OGC\text{-}GEOMETRY}_{GTYPE}{=}\mathit{circle} \,\} \tag{15}$$

$$\mathit{Moving\_LineString} =_d \{\, \mathit{line}: \mathit{set}\langle \mathit{Unit\_Moving\_LineString}\rangle \mid \forall i, j \in \mathit{ulong},\ 1 \le i, j \le |\mathit{set}\langle \mathit{Unit\_Moving\_LineString}\rangle| : j = i+1 \Rightarrow \mathit{line}_i.l_1.b.p < \mathit{line}_j.l_1.e.p \wedge \neg \mathit{overlaps}(\mathit{line}_i.l_1.b.p, \mathit{line}_j.l_1.e.p) \wedge \forall t \in \mathit{double}: \mathit{inside}(t, \mathit{line}_i.l_1.b.p) \Rightarrow \mathit{at\_instant}(\mathit{line}, t) \in \mathit{OGC\text{-}GEOMETRY}_{GTYPE}{=}\mathit{linestring} \,\} \tag{16}$$

$$\mathit{Moving\_Polygon} =_d \{\, \mathit{pol}: \mathit{set}\langle \mathit{Unit\_Moving\_Polygon}\rangle \mid \forall i, j \in \mathit{ulong},\ 1 \le i, j \le |\mathit{set}\langle \mathit{Unit\_Moving\_Polygon}\rangle| : j = i+1 \Rightarrow \mathit{pol}_i.l_1.b.p < \mathit{pol}_j.l_1.e.p \wedge \neg \mathit{overlaps}(\mathit{pol}_i.l_1.b.p, \mathit{pol}_j.l_1.e.p) \wedge \forall t \in \mathit{double}: \mathit{inside}(t, \mathit{pol}_i.l_1.b.p) \Rightarrow \mathit{at\_instant}(\mathit{pol}, t) \in \mathit{OGC\text{-}GEOMETRY}_{GTYPE}{=}\mathit{polygon} \,\} \tag{17}$$

Similarly, in order to model homogeneous collections of moving types, *multi*-moving types are defined as collections of the corresponding moving types. Consequently, the proposed spatio-temporal model is augmented by the following object types: *Multi_Moving_Point*, *Multi_Moving_Circle*, *Multi_Moving_Rectangle*, *Multi_Moving_LineString* and *Multi_Moving_Polygon*. Formally (and assuming that the spatial extension of the underlying ORDBMS supports *multi*-spatial types):

$$\mathit{Multi\_Moving\_Point} =_d \{\, \mathit{multi\_mpoint}: \mathit{set}\langle \mathit{Moving\_Point}\rangle \mid \forall i, j \in \mathit{ulong} \wedge \forall t \in \mathit{double}: \mathit{inside}(t, \mathit{multi\_mpoint}_i.p_j.p) \Rightarrow \cup_i\, (\mathit{at\_instant}(\mathit{multi\_mpoint}_i, t)) \in \mathit{OGC\text{-}GEOMETRY}_{GTYPE}{=}\mathit{multi\text{-}point} \,\} \tag{18}$$

$$\mathit{Multi\_Moving\_LineString} =_d \{\, \mathit{multi\_mline}: \mathit{set}\langle \mathit{Moving\_LineString}\rangle \mid \forall i, j \in \mathit{ulong} \wedge \forall t \in \mathit{double}: \mathit{inside}(t, \mathit{multi\_mline}_i.\mathit{line}_j.l_1.b.p) \Rightarrow \cup_i\, (\mathit{at\_instant}(\mathit{mult}_i\mathit{\_mline}_i, t)) \in \mathit{OGC\text{-}GEOMETRY}_{GTYPE}{=}\mathit{multi\text{-}linestring} \,\} \tag{19}$$

$$\mathit{Multi\_Moving\_Circle} =_d \{\, \mathit{multi\_mcircle}: \mathit{set}\langle \mathit{Moving\_Circle}\rangle \mid \forall i, j \in \mathit{ulong} \wedge \forall t \in \mathit{double}: \mathit{inside}(t, \mathit{multi\_mcircle}_i.c_j.f.p) \Rightarrow \cup_i\, (\mathit{at\_instant}(\mathit{multi\_mcircle}_i, t)) \in \mathit{OGC\text{-}GEOMETRY}_{GTYPE}{=}\mathit{multi\text{-}polygon} \,\} \tag{20}$$

$$\mathit{Multi\_Moving\_Rectangle} =_d \{\, \mathit{multi\_mrectangle}: \mathit{set}\langle \mathit{Moving\_Rectangle}\rangle \mid \forall i, j \in \mathit{ulong} \wedge \forall t \in \mathit{double}: \mathit{inside}(t, \mathit{multi\_mrectangle}_i.r_j.ll.p) \Rightarrow \cup_i\, (\mathit{at\_instant}(\mathit{multi\_mrectangle}_i, t)) \in \mathit{OGC\text{-}GEOMETRY}_{GTYPE}{=}\ \mathit{multi\text{-}polygon} \,\} \tag{21}$$

$$\mathit{Multi\_Moving\_Polygon} =_d \{\, \mathit{multi\_mpolygon}: \mathit{set}\langle \mathit{Moving\_Polygon}\rangle \mid \forall i, j \in \mathit{ulong} \wedge \forall t \in \mathit{double}: \mathit{inside}(t, \mathit{multi\_mpolygon}_i.\mathit{pol}_j.l_1.b.p) \Rightarrow \cup_i\, (\mathit{at\_instant}(\mathit{multi\_mpolygon}_i, t)) \in \mathit{OGC\text{-}GEOMETRY}_{GTYPE}{=}\ \mathit{multi\text{-}polygon} \,\} \tag{22}$$

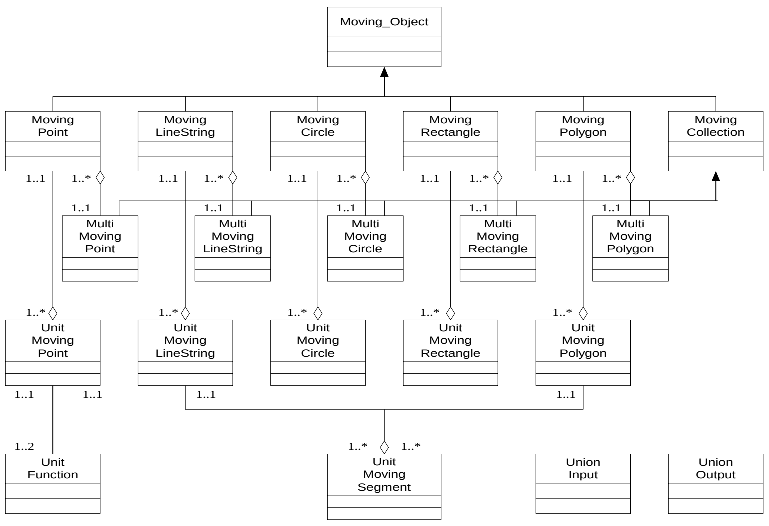


Figure 5 The moving types of MOD type system

An interesting issue here is that the previously mentioned multi-moving types do not carry their own methods interface. All the functionality for these types can be invoked by the methods of another object type, called *Moving_Collection*, standing as the supertype and aggregating the interfaces, the object methods and the spatio-temporal semantics of all the multi moving types. Furthermore, the moving-collection type is able to represent heterogeneous collections of moving types, i.e., collections of different time-varying spatial geometries. Formally:

$$\begin{aligned}
&Moving\_Collection =_d \{ \langle\langle\, multi\_mpoint: Multi\_Moving\_Point,\ multi\_mline: \\
&\quad Multi\_Moving\_LineString,\ multi\_mcircle: Multi\_Moving\_Circle, \\
&\quad multi\_mrectangle: Multi\_Moving\_Rectangle,\ multi\_mpolygon: \\
&\quad Multi\_Moving\_Polygon \,\rangle\rangle \mid \\
&\quad \forall i, j \in ulong \wedge \forall t \in double: inside(t, multi\_mpoint_i.p_j.p) \wedge inside(t, \\
&\quad multi\_mline_i.line_j.l_1.b.p) \wedge inside(t, multi\_mcircle_i.c_j.f.p) \wedge inside(t, \qquad (23) \\
&\quad multi\_mrectangle_i.r_j.ll.p) \wedge inside(t, multi\_mpolygon_i.pol_j.l_1.b.p) \Rightarrow [\, (\cup_i \\
&\quad (at\_instant(multi\_mpoint_i, t))) \cup (\cup_i (at\_instant(multi\_mline_i, t))) \cup (\cup_i \\
&\quad (at\_instant(multi\_mcircle_i, t))) \cup (\cup_i (at\_instant(multi\_mrectangle_i, t))) \cup \\
&\quad (\cup_i (at\_instant(multi\_mpolygon_i, t))) \,] \in OGC\text{-} \\
&\quad GEOMETRY_{GTYPE}{=}collection \}
\end{aligned}$$

The concept of inheritance is also utilized at the level of moving types by introducing an object that encapsulates all semantics and functionality offered by moving types, including *Moving_Collection*. The so-called *Moving_Object* type is the conjunction of all the previously defined object types, which implies that this object can completely substitute any other moving type. Furthermore, the *Moving_Object* models any moving type that can be the result of an operation between moving objects. For example, the intersection of a *Moving_Point* with a (static) polygon geometry is obviously another *Moving_Point* that is the restriction of the first *Moving_Point* inside the polygon. This result can be modeled as a *Moving_Object*. If the result of an operation is not a moving geometry then *Moving_Object* plays the role of a degenerated moving type. In other words, if there is an operation that requests the perimeter of *Moving_Polygon*, then obviously the result of this method is a time-varying real number (*Moving_Real*). Such collapsed moving types like *Moving_Real*, *Moving_String*, and *Moving_Boolean* do not formally exist in our type system but are modeled using the *Moving_Object* object type. Formally:

$$\begin{aligned}&\mathit{Moving\_Object} =_d \{\langle\!\langle\ \mathit{mobject: Moving\_Object,\ mpoint: Moving\_Point,\ mline:}\\ &\quad \mathit{Moving\_LineString,\ mcircle: Moving\_Circle,\ mrectangle: Moving\_Rectangle,}\\ &\quad \mathit{mpolygon: Moving\_Polygon,\ mcolection: Moving\_Collection,\ geometry:}\\ &\quad \mathit{GEOMETRY,\ gtype: GeometryType,\ optype: string,\ arg1: ushort,\ arg2: ushort,}\\ &\quad \mathit{input: Union\_Input}\ \rangle\!\rangle\}\end{aligned} \tag{24}$$

where *gtype* is a flag that makes *Moving_Object* behave as if it were a simple moving type, Π *GeometryType* T = { *MOBJECT, MPOINT, MLINE, MCIRCLE, MRECTANGLE, MPOLYGON, MCOLLECTION* } and *Union_Input* $=_d$ $\langle\!\langle$*mask: string, tolerance: double, distance: double*$\rangle\!\rangle$.

Summarizing, Figure 5 illustrates a UML class diagram for the moving types supported in the proposed *MOD* type system.

2.4 Spatiotemporal Indexing in Hermes

In this section we briefly introduce the basic notions of spatio-temporal indexing and present the TB-tree which is adopted in this work and implemented in HERMES. Similar to the original R-tree, the TB-tree is a height-balanced tree with the index records in its leaf nodes; leaf nodes contain entries of the same trajectories, and are of the form S = ⟨*MBB*, *Orientation*⟩, where MBB is the 3D bounding box of the 3D line segment belonging to an object's trajectory (handling time as the third dimension) and *Orientation* is a flag used to reconstruct the actual 3D line segment inside the MBB among four different alternatives that exist (see figure Figure 7). Moreover, contrary to the well-known B-tree, and similarly to the original R-tree, internal and leaf node MBBs belonging to the same tree level are allowed to overlap. Each internal or leaf node in the tree corresponds to a physical *disk page* (or disk block, which is the fundamental element on which the actual disk storage is organized) and contains between *m* and *M* entries (*M* is the node capacity and *m* in the case of TB-tree is set to 1).

Since each leaf node contains entries of the same trajectory, the object *id* can be stored once in the leaf node header. Therefore, TB-tree leaf nodes are of the form ⟨*header*, {S_i}⟩, where each S_i = ⟨MBB_i $Orientation_i$⟩ and header = ⟨*id*, *#entries*, *ptrCur*, *ptrParent*, *ptrNext*, *ptrPrevious*⟩ (in other words, the object identifier, the number of node entries and four pointers, to the current, the parent, and the next and previous nodes of the same trajectory). On the other hand, non-leaf nodes are of the form ⟨*header*, {E_i}⟩, where each E_i = ⟨MBB_i, ptr_i⟩ with MBB_i be the enclosing 3D box of the child node pointed by ptr_i (a pointer to it), and *header* = ⟨*#entries*, *ptrCur*, *ptrParent*⟩ simply stores the number of node entries and a pointer to itself and to its parent node. Furthermore, similar to SETI 9 and in order to support high insertion rates, our TB-tree implementation uses an in-memory hashed front-line structure, which maintains tuples of the form ⟨*id*, P_{curr}, N_{curr}⟩ with the object identifier *id*, its latest position P_{curr} = ⟨t_{curr}, x_{curr}, y_{curr}⟩ and a pointer N_{curr} to the leaf node containing P_{curr}.

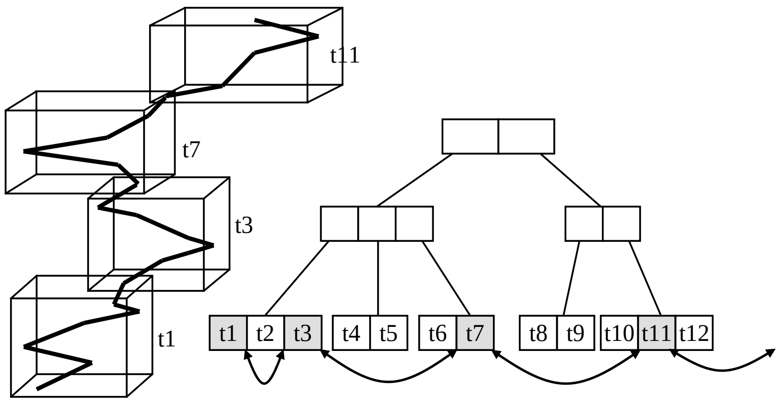


Figure 6 The TB-tree structure

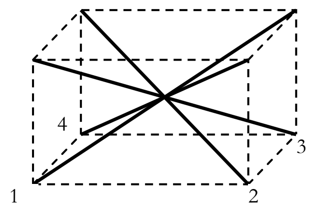


Figure 7 Alternative ways that a 3D line segment can be contained inside a *MBB*

Given the size of a disk block, which is predetermined by the operation system, the number of elements contained in a leaf of internal node in the tree is resticted by it. Specifically, given that each S_i is contained in 25 bytes (4 bytes for each one of the 6 double precission numbers needed to describe the MBB and 1 byte for the orientation flag) and the header of each leaf node has the size of 16 bytes (4 bytes for each one of the object identifier, the number of entries, and the four pointers), the total leaf capacity in terms of trajectory segments is given by *Int(([page size]*-16*)/*25*)*; this number for a typical page size of 4096 bytes results in 163 trajectory segments inside each leaf node. Following the same reasoning each internal tree node has a capacity of 170 entries, resulting in 170 child nodes.

The difference of the TB-tree with the majority of the R-tree variations relies on the way the index is built. Specifically, its insertion algorithm is not based upon the spatial and temporal relations of moving objects (or moving object segments) but it relies only on the moving object identifier (*id*). When new line segments are inserted, the algorithm searches for the leaf node containing the last entry of the same trajectory, and simply inserts the new entry in it, thus forming leaf nodes that contain line segments from a single trajectory. Furthermore, its split strategy is very simple: when a leaf node is full, a new one is created and is inserted at the right-end of the tree; due to the monotonicity of time, this strategy ensures that trajectories are organized monotonically inside the tree structure, e.g., trajectory segments are organized by time. For each trajectory, a double linked list connects the leaf nodes that contain its portions together (Figure 6), resulting in a structure that can efficiently answer trajectory-based queries.

2.5 The TB-tree Data Types

In this section we introduce the data types required for embedding the TB-tree in an ORDBMS that supports moving objects. We should note that these data types are transparent to the user of HERMES and their usage is just for the internal construction of the tree. The implementation of a tree-based index under the object-relational model follows a number of well-known rules and techniques, such as implementing different object classes for each one of the basic tree elements, namely, tree nodes (root, internal nodes, leafs) and node elements. Figure 8 below provides an abstract, though insightful, view of the index organization, along with the connection with the rest of the HERMES data types in the form of a UML class diagram describing the structure's primitives. The left part of the diagram depicts the objects participating in the index formation. Following a top-down description, the *tbTreeIdx* class is used mainly for completeness as an abstraction of the corresponding part of the model and it refers to the definition of TB-tree index on the table where the actual trajectory data are stored. Since the main trajectory table may initially be empty, the corresponding aggregation with the lower-level *tbTreeElement* class possesses a cardinality of «0..*».

Descending the diagram, we observe that the whole arrangement is separated in two kinds of *TB-tree Node* types. Namely, the *tbTreeNode* Class regarding the internal nodes of the tree structure and the *tbTreeLeaf* class used to represent the leaf nodes of the index where trajectory segments are stored. Given that the size of each leaf node is predermined and equivalent to the chosen disc block size, its capacity in terms of trajectory line segments is also predefined (following the previous discussion, a page size of 4096 bytes results in leaf nodes fitting no more than 163 segments). As a consequence, exceeding the aforementioned size, in terms of leaf node entries, causes segments of the same trajectory to be stored in different leaf nodes which remain connected by means of a double linked list. This is denoted using an association termed as *"linked"*. Note that the head leaf of the list might be connected to at most 1 (or 0 when the trajectory fits in a block) other leaves and the same holds for the tail of the arrangement. Each intermediate node is essentially linked to two other peers. This explains the cardinality of the respective association.

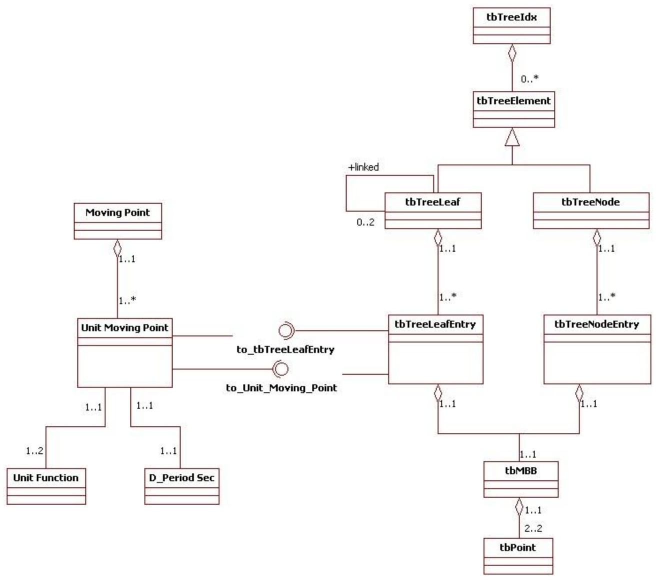


Figure 8 TB-tree data types

A *tbTreeLeaf* includes a number of leaf entries (*tbTreeLeafEntry* in Figure 8), each consisting of the MBB (*tbMBB* in the figure) that surrounds the trajectory segment kept in the leaf entry, along with an integer number 1-4 denoting its orientation; *tbMBB*s is composed by a *MinPoint* and a *MaxPoint* of *tbPoint* type which are the lower left and upper right of the box, respectively in the spatio-temporal space, while *tbpoint* has only a property of *tbX* collection type, which is an array of size 3 used to hold triplets (*x*,*y*,*t*) of time-stamped positions forming the entire object's trajectory. More spectifically, the attributes of *tbTreeLeaf* are:

- *MoID* of integer type which is the global trajectory identifier,
- *ptrCurrentNode* of integer type, being the current node's identifier encapsulated in the object to facilitate implementation issues,

- *ptrParentNode* of integer type, representing a pointer to the parent of the current node used to ascend the tree when necessary,
- *ptrPreviousNode* of type integer, which is a pointer to the node containing the previous fragment of the same trajectory,
- *ptrNextNode* of type integer, which is a pointer to the node containing the next fragment of the same trajectory,
- *LeafEntries,* a collection of *tbTreeLeafEntry* type with fixed capacity, which involves the current leaf entries as previously described, and,
- *count* of integer type that holds the cardinality of *LeafEntries*.

Formaly, given the leaf capacity *LeafCapacity*, i.e., the maximum number of leaf entries that may be contained in a leaf node, we define the following types:

$$tbPoint =_{d} \{tbX{:}set\langle double\rangle \mid |tbX|=3\} \tag{25}$$

$$\begin{aligned} tbMBB =_{d} \{&\langle MinPoint{:}tbPoint,\ MaxPoint{:}tbPoint\rangle \mid \forall\, 0\leq i\leq 2,\\ &MinPoint.x(i)<=MaxPoint.x(i)\} \end{aligned} \tag{26}$$

$$tbTreeLeafEntry =_{d} \{\langle MBB{:}tbMBB,\ Orientation{:}\ short\rangle \mid Orientation<4\} \tag{27}$$

$$\begin{aligned} tbTreeLeaf =_{d} \{&\langle MovingObjectId{:}long,\ ptrCurrentNode{:}long,\\ &ptrParentNode{:}long,\ ptrNextNode{:}long,\ ptrPreviousNode{:}long,\\ &LeafEntries{:}\ set\langle tbTreeLeafEntry\rangle,\ count{:}long \mid |LeafEntries|\leq\\ &LeafCapacity,\ count=|LeafEntries|\rangle\} \end{aligned} \tag{28}$$

Similarly, a *tbTreeNode* contains a set of *tbTreeNodeEntry* objects; each *tbTreeNodeEntry* encloses all the the leaf or node entries contained in the sub-tree starting with this node as root. More spectifically, its attributes involve:

- *ptrCurrentNode* of integer type, which is the current node's identifier encapsulated in the object to facilitate implementation issues,
- *ptrParentNode* of integer type, which is a pointer to the parent of the current node used to ascend the tree when necessary,
- *NodeEnties,* a collection of *tbTreeNodeEntry* type with fixed capacity, which involves the current node entries as previously described, and,
- *count* of integer type to hold the cardinality of *NodeEntries*.

Formally, given the node capacity *NodeCapacity* we define:

$$tbTreeNodeEntry =_d \{ \langle MBB{:}tbMBB,\ ptr{:}\ long \rangle \} \tag{29}$$

$$\begin{aligned} tbTreeNode =_d \{ \langle & ptrParentNode{:}long,\ ptrCurrentNode{:}long,\ NodeEntries{:} \\ & set\langle tbTreeNodeEntry \rangle,\ count{:}long \mid |NodeEntries| \leq NodeCapacity, \\ & count = |NodeEntries| \rangle \} \end{aligned} \tag{30}$$

Eventually, the two interfaces of Figure 8 *to_tbTreeLeafEntry*, *to_Unit_Moving_Point* provide essential mechanisms for object transformation from one type to the other.

The following sections describe the design decisions and the implementation details for mapping the *MOD* type system into extensible ORDBMS, as well as essential functionality for extending SQL-like query languages with *MOD* querying constructs.

3 Physical Mapping of the Hermes MOD Type System

The physical representation of the data types reflects the structures that are necessary in order to capture the semantics and implement the methods of these data types. In this section, we discuss how *MOD* types (abstractly described in Section 3) are mapped to physical structures for storing continuously and discretely time-evolving geometric data into an ORDBMS with OGC-compliant spatial extension. The following subsections propose low-level constructs for the implementation of such objects and illustrate the design decisions and implementation issues considered during development.

3.1 Unit Function

Unit_Function is constructed as an octave of real numbers and a flag indicating the type of the simple function. In the current version, three types of functions are supported, namely polynomial of first degree, circular arc and the constant function.

The modeling of *Unit_Function* is extensible; for example, if one wishes to add interpolations with spline or polynomials with degree higher than one, then what is only needed to be done is the addition (if necessary) of the appropriate variables as attributes of the object and the implementation of such a function.

We should note that we model a moving type that changes discretely for a period of time by setting all *Unit_Function* objects of the corresponding unit-moving type to be constant functions. Due to the fact that the coordinates represented by these *Unit_Function* objects do not change for this period of time, it is equivalent to taking a snapshot of the moving geometry, which is valid for the entire period. If at least one of these unit functions is not constant then the moving type change is continuous for this period of time. In case of a moving linestring and in order to model a discrete change for a period, the above assignment should take place for all unit-moving points that compose the corresponding unit-moving linestring. What is more, if this process were continued to all unit-moving types the result would be a completely discretely changing moving geometry.

3.2 Moving Point, Moving Circle and Moving Rectangle

We construct *Moving_Point* object type as a collection of *Unit_Moving_Point* objects (i.e. pointer to a nested table or a varying length array (i.e. varray), depending on the underlying ORDBMS, of *Unit_Moving_Point* objects), which in turn are defined as objects consisting of two attributes. The first attribute is the time period during which the other attribute is defined. The time period is expressed as an open-closed *Period* object, while the other attribute is of *Unit_Function* object type, whose domain of definition is the set of real numbers inside the open interval [t_1, t_2), where t_1 is the starting point of the period and t_2 is the ending point of the period.

Similarly to the *Moving_Point* object, *Moving_Circle* and *Moving_Rectangle* object types are constructed as pointers to collections of *Unit_Moving_Circle* and *Unit_Moving_Rectangle*, respectively. Even though these two types could be modeled as special instances of *Moving_Polygon* object, it is a design decision to distinguish them both for simplicity and flexibility reasons as well as for implementation reasons. The motivation for defining distinct object constructors for these moving types is that both of them need just a small, predefined number of unit types, in contrast to the moving polygon, where the number of its sub-elements is unknown and generally large. What is more, this important distinction facilitates the mapping of these moving types to their corresponding pure spatial geometries and makes the process of finding degenerated moving types at specific time instants easier.

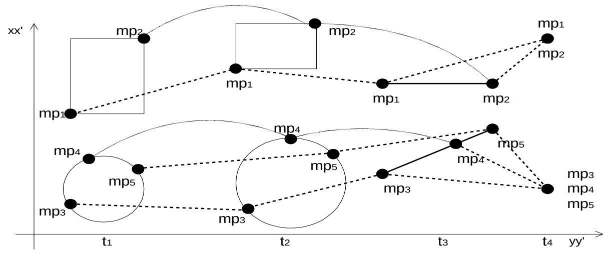


Figure 9 Instances of Moving_Circle and Moving_Rectangle type objects (and of degenerated cases)

Let us now examine the structure of *Unit_Moving_Circle* and *Unit_Moving_Rectangle* objects. *Unit_Moving_Circle* consists of three *Unit_Moving_Point* objects, representing the three points needed to define a valid circle. In the same way, *Unit_Moving_Rectangle* is composed of two *Unit_Moving_Point* objects, modeling the lower-left and upper-right point needed to define a valid rectangle. Figure 9 illustrates a moving circle and a moving rectangle instantiated at four different time points t_1, t_2, t_3, and t_4, respectively. At time point t_2, it is clear to see the effect of the different interpolation functions and how they affect the position and extent of the mapped geometries, in contrast to time point t_1. At time point t_3, a degenerated moving circle and a degenerated moving rectangle are presented, meaning that the three unit moving points that compose the moving circle become co-linear and the two unit moving points that compose the moving rectangle form a line segment that is parallel to either *xx'* or *yy'* axis. At timepoint t_4, another collapsed state is depicted, where all unit-moving points become equal. HERMES implementation is responsible to deal with such degeneracies as will be discussed in Section 5.1.

3.3 Moving LineString and Moving Polygon

Moving_LineString is a moving type that is also constructed as a pointer to a nested table consisting of *Unit_Moving_LineString* objects. The difference between this moving type and the previously defined is that the *Unit_Moving_LineString* is also defined as a pointer to another nested table comprising of *Unit_Moving_Segment* objects. *Unit_Moving_Segment* in its turn is formed by three *Unit_Moving_Point* objects and a flag indicating the kind of interpolation between the starting and the ending point of the *LineString* geometry. The simplest part of a *LineString* geometry can be either a linear or an arc segment. In other words, this flag exemplifies the usage of the other attributes of the *Unit_Moving_Segment* object. Figure 10 illustrates the structure of the *Moving_LineString* object.

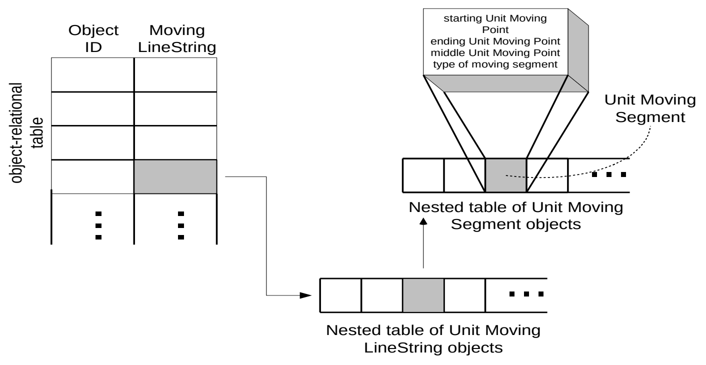


Figure 10 Structure of the Moving_LineString Object

The *Moving_Polygon* definition is very close to that of *Moving_LineString*. The main difference in the two definitions is on the construction of the corresponding unit moving type. More specifically, apart from a pointer to a collection of *Unit_Moving_Segment* objects, the *Unit_Moving_Polygon* object has an additional attribute, a flag that indicates if this set of moving segments forms the exterior ring of a polygon or is an interior (hole) ring. In other words, this extra attribute adds the logic that disjoint moving holes may exist inside a moving polygon, with boundaries not crossing or touching the exterior boundary. Considering the rest aspects of the definition of *Unit_Moving_Polygon*, there is no difference between the two object types.

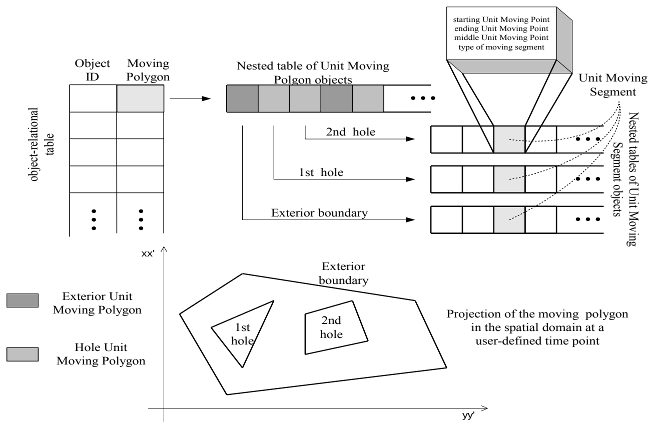


Figure 11 Structure of the Moving_Polygon Object

Actually, the difference between these two moving types comes from the different utilization of their collections of moving segments by the object methods. For example, an operation that maps a *Moving_LineString* to a *LineString* geometry checks for inequality on the starting and ending points of the line and this is a prerequisite for constructing the OGC geometry. On the contrary, the corresponding method for a moving polygon checks for the opposite, in order to be able to construct a valid OGC polygon. Another discrepancy of *Moving_Polygon*, in contrast to all the other moving types, is that in case it includes interior moving holes, then several *Unit_Moving_Polygon* objects need to be accessed in order to transform it to its corresponding spatial geometry at a specific instant (see Figure 11).

3.4 Moving Collection and Moving Object

Moving_Collection is the object type that models both homogeneous and heterogeneous collections of moving types. This is accomplished by defining this object as a set of five pointers to each of the following types: *Multi_Moving_Point*, *Multi_Moving_LineString*, *Multi_Moving_Circle*, *Multi_Moving_Rectangle* and *Multi_Moving_Polygon*. Each of these moving types represents a homogeneous collection of moving points, linestrings, circles, rectangles and polygons, constructed as a pointer to a nested table of *Moving_Point*, *Moving_LineString*, *Moving_Circle*, *Moving_Rectangle* and *Moving_Polygon* object types, respectively.

On the other hand, *Moving_Object* is the outcome of the conjunction of all the previous presented moving objects, and can be considered as the supertype of these types. Practically speaking, it is not intended to be directly used or constructed by a data cartridge user. On the contrary, it is intended to be the result type of operations of the other moving types (i.e., system generated). As inferred from the structure of *Moving_Object* (cf. formula (24)), the pointers to the moving types presented in the preceding sections model the subtypes of the current (super) type simulating inheritance.

3.5 Implementation of the TB-tree in HERMES

Regarding the data types required for the TB-tree index, they are mainly implemented as objects with simple attributes and arrays of attributes. Specifically:

- *tbPoint* is constructed as a standard array of real values with its cardinality set to 3 (*x*, *y* and *t*)
- *tbMBB* is constructed by two attributes of type *tbPoint*
- *LeafEntry* is constructed by an attribute of *tbMBB* type and another one of integer type taking values from 1 to 4, representing one among the four possible orientations that a line segment may have inside its MBB.
- *tbTreeLeaf* is constructed by the integer value of *MovingObjectId*, and a set of pointers (integer values), i.e., *ptrCurrentNode*, *ptrParentNode*, *ptrPreviousNode* and *ptrNextNode*. It also contains a standard array of *tbTreeLeafEntries* with predetermined size *LeafCapacity*, and an integer value containing the number of occupied entries inside the array.
- Similarly, a *tbTreeNode* is constructed by the two pointers (integer values), *ptrCurrentNode* and *ptrParentNode,* and a standard array of *tbTreeNodeEntries* with predetermined size *NodeCapacity*. Finally, an integer value containing the number of occupied entries inside the aforementioned array is employed inside the *tbTreeNode* structure.

Regarding the implementation of the TB-tree in the HERMES a number tables constituting the primary storage elements of index data are employed. Specifically, following the UML of Figure 8, the basic data types are stored in the following tables. Note also that these tables are automatically created/dropped upon the respective index creation/drop:

- *movingobjects*: The *movingobjects* is an auxiliary table used to store a pointer to the index leaf where the last part of a trajectory is stored 18. As such, it contains only 2 columns for the *trajectory id*, and for the pointer integer values.
- *tbTtreeidx_non_leaf*: This is the table storing the internal tree nodes. It actually contains tuples of the form (*NodeId*, *tbTreeNode*), where *NodeId=tbTreeNode.ptrCurrentNode*.
- *tbTreeidx_leaf*: This is the table storing the the tree leaf nodes; it also contains tuples of the form (*LeafId*, *tbTreeLeaf*) where *LeafId=tbTreeLeaf.ptrCurrentNode*.

4 Operations on Moving Object Data Types

Following, we classify the operations of the moving types introduced by HERMES into appropriate categories that enable us to describe and analyze the new query capabilities. The initial set of operations is the union of the methods supported by the simple moving types (namely, *Moving_Point*, *Moving_LineString*, *Moving_Circle*, *Moving_Rectangle*, *Moving_Polygon* and *Moving_Collection*). This set of operations is equivalent to the methods provided by the generic *Moving_Object* type as it models all the previous.

The identifiable classes of operations that HERMES supports are:

i) *Consistency operations*: operations responsible for keeping the database in a consistent state (checking ordering and consecutiveness of periods of unit moving types, realizing degenerated cases, etc.).

ii) *Predicates*: operations that return boolean values concerning topological and other relationships between moving types (within distance, meet, overlap, etc.).

iii) *Projection operations*: operations that restrict and project moving types to temporal (e.g. at_instant, at_period) and spatial domain (e.g. trajectory, buffer).

iv) Distance and Direction operations

v) *Set operations*: basic set relationship operations (union, intersection, set difference).

vi) *Numeric operations*: functions that compute a numeric value (e.g., the perimeter or the area of a moving polygon, the speed of a moving point).

vii)*Similarity functions*: a set of trajectory distance functions based on primitive (space and time) as well as derived parameters of trajectories (speed, acceleration, and direction).

viii) *Index maintenance*: necessary operations for creating, dropping and updating the TB-tree index.

ix) *Index operators*: several advanced algorithms for efficient query processing of movement data.

The following sections describe the functionality of selected operations, representative of each class. The interested reader may find signatures and more algorithms in 43.

The interested reader may find signatures, more algorithms and special behavior of the operations in Appendix D.

4.1 Maintaining the Database Consistent

HERMES-MDC provides a set of object methods that enable the user to check the construction data of moving objects and maintain the database in a consistent state. These operations impose some integrity constraints that need to be followed for time-varying spatial data and, as such, protect the user from errors that have to do with the complex internal structure of the moving types. There are six such object methods, which we illustrate below:

boolean ***check_periods_equality*** *(): Check_periods_equality* checks if the periods of the *Unit_Moving_Point* objects of each one of the unit moving types that form a moving geometry are equal. In other words, we do not permit the existence of a moving type that consists of several unit moving types and at least one of them describes the motion of its component *Unit_Moving_Point* objects with different *D_Periods_Sec* objects. Of course, such a method does not have any meaning for *Moving_Point*, as each of its unit moving types consists of only one *Unit_Moving_Point* object.

boolean ***check_sorting*** *(): Check_sorting* does not force any constraint per unit moving level. On the contrary, the rule it entails, is that there should be an ascending sorting of the periods between the unit moving types, each one represented by such a period. Such a constraint is required to model the evolution of the moving types in the time line. The evolution of an object is represented by its consecutive unit moving types and the corresponding time periods should follow the same development.

boolean ***check_disjoint*** *(): Check_disjoint* assures that the *D_Periods_Sec* objects that represent the time period for which the unit moving types are defined, are disjoint and that they do not intersect in any point in the time axis. More specifically, this operation checks if a period *”overlaps”* with the next in sorting-order period, namely the period of the next unit moving type.

boolean ***check_meet*** *(): Check_meet* is an operation that can be invoked only by a user and is not utilized internally by the data cartridge. It checks if a period *”meets”* with the next period in the unit-type-order. This object method has as a precondition the three previous operations, meaning that except the *“meet”* criterion that should stand between periods of sequential unit moving types, all the previous operations should return true. The meaning of this operation is to assure that there is a smooth transformation of the time-changing geometries between sequential unit moving types and there are not *temporal gaps* between them. Figure 2 is an example where the *“meet”* constraint is satisfied in the transition of a moving point, as well as the *“sort”* and *“disjoint”* constraints.

boolean ***check_degeneracies*** *(D_Timepoint_Sec): Check_degeneracies* is a method that checks if the geometry associated with a moving type at a specific point in the continuous time axis is a non-degenerated geometry. More specifically, this method finds the unit moving type (if there is one), whose period attribute (*D_Period_Sec* object) *“contains”* the time point (*D_Timepoint_Sec* object) passed as argument to the method. Afterwards, it interpolates the internal unit functions for that instant of time, imposing some rules and constraints upon the produced points in the Cartesian system of coordinates.

Depending on the type, *Check_degeneracies* imposes different restrictions on the development of these moving objects at user-defined time points. For *Moving_Point* there is not such an operation as there is no combination of mapped coordinates that could form an invalid geometry. For the rest of the *simple* moving types, the reader can find below some characteristic constraints enforced by HERMES-MDC:

Moving_LineString: (a) Checks if the *Unit_Moving_Point* objects (two for line segments; three for arc segments) that define the *Unit_Moving_Segment* objects become equal at a specific time point, thus degenerating a segment to a point; (b) Checks for overlapping between consequent *Unit_Moving_Segment* objects, meaning that the two time-varying ordinates of a *Unit_Moving_Point* *“fall”* upon the segment that is defined by the two previous *Unit_Moving_Point* objects; (c) Checks the coordinates of the starting *Unit_Moving_Point* of the first *Unit_Moving_Segment* *not* to be equal at an instant, with the coordinates of the ending *Unit_Moving_Point* of the last *Unit_Moving_Segment*. In such a situation, the potential LineString is degenerated to a *Polygon* geometry, regardless the fact that this polygon may have other anomalies (e.g. self-intersected segments that are acceptable in a *LineString* geometry); (d) In case of *"arc" Unit_Moving_Segment* the method checks for co-linearity at a specific time point between the three *Unit_Moving_Point* objects that form the *"arc"* moving segment. In this situation the arc segment becomes a degenerated linear segment.

Moving_Circle: (a) Checks if the three *Unit_Moving_Point* objects that define a *Unit_Moving_Circle* object become equal at a specific time point, thus degenerating a circle to a point; (b) Assures that the three *Unit_Moving_Point* objects do not become co-linear.

Moving_Rectangle: (a) Checks if the lower left and upper right *Unit_Moving_Point* objects that define a *Unit_Moving_Rectangle* object become equal at a specific time point, thus degenerating a rectangle to a point; (b) Checks if the *X* or the *Y* ordinates of the projected lower left and upper right *Unit_Moving_Point* objects become equal, meaning that the produced rectangle is collapsed to a linear segment parallel to *xx'* or *yy'* axis, respectively.

Moving_Polygon: (a) Checks for the same rules and constraints as in the case of *Moving_Linestring*, with the difference that, instead of inequality, it imposes equality between the starting and ending *Unit_Moving_Point*; (b) Checks if the *Unit_Moving_Polygon* objects that represent holes of a *Moving_Polygon* are always *"disjoint"* and *"inside"* the exterior boundary.

Varchar2 ***validate_geometry*** *(D_Timepoint_Sec, err_msg): Validate_geometry* is a generic method that performs a consistency check for valid moving geometry types. More specifically, this operation utilizes all the previous *"check"* methods by executing them in the order that we presented them, by this way producing a *control pattern* for each moving type. After applying this control pattern, the *validate_geometry* method invokes the *"at_instant"* operation, which maps a moving type to an *Sdo_Geometry* at a specific time point. Subsequently, this pure spatial object is examined under some principles that stand for the geometry model of Oracle10g. For example, polygons should have at least four points, which includes the point that closes the polygon, linestrings should have at least two points and in a multi-polygon, all polygons must be disjoint. Finally, the *validate_geometry* method following these tests returns *'TRUE'* if the moving type is valid, an Oracle error message number based on the specific reason the time-varying geometry is invalid or *'FALSE'* if the moving type fails for some other reason.

In the previous paragraphs, we described the operations concerning the constraints that should hold in a database of *"simple"* moving objects. The corresponding methods of a homogeneous or heterogeneous collection of such *"simple"* moving types, represented by the *Moving_Collection* object follow a different strategy. In other words, these operations traverse one by one all the component objects of the multi moving types that compose a *Moving_Collection* object, and apply the previous discussed operations to them. The first moving type that causes an error or is detected to be invalid or degenerated stops this process and informs the cartridge user with an appropriate message.

In the case of *Moving_Object*, these methods function differently according to the kind of *Moving_Object*. If a *Moving_Object* is just a wrapper of a simple moving type or a homogeneous or heterogeneous collection of them, then these operations just invoke the corresponding method of the wrapped moving type and return the result. If *Moving_Object* represents a time-varying object as the result of an operation between moving types (including *Moving_Collection*), or between a moving type and a static geometry, then HERMES-MDC applies the corresponding method to the moving types that participate on the construction of the *Moving_Object* and combines the separate outcomes to form the concluding result.

4.2 Predicates Modeling Topological and Distance Relationships

HERMES-MDC provides object methods in the form of predicates to describe relationships between moving types. There are two sets of predicates supported by HERMES-MDC, namely *within_distance* and *relate*. Each set of predicates consists of eight operations, each of which models the relationship of the current moving type with a *Moving_Point*, a *Moving_LineString*, a *Moving_Circle*, a *Moving_Rectangle*, a *Moving_Polygon*, a *Moving_Collection*, a *Moving_Object* and a *Sdo_Geometry* object. Each operation comes with two different overloaded signatures, modeling different semantics: the first signature is time-dependent, meaning that the outcome of the operation is related to a user-defined time point, while the second is independent to the time dimension. Below, the reader can find the pair of signatures of only one of the eight operations, and more specifically, those describing relationship with a *Moving_Polygon*. The time-dependent signature of the method is the one without the brackets, while the time-independent version of the operation can be obtained by substituting the return type of the operation with the type in the brackets { } and by removing the *D_Timepoint_Sec* argument from the parameter list. This is a common notation in the remainder of the paper.

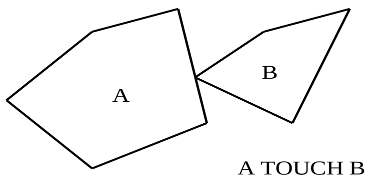
boolean {Moving_Object} ***f_within_distance*** *(distance, Moving_Polygon, tolerance, D_Timepoint_Sec):* The time-dependent predicate determines whether two moving objects are within some specified Euclidean distance from each other at a user-defined time point. After mapping the moving objects to physical spatial geometries at the given instant, the function returns *TRUE* for object pairs that are within the specified distance; returns *FALSE* otherwise. The distance between two non-point objects (such as lines and polygons) is defined as the minimum distance between these two objects. Thus, the distance between two adjacent polygons is zero.

Many object methods in HERMES-MDC accept a *tolerance* parameter. If the distance between two points is less than or equal to the tolerance, the cartridge considers the two points to be a single point. Thus, tolerance is usually a reflection of how accurate or precise users perceive their spatio-temporal data to be. *Within_distance* is a characteristic example for understanding the semantics of the *tolerance* parameter. Also, the time-independent *within_distance* operation differs from the above predicate in that the return value is a *Moving_Object* that represents a time-varying boolean value. This implicitly defined *"moving boolean"* object models the sequence of the time intervals that the two related objects are within or not some specified Euclidean distance.

Varchar2 {Moving_Object} ***f_relate*** *(mask, Moving_Polygon, tolerance, D_Timepoint_Sec):* This generic predicate examines two moving objects and determines their topological relationship. As previously, the *"relate"* predicate appears with two overloaded versions. The first evaluates the topological relationship upon a specific user-defined time point, while the second version returns a *Moving_Object* modeling a time-varying string (*"moving string"*), which describes the evolution in the topological relationship between the related objects. The user can specify the kind of relationships that he/she requires to check via the *mask* parameter.

The *"relate"* operator implements a 9-intersection model for categorizing binary topological relations between moving geometries [EF91]. At any time, each object has an interior, a boundary, and an exterior. The boundary consists of points or lines that separate the interior from the exterior. The boundary of a line consists of its end-points. The boundary of a polygon is the line that describes its perimeter. The interior consists of points that are in the object but not on its boundary and the exterior consists of those points that are not in the object.

Given that an object A has three components (a boundary A_b, an interior A_i, and an exterior A_e), any pair of objects has 9 possible interactions between their components. Pairs of components have an empty (0) or a non-empty (1) set intersection. The set of interactions between two projected moving geometries is represented by a 9-intersection matrix that specifies which pairs of components intersect and which do not. Figure 12 shows the 9-intersection matrix for two polygons that are adjacent to one another. This matrix yields the following bit mask, generated in row-major form: "101001111". For more details on topological relationships supported and respective values of *mask* parameter, see Appendix D.



$$\begin{array}{cc|ccc|}
 & & & B & \\
 & & b & i & e \\
 & b & 1 & 0 & 1 \\
A & i & 0 & 0 & 1 \\
 & e & 1 & 1 & 1
\end{array}$$

Figure 12 9-Intersection Matrix

4.3 Projection and Interaction to Temporal and Spatial Domain

HERMES-MDC provides object methods of special interest that have been proposed in the literature. Subsequently, we present the operations as these are defined for *Moving_Object* and the semantics behind these methods and we differentiate our presentation in case of change in the semantics of other moving types.

Unit_Moving_Point ***unit_type*** *(D_Timepoint_Sec):* This operation is the single method not defined for a *Moving_Object* type. Generally speaking, this operation is defined only for the *simple* moving objects that their construction is closely related with a collection of unit moving objects. For the rest of the *simple* moving objects the above signature changes the result type to their corresponding unit moving object (see [Pel02]). The simple but very important task that this function performs is that it finds (and returns) the unit-moving object whose attribute time period (*D_Period_Sec* object) *"contains"* the user-defined time point (*D_Timepoint_Sec* object). In other words, it returns that unit-moving type where the time instant represented by the argument *D_Timepoint_Sec* object is *"inside"* the time period that characterizes the unit-moving type. What is more, the *unit_type* method carries out all the necessary checks to maintain the database consistent and to ensure the validity of the moving object.

Union_Output ***at_instant*** *(D_Timepoint_Sec):* The *at_instant* operation is the most important method for the moving types introduced in HERMES-MDC, firstly because it is the operation that maps the abstract variables of mathematical functions to meaningful spatial objects conceivable by end-users and, secondly, because it is the base of implementation for many other object methods. As already mentioned, the above signature concerns the *at_instant* operation for the *Moving_Object* type. The return type (*Union_Output*) is an object that represents the union of all the possible results of the projection of a *Moving_Object* at a user-defined time point. In other words, if *Moving_Object* represents a time-varying geometry then *Union_Output* is basically an *Sdo_Geometry* object. If *Moving_Object* represents a *"moving"* real or string then *Union_Output* is a real number or a character string, respectively.

In the case of a *Moving_Object* the *at_instant* operation invokes the *at_instant* operations of the moving types that construct the *Moving_Object*. If *Moving_Object* represents a moving geometry then the result of the previous operation is immediately returned. If *Moving_Object* represents a *"moving"* type as the result of an operation between moving objects then the projected geometries of the previous step are applied against this operation and the outcome of this second step is returned.

In the case of *Moving_Collection*, this operation invokes the *at_instant* operations of all the moving types of the multi moving objects and subsequently applies a special *"union"* operation upon the projected geometries by *"concatenating"* them in a collection object and returns the result of the *"concatenation"*.

Moving_Object ***at_period*** *(D_Period_Sec):* The *at_period* object method is an operation that restricts the moving object to the temporal domain. In other words, by using this function the user can delimit the time period that is meaningful to ask the projection of the moving object to the spatial domain. More specifically, the time period passed as argument to the method is compared with all *D_Period_Sec* objects that characterize the unit moving objects. If the parameter period does not overlap with the compared period then the corresponding unit type is omitted. If it overlaps, then the time period that defines a unit-moving object becomes its *"intersection"* with the given period.

D_Temp_Element_Sec ***f_temp_element*** *():* The *f_temp_element* operation gives HERMES-MDC user the capability to project the time periods that form the unit moving objects that compose a moving type on the time line and subsequently *"concatenate"* all these distinct time periods to construct a temporal element. Figure 13 depicts the result of the *f_temp_element* operation when applied to a *Moving_Point* object.

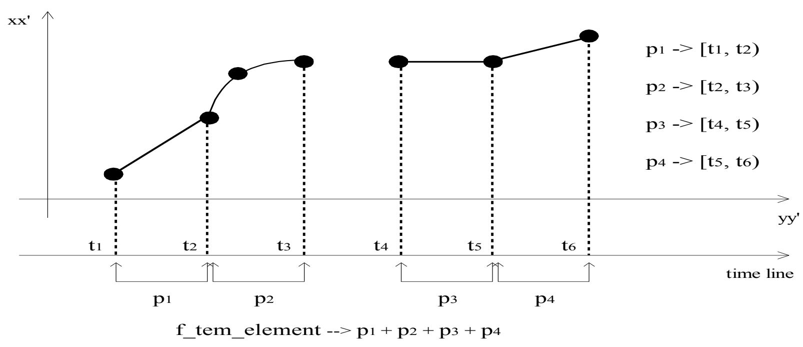


Figure 13 Projection of a Moving Point on the temporal domain

Moving_Object ***at_temp_element*** *(D_Temp_Element_Sec):* Similarly to the *at_period* operation, the *at_temp_element* object method restricts the moving object to the temporal domain, but the process of restricting the periods between which the moving object is valid is driven by a collection of *D_Period_Sec* objects and not just one *D_Period_Sec* object as in the previous case.

Sdo_Geometry {Moving_Object} ***f_buffer*** *(distance, tolerance, D_Timepoint_Sec):* The *f_buffer* operation comes with two overloaded versions. The first generates a buffer polygon around a moving geometry object at a specific user-defined time point, while the second version returns a *Moving_Object* modeling a time-varying polygon, which describes a moving rounded buffer around a moving geometry. Obviously, this method is meaningless for a *Moving_Object* that represents a time-varying real number or string. Calling the *f_buffer* method for such a *Moving_Object* triggers the error handling mechanism of HERMES-MDC, which informs the user with an appropriate message.

The *f_buffer* operation for a homogeneous collection of moving geometries at a specific timepoint returns a multi-polygon where each polygon represents the buffer of its corresponding element in the collection. An interesting case is the buffer of a heterogeneous collection of moving objects, which is just one polygon that buffers all the different projected geometries together. The above-mentioned issues are visualized in Figure 14, where snapshots of different moving types and their corresponding buffer polygons are presented.

What is not illustrated in the description of the operation is the specific structure of these buffers for each corresponding moving type. Starting with the *Moving_Point*, someone would expect that the buffer of this type at a specific instant would be a circle geometry with radius the user-specified distance of the buffer. Surprisingly, the geometry returned by *f_buffer* operation is a polygon consisting of two arc segments that circle the point at the specified distance. The same happens in the case of the *Moving_Circle* where the buffer at a specific timepoint is defined as the buffer of its centre but the distance of the buffer now is the initial user-specified distance plus the radius of the moving circle at that instant. The buffer of a *Moving_LineString*, a *Moving_Rectangle* and a *Moving_Polygon* at a specific timepoint is a compound polygon whose number of linear segments is equal to the number of linear segments that exist in the corresponding projected geometries and whose number of arc segments is equal to the number of vertices plus the number of arc segments.

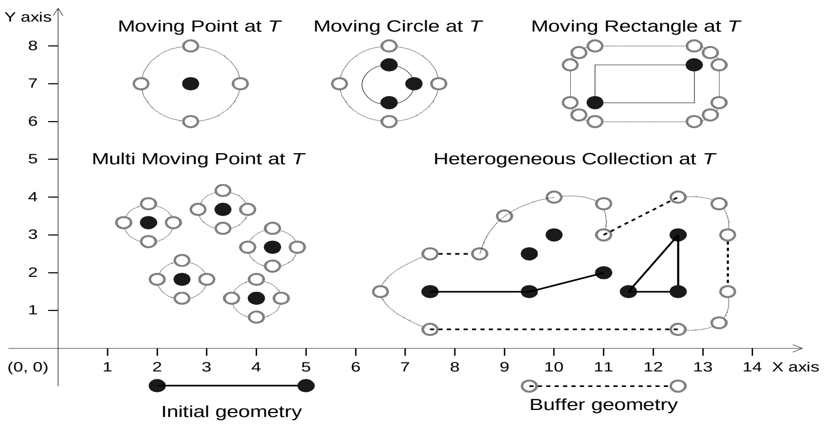


Figure 14 Demonstrating f_buffer operation

Sdo_Geometry {Moving_Object} ***f_centroid*** *(tolerance, D_Timepoint_Sec):* The *f_centroid* operation returns the centre of a moving polygon object at user-defined time points. The centre is also known as the *"centre of gravity"*. The overloaded *f_centroid* function represents a moving point that at any time is the centre of gravity of the moving polygon object. The method is meaningful only for moving types that model single time-varying areas. In all other cases, (collections of moving geometries) an application error is raised informing the cartridge user.

An interesting case presented when utilizing this operation is once the centre of gravity of the moving region falls out of its area. This could happen when the moving hole inside a moving polygon includes the centre and when a moving polygon becomes too concave at a specific timepoint. Both cases are visualized in Figure 15.

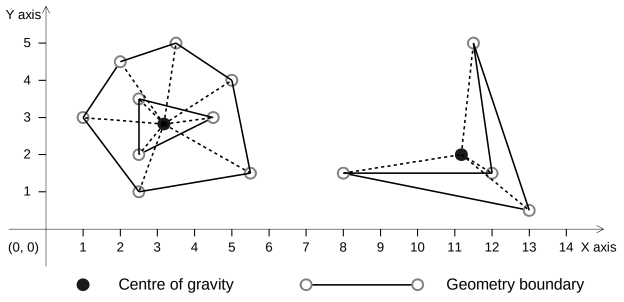


Figure 15 Demonstrating f_centroid operation

Sdo_Geometry {Moving_Object} ***f_convexhull*** *(tolerance, D_Timepoint_Sec):* The *f_convexhull* method returns a simple convex polygon that completely encloses the moving geometry object at a specific instant of time. The *Moving_Object* returned by the second time-independent *f_convexhull* function, models a moving polygon that is the convex hull of a moving object at any time point. HERMES-MDC uses as few straight-line sides as possible to create the smallest polygon that completely encloses an instantiated moving object (see dashed lines in Figure 16). A convex hull is a convenient way to get an approximation of a complex geometry object.

Sdo_Geometry {Moving_Object} ***f_pointonsurface*** *(tolerance, D_Timepoint_Sec):* This function returns a point geometry object representing a point that is guaranteed to be on the surface of a moving polygon when projected to the spatial domain at the time point used as argument. The returned point can be any point on the surface. The user should not make any assumption about where on the surface the returned point is, or whether the point is the same or different when the function is called multiple times with the same input parameter values. The second version of the *f_pointonsurface* operation returns a *Moving_Object*, which models a moving point whose mapping at any instant will be a point that is guaranteed to be on the surface of the corresponding projected polygon at the same time point.

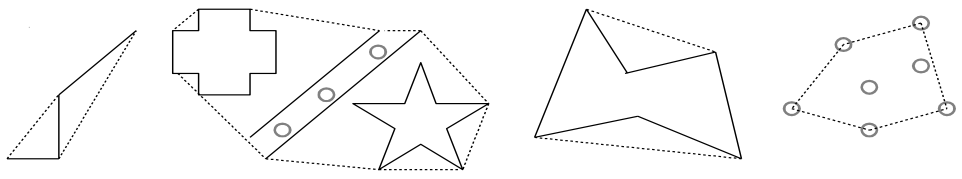


Figure 16 Convex polygons containing snapshots of several moving geometries

Union_Output ***f_initial*** *():* The *f_initial* object method is basically the *at_instant* operation invoked at the first instant of time that the moving object is valid, meaning the first second of the closed-open period that identifies the least recent unit moving object.

Union_Output ***f_final*** *():* Similarly to the *f_initial* object method, the *f_final* operation projects the moving object at the last valid instant of the time period that characterizes the most recent unit moving object.

Sdo_Geometry ***f_traversed*** *():* The geometry returned by this function models all the places that a moving geometry *"traverses"* along its motion during the periods that characterize the unit moving objects. Such a geometry object is of polygon type. In the case of *Moving_Point* objects, the *f_traversed* method is transformed to a special operator (*f_trajectory*) described in the subsequent paragraph. Figure 17 illustrates four examples of traversed areas, one for each of the simple moving types. In the case of the traversed *Moving_LineString*, we notice that the returned geometry is not a single polygon but a multi polygon due to the fact that the periods of the unit moving objects that compose the *Moving_LineString* do not *"meet"* each other or the variables that define the unit functions between subsequent unit moving objects present a substantial difference.

Sdo_Geometry ***f_trajectory*** *():* This function is the *f_traversed* method for the case of a *Moving_Point* object. In other words, this operation simulates the trajectory traversed by a *Moving_Point*. More specifically, this projection of the movement of a *Moving_Point* to the Cartesian plane is done by mapping the time-dependent ordinates of the object at the beginning, ending and a random intermediate time instant of each one of the periods that identify the *Unit_Moving_Point* objects that compose the *Moving_Point*. Subsequently, the algorithm examines whether the intermediate projected co-ordinates "fall" upon the line formed by the other two pairs of co-ordinates. Depending on the result, a linear or arc segment connecting the beginning and ending projected co-ordinates is implied. A process of merging these segments follows, to form the returned *LineString* geometry.

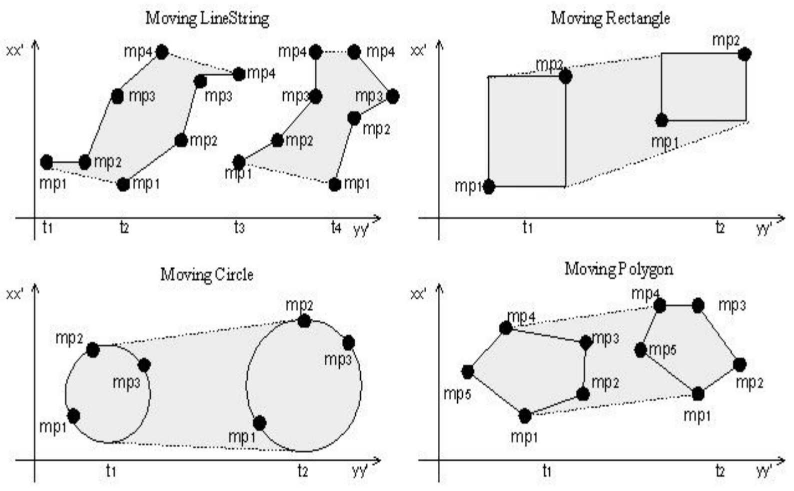


Figure 17 Areas Traversed by Moving Geometries

Sdo_Geometry ***f_locations*** *():* The *f_locations* object method is defined only for a Moving_Point object or a Moving_Object and follows the same algorithm as the *f_trajectory* operation with the difference that the returned type is a multipoint geometry representing the previously discussed projected co-ordinates at the beginning and ending timepoints of the periods that characterize the *Unit_Moving_Point* objects.

4.4 Numeric operations

HERMES-MDC supports a special category of object methods that either compute a numeric value of a moving object at a specific timepoint (e.g., the current perimeter of a moving polygon) or construct a *Moving_Object* representing the same time-varying numeric value. More analytically, we provide the subsequent numeric operations:

number {Moving_Object} ***f_area*** *(tolerance, D_Timepoint_Sec):* The *f_area* operation is defined for those moving types that their projection to the Cartesian plane depicts a closed region and computes the area for this region. The second (time-independent) version of the method returns a *Moving_Object* representing the time-varying area of a moving, extending and/or shrinking region. This function works with any moving polygon, including polygons with moving holes.

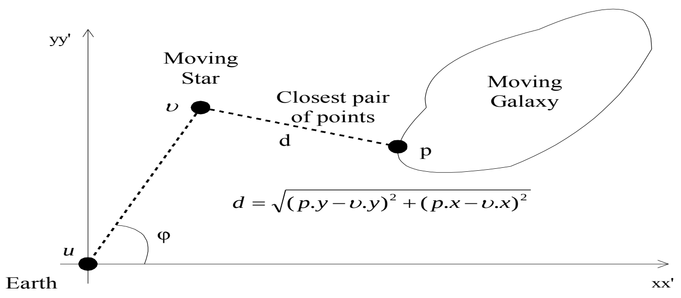
number {Moving_Object} ***f_length*** *(tolerance, D_Timepoint_Sec):* The *f_length* object method computes the length of a Moving_LineString object or the perimeter of a Moving_Circle, Moving_Rectangle or Moving_Polygon projected at the Cartesian plane at a user-defined time point. For a Moving_Polygon that contains one or more holes, this function calculates the perimeters of the exterior boundary and all holes at the given time point, and returns the sum of all the perimeters. The second version of the method returns a Moving_Object representing the time-varying length or perimeter of the moving type that invokes the operation.

Varchar2 {pls_integer} ***f_num_of_components*** *({mtype Varchar2}):* This operation is introduced only for *Moving_Collection* objects and its functionality is to estimate and return a structured string that describes the number of components that compose the collection of moving types. The second version of this object method takes a string describing a moving geometry as parameter and returns the number of the objects of the same type that participate in the construction of the moving collection.

4.5 Distance and Direction operations

The following two methods assist the cartridge user to calculate the minimum distance between moving objects or the angle formed between moving points.

number {Moving_Object} ***f_distance*** *(Moving_Polygon, tolerance, D_Timepoint_Sec):* HERMES-MDC provides a distance measure that exists for all moving types, which either computes the distance between two instantiated moving objects (the time-dependent version) or returns a time-varying real number that represents the minimum distance between these moving types at all time points (the time-independent version). The distance between two objects is the distance between the closest pair of points or segments of the two objects.



$$\phi(u,\upsilon)=\begin{cases}\perp, & if\ u=\upsilon\\ \arctan\left(\frac{\upsilon.y-u.y}{\upsilon.x-u.x}\right), & if\ \upsilon.x>u.x\ and\ \upsilon.y>u.y\\ 90, & if\ \upsilon.x=u.x\ and\ \upsilon.y=u.y\\ 180+\arctan\left(\frac{\upsilon.y-u.y}{\upsilon.x-u.x}\right), & if\ \upsilon.x<u.x\\ 270, & if\ \upsilon.x=u.x\ and\ \upsilon.y<u.y\\ 360+\arctan\left(\frac{\upsilon.y-u.y}{\upsilon.x-u.x}\right), & if\ \upsilon.x>u.x\ and\ \upsilon.y<u.y\end{cases}$$

Figure 18 Distance & Direction Operations

number {Moving_Object} ***f_direction*** *(Moving_Point, D_Timepoint_Sec):* The *f_direction* function is defined only for *Moving_Point* objects returning the angle of the line from the first to the second moving point (measured in degrees, $0 \le angle < 360$), after these have been projected to the Cartesian plane at a specific time point. The time-independent version of the function returns a *Moving_Object* modeling a *"moving real"*, which corresponds to the time-changing angle formed by the conceptual line segment that joins the two moving points and the *xx'* axis. Figure 18 illustrates the distance between a star (*Moving_Point*) and a galaxy (*Moving_Polygon*) projected at the spatial domain in a user-defined timepoint, as well as the angle formed by the moving star and the earth.

4.6 Set Relationships

HERMES-MDC provides four object methods for describing set-relationships between moving types. Each comes with two overloaded versions, one for describing a geometry object as the result of applying the set-relationship at a user-defined time point and one for describing a moving geometry that is defined as the set-relationship at all the time periods that this relationship is meaningful. For example the intersection of a *Moving_Point* with a *Moving_Polygon* results in a *Moving_Object* that represents another moving point, which is the restriction of the initial *Moving_Point* inside or on the boundary of the *Moving_Polygon*.

Subsequently, we present the supported set-relationships operations between any moving type and a *Moving_Polygon* object. Similar operations are defined for all the other moving types, as well as operations describing set-relationships of a moving type with a pure spatial object.

Sdo_Geometry {Moving_Object} ***f_intersection*** *(Moving_Polygon, tolerance, D_Timepoint_Sec):* The *f_intersection* object method returns either a geometry object that is the topological intersection (*AND* operation) of the two associated moving types projected at a user-defined time point or a *Moving_Object* whose mapping at each instant represents a geometry that is the outcome of this set operation. Invoking *f_intersection* method for the simplest moving object (*Moving_Point*), as one would expect, the result of this operation is the projection of itself on the spatial domain (point geometry) at time instants that intersects with other moving types or static geometries and *null* at time instants where it is not on the boundary or the interior of linestrings and polygons or it coincides with none of the points in a collection of them. Let us now present some motivating cases when invoking *f_intersection* method for moving linestring and polygon objects with other single or multi moving types that have more than one common points, segments or areas. Figure 19 below depicts the instantiation of a *Moving_Object* modeling the intersection of a *Moving_LineString* with a polygon, at three different timepoints t_1, t_2, and t_3. At timepoint t_1 it is obvious the result of such an operation, which is a linestring geometry. At timepoint t_2 this intersection has as result a multi-linestring geometry due to the development of *Moving_LineString*, while at timepoint t_3 the resulted geometry is a heterogeneous collection of lines and points.

Sdo_Geometry {Moving_Object} ***f_union*** *(Moving_Polygon, tolerance, D_Timepoint_Sec):* The *f_union* object method returns either a geometry object that is the topological union (*OR* operation) of the two associated moving types projected at a user-defined time point or a *Moving_Object* whose mapping at each instant represents a geometry that is the outcome of this set operation.

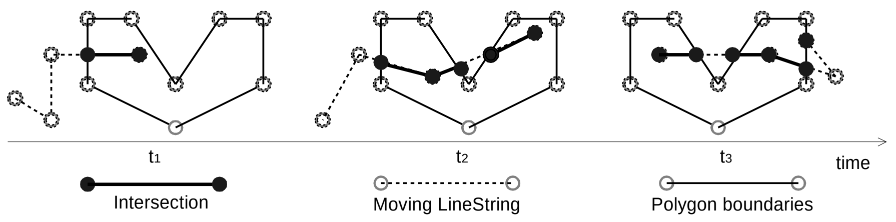


Figure 19 Demonstrating f_intersection Operation

One could extract a series of rules that stand for the outcome of the *f_union* object method, except the common one. More specifically, the union of a single moving geometry or a homogeneous moving collection with a disjoint moving (or static) geometry of the same type at a specific timepoint, results in a multi-geometry of that type. If the argument object is of different type from the caller and do not have common boundaries and/or interior areas, then the result in any case will be a heterogeneous collection. A noteworthy case is the union of a moving point or linestring with linestring or polygon geometries when at the time of the query their projection falls upon the linestring or the boundary of the polygon, respectively. In such case, the points of the moving point or linestring are interleaved as additional points in the sequence of points that defines the linestring or the boundary of the polygon.

Sdo_Geometry {Moving_Object} ***f_difference*** *(Moving_Polygon, tolerance, D_Timepoint_Sec):* The *f_difference* object method returns either a geometry object that is the topological difference (*MINUS* operation) of the two associated moving types projected at a user-defined time point or a *Moving_Object* whose mapping at each instant represents a geometry that is the outcome of this set operation. Generally speaking, the *f_difference* operation returns the part of the caller object that does not belong to the argument object. More specifically, applying this method to a moving geometry at a specific timepoint, the result is the projection of this moving type if the argument object is disjoint with this projection. In a different case where the argument object completely encloses the caller's projection the result is the *null* value. For example this happens when a user requires the difference of a moving point or linestring whose instantiation falls on the boundary or the interior of a polygon or upon the segment of a linestring. An interesting case happens when the *f_difference* operation is invoked between two moving polygons at an instant where the argument polygon has been moved wholly inside the caller moving polygon. The result in such case is a polygon with a hole.

Sdo_Geometry {Moving_Object} ***f_xor*** *(Moving_Polygon, tolerance, D_Timepoint_Sec):* The *f_xor* object method returns either a geometry object that is the topological symmetric difference (*XOR* operation) of the two associated moving types projected at a user-defined time point or a *Moving_Object* whose mapping at each instant represents a geometry that is the outcome of this set operation. The *f_xor* operation provides the union of the caller with the argument object, "subtracting" their intersection. As such, similarly to the *f_union* case, the *f_xor* for a moving polygon with another one that is totally inside the first returns also a polygon with a hole. If the first moving polygon does not *cover* completely the parameter moving polygon but just overlap, the result of the *f_xor* operation at a specific timepoint is a multi-polygon geometry. What is more, invoking this operation for a moving point with argument another moving point, the outcome at a specific instant is a multi-point if their projections are not the same and *null* if they are.

4.7 Rate of Change

An important property of any time-dependent value is its rate of change, i.e., its *derivative*. To determine which of our data types is applicable to this concept, consider the following definition of the derivative.

$$f'(t)=\frac{\partial f(t)}{\partial \mathrm{t}}=\lim_{\Delta t\longrightarrow 0}\frac{f(t+\Delta \mathrm{t})-f(t)}{\Delta t}$$

This definition, and thus the notion of derivation, is applicable to the moving types that firstly support a difference operation and secondly support division by a real number. *Moving_Point* type is the single type that clearly qualifies the above prerequisites. At least three operations assume the rule of difference in the definition, namely the Euclidian distance, the direction between two points and the vector difference (viewing points as two-dimensional vectors). This leads to three different derivative operations, called *speed*, *turn* and *velocity*, respectively.

number {Moving_Object} ***f_speed*** *(D_Timepoint_Sec)*: The *speed* operation comes in two overloaded signatures. The time-dependent version returns a number representing the *speed* of a moving point at a specific timepoint, while the time-independent version returns a *Moving_Object* modeling the time-varying *speed* at any time instant.

The algorithm that implements the *speed* method is based on its formal definition:

$$speed(t)=\sqrt{\upsilon_x^2(t)+\upsilon_y^2(t)}=\sqrt{\left(\frac{\partial s_x(t)}{\partial t}\right)^2+\left(\frac{\partial s_y(t)}{\partial t}\right)^2}$$

where υ_x, υ_y are the corresponding speeds of the moving point along *xx'* and *yy'* axes, which are expressed as the time derivatives of the distance functions, namely $\frac{\partial s_x(t)}{\partial t},\frac{\partial s_y(t)}{\partial t}$. These functions are not other than the two *Unit_Function* objects needed to define a *Unit_Moving_Point*.

number {Moving_Object} ***f_turn*** *(D_Timepoint_Sec)*: Similarly, *turn* operation is provided by the following two signatures, one representing the rate of change of the angle between the *xx'* axis and the motion vector at a specific timepoint and one expressing the same derivative value at any time instant.

The above-mentioned time-varying angle $\phi(t)$ can be computed as the tangent between s_x and s_y. Utilizing the derivative of the *arctan* function $\frac{\partial \arctan(\mathrm{x})}{\partial \mathrm{x}}=\frac{1}{1+x^2}$ and the definition of derivatives of composite functions $(g(f(x)))' = g'(f(x)) \cdot f'(x)$, the derivative of $\phi(t)$ can be computed as follows:

$$\tan(\phi(t)) = \frac{s_y(t)}{s_x(t)} \Leftrightarrow \arctan(\tan(\phi(t))) = \arctan\left(\frac{s_y(t)}{s_x(t)}\right) \Leftrightarrow \phi(t) = \arctan\left(\frac{s_y(t)}{s_x(t)}\right) \Leftrightarrow$$

$$\phi(t)' = \frac{1}{1+\left(\frac{s_y(t)}{s_x(t)}\right)^2} \cdot \left(\frac{s_y(t)}{s_x(t)}\right)' = \frac{1}{1+\left(\frac{s_y(t)}{s_x(t)}\right)^2} \cdot \frac{(s_y(t))' \cdot s_x(t) - s_y(t) \cdot (s_x(t))'}{(s_x(t))^2} = \frac{1}{1+\left(\frac{s_y(t)}{s_x(t)}\right)^2} \cdot \frac{\upsilon_y(t) \cdot s_x(t) - s_y(t) \cdot \upsilon_x(t)}{(s_x(t))^2}$$

Sdo_Geometry {Moving_Object} ***f_velocity*** *(D_Timepoint_Sec)*: Finally, the *velocity* of a moving point at a specific timepoint or at any instant during its development, is represented as a point geometry or a *Moving_Point* object, respectively.

Viewing a Moving_Point as a two-dimensional vector $\vec{s}(t) = (s_x(t), s_y(t))$, the derivative of this vector, which implements the *velocity* operation, is given by the following equation $(\vec{s}(t))' = ((s_x(t))', (s_y(t))')$.

Based on the *f_direction* method HERMES supports two sets of operations that provide predicate functionality on directional relationships between moving objects. The first set consists of four operations (namely, *f_west*, *f_east*, *f_north*, and *f_south*) each of which returns a Boolean value depending on whether the moving object is e.g. *west* from the a given moving or static geometry parameter, as well as a range of angles that puts some constraints in the directional relationship. Similarly, the second set consists of four operations (namely, *f_left*, *f_right*, *f_above*, and *f_behind*) that represent implicit directional relationships w.r.t. the motion of the query object.

4.8 Similarity functions

HERMES supports a set of query operators for similarity search between moving points as these have been introduced in 41, 39. Two main types of similarities are defined, namely, *spatiotemporal* and (temporally-relaxed) *spatial* similarity, followed by three variations, namely *speed-pattern based*, *acceleration-pattern based*, and *directional* similarity. More specifically:

number GenLIP(Moving_Point): The *Generalized Locality Inbetween Polylines* (GenLIP) distance between two moving points, returns an intuitive value that implies the area (see the shaded area in Figure 20) between the spatial projections of the two trajectories.

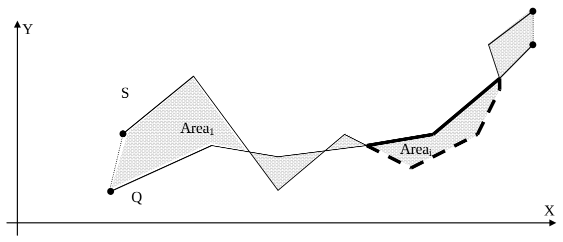


Figure 20: Locality In-between 2D Polylines

number GenSTLIP(Moving_Point): The *Generalized SpatioTemporal LIP* (GenSTLIP) function takes into account time, it operates on the original 3D representation of moving points and as such eliminates the time-relaxation of the GenLIP method by requiring co-location and co-existence during the lifetime of the moving points.

number GenSPSTLIP(Moving_Point):

number GenACSTLIP(Moving_Point): The *Generalized Speed-Pattern and Acceleration-Pattern STLIP* functions take also into account whether the two involved moving points move with similar speed or acceleration patterns.

number DDIST(Moving_Point):

number TDDIST(Moving_Point): The *Directional Distance* (DDIST) and *Temporal DDIST* (TDDIST) are two other variations that quantify the similarity of two moving objects according to their heading pattern. The first variation operates on the spatial projection of the objects, while the second checks whether the change in the heading happens in a synchronized way.

4.9 Index maintenance

Based on the extensible indexing capabilities provided by an ORDBMS each TB-tree owns the following functions:

IndexCreate: creates the index tables (i.e *tbtreeidx_leaf*, *tbtreeidx_non_leaf*) and populates the data already inserted in the table on which the index is created.

IndexInsert: performs insertions in the tree, triggered by the insertion of a new trajectory on the indexed table.

IndexUpdate: updates the tree every time a new trajectory segment (i.e unit_moving_point) is inserted.

IndexDrop: drops the tables that store the index data. This method is called when a DROP INDEX statement is issued

Functions *IndexInsert* and *IndexUpdate* call function *TBINSERT* which implements the TB-tree's insertion algorihm as described in 46.

4.10 Index operators

Range/timeslice queries, of the form "*find all objects located within a given area during a certain time interval or time instance*", (*Q2*/*Q*1 in Figure 21), is a straightforward extension of the respective 2D R-tree algorithm, in the 3D space formed by the two spatial and the one temporal dimension. This algorithm recursively visits tree nodes, rejecting node *MBB*s that does not overlap the query window, while following the pointers from overlapping *MBB*s to their respective child nodes until all candidate leaf nodes have been found. The algorithm starts by visiting the tree root, checking whether the *MBB*s of the root entries overlap the spatio-temporal query window *Q*. If a node entry overlaps *Q*, the algorithm follows the pointer to the corresponding child node, where it repeats recursively the same task. If the algorithm reaches a leaf node, leaf entries are examined against *Q* and if their *MBB* overlap, the algorithm reports their *ids.*

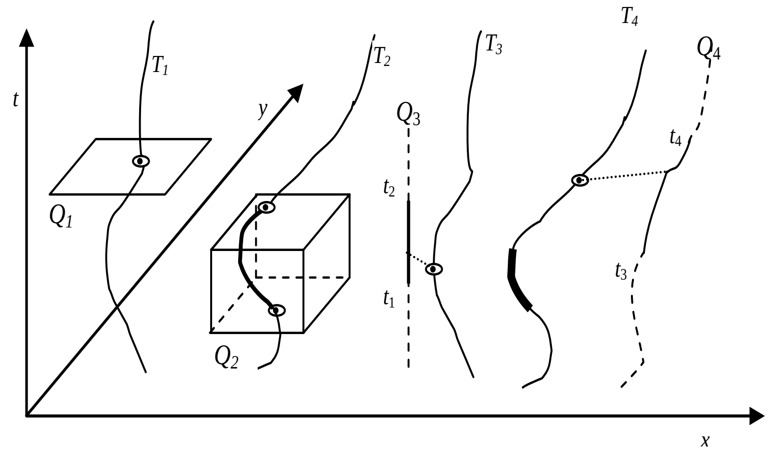


Figure 21 Querying trajectory databases

Regarding the *k* nearest neighbor (*k*-NN) search, 19 proposed a variety of solutions for answering such queries. More specifically, given as an example the trajectory database of Figure 21 given a stationary (or moving) query point *Q*3 (*Q*4) and a temporal query window [t_1, t_2] ([t_3, t_4]), 19 proposed several algorithms for finding the moving object trajectory T_3 (T_4) that is closer to the query object. Among them, the incremental varations of the algorithms proposed in 19 (IncPointNNSearch and IncTrajectoryNNSearch) are shown to be more scalable, thus, being good solutions to be implemented in the HERMES. Here, we have also to point that the aforementioned algorithms are capable to answer *k*-NN versions of the respective queries as well.

More specifically, the algorithms proposed in 19 are based on the respective NN algorithm for static objects presented in 26 which traverses the tree structure in a best-first way. The proposed algorithms use a priority queue, in which the (node or leaf) entries of the tree nodes are stored in increasing order of their distance from the query object. At each tree node the algorithm iterates through its entries checking whether the lifetime of an entry overlaps the time period of the query, calculating at the same time its distance from the query object, which is used to store them in the priority queue. At each algorithm's iteration the first entry is requested from the queue, until a leaf entry is found, which is then reported as the query result. The algorithms proposed in 19 are incremental in the sense that the *k*-th NN can be obtained with very little additional work once the (*k*-1)-th NN has been found; therefore, are easily generalized to the case where are the *k*>1 nearest neighbors of a query object (stationary of moving point) are requested.

Given the above discussion, HERMES supports the following set of operators, namely, *range* 46, *Point* and *Trajectory Nearest Neighbor* 19 and spatio-temporal *topological* 46 queries.

Function *tb_mp_in_spatiotemporal_window* executes a range query against a table storing indexed trajectories. It takes as arguments a standard spatial rectangular window, as well as a temporal period, and returns trajectory ids as well as trajectory fractions fully contained inside the given spatio-temporal window.

Function *IncPointNNSearch* executes a Point Nearest Neighbor query against a table storing indexed trajectories. It takes as arguments the coordinates of the query point (*X* and *Y* as reals), a temporal period, and the number of *k* closest nearest neighbors to be returned. It returns trajectory ids, as well as the spatiotemporal trajectory line segments that are closest to the query point at the given time period.

Function *IncTrajectoryNNsearch* executes a trajectory Nearest Neighbor query against a table storing indexed trajectories. It takes as arguments the identifier of the trajectory to be used as query, and the number of *k* closest nearest neighbors to be returned. It returns trajectory ids, as well as the spatiotemporal trajectory line segments that are closest to the query trajectory during its life time.

Function *TopologicalQuery* is used to retrieve the trajectories that enter and/or leave a spatio-temporal query window. The query parameters involve the lower left (X_1,Y_1) and upper right (X_2,Y_2) of a (rectangular) area as well as a time period for the temporal part of the query. Finally, a *MASK* must be defined to clarify the type of topological query. Possible *MASK* (string) values can be '*ENTER*', '*LEAVE*', '*ENTER_LEAVE*' depending whether the users are interested in trajectories that enter/leave or enter&leave the area within the given time period.

5 Architectural Aspects of HERMES-MDC and an Application Example

HERMES can be utilized in a real world scenario to assist a database developer in modeling, querying and analyzing moving object databases. A straightforward utilization scenario is to design and construct a spatio-temporal object-relational database schema using HERMES and build an application by transacting with this database. Figure 22 illustrates such a scenario on the top of *Oracle* ORDBMS. In this case and in order to specify the database schema, the database designer writes scripts in the syntax of the *Data Definition Language* (DDL), which in this case is the PL/SQL, extended with the spatio-temporal operations previously introduced.

To build an application on top of such a database for creating objects, querying data and manipulating information; the application developer writes a source program (for instance) in *Java* (or *JSP* in case of web-based applications) wherein he/she can embed *PL/SQL scripts* that invoke object constructors and methods from HERMES. The *JDBC pre-processor* integrates the power of the programming language with the database functionality offered by the extended PL/SQL and together with the *ORDBMS Runtime Library* generate the application's executable. By writing independent stored procedures that take advantage of HERMES functionality and by compiling them with the *PL/SQL Compiler*, is another way to build a spatio-temporal application. Figure 23 depicts such an application which also acts as a web-based visual query builder for HERMES.

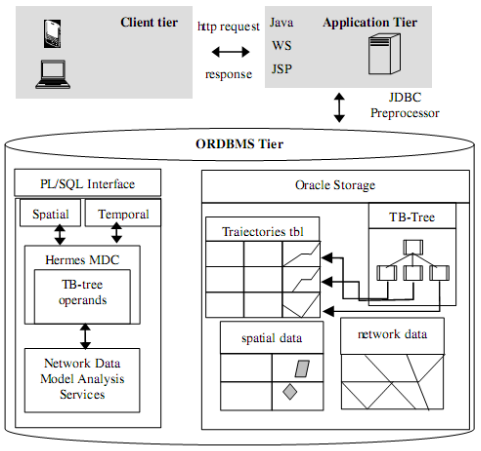
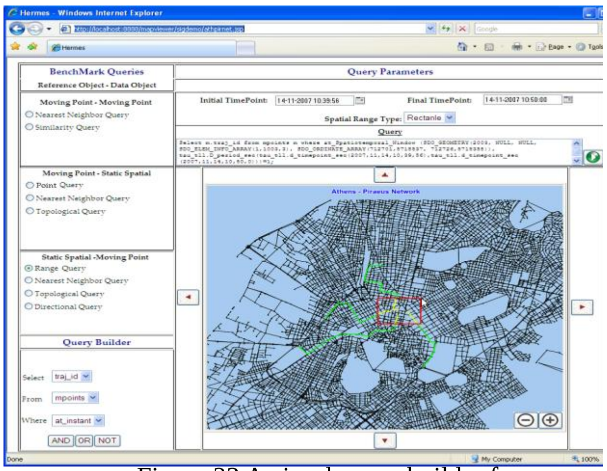


Figure 22 The architecture of the HERMES



Initial TimePoint: 14-11-2007 10:39:56 Final TimePoint: 14-11-2007 10:50:00

Spatial Range Type: Rectanle

Query

```
Select m.traj_id from mpoints m where at_Spatiotemporal_Window (SDO_GEOMETRY(2003, NULL, NULL,
SDO_ELEM_INFO_ARRAY(1,1003,3), SDO_ORDINATE_ARRAY(712701,4718887, 712726,4718886)),
tau_tll.D_period_sec(tau_tll.d_timepoint_sec(2007,11,14,10,39,56),tau_tll.d_timepoint_sec
(2007,11,14,10,50,0)))=1;
```

Select traj_id
From mpoints
Where at_instant

Figure 23 A visual query builder for HERMES

To demonstrate the functionality of the proposed HERMES, in the following paragraphs we present an application example related to vehicle traffic analysis. The motivation is that a courier company, whose vehicles are enhanced with GPS devices transmiting their space-time location to a central MOD, needs a flexible way to manage and analyse the motion of the vehicles. One can enumerate a series of benefits to be gained from a possible optimization of the movements of the couriers, such as, personnel's control, better and faster customer services, minimization of operational costs, enhanced decision making. By utilizing this application example, the expressive power and the applicability of HERMES in such a commercial domain are demonstrated. We note that the subsequent discussion and terminology follows the syntax of HERMES as implemented in Oracle ORDBMS. We have already mentioned that the core of HERMES has also been implemented in 7 inside another ORDBMS, namely the PostGIS. This actually proves the correctness of the design of HERMES on top of extensible ORDBMS that have OGC-compliant spatial extensions. The differences in the syntax between the two implementations are minor (and mainly due to the syntax differences of the two static spatial extensions) 63, while we are in the process of testing the compatibility between the results of the operations.

In order to present the capabilities of HERMES, we build the following database:

Highways (name: Varchar2, line: SDO_GEOMETRY)
Landmarks (name: Varchar2, kind: Varchar2, location: SDO_GEOMETRY)
Vehicles (id: Varchar2, type: Varchar2, route: Moving_Point)
High-Traffic-Areas (name: Varchar2, extent: Moving_Polygon)

Highways relation is a set of linestring geometries along which the vehicles are supposed to be moving. Landmarks relation contains locations of certain landmarks, such as petrol stations, etc. Vehicles relation identify the route of a lorry that is modeled as a moving point, while type attribute stamps each vehicle with a characteristic description of each kind (e.g. truck, motorbike, etc.). Furthermore, field route of relation Vehicles is indexed by a TB-tree. High-Traffic-Areas relation records areas that could influence the route or the schedule of a vehicle. These areas are given a name for identification purposes, while extent attribute provides the time-varying regions for traffic jams.

In the following paragraphs, we illustrate a composite spatio-temporal scenario (in the form of a series of MOD queries) in the domain of our application example. The linguistic description of each query is followed by the implementation of the query in the form of a *PL/SQL* block, as well as by an abstract presentation of the way that such a query is resolved. This scenario illustrates the expressive power and the spatio-temporal query capabilities added to *PL/SQL* by HERMES.

(Q1) Which vehicles are moving inside a given region right now?

PL/SQL block for Q1:

```
DECLARE
   region SDO_GEOMETRY := SDO_GEOMETRY(2003, NULL, NULL,
   SDO_ELEM_INFO_ARRAY(1,1003,3), SDO_ORDINATE_ARRAY(489048,4203749,
   90032,4205990));
BEGIN
   SELECT id FROM Vehicles
   WHERE route.f_relate('INSIDE', region, 0.005, TAU_TLL.now()) = 'INSIDE';
END;
```

In order to answer Q1 we invoke a typical SQL statement that selects from the Vehicles relation the ids of the couriers that satisfy the WHERE-clause, which is the time-dependent version of f_relate operation. A slight variant of Q1 is the classic spatio-temporal range query (see Q2) that may also be answered with the employment of the TB-tree operators, by simply invoking function *tb_mp_in_spatiotemporal_window*. Actually, this is the query depicted in the query builder in Figure 23.

(Q2) Find all vehicles moving inside a given region and time period?

```
PL/SQL block for Q2:
DECLARE
region SDO_GEOMETRY := SDO_GEOMETRY(2003, NULL, NULL,
SDO_ELEM_INFO_ARRAY(1,1003,3), SDO_ORDINATE_ARRAY(489048,4203749,
                                                        90032,4205990));
BEGIN
SELECT TB_MP_IN_SPATIOTEMPORAL_WINDOW (region,
                tau_tll.d_period_sec(
                        tau_tll.D_Timepoint_Sec(2010,7,9,10,35,0),
                        tau_tll.D_Timepoint_Sec(2010,7,9,10,55,0)))
FROM Vehicles;
END;
```

(Q3) If vehicle 'X' is in the result set of Q1, when and where did it enter the region?

```
PL/SQL block for Q3:
DECLARE
truckX Moving_Point;
truckX_IN_region Moving_Object;
temp_projection TAU_TLL.TEMP_ELEMENT_SEC;
when TAU_TLL.TIMEPOINT_SEC;
where SDO_GEOMETRY;
BEGIN
SELECT route INTO truckX FROM Vehicles WHERE id='X';
truckX_IN_region := truckX.f_intersection(region);
temp_projection := truckX_IN_region.f_temp_element();
when := temp_projection.te(temp_projection.te.FIRST).b;
where := truckX_IN_region.f_initial();
END;
```

To address Q3, we demonstrate how we can restrict a moving point inside a static spatial region and how to temporally and spatially project this restricted moving point in its initial position. The result of such an operation (f_intersection) in all cases is a Moving_Object that can be handled as any other moving geometry. By temporally projecting it (f_temp_element) on the continuous time line and finding the temporal element that consists of the time periods for which are defined the unit moving objects of the moving courier, we can estimate the timepoint when initially entered the given region. In addition, by applying the f_initial method, we can locate the point that this happened.

(Q4) A variant of Q3 would be to find all entering points of trajectories in the given spatio-temporal range.

PL/SQL block for Q4:

```
SELECT * FROM TABLE(TB_TOPOLOGICAL_QUERY(
      SDO_GEOMETRY(2003, NULL, NULL,
SDO_ELEM_INFO_ARRAY(1,1003,3),
      SDO_ORDINATE_ARRAY(489048,4203749, 90032,4205990)),
              tau_tll.d_period_sec(
                        tau_tll.D_Timepoint_Sec(2010,7,9,10,35,0),
                        tau_tll.D_Timepoint_Sec(2010,7,9,10,55,0)),
'ENTER') )
```

(Q5) What distance has vehicle 'X' travelled inside the region?

PL/SQL block for Q5:

```
DECLARE
distance double;
BEGIN
distance:= LENGTH (INTERSECTION (region, truckX.f_trajectory()));
END;
```

This query is resolved by finding the intersection of the region with the trajectory followed by the courier (f_trajectory operation). This intersection is a *LineString* geometry that restricts the route of the courier *inside* the region and by applying the LENGTH spatial operator upon the resulted *LineString* we compute the required distance.

(Q6) Give a list of options to the driver of vehicle 'X' to refuel the vehicle within the next 2km

PL/SQL block for Q6:

```
BEGIN
SELECT name, location FROM Landmarks
WHERE kind = 'petrol station' AND
truckX.f_within_distance(2000, location, 0.005, TAU_TLL.now()) = 'TRUE';
END;
```

In order to provide the list of petrol stations (Q6), we select the landmarks that are petrol stations and the courier is within the specified distance (f_within_distance operation) from them at the time the query is invoked.

(Q7) Which is the best route, in terms of distance, that this courier should follow in order to avoid traffic jam ‘A’?

PL/SQL block for Q7:

```
DECLARE
  jamA Moving_Polygon;
  jamA_region SDO_GEOMETRY;
  CURSOR highways IS SELECT * FROM Highways;
  highway_length, min_length number := 0;
  best_highway SDO_GEOMETRY;
  BEGIN
  SELECT extent INTO jamA FROM High-Traffic-Areas WHERE name=’A’;
  jamA_region := jamA.f_traversed();
  FOR highways_rec IN highways LOOP
      IF RELATE (highways_rec.line, ‘DISJOINT’, jamA_region) = ‘DISJOINT’
      THEN
    highway_length := LENGTH(highways_rec.line);
    IF highway_length < min_length THEN
     min_ length:= highway_length;
     best_highway := highways_rec.line;
    END IF;
   END IF;
  END LOOP;
  END;
```

To simplify the presentation of Q7 let us assume that the courier resides at the beginning of a series of highways and that its destination is the ending point of these highways. As such, having a cursor to traverse (FOR LOOP) all highways, we choose that highway that is *disjoint* (RELATE operator) with the region traversed (f_traversed operation) by jam ‘A’ and it has the smallest length (LENGTH operator).

Based on related research work 44 queries like the above constitute a minimum functionality a MOD system should provide. Furthermore, the usefulness and applicability of the server-side extensions provided by HERMES have been proved in 45 and 40 by developing benchmark queries proposed in 54 for the evaluation of systems supporing Location-Based Services.

6 Building Real MOD Applications on Top of Hermes

The best way to evaluate HERMES is to assess the realization of its initial goal, which is to provide a complete framework for developing MOD-related applications. In the previous section we provided a sketch for building a specific application related to vehicle traffic analysis, while in this section we demonstrate this by briefly presenting successfull applications of HERMES in four different domains, namely in *trajectory data warehouses* (i.e. TDW 31), in *moving object data mining query languages* (i.e. DAEDALUS tool 36), in *semantic enrichment of movement patterns* (i.e. ATHENA tool 5), and in *privacy-aware trajectory tracking query engines* (i.e. HERMES++ tool 22). We would like to note that the above works are a subset of tools and methods developed as a result of a European-wide research project called GeoPKDD – (Geographic Privacy-Aware Knowledge Discovery and Delivery) 21. HERMES is also a prototype outcome of GeoPKDD designed to be the MOD management infrastructure of such tools. Of cource, in order to support such diverse applications domains we have designed and incorporated into HERMES several specialized operations (e.g. a trajectory anomymizer operator for 22), however their description is ommited here due to to space constraints.

Trajectory data warehouses – TDW aim at developing a multi-dimensional model suitable for online analytical processing (OLAP) of trajectory data, such as drill-down and roll-up operations. In order to design a trajectory warehouse architecture, one should first identify the differences from conventional warehouse approaches and then to devise appropriate extensions. There are three steps so as to realize the development of a TDW. At the first step the design of a MOD and of a multidimensional data model (i.e. trajectory data cube) takes place. At the second step, preprocessing (i.e. cleaning, consistency checking) and loading of raw movement data into the MOD occurs, while once trajectories have been stored in the MOD, the Extract-Transform-Load (ETL) phase is executed in order to feed the TDW and the measures of the data cube are calculated. In 31, 30, 48 HERMES has been employed as the infrastructure to develop the above described process in a huge, real trajectory dataset, where due to the size of the dataset, the existence of efficient, scalable querying processing operators to support ETL was a key requirement.

Moving object data mining query languages (MO-DMQL) – In 36 the authors proposed DAEDALUS, a formal framework and system, that defines knowledge discovery processes as a progressive combination of mining and querying operators. The heart of DAEDALUS is the MO-DMQL query language that extends SQL in two aspects, namely a pattern definition operator and functionality to uniform manipulate both raw trajectory data and unveiled movement patterns. DAEDALUS system has been implemented as a query execution layer on top of the HERMES. More specifically, the role of HERMES in DAEDALUS is two-fold; to act as a repository for movement data and secondly to give the basic building block that allows defining models' representation and storage.

Semantic enrichment of movement patterns – Having as aim to provide a model for the conceptual representation and deductive reasoning of trajectory patterns obtained from mining raw trajectories, the authors in 5 have developed ATHENA tool, which employs ontologies for the semantic enrichment of trajectories. This is achieved by means of a semantic enrichment process, where raw trajectories are enhanced with semantic information and integrated with geographical knowledge encoded in an ontology. To highlight this process imagine that a user poses a query using the ontology concepts where trajectories/patterns are classified by a reasoner. The ontology is then populated by instances coming from a MOD storing semantic trajectories, patterns and auxiliary geographical features. Again, HERMES supports all the spatio-temporal data management requirements raised by ATHENA. The overall undertaking was evaluated in a real-world case study posing as objective of the analysis to understand tourist movements in Milan's metropolitan area.

Privacy-aware trajectory tracking query engines – Due to the very nature of movement data, lately a new line of research has emerged that investigates safeguards to enforce so as to ensure the privacy of the individuals, whose movement is recorded. HERMES++ 22 which has been designed on top of HERMES describes such a privacy aware trajectory tracking query engine, where subscribed users can gain restricted access to an in-house trajectory data warehouse, to perform certain analysis tasks. In addition to regular queries involving non-spatial non-temporal attributes, the engine supports a variety of spatiotemporal queries, including range queries, nearest neighbor queries and queries for aggregate statistics. The query results are augmented with fake trajectory data (dummies) to fulfil the requirements of K-anonymity.

7 A Real Case Study

This Section includes the description of a real world application scenario and at the same time presents usage instructions involving the desktop module of Hermes web application as demonstrated in 40. We will analyze into detail the elements of the query language used in the presented operations and corresponding parameters so as to facilitate interested users.

As regards the web interface, for best presentation results, we recommend that you use IE6 (or higher) or Safari web browsers though also Mozilla Firefox has been tested and works properly.

7.1 Usage Scenario

To better perceive the functionality of the underlying Trajectory Database Engine we exhibit the usage of implemented operators utilized in query operations assuming a specific application scenario. The scenario described below constitutes a representative example of HERMES potentials and its ability to provide real-world LBS support. However, note that the set of supported services are not restricted by application specific factors but can serve as the infrastructure for every modern application that demands advanced trajectory data management and querying (i.e fleet management, asset tracking, mobile advertising etc).

In the demonstration scenario, we assume a fleet of taxis that move in the metropolitan area of Athens, Greece. Taxis are moving on the underlying road network and periodically request routing to certain destinations. A routing module indicates the shortest path as the preferable path to reach the aforementioned destination. Afterwards, for the purposes of the demonstration, each taxi driver is supposed to accept and follow the proposed path. In that way, moving object trajectories are expected to be known in hand.

7.2 Database Design

The underlying data infrastructure consists of the following types of data:

Spatial entities:
- Athens Road Network Data (Nodes, Links)
- Landmarks (ID, geometry, address, area, type)
- Regions (ID, name, geometry)

Note that Landmarks are possible POIs that a taxi driver may wish to be aware of their existence and their proximity to his way towards a destination. Regions involve a set of municipalities that cover the underlying road network. We are going to further discuss the role of landmarks and regions in query operations in the next subsection.

“Moving” entities:
- Vehicles (obj_id, traj_id, trajectory)

7.3 Query Operations

Query operations are categorized based on the type of the reference object and the type of the data objects. In brief, a reference object involves the type of the object (trajectory or spatial entity) based on which query answers are retrieved while the data object regards the type of objects (trajectories or spatial entities) that participate in the posed query answer.

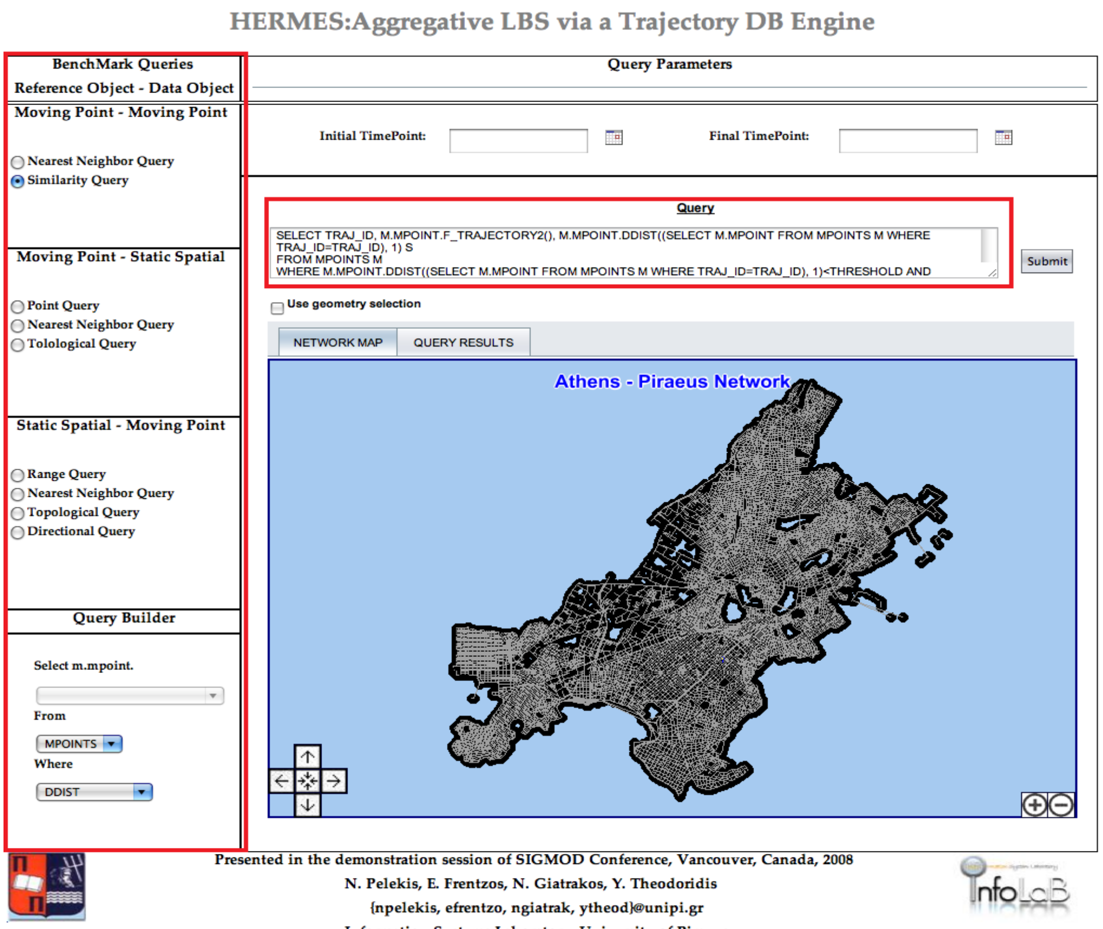
Based on the above the queries are categorized as:

Moving Point – Moving Point: both the reference and data objects are trajectories

Moving Point – Static Spatial: the reference object is a given trajectory while the query answer is expected to be a set of spatial entities

Static Spatial – Moving Point: the reference object is a given spatial entity while the query answer is expected to be a set of trajectories or trajectory parts

The user is expected to choose the desired query from the categories at the left of the screen. Upon a specific query election the query is formed in the corresponding textbox at the center of the screen where the user is supposed to provide required parameters.



Moving Point - Moving Point: Nearest Neighbor Query [✓ Similarity Query]

```
SELECT TRAJ_ID, M.MPOINT.F_TRAJECTORY2(), M.MPOINT.DDIST((SELECT M.MPOINT FROM MPOINTS M WHERE
TRAJ_ID=TRAJ_ID), 1) S
FROM MPOINTS M
WHERE M.MPOINT.DDIST((SELECT M.MPOINT FROM MPOINTS M WHERE TRAJ_ID=TRAJ_ID), 1)<THRESHOLD AND
```

Presented in the demonstration session of SIGMOD Conference, Vancouver, Canada, 2008
N. Pelekis, E. Frentzos, N. Giatrakos, Y. Theodoridis
{npelekis, efrentzo, ngiatrak, ytheod}@unipi.gr
Information Systems Laboratory, University of Piraeus

Figure 24: Query Selection and Formation Areas

a. Moving Point – Moving Point

In this category we have two types of operations, namely Nearest Neighbor and Similarity queries.

i. Nearest Neighbor Query

Usage: Given a trajectory T find the k nearest parts (during T's lifetime) of other trajectories

Query:

```
SELECT TBFUNCTIONS.MV_INCTRAJECTORYNNSEARCH(TRAJ_ID ,K)
FROM DUAL
```

Parameters: The query parameters (in bold) involve the ID of the object's trajectory (reference object) and the desired number of nearest neighbors k (data objects).

Example:

```
SELECT TBFUNCTIONS.MV_INCTRAJECTORYNNSEARCH(1 ,10)
FROM DUAL
```

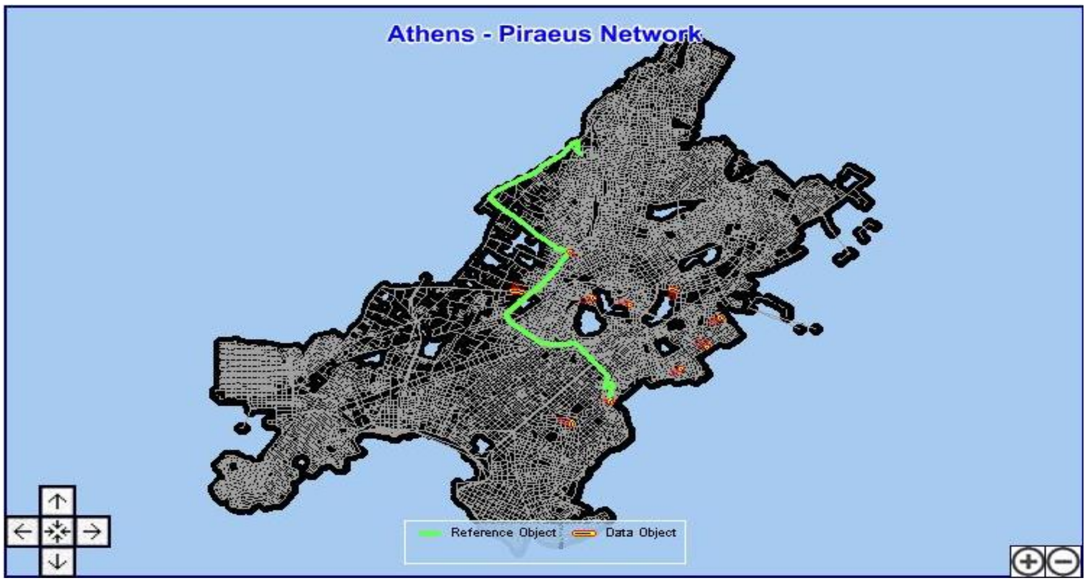
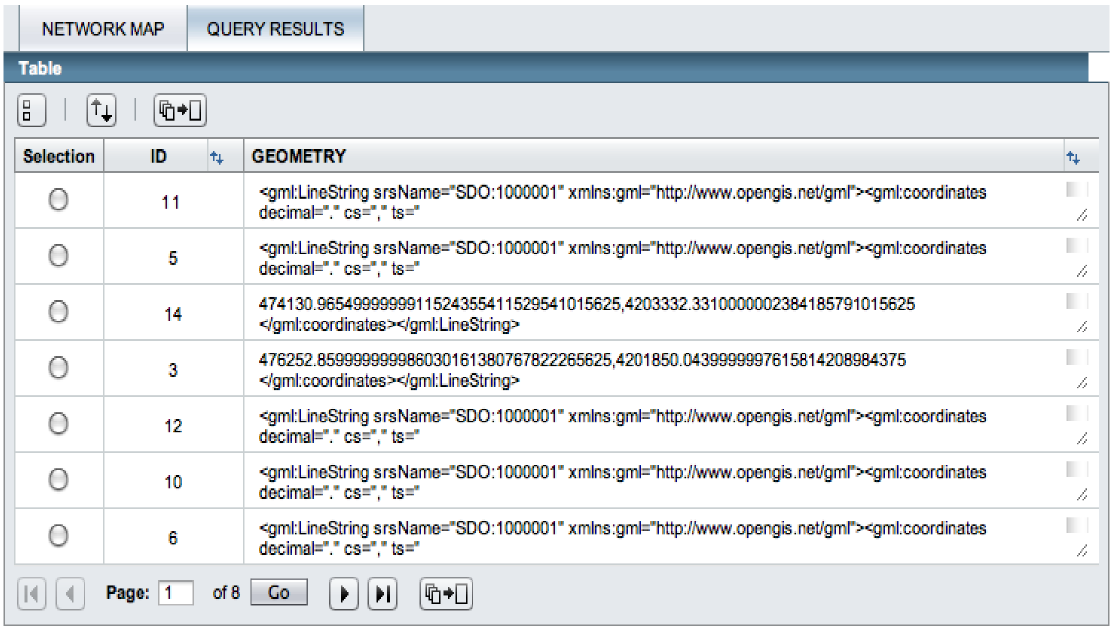


Figure 25: Answer of an incremental trajectory NN query visualized on the map



Selection	ID	GEOMETRY
○	11	<gml:LineString srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:coordinates decimal="." cs="," ts="
○	5	<gml:LineString srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:coordinates decimal="." cs="," ts="
○	14	474130.9654999999911524355411529541015625,4203332.3310000002384185791015625 </gml:coordinates></gml:LineString>
○	3	476252.859999999986030161380767822265625,4201850.0439999997615814208984375 </gml:coordinates></gml:LineString>
○	12	<gml:LineString srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:coordinates decimal="." cs="," ts="
○	10	<gml:LineString srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:coordinates decimal="." cs="," ts="
○	6	<gml:LineString srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:coordinates decimal="." cs="," ts="

Page: 1 of 8 Go

Figure 26: Answer of an incremental trajectory NN query in gml format

ii. Similarity Query

Usage: Given a trajectory T find at most N similar trajectories that satisfy a given similarity threshold. In this type of query we distinguish the following types of similarity:

- **DDIST:** The operator finds trajectories that are considered similar based on the resemblance they exhibit in their direction during their lifetime. This function ignores the temporal information part of the trajectories.

Query:

```
SELECT TRAJ_ID, M.MPOINT.F_TRAJECTORY2(),
      M.MPOINT.DDIST(
            (SELECT M.MPOINT
             FROM MPOINTS M
             WHERE TRAJ_ID=TRAJ_ID), 1) S
FROM MPOINTS M
WHERE M.MPOINT.DDIST(
      (SELECT M.MPOINT
       FROM MPOINTS M WHERE TRAJ_ID=TRAJ_ID), 1)<THRESHOLD
       AND ROWNUM<N
       ORDER BY S ASC
```

Parameters: The query parameters (in bold) involve the ID of the object's trajectory (reference object), a similarity threshold which is a real value between 0-1 and the expected, maximum number of similar trajectories – data objects. Note that the trajectory Id of the reference object should be declared twice. The first declaration involves the similarity value and its projection to the query results while the second regards the selection part of the query.

Example:

```
SELECT  TRAJ_ID,  M.MPOINT.F_TRAJECTORY2(),  M.MPOINT.DDIST((SELECT  M.MPOINT
FROM MPOINTS M WHERE TRAJ_ID=1), 1) S
FROM MPOINTS M
WHERE  M.MPOINT.DDIST((SELECT  M.MPOINT  FROM  MPOINTS  M  WHERE  TRAJ_ID=1),
1)<0.5 AND ROWNUM<4
ORDER BY S ASC
```

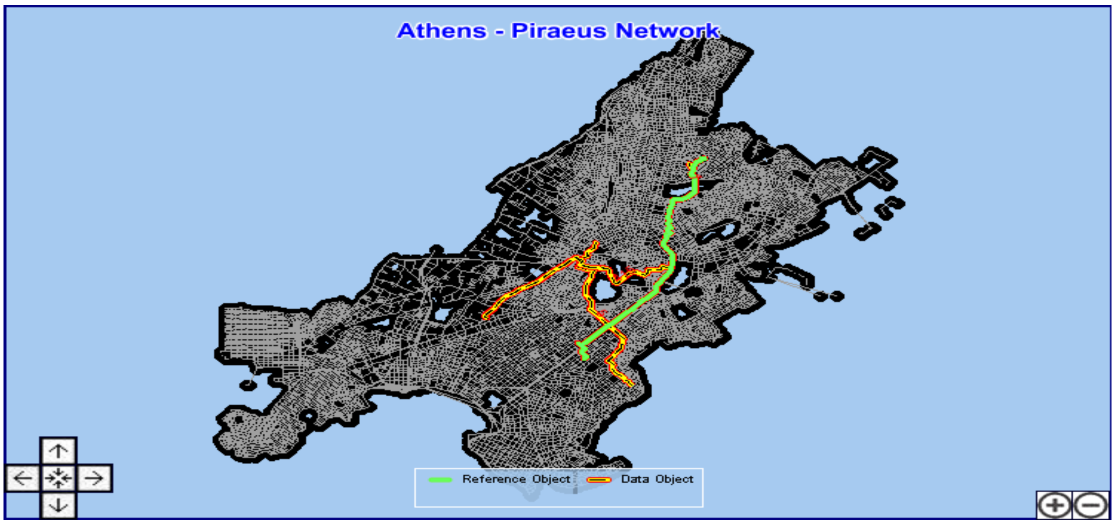
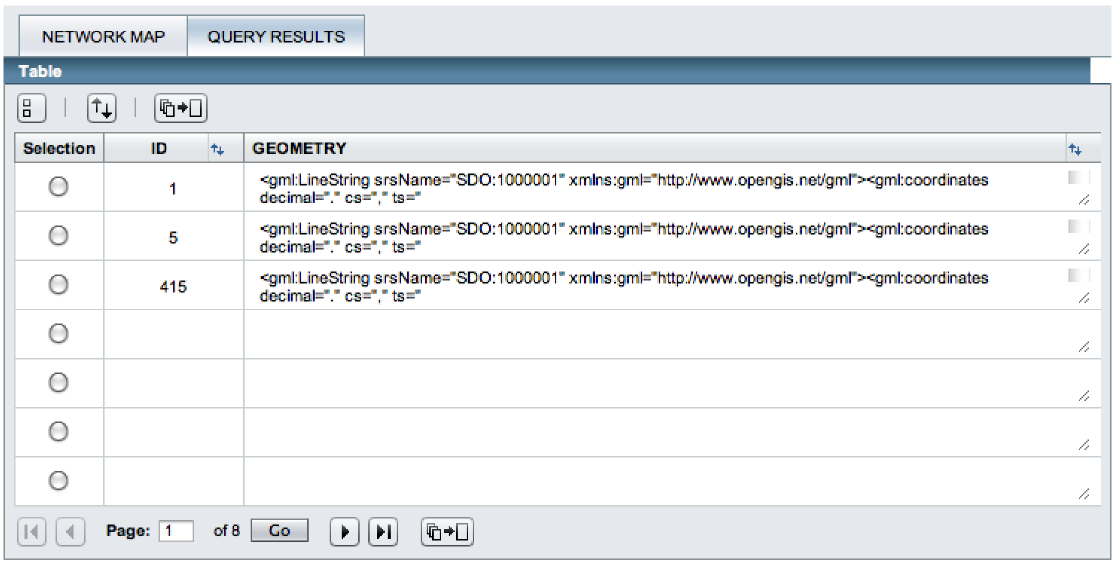


Figure 27: Answer of DDIST similarity query visualized on the map



Selection	ID	GEOMETRY
○	1	<gml:LineString srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:coordinates decimal="." cs="," ts="
○	5	<gml:LineString srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:coordinates decimal="." cs="," ts="
○	415	<gml:LineString srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:coordinates decimal="." cs="," ts="
○		
○		
○		
○		

Figure 28: Answer of DDIST similarity query in gml format

- **TDDIST**: A time aware version of DDIST similarity function, in that it takes into account (apart from their heading) temporal information of trajectories.

Query:

```
SELECT TRAJ_ID, M.MPOINT.F_TRAJECTORY2(),
      M.MPOINT. TDDIST (
             (SELECT M.MPOINT
              FROM MPOINTS M
              WHERE TRAJ_ID=TRAJ_ID), 1) S
FROM MPOINTS M
WHERE M.MPOINT. TDDIST(
      (SELECT M.MPOINT
       FROM MPOINTS M WHERE TRAJ_ID=TRAJ_ID), 1)<THRESHOLD
       AND ROWNUM<N
       ORDER BY S ASC
```

Parameters: Parameters are equivalent with those in DDIST

Example:

```
SELECT  TRAJ_ID,  M.MPOINT.F_TRAJECTORY2(),  M.MPOINT.TDDIST((SELECT  M.MPOINT
FROM MPOINTS M WHERE TRAJ_ID=1), 1) S
FROM MPOINTS M
WHERE  M.MPOINT.  TDDIST((SELECT  M.MPOINT  FROM  MPOINTS  M  WHERE  TRAJ_ID=1),
1)<0.5 AND ROWNUM<4
ORDER BY S ASC
```

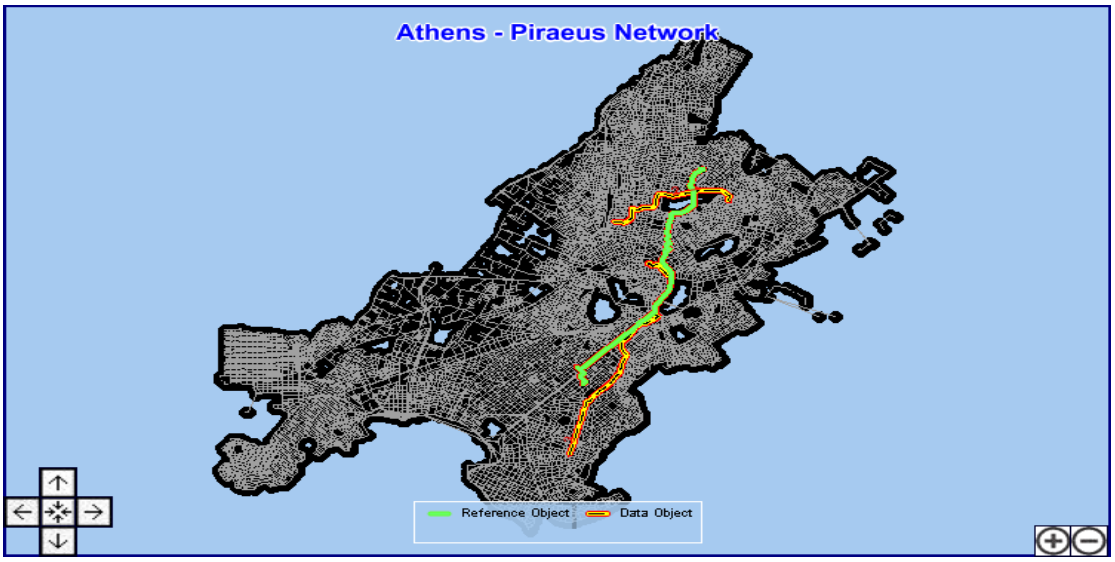
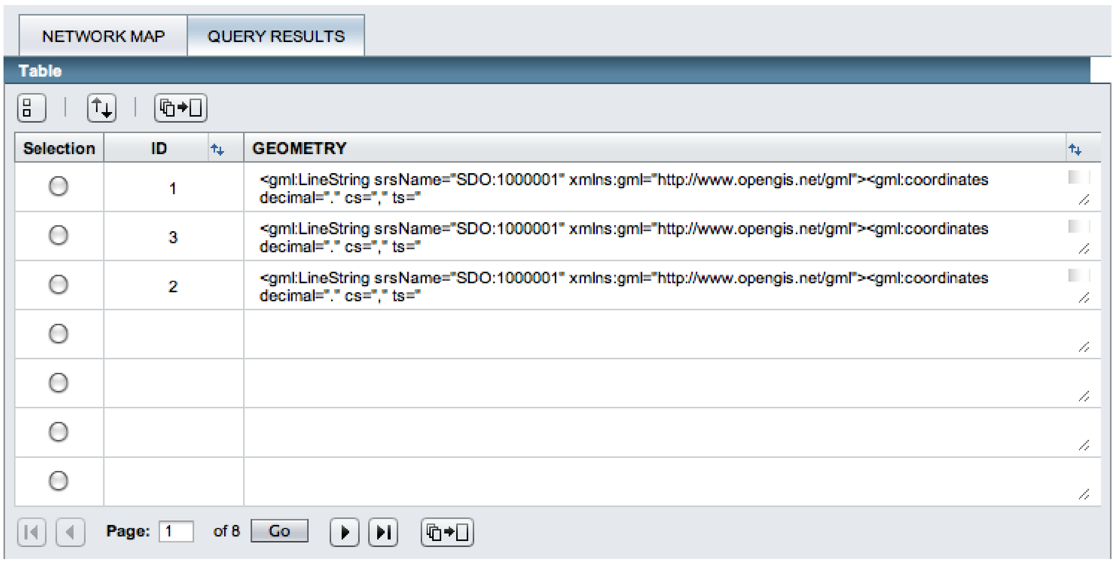


Figure 29: Answer of TDDIST similarity query visualized on the map



Selection	ID	GEOMETRY
○	1	<gml:LineString srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:coordinates decimal="." cs="," ts="
○	3	<gml:LineString srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:coordinates decimal="." cs="," ts="
○	2	<gml:LineString srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:coordinates decimal="." cs="," ts="
○		
○		
○		
○		

Page: 1 of 8 Go

Figure 30: Answer of TDDIST similarity query in gml format

- **LIP**: the operator involves spatial similarity based on a distance function upon the projections of trajectories in the Cartesian plain. The idea is to calculate the area of the shape formed by two 2D polylines, which are the outcome of the projection.

Query:

```
SELECT TRAJ_ID, M.MPOINT.F_TRAJECTORY2(),
      M.MPOINT. LIP (
             (SELECT M.MPOINT
              FROM MPOINTS M
              WHERE TRAJ_ID=TRAJ_ID), 1
             ) S
FROM MPOINTS M
WHERE M.MPOINT. LIP(
       (SELECT M.MPOINT
        FROM MPOINTS M WHERE TRAJ_ID=TRAJ_ID), 1)<THRESHOLD
        AND ROWNUM<N
        ORDER BY S ASC
```

Parameters: The **TRAJ_ID** and **N** parameters are equivalent with those in DDIST, TDDIST. The **THRESHOLD** parameter differs in that it refers to the area which is the outcome of the distance function. As such it receives real values regarding that mensuration.

Example:

```
SELECT TRAJ_ID, M.MPOINT.F_TRAJECTORY2(), M.MPOINT.LIP((SELECT M.MPOINT FROM
MPOINTS M WHERE TRAJ_ID=1), 1) S
FROM MPOINTS M
WHERE M.MPOINT. LIP((SELECT M.MPOINT FROM MPOINTS M WHERE TRAJ_ID=1), 1)<
10000000 AND ROWNUM<4
ORDER BY S ASC
```

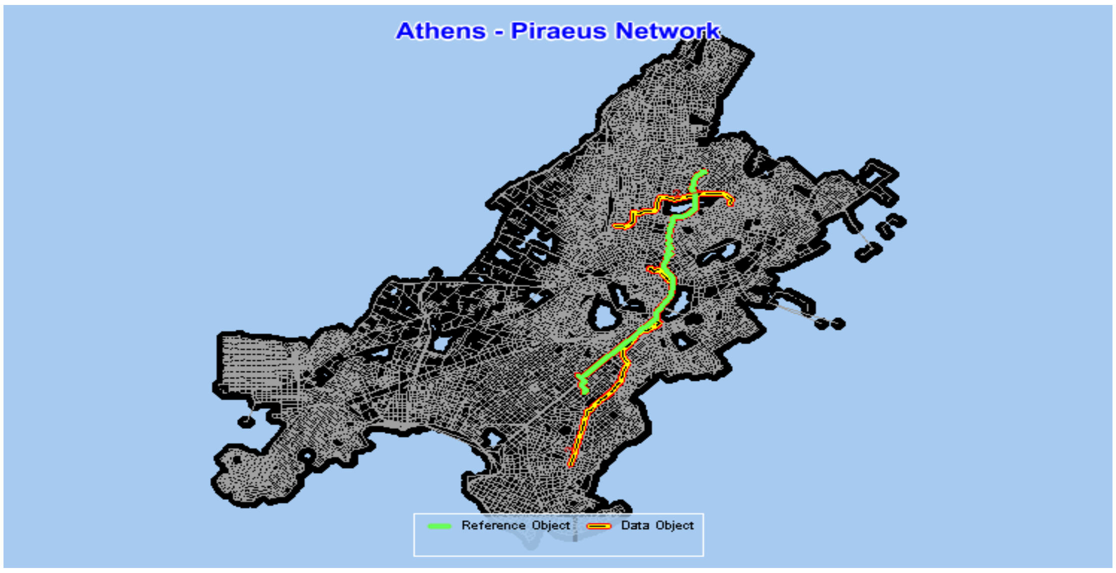
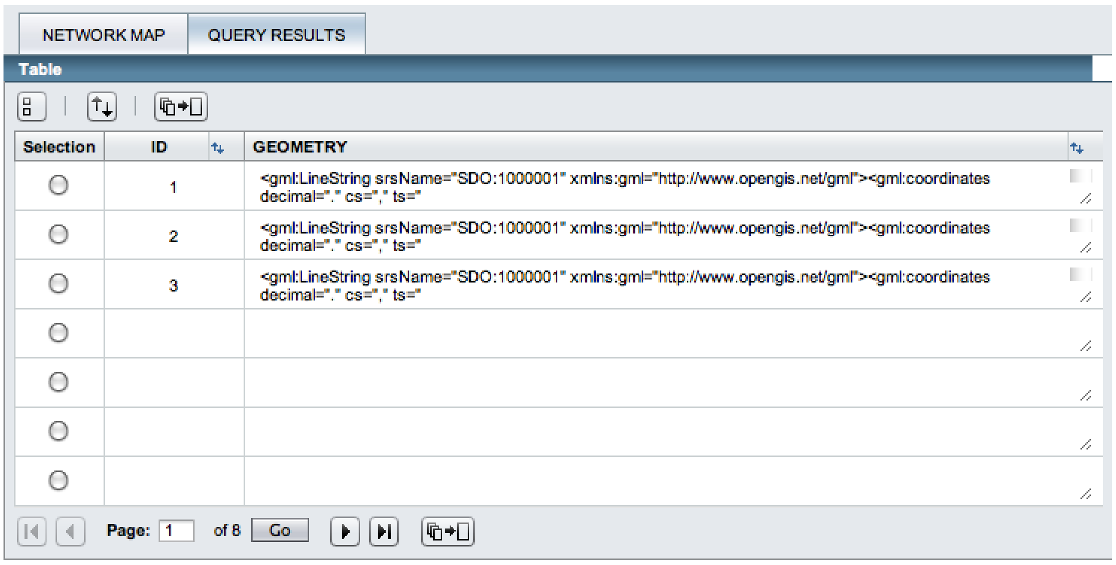


Figure 31: Answer of LIP similarity query visualized on the map



Selection	ID	GEOMETRY
○	1	<gml:LineString srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:coordinates decimal="." cs="," ts="
○	2	<gml:LineString srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:coordinates decimal="." cs="," ts="
○	3	<gml:LineString srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:coordinates decimal="." cs="," ts="
○		
○		
○		
○		

Figure 32: Answer of LIP similarity query in gml format

- **GenSTLIP_OSP**: An operator that measures spatiotemporal similarity between trajectories. Intuitively, two moving objects are considered similar in both space and time when they move close at the same time.

Query:

```
SELECT TRAJ_ID, M.MPOINT.F_TRAJECTORY2(),
      M.MPOINT.GENSTLIP_OSP(
      (SELECT M.MPOINT
       FROM MPOINTS M
       WHERE TRAJ_ID=TRAJ_ID),
      1,0,
      MDSYS.SDO_GEOM.SDO_LENGTH(M.MPOINT.F_TRAJECTORY2(),0.00005),
      (SELECT MDSYS.SDO_GEOM.SDO_LENGTH(M.MPOINT.F_TRAJECTORY2(),
                                                             0.00005)
      FROM MPOINTS M
      WHERE TRAJ_ID=TRAJ_ID), 1,10) S
FROM MPOINTS M
WHERE M.MPOINT.GENSTLIP_OSP(
      (SELECT M.MPOINT
       FROM MPOINTS M
       WHERE TRAJ_ID=TRAJ_ID),
       1,0,
       MDSYS.SDO_GEOM.SDO_LENGTH(M.MPOINT.F_TRAJECTORY2(), 0.00005),
      (SELECT MDSYS.SDO_GEOM.SDO_LENGTH(M.MPOINT.F_TRAJECTORY2(),
                                                             0.00005)
      FROM MPOINTS M
      WHERE TRAJ_ID=TRAJ_ID), 1,10
      )<THRESHOLD AND ROWNUM<N
ORDER BY S ASC
```

Parameters: The **TRAJ_ID**, **N** and **THRESHOLD** parameters are equivalent with those in LIP.

Example:

```
SELECT   TRAJ_ID,   M.MPOINT.F_TRAJECTORY2(),   M.MPOINT.GENSTLIP_OSP((SELECT
M.MPOINT FROM MPOINTS M WHERE TRAJ_ID=1), 1,0,
MDSYS.SDO_GEOM.SDO_LENGTH(M.MPOINT.F_TRAJECTORY2(), 0.00005),
(SELECT  MDSYS.SDO_GEOM.SDO_LENGTH(M.MPOINT.F_TRAJECTORY2(),  0.00005)  FROM
MPOINTS M WHERE TRAJ_ID=1), 1,10) S
FROM MPOINTS M
WHERE   M.MPOINT.GENSTLIP_OSP((SELECT   M.MPOINT   FROM   MPOINTS   M   WHERE
TRAJ_ID=1), 1,0,
MDSYS.SDO_GEOM.SDO_LENGTH(M.MPOINT.F_TRAJECTORY2(), 0.00005),
(SELECT  MDSYS.SDO_GEOM.SDO_LENGTH(M.MPOINT.F_TRAJECTORY2(),  0.00005)  FROM
MPOINTS M WHERE TRAJ_ID=1), 1,10)< 10000000 AND ROWNUM<4    ORDER BY S ASC
```

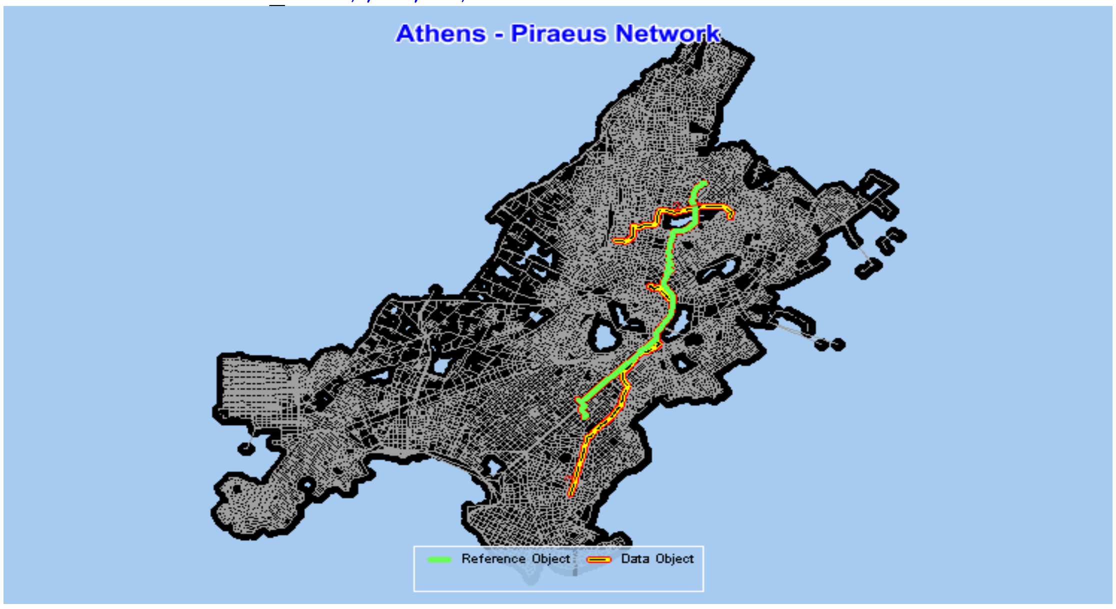
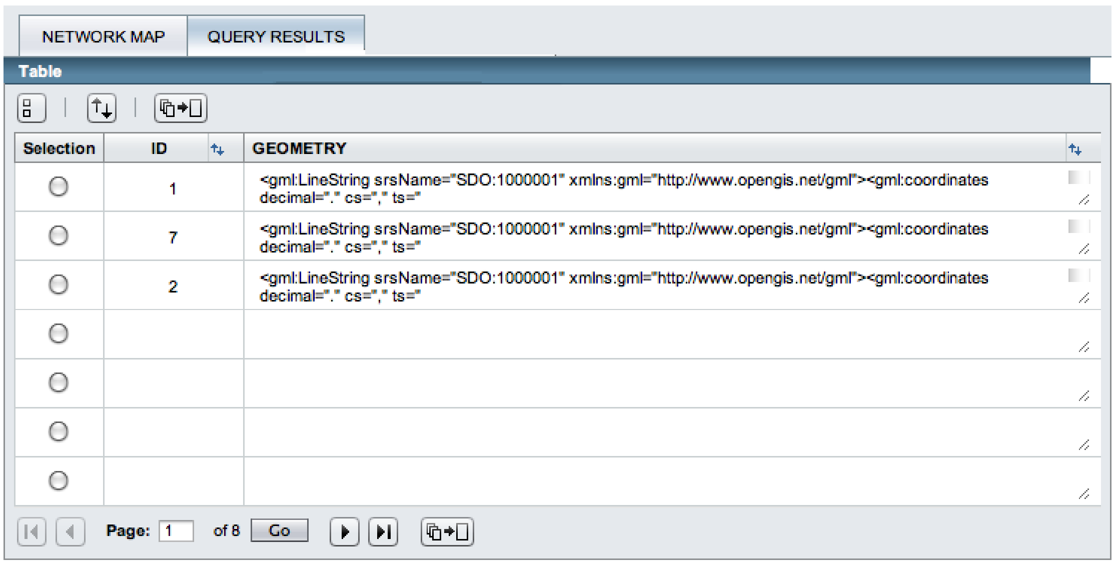


Figure 33: Answer of LIP similarity query visualized on the map



Selection	ID	GEOMETRY
○	1	<gml:LineString srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:coordinates decimal="." cs="," ts="
○	7	<gml:LineString srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:coordinates decimal="." cs="," ts="
○	2	<gml:LineString srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:coordinates decimal="." cs="," ts="
○		
○		
○		
○		

Figure 34: Answer of LIP similarity query in gml format

Notes: In the web interface upon the selection of similarity query in the corresponding category, the user is able to further define the type of similarity that will be taken into consideration in the query using the query builder at the left-down part of the screen.

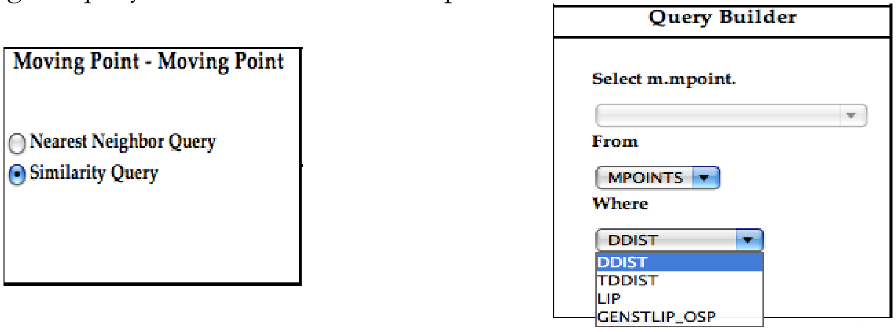


Figure 35: Similarity function selection

b. Moving Point – Static Spatial

In this category we have three types of operations, namely Point, Nearest Neighbor and Topological queries.

i. Point Query

Usage: Given a trajectory T find the regions (municipalities of Athens) that T visits during its lifespan.

Query:

```
SELECT RG_NAME, RG_REGION
FROM SP_REGIONS R WHERE MDSYS.SDO_RELATE(R.RG_REGION,
(SELECT M.MPOINT.F_TRAJECTORY2() FROM MPOINTS M WHERE
TRAJ_ID=TRAJ_ID),'MASK=ANYINTERACT QUERYTYPE=WINDOW')='TRUE'
```

Parameters: The query parameters (in bold) involve the ID of the object's (reference object) trajectory. In the mask parameter of SDO_RELATE we use ANYINTERACT denoting that the trajectory and region objects are not disjoint. The QUERYTYPE taking place in the mask is used for backward compatibility.

Example:

```
SELECT RG_NAME, RG_REGION
FROM SP_REGIONS R WHERE MDSYS.SDO_RELATE(R.RG_REGION,
(SELECT   M.MPOINT.F_TRAJECTORY2()   FROM   MPOINTS   M   WHERE   TRAJ_ID=
1),'MASK=ANYINTERACT QUERYTYPE=WINDOW')='TRUE'
```

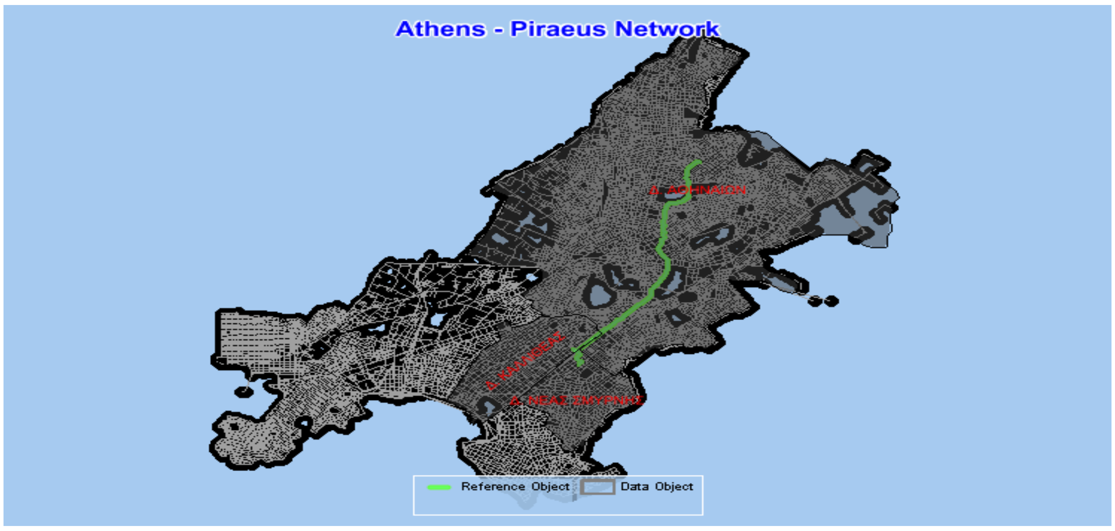
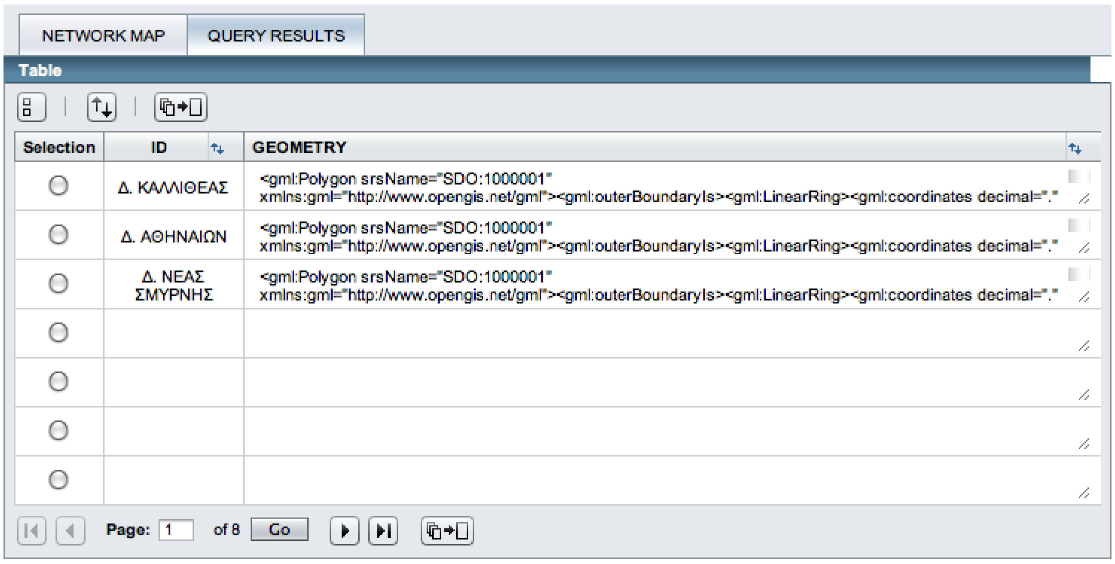


Figure 36: Answer of point query visualized on the map



Selection	ID	GEOMETRY
○	Δ. ΚΑΛΛΙΘΕΑΣ	<gml:Polygon srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:outerBoundaryIs><gml:LinearRing><gml:coordinates decimal="."
○	Δ. ΑΘΗΝΑΙΩΝ	<gml:Polygon srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:outerBoundaryIs><gml:LinearRing><gml:coordinates decimal="."
○	Δ. ΝΕΑΣ ΣΜΥΡΝΗΣ	<gml:Polygon srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:outerBoundaryIs><gml:LinearRing><gml:coordinates decimal="."
○		
○		
○		
○		

Figure 37: Answer of point query in gml format

ii. Nearest Neighbor Query

Usage: Given a trajectory T find the k nearest landmarks (POIs)
Query:

```
SELECT ADDRESS,GEOMETRY
FROM (SELECT * FROM LANDMARKS L WHERE L.TYPE='TYPE') L
WHERE MDSYS.SDO_NN(L.GEOMETRY,
(SELECT     M.MPOINT.F_TRAJECTORY2()     FROM     MPOINTS     M     WHERE
TRAJ_ID=TRAJ_ID),'SDO_NUM_RES=K')='TRUE'
```

Parameters: The query parameters (in bold) involve the **TRAJ_ID** of the object's trajectory (reference object). Furthermore the desired number of **K**-NNs needs to be specified. Eventually, the user is able to choose the **TYPE** of POIs they desire to be retrieved. Possible values (strings) for TYPE are: PORT AUTHORITIES, BUS TERMINALS, PHARMACIES, OLYMPIC VENUES, NEIGHBOURHOODS, LANDMARKS.

Example:

```
SELECT ADDRESS,GEOMETRY
FROM (SELECT * FROM LANDMARKS L WHERE L.TYPE='PHARMACIES') L
WHERE MDSYS.SDO_NN(L.GEOMETRY,
(SELECT M.MPOINT.F_TRAJECTORY2() FROM MPOINTS M WHERE
TRAJ_ID=1),'SDO_NUM_RES=4')='TRUE'
```

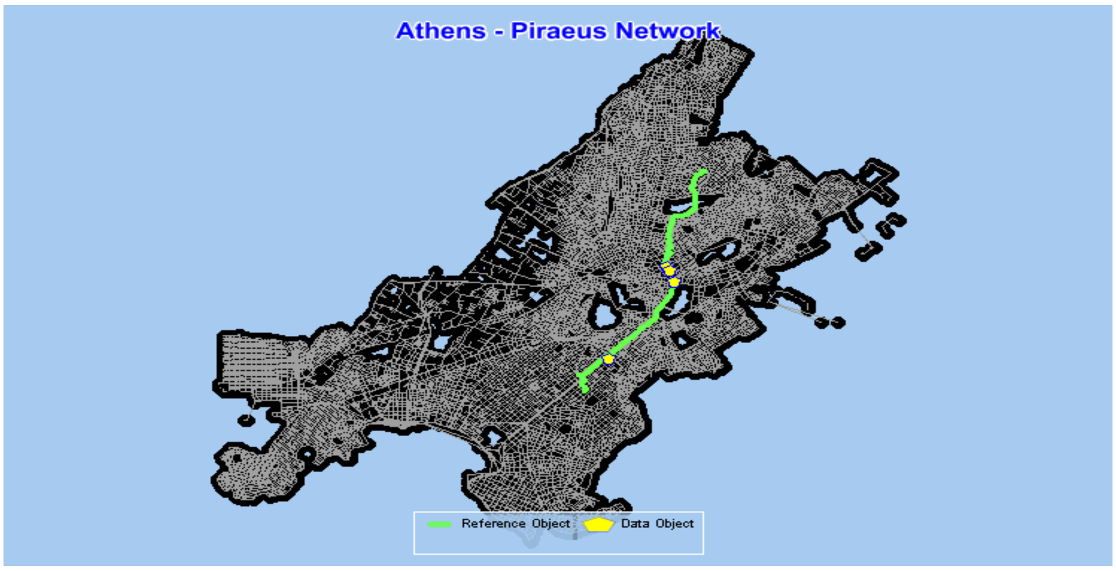
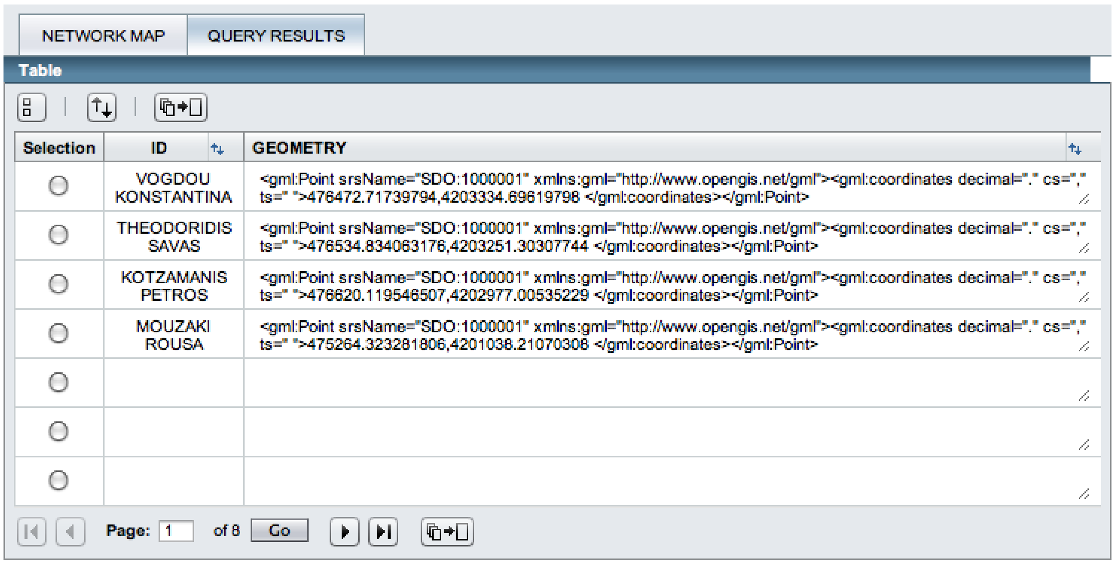


Figure 38: Answer of K-nn query visualized on the map



Selection	ID	GEOMETRY
○	VOGDOU KONSTANTINA	<gml:Point srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:coordinates decimal="." cs="," ts=" ">476472.71739794,4203334.69619798 </gml:coordinates></gml:Point>
○	THEODORIDIS SAVAS	<gml:Point srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:coordinates decimal="." cs="," ts=" ">476534.834063176,4203251.30307744 </gml:coordinates></gml:Point>
○	KOTZAMANIS PETROS	<gml:Point srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:coordinates decimal="." cs="," ts=" ">476620.119546507,4202977.00535229 </gml:coordinates></gml:Point>
○	MOUZAKI ROUSA	<gml:Point srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:coordinates decimal="." cs="," ts=" ">475264.323281806,4201038.21070308 </gml:coordinates></gml:Point>
○		
○		
○		

Figure 39: Answer of K-nn query in gml format

iii. Topological Query

Usage: Given a trajectory T find the regions that OVERLAPBDYDISJOINT, OVERLAPBDYINTERSECT, CONTAINS etc with it

Query:

```
SELECT RG_NAME, RG_REGION
FROM SP_REGIONS R
WHERE MDSYS.SDO_RELATE(R.RG_REGION,
(SELECT M.MPOINT.F_TRAJECTORY2() FROM MPOINTS M WHERE
TRAJ_ID=TRAJ_ID),'MASK=REL_TYPE QUERYTYPE=WINDOW')='TRUE'
```

Parameters: The query parameters (in bold) involve the **TRAJ_ID** of the object's trajectory (reference object) as well as the topological relation type (**REL_TYPE**). Possible choices for the mask element are:

- DISJOINT -- The boundaries and interiors do not intersect.
- TOUCH -- The boundaries intersect but the interiors do not intersect.
- OVERLAPBDYDISJOINT -- The interior of one object intersects the boundary and interior of the other object, but the two boundaries do not intersect. This relationship occurs, for example, when a line originates outside a polygon and ends inside that polygon.
- OVERLAPBDYINTERSECT -- The boundaries and interiors of the two objects intersect.
- EQUAL -- The two objects have the same boundary and interior.
- CONTAINS -- The interior and boundary of one object is completely contained in the interior of the other object.
- COVERS -- The interior of one object is completely contained in the interior or the boundary of the other object and their boundaries intersect.
- INSIDE -- The opposite of CONTAINS. A INSIDE B implies B CONTAINS A.
- COVEREDBY -- The opposite of COVERS. A COVEREDBY B implies B COVERS A.
- ON -- The interior and boundary of one object is on the boundary of the other object (and the second object covers the first object). This relationship occurs, for example, when a line is on the boundary of a polygon.

Example:

```
SELECT RG_NAME, RG_REGION
FROM SP_REGIONS R
WHERE MDSYS.SDO_RELATE(R.RG_REGION,
(SELECT M.MPOINT.F_TRAJECTORY2() FROM MPOINTS M WHERE
TRAJ_ID=2),'MASK=CONTAINS QUERYTYPE=WINDOW')='TRUE'
```

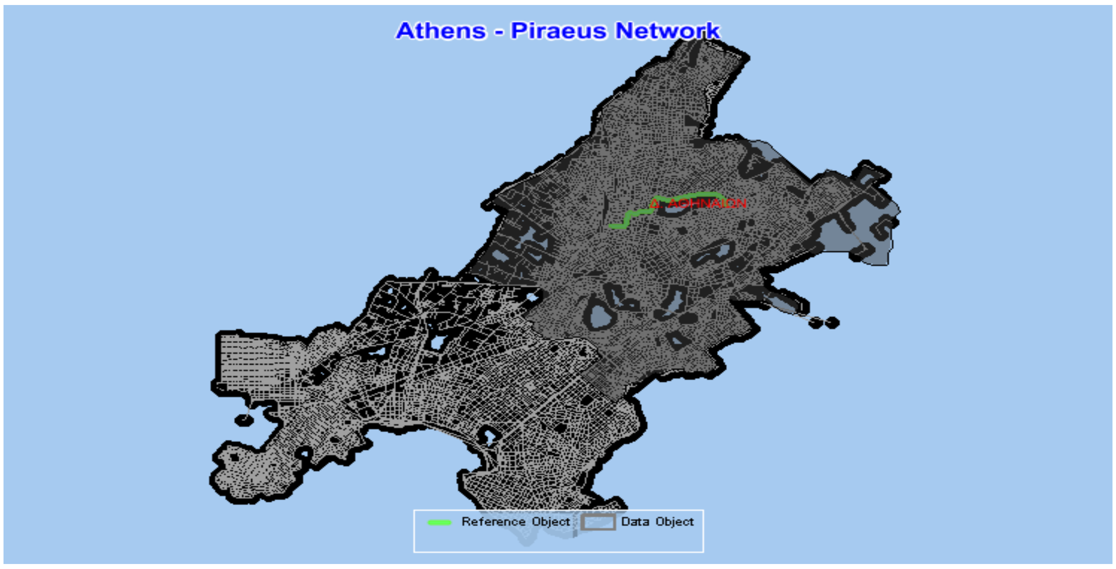
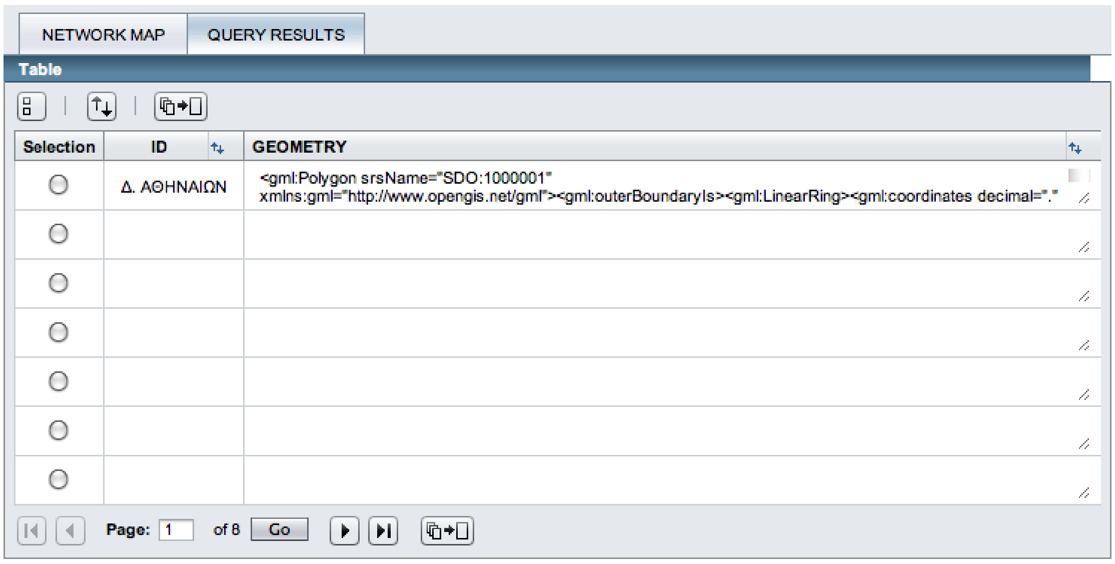


Figure 40: Answer of topological query (MASK=CONTAINS) visualized on the map



Selection	ID	GEOMETRY
○	Δ. ΑΘΗΝΑΙΩΝ	<gml:Polygon srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:outerBoundaryIs><gml:LinearRing><gml:coordinates decimal="."
○		
○		
○		
○		
○		
○		

Figure 41: Answer of topological query (MASK=CONTAINS)
in gml format

c. Static Spatial – Moving Point

In this category we have four types of operations, namely Range, Nearest Neighbor, Topological and Directional Queries.

i. Range Query

Usage: Find trajectory parts contained in a given spatiotemporal window
Query:
```
SELECT TBFUNCTIONS.MV_QUERY_WINDOW(
                               SDO_GEOMETRY(2003,      1000001,      NULL,
                               SDO_ELEM_INFO_ARRAY(1,1003,3),
                               SDO_ORDINATE_ARRAY(X1,Y1,X2,Y2)),
                                             TAU_TLL.D_PERIOD_SEC(

                                   TAU_TLL.D_TIMEPOINT_SEC(timepoint1),

                                   TAU_TLL.D_TIMEPOINT_SEC(timepoint2))

                                                                  )
FROM DUAL
```
Parameters: The query parameters (in bold) involve the lower left (**X1,Y1**) and upper right (**X2,Y2)** of the (rectangular) spatial window as well as a time period (**timepoint1, timepoint2)** for the temporal part of the query
Example:
```
SELECT TBFUNCTIONS.MV_QUERY_WINDOW(
                               SDO_GEOMETRY(2003,      1000001,      NULL,
                               SDO_ELEM_INFO_ARRAY(1,1003,3),
SDO_ORDINATE_ARRAY(465000, 4200000, 480000, 4201900)),
                                             TAU_TLL.D_PERIOD_SEC(

                              TAU_TLL.D_TIMEPOINT_SEC(2009,07,04,21,12,43),

                              TAU_TLL.D_TIMEPOINT_SEC(2009,07,04,21,15,56))

                                                                  )
FROM DUAL
```

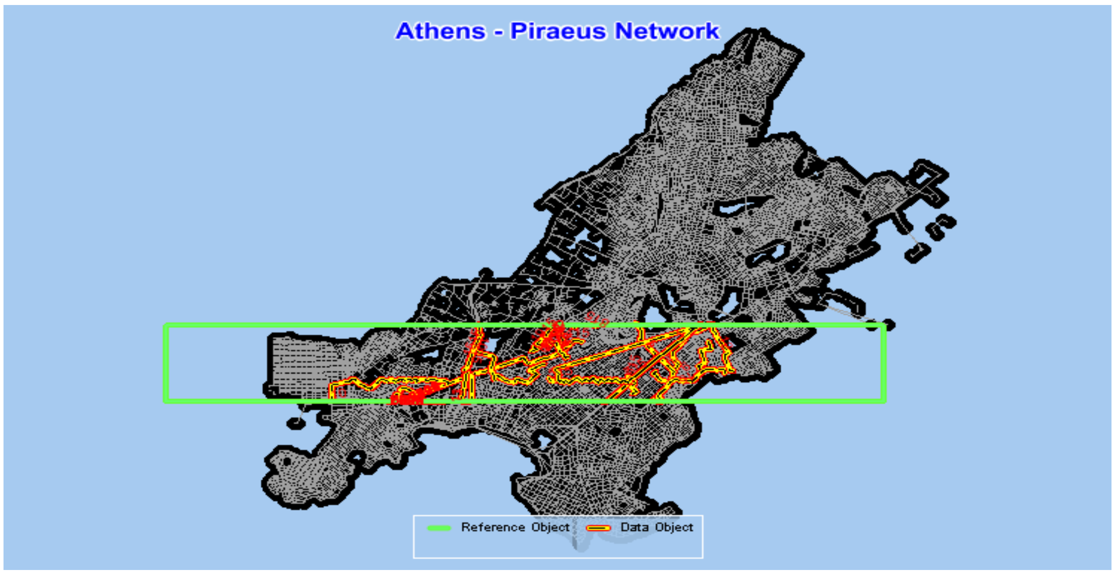
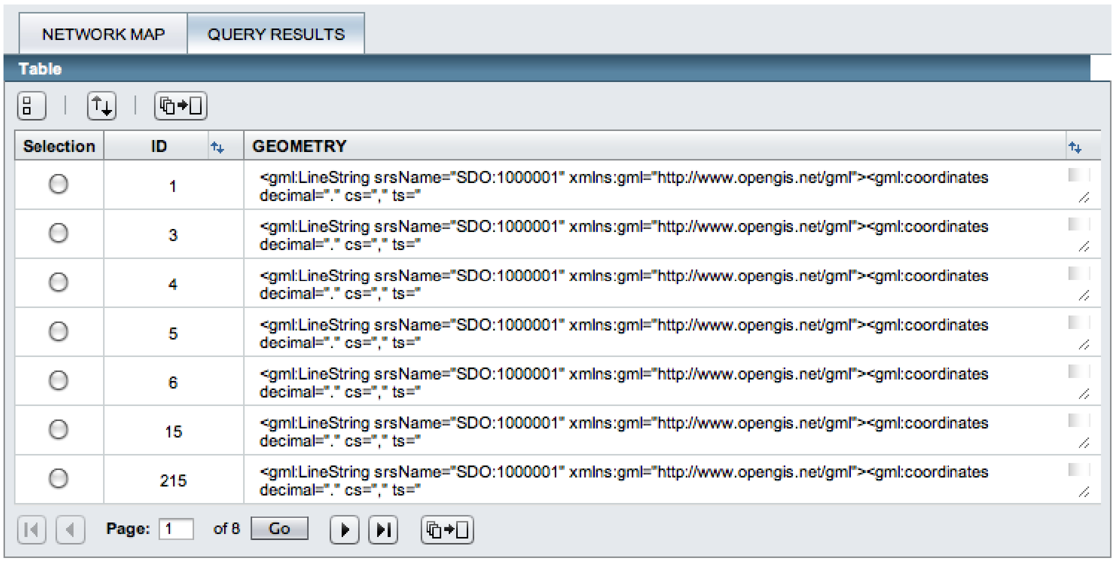


Figure 42: Answer of range query visualized on the map

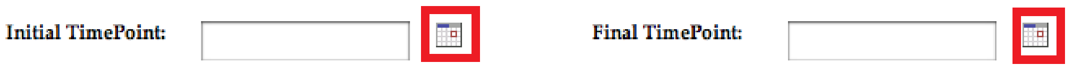


Selection	ID	GEOMETRY
○	1	<gml:LineString srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:coordinates decimal="." cs="," ts="
○	3	<gml:LineString srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:coordinates decimal="." cs="," ts="
○	4	<gml:LineString srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:coordinates decimal="." cs="," ts="
○	5	<gml:LineString srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:coordinates decimal="." cs="," ts="
○	6	<gml:LineString srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:coordinates decimal="." cs="," ts="
○	15	<gml:LineString srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:coordinates decimal="." cs="," ts="
○	215	<gml:LineString srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:coordinates decimal="." cs="," ts="

Page: 1 of 8 Go

Figure 43: Answer of range query in gml format

Notes: The insertion of the appropriate time interval for a range query can be made by using the text boxes labeled Initial and Final Timepoint, placed at the top of the web interface.



Initial TimePoint: Final TimePoint:

Figure 44: Initial and Final Timepoint Selection

By choosing the icon (surrounded by red rectangles in Figure 44) at the right side of time point selection textboxes, a popup window will appear presenting a calendar as well as a clock to enable users define corresponding time points. Note that after identifying a time point by clicking or typing in the popup calendar window, you need to additionally click inside each textbox to confirm your selection which is then passed inside the formed query.

Furthermore, we should note that the specification of X1,Y1,X2,Y2 by the user could have been graphically made by choosing the construction of map images in SVG format. However, making every node of the road network (or point in the map) selectable tremendously affects response time. As a result we allow users type the exact coordinates of the desired spatial rectangle. In the query refinement section we will examine an alternative way to pass the spatial window parameter.

ii. Nearest Neighbor Query

Usage: Find the K nearest to a POI trajectory parts, within a given time period

Query:

```
SELECT TBFUNCTIONS.MV_INCPOINTNNSEARCH(
                            X,Y,
                            TAU_TLL.D_TIMEPOINT_SEC(timepoint1),
                            TAU_TLL.D_TIMEPOINT_SEC(timepoint2),
                            K)
FROM DUAL
```

Parameters: The query parameters (in bold) involve the coordinates (**X,Y**) of the POI, the time period (**timepoint1,timepoint2)** and the desired number of nearest neighbors **K**

Example:

```
SELECT TBFUNCTIONS.MV_INCPOINTNNSEARCH(
                            480000,4201900,
                            TAU_TLL.D_TIMEPOINT_SEC(2009,06,30,00,59,40),TAU_TLL
                            .D_TIMEPOINT_SEC(2009,07,17,00,59,55), 15)
FROM DUAL
```

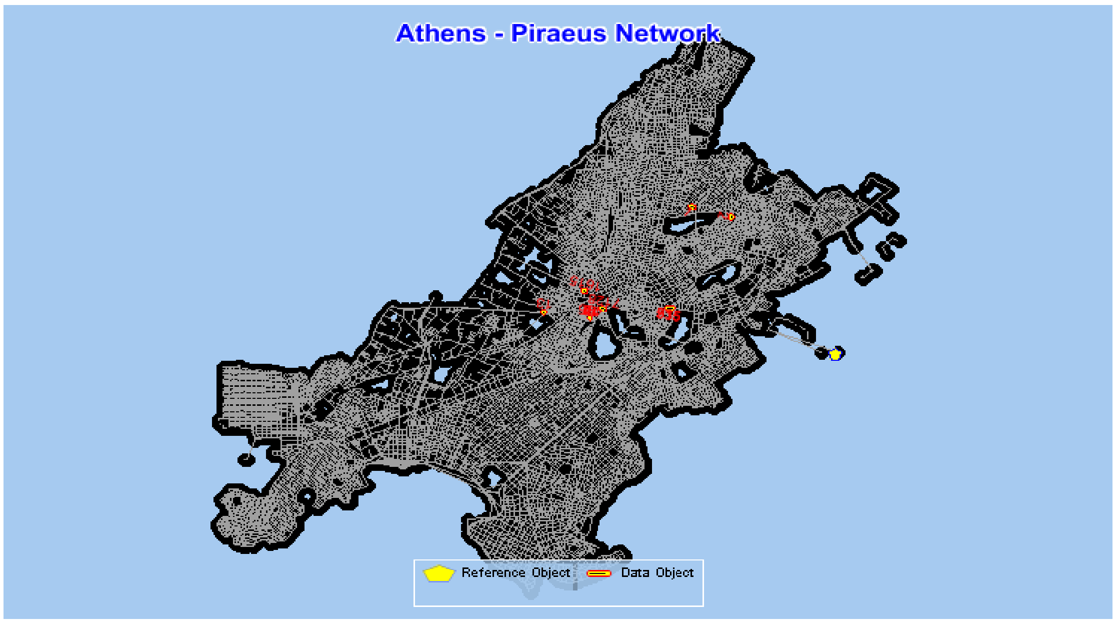
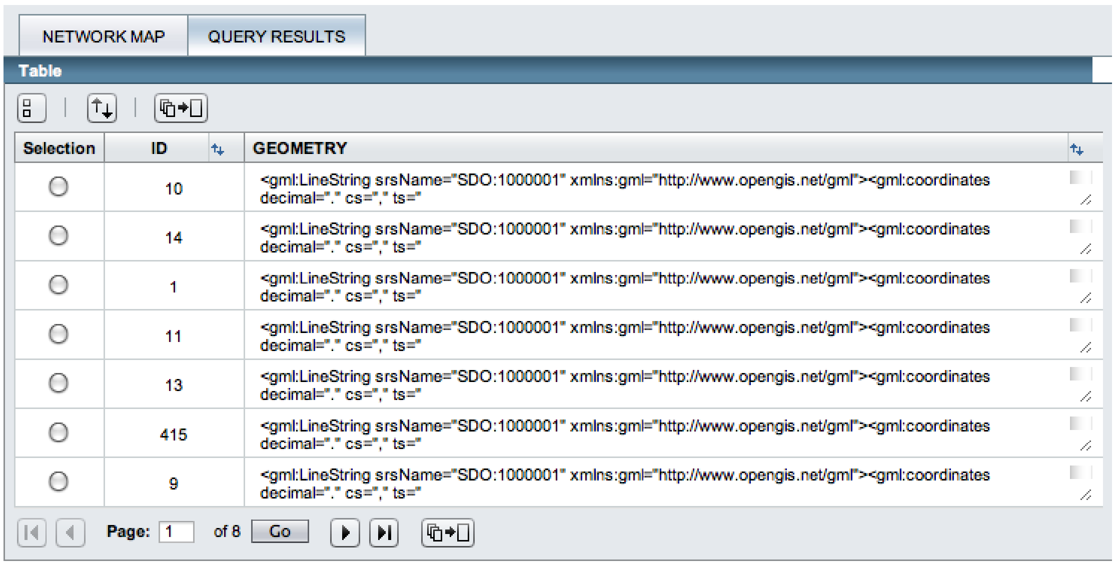


Figure 45: Answer of incremental point NN query visualized on the map



Selection	ID	GEOMETRY
○	10	<gml:LineString srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:coordinates decimal="." cs="," ts="
○	14	<gml:LineString srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:coordinates decimal="." cs="," ts="
○	1	<gml:LineString srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:coordinates decimal="." cs="," ts="
○	11	<gml:LineString srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:coordinates decimal="." cs="," ts="
○	13	<gml:LineString srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:coordinates decimal="." cs="," ts="
○	415	<gml:LineString srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:coordinates decimal="." cs="," ts="
○	9	<gml:LineString srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:coordinates decimal="." cs="," ts="

Figure 46: Answer of incremental point NN query in gml format

Notes: The insertion of the appropriate time interval for the incremental point NN query can be made by using corresponding textboxes as we have already discussed.

iii. Topological Query

Usage: Find the trajectories that enter/leave an area within a given timeperiod

Query:

```
SELECT TRAJ_ID,M.MPOINT.F_TRAJECTORY2()
FROM MPOINTS M
WHERE TRAJ_ID IN (
      SELECT DISTINCT * FROM TABLE(

TBFUNCTIONS.TB_TOPOLOGICAL_QUERY(

                                            SDO_GEOMETRY(2003, NULL, NULL,

                                            SDO_ELEM_INFO_ARRAY(1,1003,3),

                                         SDO_ORDINATE_ARRAY(X1,Y1,X2,Y2)),

                                                     TAU_TLL.D_PERIOD_SEC(
                               TAU_TLL.D_TIMEPOINT_SEC(timepoint1),
                                     TAU_TLL.D_TIMEPOINT_SEC(timepoint2)),
                               'MASK')

                                                                        )
                  )
```

Parameters: The query parameters (in bold) involve the lower left (**X1,Y1**) and upper right (**X2,Y2)** of a (rectangular) area as well as a time period (**timepoint1, timepoint2)** for the temporal part of the query. Finally, a **MASK** must be defined to clarify the type of topological query. Possible MASK (string) values can be 'ENTER', 'LEAVE', 'ENTER_LEAVE' depending whether the users are interested in trajectories that enter/leave or enter&leave the area within the given timeperiod.

Example:

```
SELECT TRAJ_ID,M.MPOINT.F_TRAJECTORY2()
FROM MPOINTS M
WHERE TRAJ_ID IN (SELECT DISTINCT * FROM TABLE
(TBFUNCTIONS.TB_TOPOLOGICAL_QUERY(SDO_GEOMETRY(2003, NULL, NULL,
SDO_ELEM_INFO_ARRAY(1,1003,3),
SDO_ORDINATE_ARRAY(465000,4200000,480000,4201900)),
TAU_TLL.D_PERIOD_SEC(TAU_TLL.D_TIMEPOINT_SEC(2009,06,22,01,43,17),
TAU_TLL.D_TIMEPOINT_SEC(2009,06,22,01,49,10)),'ENTER_LEAVE')))
```

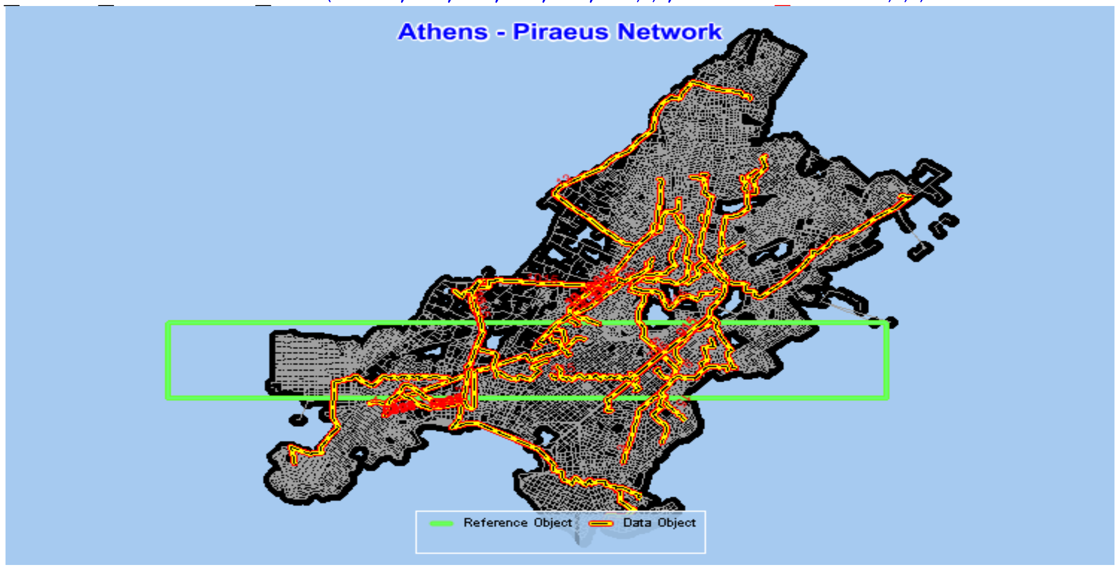
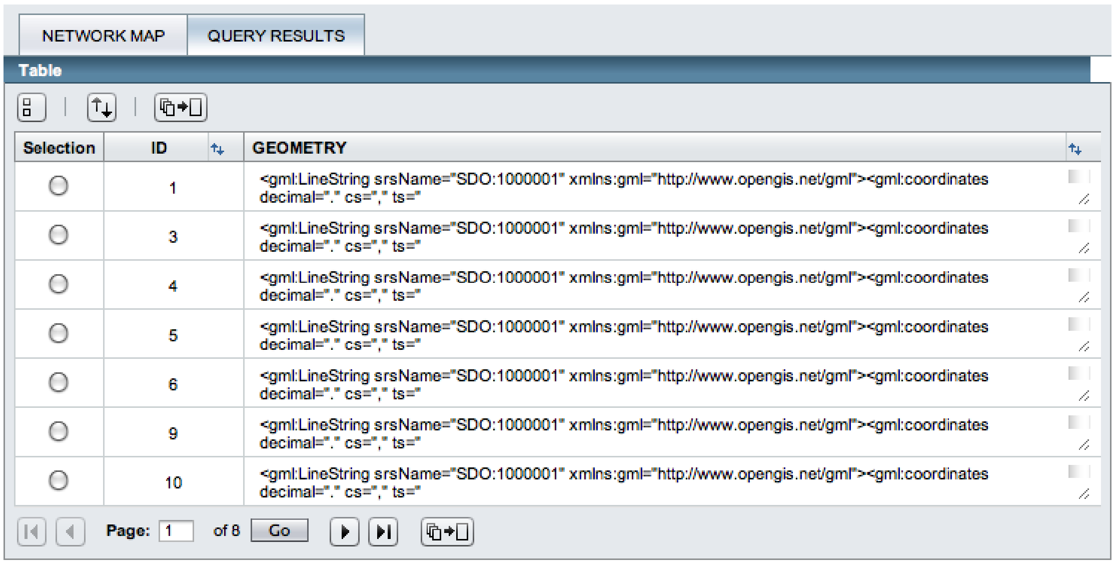


Figure 47: Answer of topological query visualized on the map



Selection	ID	GEOMETRY
○	1	<gml:LineString srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:coordinates decimal="." cs="," ts="
○	3	<gml:LineString srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:coordinates decimal="." cs="," ts="
○	4	<gml:LineString srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:coordinates decimal="." cs="," ts="
○	5	<gml:LineString srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:coordinates decimal="." cs="," ts="
○	6	<gml:LineString srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:coordinates decimal="." cs="," ts="
○	9	<gml:LineString srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:coordinates decimal="." cs="," ts="
○	10	<gml:LineString srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:coordinates decimal="." cs="," ts="

Page: 1 of 8 Go

Figure 48: Answer of topological query in gml format

Notes: The insertion of the appropriate time interval for the incremental point NN query can be made by using corresponding textboxes as it already has been discusses.

iv. Directional Query

Usage: Find trajectories the location of which is east, west, north, south, front, behind, left, right of a Point at a given time instant (time point). In this type of query we distinguish the following directional functions:

- F_LEFT: The *f_left* operation returns true if the location of the point at the user defined timepoint is left from the argument geometry - point
- Similarly, operations *f_right, f_front, f_behind* return true if the location of the point at the user defined timepoint is on the right, in front or behind the argument geometry – point, respectively.

- Furthermore, we augment our operator set with a related set of methods that identify whether a moving point is located *west*, *east*, *north*, *south* of a geometry. These methods are differentiated from the previous as we do not care for the heading of the moving point.

Query:

```
SELECT M.MPOINT.F_TRAJECTORY2()
FROM MPOINTS M
WHERE M.MPOINT.F_EAST(

                                                  SDO_GEOMETRY(2001,1000001,

                                       SDO_POINT_TYPE(X,Y, NULL),NULL,NULL),

                                        TAU_TLL.D_TIMEPOINT_SEC(timepoint1),
         A1,A2)=1
```

Parameters: The query parameters (in bold) involve the coordinates (**X,Y**) of the reference point geometry, the time instant (**timepoint)** and the angle range **(A1, A2)** that dictates the search space towards the specified direction.

Example:

```
SELECT M.MPOINT.F_TRAJECTORY2()
FROM MPOINTS M
WHERE M.MPOINT.F_LEFT(SDO_GEOMETRY(2001,1000001,
                 SDO_POINT_TYPE(465000,4200000, NULL),NULL,NULL),
                 TAU_TLL.D_TIMEPOINT_SEC(2009,06,10,21,39,11),30,150)=1
```

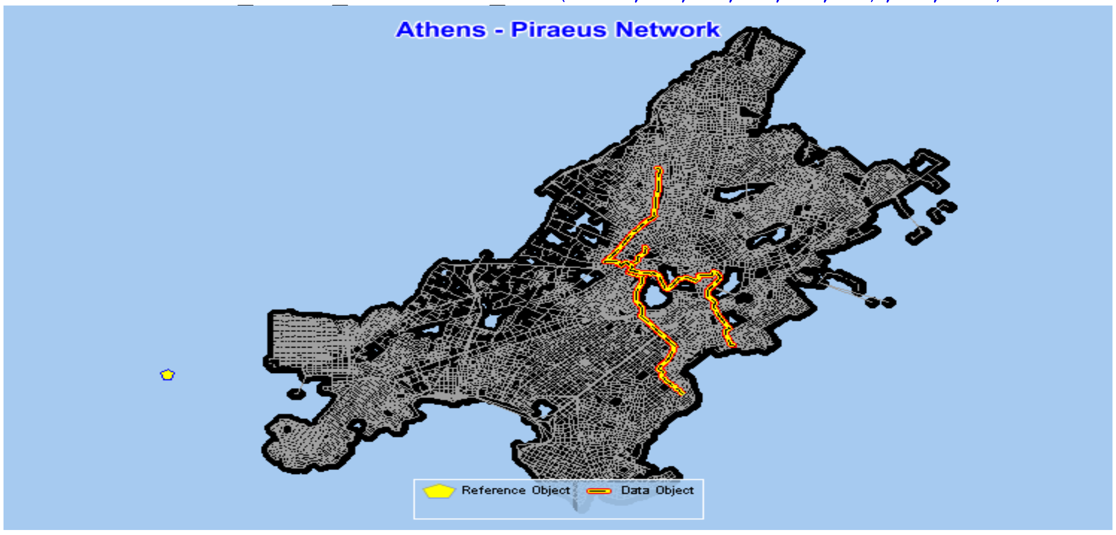
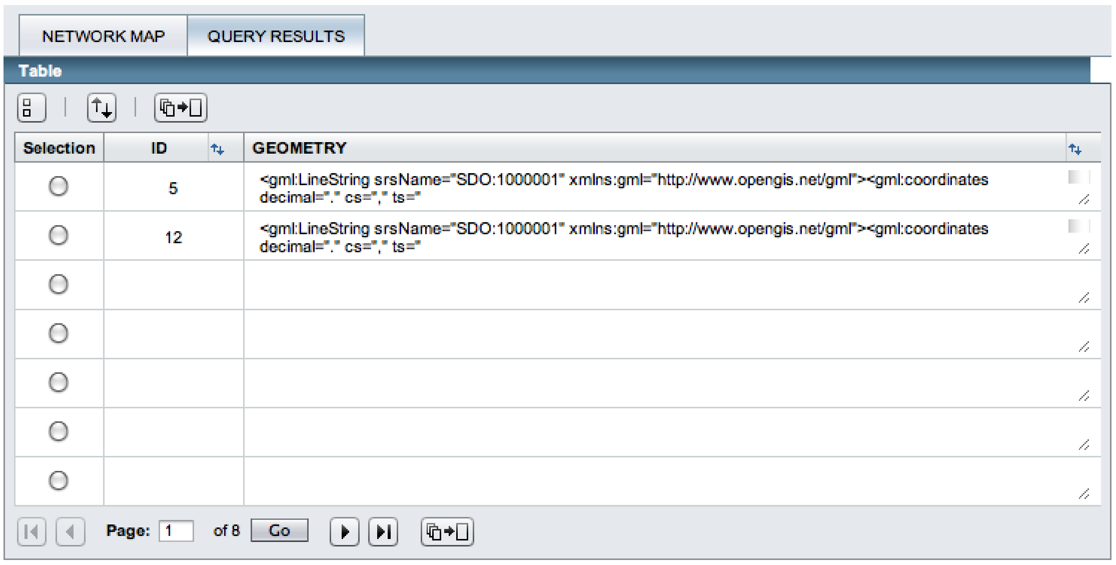


Figure 49: Answer of directional (f_left) query visualized on the map



Selection	ID	GEOMETRY
○	5	<gml:LineString srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:coordinates decimal="." cs="," ts="
○	12	<gml:LineString srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:coordinates decimal="." cs="," ts="
○		
○		
○		
○		
○		

Figure 50: Answer of directional (f_left) query in gml format

Notes: The insertion of the appropriate time point can be made by using the "Initial Timepoint" textbox only.

In the web interface upon the selection of directional query in the corresponding category, the user is able to further select the type that will be taken into consideration in the query using the query builder at the left-down part of the screen.

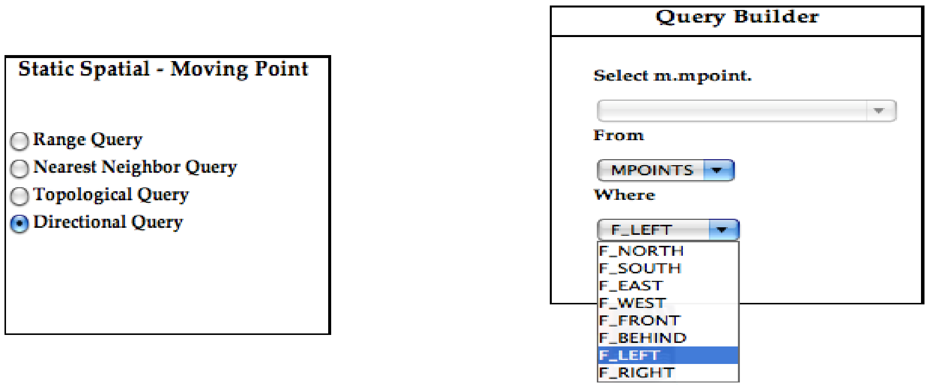


Figure 51: Directional function selection

d. Complementary operations

Apart from the previously presented query categorization, a set of complementary operators are provided. They can be used by performing a selection in the "Select" part of the query builder (provided that neither similarity nor directional query category have been selected). The following figure presents the set of complementary functions.

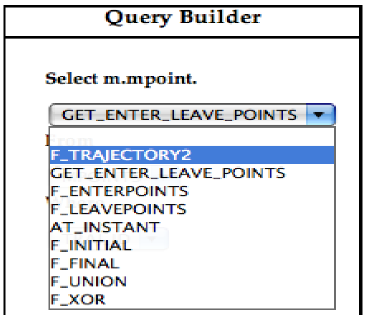


Figure 52: Set of complementary functions

F_TRAJECTORY2

Usage: Function used to visualize a specific trajectory on the map

Query:

```
SELECT M.MPOINT.F_TRAJECTORY2()
FROM MPOINTS M
WHERE TRAJ_ID=TRAJ_ID
```

Parameters: The only parameter involves the ID of the trajectory that the user wishes to visualize. In case "TRAJ_ID" is left as is, the whole set of stored trajectories is presented on the map.

Example:

```
SELECT M.MPOINT.F_TRAJECTORY2()
FROM MPOINTS M
WHERE TRAJ_ID=10
```

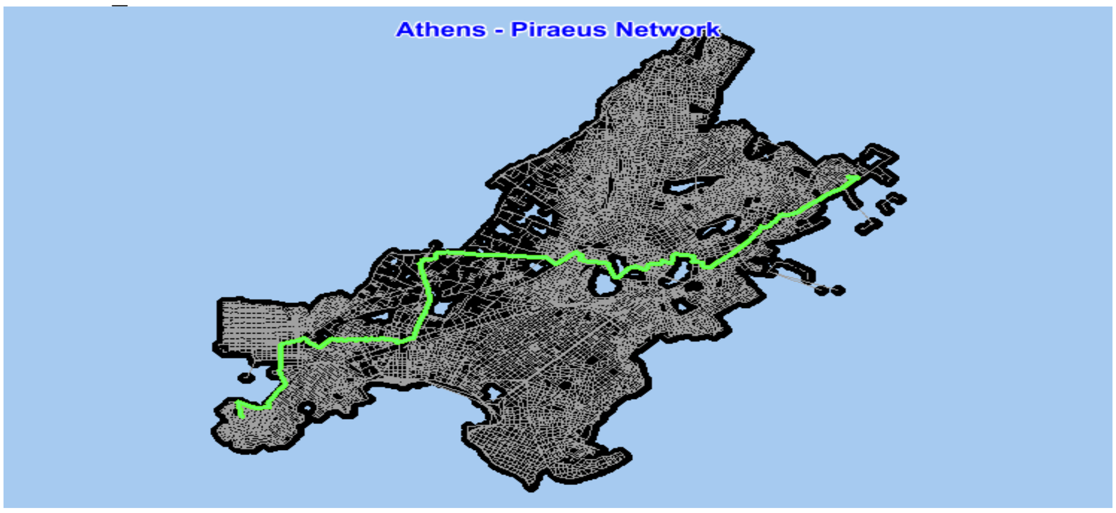


Figure 53: Visualization of TRAJ_ID=10

GET_ENTER_LEAVE_POINTS, F_ENTERPOINTS, F_LEAVEPOINTS

Usage: This set of functions is used to identify the enter/leave (or both) points of a specific trajectory in respect to a spatial region on the map

Query:

```
SELECT M.MPOINT.GET_ENTER_LEAVE_POINTS(
                        SDO_GEOMETRY(2003, NULL, NULL,
                        SDO_ELEM_INFO_ARRAY(1,1003,3),
                        SDO_ORDINATE_ARRAY(X1,Y1,X2,Y2)))
FROM MPOINTS M
WHERE TRAJ_ID=TRAJ_ID
```

Parameters: The query parameters involve the lower left (`X1,Y1`) and upper right (`X2,Y2)` of the region as well as a trajectory ID. F_ENTERPOINTS and F_LEAVEPOINTS receive similar parameters.

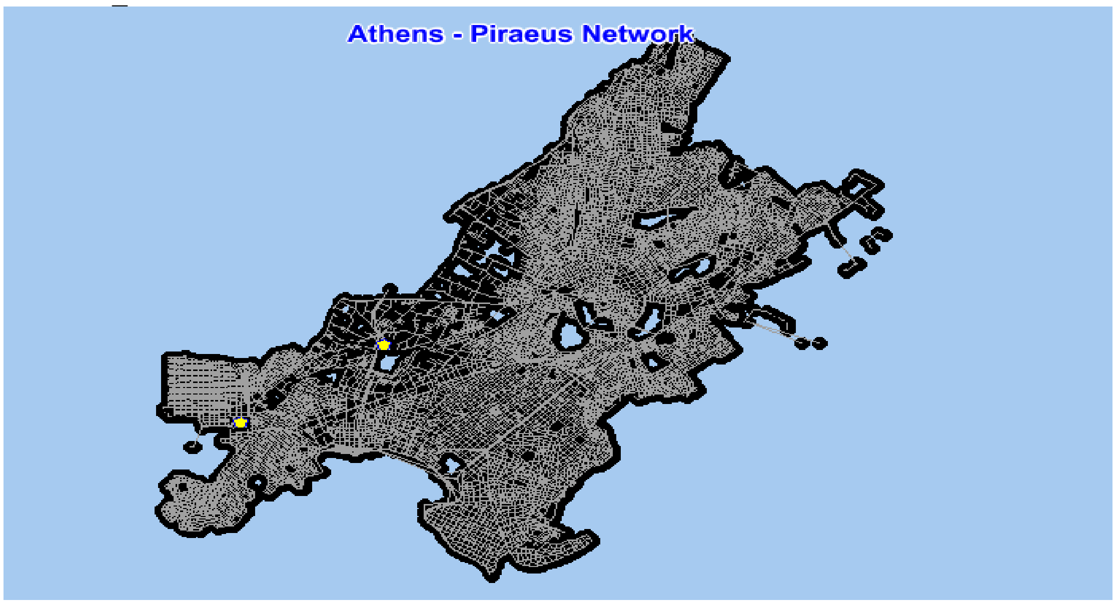
Example:
```
SELECT    M.MPOINT.GET_ENTER_LEAVE_POINTS(SDO_GEOMETRY(2003,    NULL,    NULL,
SDO_ELEM_INFO_ARRAY(1,1003,3),
SDO_ORDINATE_ARRAY(465000,4200000,480000,4201900)))
FROM MPOINTS M
WHERE TRAJ_ID=10
```


Figure 54: Enter, Leave points of TRAJ_ID=10 in respect with the selected region

AT_INSTANT

Usage: The function returns the location of a moving object (specific TRAJ_ID) at a given time point.

Query:

```
SELECT M.MPOINT.AT_INSTANT(TAU_TLL.D_TIMEPOINT_SEC(timepoint))
FROM MPOINTS M
WHERE TRAJ_ID=TRAJ_ID
```

Parameters: The query parameters involve the specification of the exact time point (using the Initial Time Point Textbox) and the ID of the desired trajectory. In case 'TRAJ_ID' is used as is instead of a specific trajectory identificator, the locations of all available moving objects for the given time point are projected on the map.

Example:
```
SELECT M.MPOINT.AT_INSTANT(TAU_TLL.D_TIMEPOINT_SEC(2009,06,10,21,39,11))
FROM MPOINTS M
WHERE TRAJ_ID=TRAJ_ID
```

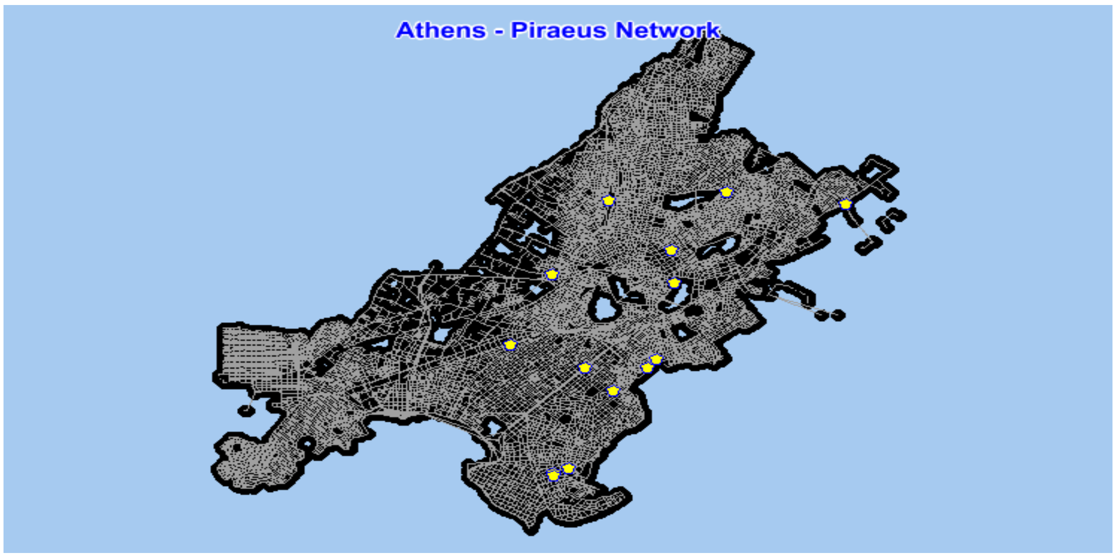


Figure 55: Locations of moving objects at the specified time point

F_INITIAL, F_FINAL

Usage: The function returns the starting, ending point of a given trajectory, respectively
Query:

```
SELECT M.MPOINT.F_INITIAL()
FROM MPOINTS M
WHERE TRAJ_ID=TRAJ_ID
```

Parameters: The only parameter involves the ID of the trajectory that the user wishes to visualize its initial/ final location.
Example:

```
SELECT M.MPOINT.F_INITIAL()
FROM MPOINTS M
WHERE TRAJ_ID=1
```

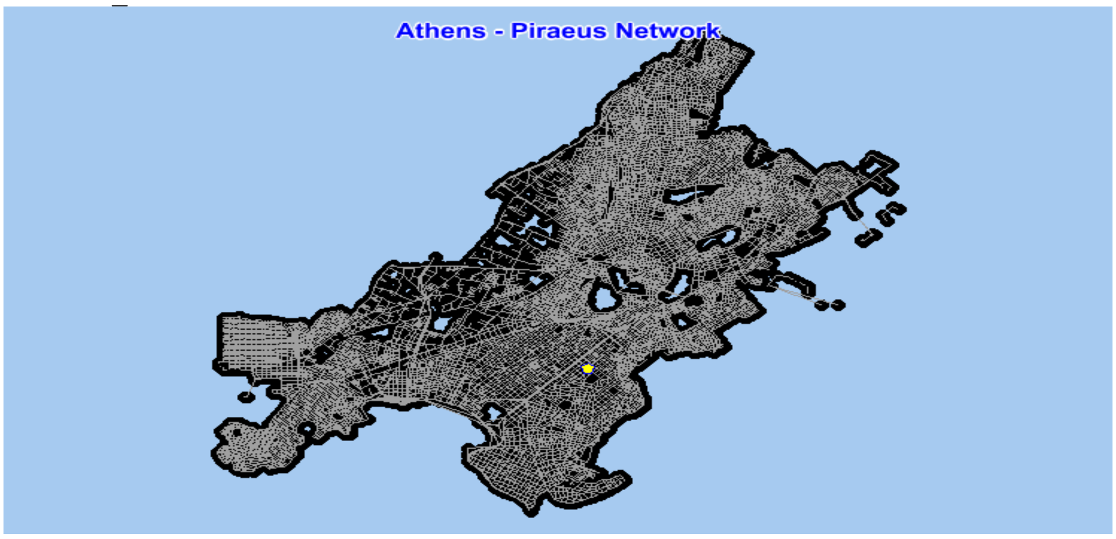


Figure 56: Initial location of TRAJ_ID=1

F_UNION

Usage: The function returns a geometry object that is the topological union (OR operation) of an instanced point with a given trajectory at a specific time point

Query:
```
SELECT    M.MPOINT.F_UNION((SELECT    M.MPOINT    FROM    MPOINTS    M    WHERE
TRAJ_ID=TRAJ_ID1),TOLERANCE,TAU_TLL.D_TIMEPOINT_SEC(timepoint))
FROM MPOINTS M
WHERE TRAJ_ID=TRAJ_ID2
```
Parameters: The parameters involve the **TRAJ_ID1** of a selected moving object and the **TRAJ_ID2** of the reference, time-instanced moving point. In addition, a specific time point and a tolerance value need to be specified. Many Spatial functions accept a tolerance parameter. If the distance between two points is less than or equal to the tolerance, the two points are considered as a unique point. Thus, tolerance is usually a reflection of how accurate or precise users perceive their spatial data to be.
Example:
```
SELECT    M.MPOINT.F_UNION((SELECT    M.MPOINT    FROM    MPOINTS    M    WHERE
TRAJ_ID=1),0.001,TAU_TLL.D_TIMEPOINT_SEC(2009,06,10,21,39,11))
FROM MPOINTS M
WHERE TRAJ_ID=10
```

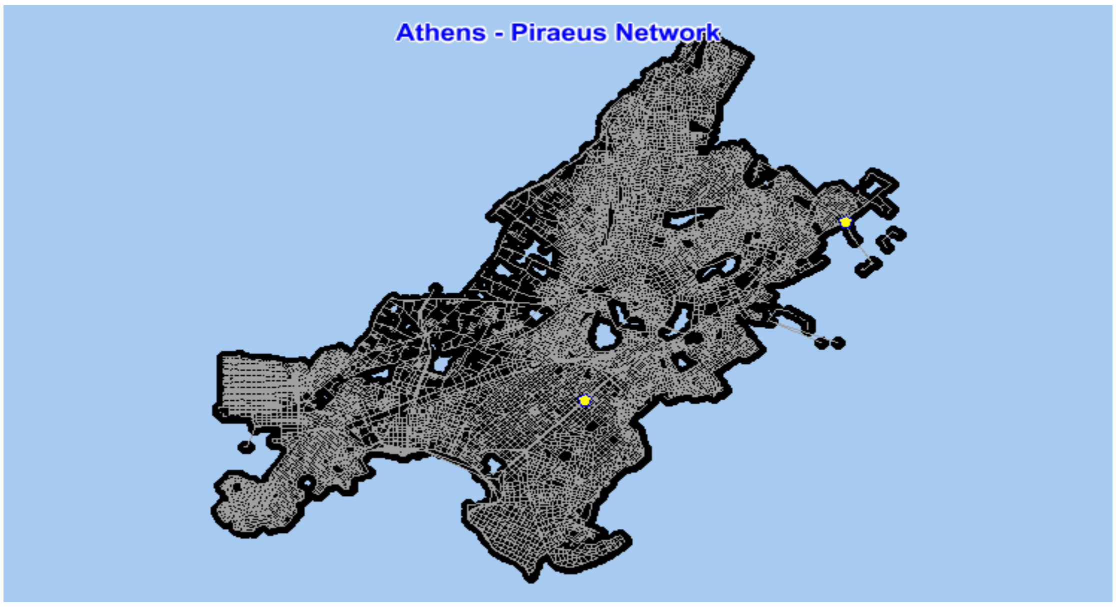


Figure 57: F_UNION between TRAJ_ID=10 at the specified time point and TRAJ_ID=1

F_XOR
Usage: The function returns a geometry object that is the topological symmetric difference (XOR operation) of an instanced point with a given trajectory at a specific time point
Query:

```
SELECT    M.MPOINT.F_XOR((SELECT    M.MPOINT    FROM    MPOINTS    M    WHERE
TRAJ_ID=TRAJ_ID1),TOLERANCE,TAU_TLL.D_TIMEPOINT_SEC(timepoint))
FROM MPOINTS M
WHERE TRAJ_ID=TRAJ_ID2
```
Parameters: The parameters are equivalent with those in F_UNION
Example:
```
SELECT    M.MPOINT.F_XOR((SELECT    M.MPOINT    FROM    MPOINTS    M    WHERE
TRAJ_ID=1),0.001,TAU_TLL.D_TIMEPOINT_SEC(2009,06,10,21,39,11))
FROM MPOINTS M
WHERE TRAJ_ID=12
```

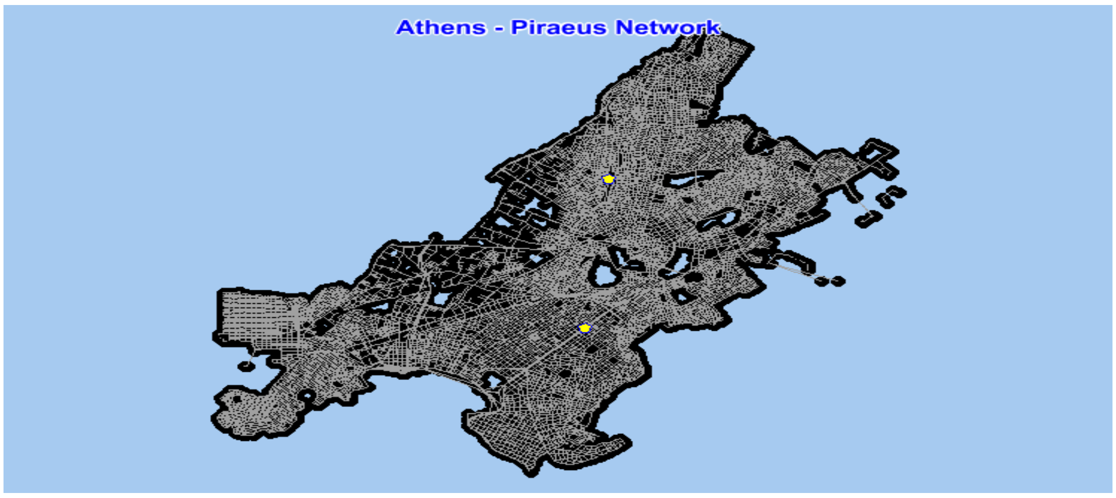


Figure 58: F_XOR between TRAJ_ID=12 at the specified time point and TRAJ_ID=1

7.4 Query Refinement

Certain types of queries can be further refined by choosing between their results and exploit them as parameters in a new query. The talk mainly regards interaction between "Moving Point – Static Spatial" and "Static Spatial – Moving Point" categories. In other words, a data object that participates in the answer of a query belonging to the "Moving Point – Static Spatial" category can be defined as the reference object in a "Static Spatial – Moving Point" query. The exact match is obviously determined by the type of parameter that "Static Spatial – Moving Point" queries accept.

In particular, the user has the ability to perform the following types of query combinations:

- Moving Point – Static Spatial: Point and Topological Query with Static Spatial – Moving Point: Range and Topological Query
- Moving Point – Static Spatial: Nearest Neighbor Query with Static Spatial – Moving Point: Nearest Neighbor and Directional Query

Moreover, the results of the "Moving Point – Static Spatial" category can be exploited in the previously presented GET_ENTER_LEAVE_POINTS, F_ENTERPOINTS, F_LEAVEPOINTS queries.

We now proceed by citing representative examples of the steps that the user should follow during the query refinement process.

a.Moving Point – Static Spatial: Point Query & Static Spatial Moving Point: Range Query

STEP1: Point Query Execution

```
SELECT RG_NAME, RG_REGION
FROM SP_REGIONS R WHERE MDSYS.SDO_RELATE(R.RG_REGION,
(SELECT M.MPOINT.F_TRAJECTORY2() FROM MPOINTS M WHERE TRAJ_ID=1),
'MASK=ANYINTERACT QUERYTYPE=WINDOW')='TRUE'
```

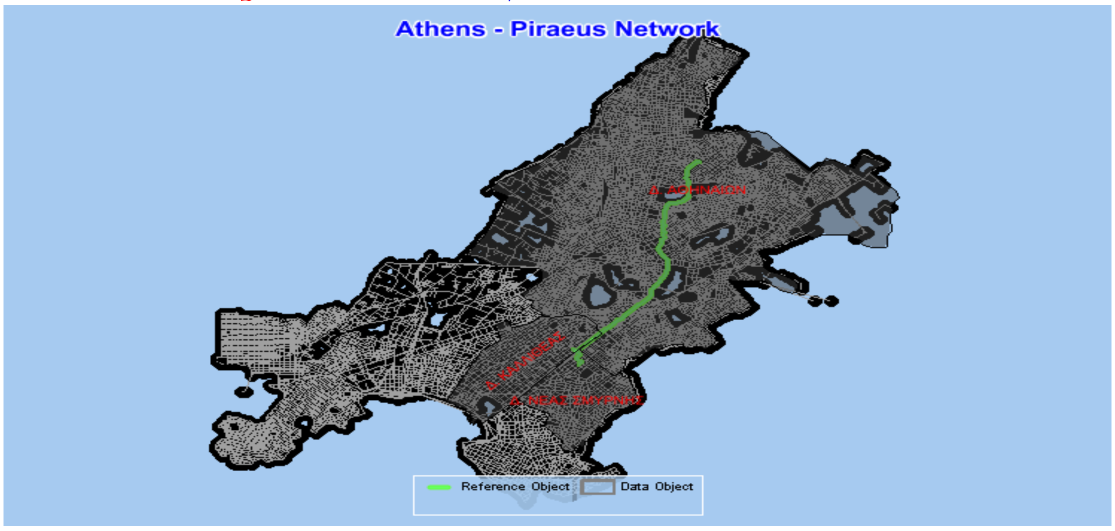
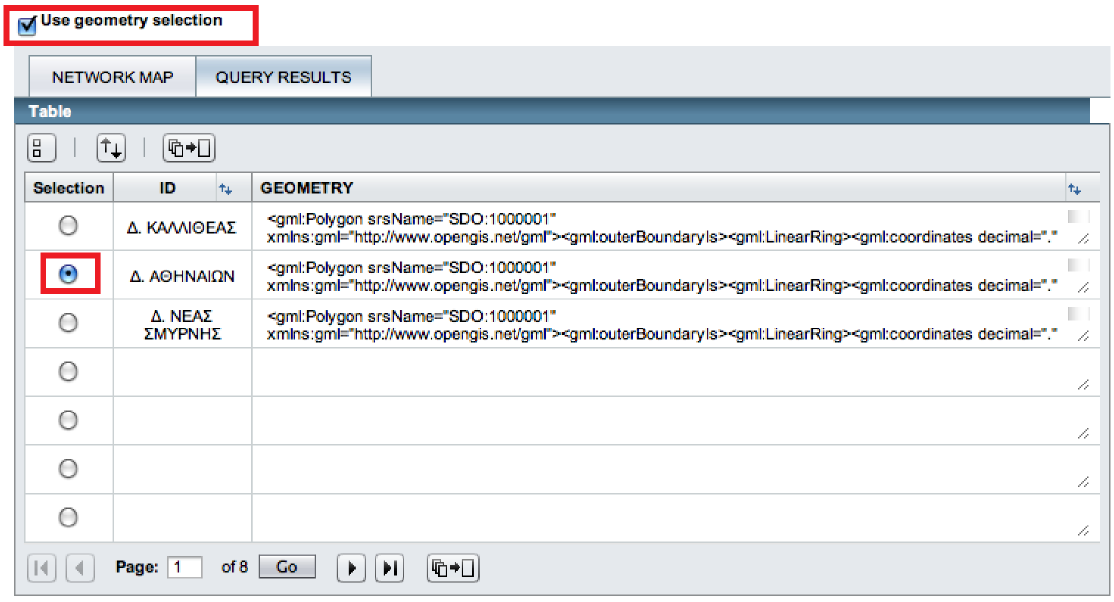


Figure 59: Answer of point query visualized on the map

STEP2: New Reference Object Selection

To select the new reference object, the user should choose the "QUERY RESULTS" tab in the interface. In the table presenting the results of the previously posed query, the selection column provides appropriate radio buttons for this purpose. Finally the user should inform the system that the chosen result is to be passed as a parameter to the next query. This is achieved by clicking on the "Use geometry selection" checkbox above the table.



☑ Use geometry selection

Selection	ID	GEOMETRY
○	Δ. ΚΑΛΛΙΘΕΑΣ	<gml:Polygon srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:outerBoundaryIs><gml:LinearRing><gml:coordinates decimal="."
●	Δ. ΑΘΗΝΑΙΩΝ	<gml:Polygon srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:outerBoundaryIs><gml:LinearRing><gml:coordinates decimal="."
○	Δ. ΝΕΑΣ ΣΜΥΡΝΗΣ	<gml:Polygon srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:outerBoundaryIs><gml:LinearRing><gml:coordinates decimal="."
○		
○		
○		
○		

Page: 1 of 8 Go

Figure 60: New reference object selection (municipality of Athens)

STEP3: Query Refinement

In the range query that will be formed the user needs to define only the desired time period since the spatial window has already been declared in the previous step (in this example the spatial window regards the municipality of Athens)1.

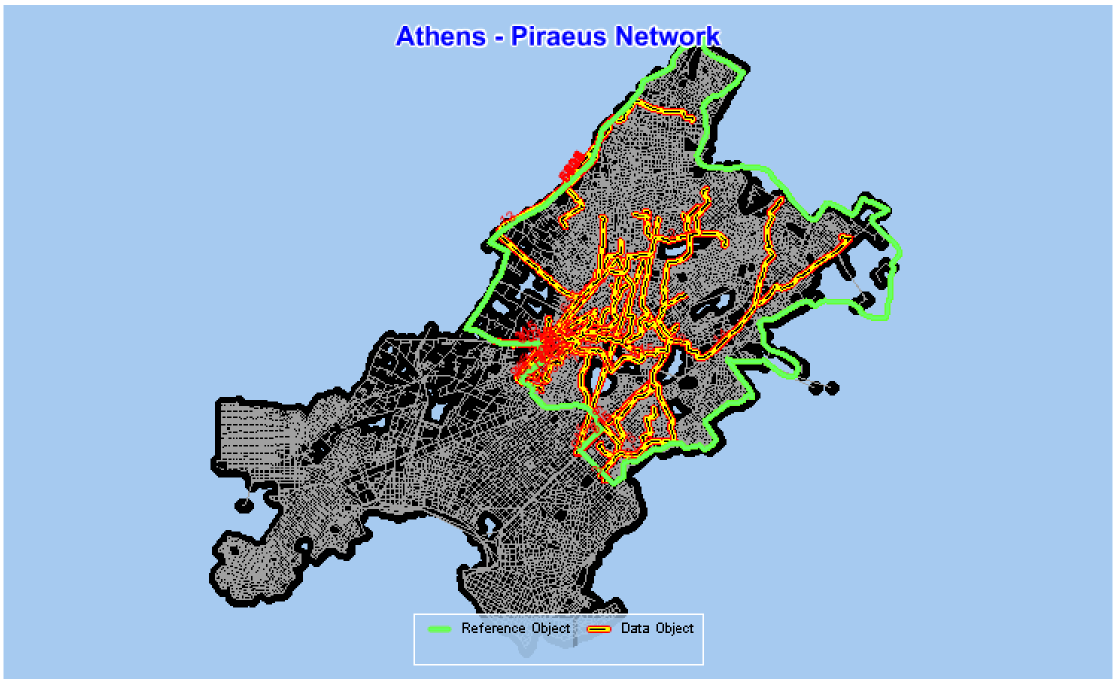


Figure 61: Range query execution in the refinement process

b. Moving Point – Static Spatial: Nearest Neighbor Query & Static Spatial Moving Point: Range Query

STEP1: Moving Point – Static Spatial: Nearest Neighbor Query Execution

```
SELECT ADDRESS,GEOMETRY
FROM (SELECT * FROM LANDMARKS L WHERE L.TYPE='LANDMARKS') L
WHERE MDSYS.SDO_NN(L.GEOMETRY,
(SELECT      M.MPOINT.F_TRAJECTORY2()      FROM      MPOINTS      M      WHERE
TRAJ_ID=4),'SDO_NUM_RES=7')='TRUE'
```

[1] The spatial window that appears to the user in the formed range query, still displays the coordinates a default rectangular area. However, the actual execution takes into account the new reference object specified in step 2. This saves some extra client-server communication.

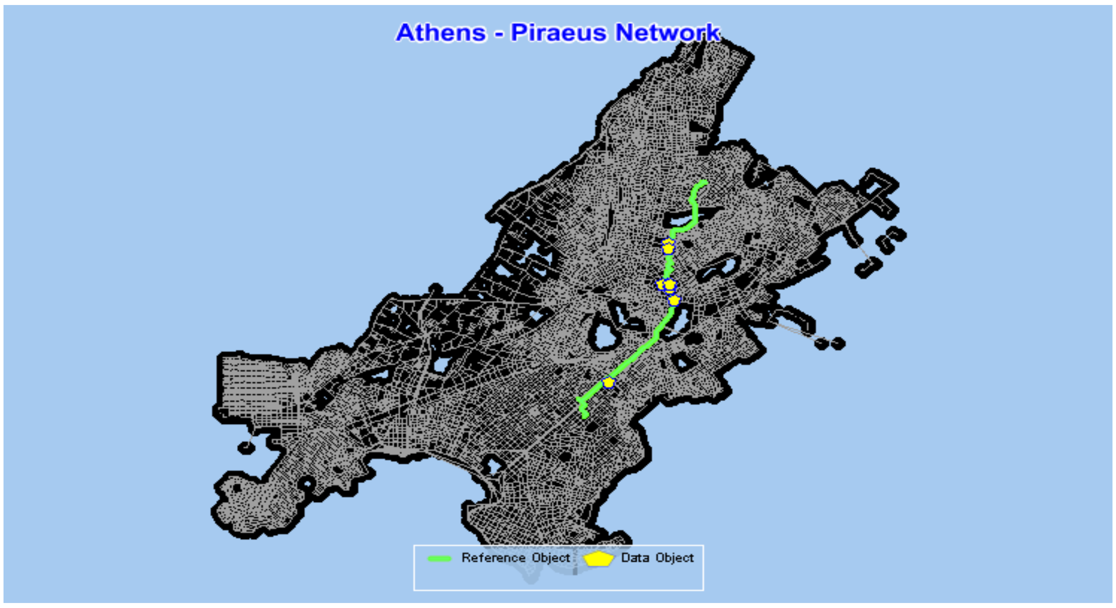
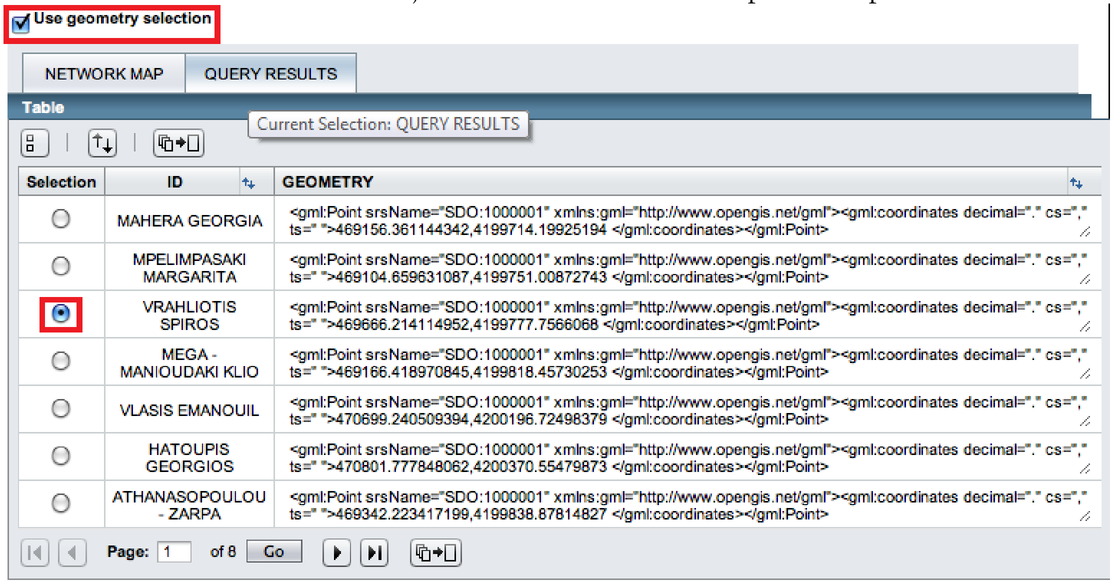


Figure 62: Answer of Moving Points – Static Spatial: NN query
visualized on the map

STEP2: New Reference Object Selection
The selection of the new reference object is similar with the example in the previous section.



[✓] Use geometry selection

Selection	ID	GEOMETRY
○	MAHERA GEORGIA	<gml:Point srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:coordinates decimal="." cs="," ts=" ">469156.361144342,4199714.19925194 </gml:coordinates></gml:Point>
○	MPELIMPASAKI MARGARITA	<gml:Point srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:coordinates decimal="." cs="," ts=" ">469104.659631087,4199751.00872743 </gml:coordinates></gml:Point>
●	VRAHLIOTIS SPIROS	<gml:Point srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:coordinates decimal="." cs="," ts=" ">469666.214114952,4199777.7566068 </gml:coordinates></gml:Point>
○	MEGA - MANIOUDAKI KLIO	<gml:Point srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:coordinates decimal="." cs="," ts=" ">469166.418970845,4199818.45730253 </gml:coordinates></gml:Point>
○	VLASIS EMANOUIL	<gml:Point srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:coordinates decimal="." cs="," ts=" ">470699.240509394,4200196.72498379 </gml:coordinates></gml:Point>
○	HATOUPIS GEORGIOS	<gml:Point srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:coordinates decimal="." cs="," ts=" ">470801.777848062,4200370.55479873 </gml:coordinates></gml:Point>
○	ATHANASOPOULOU - ZARPA	<gml:Point srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:coordinates decimal="." cs="," ts=" ">469342.223417199,4199838.87814827 </gml:coordinates></gml:Point>

Page: 1 of 8 Go

Figure 63: New reference object selection (municipality of
Athens)

STEP3: Query Refinement (Static Spatial – Moving Point: Nearest Neighbor query)
In the nearest neighbor query that will be formed the user needs to define only the desired time period since the reference point has already been declared in the previous step.

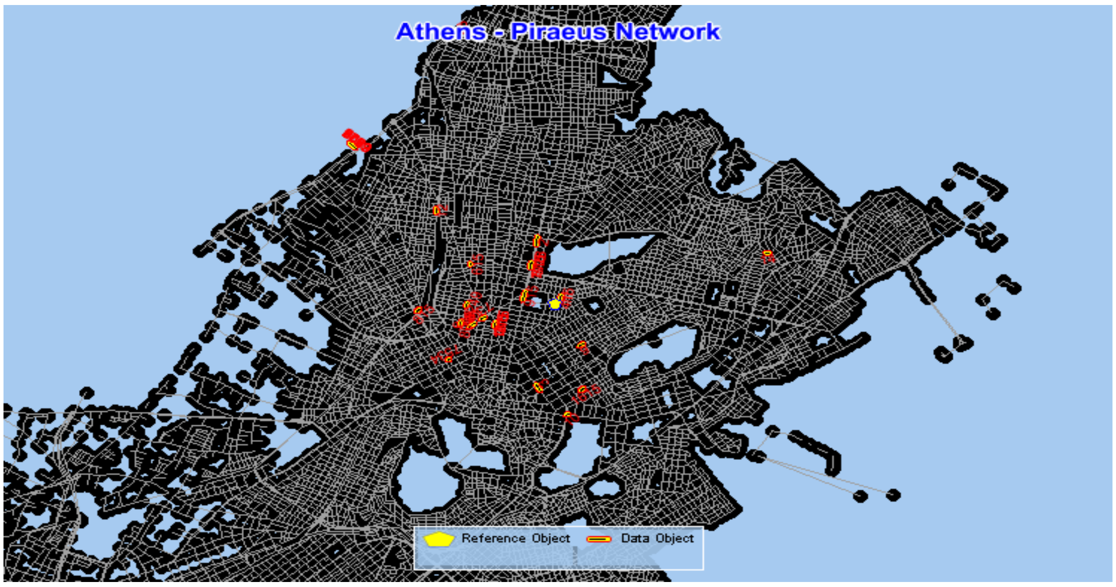


Figure 64: Static Spatial – Moving Point: Nearest Neighbor query execution in the refinement process

c. Additional

As already mentioned the results of the "Moving Point – Static Spatial" category can be exploited in GET_ENTER_LEAVE_POINTS, F_ENTERPOINTS, F_LEAVEPOINTS queries. Since the triplet of these functions receives a spatial window parameter only Moving Point – Static Spatial: Point and Topological queries can be utilized in the first step of the process.

STEP1: Moving Point – Static Spatial: Point Query Execution

```
SELECT RG_NAME, RG_REGION
FROM SP_REGIONS R WHERE MDSYS.SDO_RELATE(R.RG_REGION,
(SELECT M.MPOINT.F_TRAJECTORY2() FROM MPOINTS M WHERE TRAJ_ID=1),
'MASK=ANYINTERACT QUERYTYPE=WINDOW')='TRUE'
```

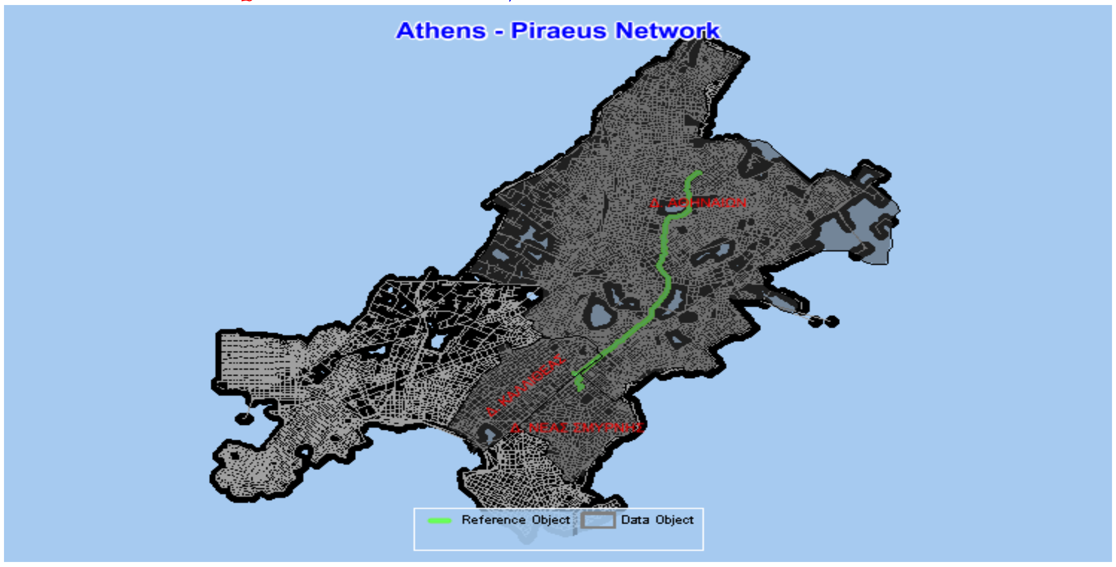
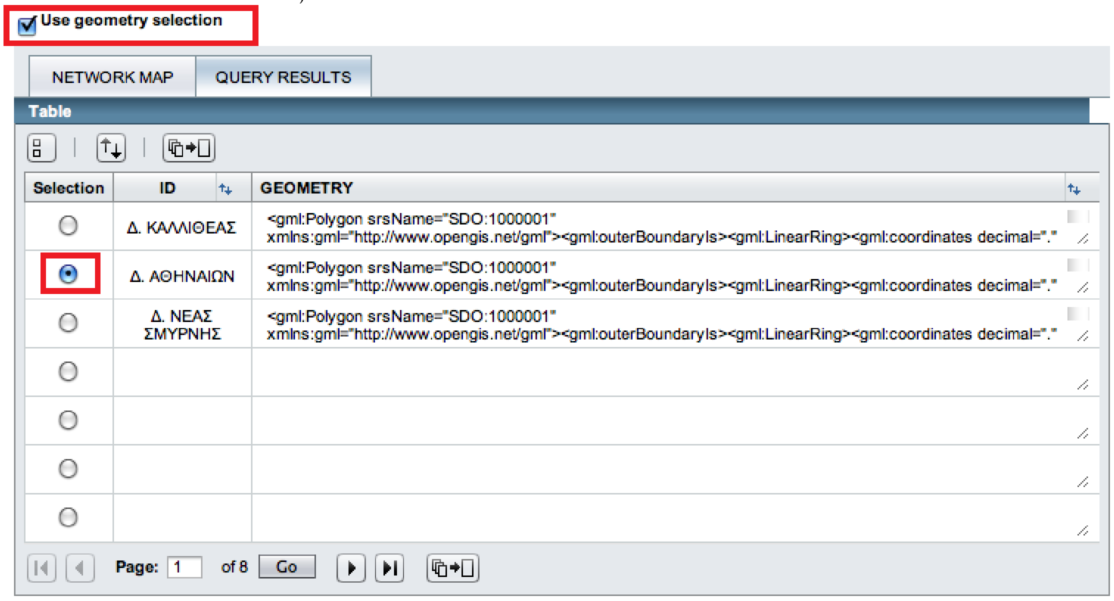


Figure 65: Answer of point query visualized on the map

STEP2: New Reference Object Selection



☑ Use geometry selection

NETWORK MAP | QUERY RESULTS

Table

Selection	ID	GEOMETRY
○	Δ. ΚΑΛΛΙΘΕΑΣ	<gml:Polygon srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:outerBoundaryIs><gml:LinearRing><gml:coordinates decimal="."
●	Δ. ΑΘΗΝΑΙΩΝ	<gml:Polygon srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:outerBoundaryIs><gml:LinearRing><gml:coordinates decimal="."
○	Δ. ΝΕΑΣ ΣΜΥΡΝΗΣ	<gml:Polygon srsName="SDO:1000001" xmlns:gml="http://www.opengis.net/gml"><gml:outerBoundaryIs><gml:LinearRing><gml:coordinates decimal="."
○		
○		
○		
○		

Page: 1 of 8 Go

Figure 66: New reference object selection (municipality of Athens)

STEP3: Query Refinement (F_ENTERPOINTS)

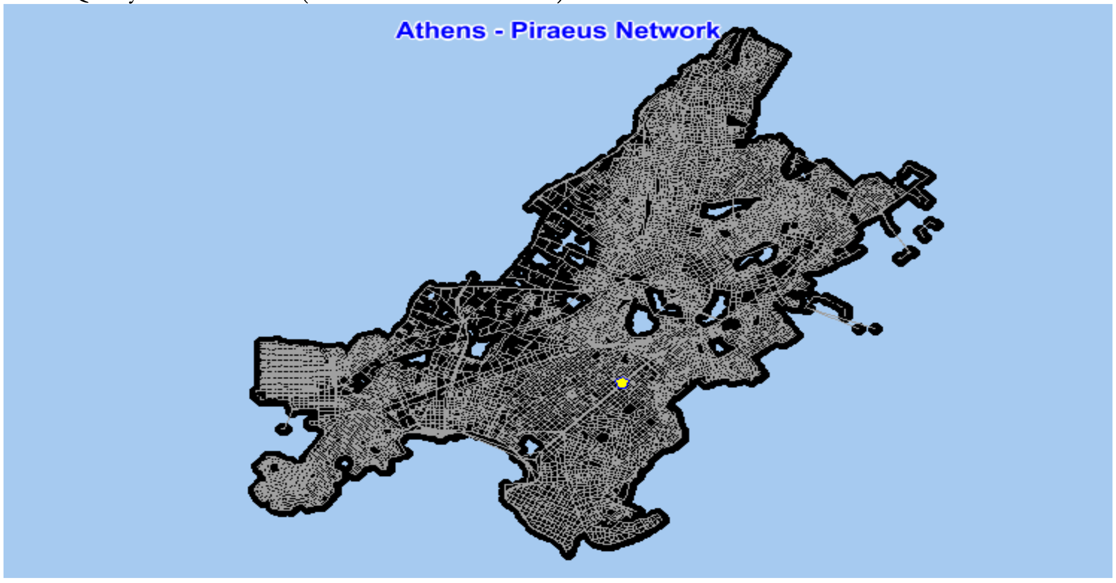


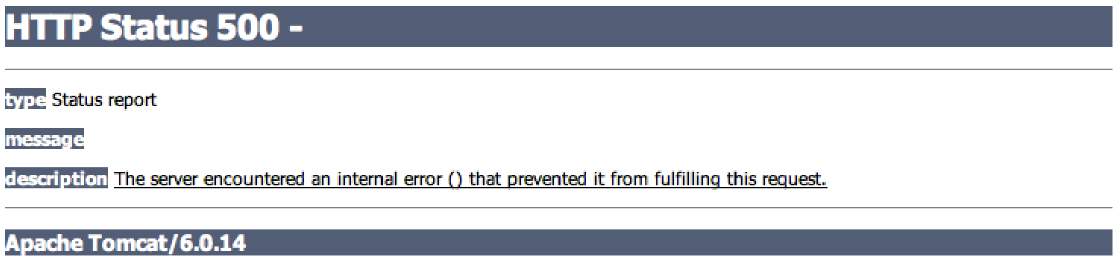
Figure 67: F_ENTERPOINTS results in the query refinement process

7.5 Troubleshooting

There are two known situations that can cause malfunction to the HERMES web application.

Leaving the session inactive for several minutes may cause the scripts that form the query upon the selection of a respective category to become inactive (i.e no query is presented in the corresponding textbox after a selection). Refreshing the session solves that problem

Server errors similar to the one presented in the figure below may occur as a result of invalid modifications to the initially formed query by the user. This happens due to the fact that there is no intermediate parser to examine the validity of the query request before passing it to the database server.



HTTP Status 500 -

type Status report

message

description The server encountered an internal error () that prevented it from fulfilling this request.

Apache Tomcat/6.0.14

Figure 68: Server Error

We recommend that you avoid changing the query, apart from the required parameter parts. Upon an error occurrence it is preferable to close the window and start a new session.

8 Comparison with Related Work

Several research efforts have tried to model spatio-temporal databases using the *moving object* concept. In 15 the authors propose a new line of research where moving points and moving regions are viewed as three-dimensional (2D + time) or higher dimensional entities whose structure and behavior is captured by modeling them as abstract data types. Such abstract data types for moving points and moving regions have been introduced in 24, together with a set of operations on such entities. The model presented in 24 was the first attempt to deal with continuous motion while in 17 the definition of the discrete representation of the above-discussed abstract data types is presented. The interesting part of the discrete model is how *"moving"* types are represented. The authors describe the *sliced representation* behind which, the basic idea is to decompose the temporal development of a value into fragments called *"slices"* such that within the slice this development can be described by some kind of *"simple"* function. The next step in this development was the study of algorithms for the rather large set of operations defined in 24. Whereas 17 just provides a brief look into this issue by presenting two example algorithms at the end, in 29 the authors present a comprehensive, systematic study of algorithms for a subset of the operations introduced in 24. Whereas some algorithms are relatively straightforward and simple, there are still a considerable number of quite involved ones. In all cases the authors analyze the complexity of the algorithms. In 29 the data structures from 17 are also refined and extended by auxiliary fields to speed up computations. This paper also offers a blueprint for implementing such a *"moving objects"* extension package for suitable extensible database architectures. More specifically, the details and the current status of a prototypical implementation of the data structures and algorithms described are presented. The final outcome of this work has been recently demonstrated in 2. The prototype is being developed as an algebra module for the experimental database system SECONDO 12.

As an extension to the abstract model in 24, the concept of *spatio-temporal predicates* is introduced in 16. The goal is to investigate temporal changes of topological relationships induced by temporal changes of spatial objects. Further work on modeling includes 52 where the authors focus on moving point objects and the inclusion of concepts of differential geometry (speed, acceleration) in a calculus based query language. In 6 discuss in detail non-linear representation for moving objects, while in 57 the authors consider movement in networks and some evaluation strategies.

Another model using moving objects is proposed by Wolfson and colleagues in 50, 61 and 60. The authors propose the so-called Moving Objects Spatio-Temporal (MOST) data model for databases with *dynamic attributes*, i.e. attributes that change continuously as a function of time, without being explicitly updated. This model enables the DBMS to predict the future location of a moving object by providing a *motion vector*, which consists of its location, speed and direction for a recent period of time. In the model, the answer to a query depends not only on the database contents, but also on the time at which the query is entered. As long as the predicted position based on the motion vector does not deviate from the actual position more than some threshold, no update to the database is necessary. An important issue here is to balance the cost of updates against the cost of imprecise information. The authors also offer a query language (Future Temporal Logic - FTL) based on temporal logic to formulate questions about the near future movement. The approach is restricted to moving points and does not address more complex time-varying geometries such as moving regions.

Related work in the field also includes our initial approach in designing HERMES. More specifically, in 45 we briefly described the envisioned architecture of HERMES framework, in 42 we presented the primitives of the proposed datatype-oriented model and provides a preliminary insight on the supported functionality, while in 40 we demonstrated the software developed theretofore, focusing in a specific (i.e. LBS) application domain. The current paper presents the complete system and describes all the necessary infrastructure for introducing our datatype system for moving objects. More specifically, we describe all the base, temporal and spatial types that compose the basic constructs for the definition of the moving objects datatypes, while we discuss in detail the fundamentals for extending the previous with moving objects. In addition, all the datatypes, which are the core of the data type system of

HERMES, are now formally defined and discussed in detail. The definition of the data type system is followed by a presentation of the design decisions and techniques for the physical representation of the proposed abstract data types. We further discuss the principles adhered by HERMES for designing moving objects operations and present in detail the full set of methods defined upon the proposed data types. Our design extends the data definition and manipulation language of OGC-compliant ORDBMS with spatio-temporal semantics and functionality, paying special attention on advanced spatio-temporal indexing and query processing techniques. The proposed operations are accompanied with a discussion regarding their development and fruitful examples and illustrations for depicting the supported functionality. We also include a description of the implementation details of our system taking advantage of extensibility interfaces provided by state-of-the-art ORDBMS. Furthermore, we focus on the resulted query language which is applied, as a proof-of-concept, to a case study related with vehicle traffic analysis. Finally, we present several systems and case studies that HERMES has been successfully applied and we provide a qualitative comparison of our research effort with related work.

In 23 the authors extended the SECONDO system with algorithms for efficient k-nearest neighbor search on moving object trajectories, while in 13 they introduced a benchmark that defines datasets and queries for experimental evaluations. Another recent approach is TrajStore 10, which focuses on supporting efficient spatio-temporal range queries in very large datasets.

In the following paragraphs and in order to place the contribution of this paper, we briefly present the differences of HERMES features proposed in this paper with the approach described in 24, 17 and 29, which is the most related to our work.

HERMES introduces time-varying geometries that change location or shape in discrete steps and/or continuously. Our approach for supporting both discretely and continuously changing spatio-temporal objects and which is based on the *Unit_Function* object is more generic and flexible than the tactic adopted in 17 that asserts the same functionality. Apart from linear interpolations of spatial and spatio-temporal (moving) types utilized in 17 and 29, HERMES also utilizes arc interpolations by proposing a categorization according to the quadrant the motion takes place and the motion heading. What is more, the user of HERMES is facilitated with a flexible and extensible interface for additional types of motion for moving types (e.g. splines, polynomials of degree higher than one etc.), which is provided via the *Unit_Function* object type.

In addition to *Moving_Point*, *Moving_LineString*, *Moving_Polygon*, proposed in 17, the proposed *MOD Type System* also includes types like *Moving_Circle*, *Moving_Rectangle*, *Moving_Collection* and *Moving_Object*. A rich set of object methods is introduced that expresses all the interesting spatio-temporal phenomena and processes. This set of operations is a superset of the operations introduced in 24. The operation set commenced in 24 at an abstract level, is reduced in 17 where specific finite representations and data structures are given for all the types of the abstract model, and is further reduced in 29 where a subset of the algorithms are selected to make the implementation manageable.

Of course, there are more differences between the two operations sets supplied by 24 and HERMES. For example, all topological operations introduced in 24 are combined in HERMES under a single operator, which distinguishes the different topological relationships via a *"mask"* parameter. Furthermore, HERMES introduces new operations describing the buffer, the convex hull, the centre of gravity and points on the surface of moving geometries. Additionally, particular attention has been paid to operations that facilitate the user to check the construction of moving objects and to keep such kind of spatio-temporal data in a consistent state. This leads to effective database maintenance and reliable error-handling mechanism. More importantly, as we aim to provide a powerful toolkit for analysts, HERMES includes higher level methods (e.g. operators for trajectory similarity search), upon which knowledge discovery tasks can be easily performed.

The *Moving_Collection* object supports not only a homogeneous collection of moving types but also a heterogeneous collection of them. In 24, heterogeneous collections are not supported and a single moving type corresponds to a homogeneous *Moving_Collection* of the proposed *MOD Type System*. The *Moving_Object* can substitute any of the other moving types, as well as moving geometries that result as operations on other moving geometries and moreover, it can model time-varying objects like the time-changing perimeter of a moving region. In 24 such degenerated moving types (moving reals, strings and booleans) are constructed as separate objects, which leads to a proliferation of object types that mainly are not spatio-temporal, which makes more difficult and unnatural the utilization of such data types by end users.

Generally speaking, the proposed *MOD Type System* is richer and more flexible than the one presented in 24. For example, it supports moving linestrings that intersect themselves during their development, while such a behavior is not allowed in 24 due to the fact that the spatial model does not accept self-intersecting linestrings. This is a very simple example of the importance that HERMES is OGC-compliant.

9 Conclusions and Future Work

In this paper, a formal framework and its implementation for managing and analyzing moving objects, called HERMES, was introduced. HERMES is a system extension that provides spatio-temporal functionality to ORDBMS offering OGC-compliant spatial extensions and supports modeling and querying of moving objects changing location either in discrete steps or continuously. A collection of data types and their corresponding operations are defined, implemented, and demonstrated through a vehicle traffic analysis application developed in Oracle. This application demonstrates that embedding the functionality offered by HERMES in ORDBMS data manipulation language provides a flexible, expressive and easy to use query language for moving object databases.

Another contribution of this work is that it prescribes straightforward future research directions. First of all, due to the fact that our study concerns only two-dimensional spatial objects as well as the change and motion of such geometries in the 2D Cartesian plane, there is need to investigate the way we could model surfaces and three-dimensional spatial objects and the time-changing variants of them. Additionally, a future direction we are planning to follow is to utilize the optimization extensibility interface of existing ORDBMS in order to enhance the performance of HERMES. Finally, we will follow and extend the benchmark introduced in 13 for a more qualitative comparison of HERMES with the approach of SECONDO.

10 Acknowledgments

Research partially supported by the FP6-14915 IST/FET Project GeoPKDD (Geographic Privacy-aware Knowledge Discovery and Delivery) (URL: www.geopkdd.eu) and by the FP7 ICT/FET Project MODAP (Mobility, Data Mining, and Privacy) funded by the European Union (URL: www.modap.org).

11 References

1 T. Abraham, J.F. Roddick. Survey of Spatio-Temporal Databases. *GeoInformatica*, 3:61-99, 1999.

2 T. Abraham, J.F. Roddick. Survey of Spatio-Temporal Databases. *GeoInformatica*, 3:61-99, 1999.

3 V.T. de Almeida, R.H. Güting, T. Behr. Querying Moving Objects in SECONDO. *Proc. 7th International Conference on Mobile Data Management* (MDM), 2006.

4 V.T. de Almeida, R.H. Güting. Indexing the Trajectories of Moving Objects in Networks. *GeoInformatica* 9(1):33-60, 2005.

5 M. Baglioni, J. A. F. de Macedo, C. Renso, R. Trasarti, M. Wachowicz. Towards Semantic Interpretation of Movement Behavior, *Proc of 12th AGILE International Conference on Geographic Information Science* (AGILE), Hannover, Germany, 2009.

6 L. Becker, H. Blunck, K. Hinrichs, J. Vahrenhold. A Framework for Representing Moving Objects. *Proc. of DEXA*, 854-863, 2004.

7 S. Boulahya. Représentation et interrogation de données spatio-temporelles : Cas d'étude sur PostgreSQL/PostGIS. Masters' Thesis, Department of Computer and Decision Engineering, Université Libre de Bruxelles, Brussels, Belgium, 2009, (in French).

8 R.G.G. Cattel, D.K. Barry (eds.). *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann Publishers, May 1997.

9 V.P. Chakka, A. Everspaugh, J. Patel. Indexing Large Trajectory Data Sets with SETI. *Proceedings of CIDR*, 2003.

10 P. Cudre-Mauroux, E. Wu, and S. Madden, TrajStore: An Adaptive Storage System for Very Large Trajectory Data Sets, *Proc. of 26th International Conference on Data Engineering* (ICDE), 2010.

11 DB2 Spatial Extender. http://www-01.ibm.com/software/data/spatial/db2spatial/ (accessed on 9 July 2010).

12 S. Dieker and R. H. Güting. Plug and Play with Query Algebras: Secondo. A Generic DBMS Development Environment. *Proc. Int'l Symp. on Database Engineering and Applications* (IDEAS), pages 380-390, September 2000.

13 C. Düntgen, T. Behr, R. H. Güting. BerlinMOD: a benchmark for moving object databases. *VLDB J.* 18(6): 1335-1368, 2009.

14 M. Egenhofer and R. Franzosa. Point-Set Topological Spatial Relations. *International Journal of Geographical Information Systems*, 5(2): 161-174, 1991.

15 M. Erwig, R.H. Güting, M. Schneider, and M. Vazirgiannis. Spatio-Temporal Data Types: An Approach to Modeling and Querying Moving Objects in Databases. *GeoInformatica*, 3(3): 265-291, 1999.

16 M. Erwig and M. Schneider. Spatio-Temporal Predicates. *IEEE Transactions on Knowledge and Data Engineering*, 14(4): 881-901, 2002.

17 L. Forlizzi, R. H. Güting, E. Nardelli, M. Schneider. A Data Model and Data Structures for Moving Objects Databases. *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, Dallas, Texas, USA, 2000.

18 E. Frentzos. Trajectory Data Management. PhD Thesis, 2008.

19 E. Frentzos, K. Gratsias, N. Pelekis, Y. Theodoridis. Algorithms for Nearest Neighbor Search on Moving Object Trajectories. *Geoinformatica* 11(2): 159-193 (2007)

20 E. Frentzos, N. Pelekis, I Ntoutsi and Y. Theodoridis. Trajectory Database Systems, In F. Giannotti and D. Pedreschi (eds), *Mobility, Data Mining and Privacy*. Springer, 2008.

21 GeoPKDD (Geographic Privacy-aware Knowledge Discovery and Delivery) FP6-14915 IST/FET Project, funded by the European Commission. URL: www.geopkdd.eu.

22 A. Gkoulalas-Divanis and V. S. Verykios. A Privacy Aware Trajectory Tracking Query Engine. *ACM SIGKDD Explorations*, 10(1): 40-49, July 2008.

23 R. H. Güting, T. Behr and J. Xu. Efficient k-nearest neighbor search on moving object trajectories. *VLDB J.*, 2010.

24 R.H. Güting, M. H. Bohlen, M. Erwig, C. S. Jensen, N. A. Lorentzos, M. Schneider, and M. Vazirgiannis. A Foundation for Representing and Querying Moving Objects. *ACM Transactions on Database Systems*, 25(1): 1-42, 2000.

25 Güting, R.H. An Introduction to Spatial Database Systems. *VLDB Journal*, 4: 357-399, 1994.

26 G. Hjaltason, H. Samet, Distance Browsing in Spatial Databases, *ACM TODS*, 24(2): 265-318, 1999.

27 I. Kakoudakis. The TAU Temporal Object Model. MPhil Thesis, UMIST, Department of Computation, 1996.

28 M. Koubarakis, T. Sellis et al. (eds.). *Spatio-temporal Databases: The Chorochronos Approach.* Springer, 2003.

29 J. A. C. Lema, L. Forlizzi, R. H. Güting, E. Nardelli, M. Schneider. Algorithms for Moving Objects Databases. *The Computer Journal* 46(6): 680-712, 2003.

30 L. Leonardi, G. Marketos, E. Frentzos, N. Giatrakos, S. Orlando, N. Pelekis, A. Raffaetà, A. Roncato, C. Silvestri, Y. Theodoridis. T-Warehouse: Visual OLAP Analysis on Trajectory Data. Proc. of the 26th IEEE International Conference on Data Engineering (ICDE'10), Long Beach, California, 2010.

31 G. Marketos, E. Frentzos, I. Ntoutsi, N. Pelekis, A. Raffaeta and Y. Theodoridis. "Building Real-World Trajectory Warehouses". *Proc. 7th International ACM SIGMOD Workshop on Data Engineering for Wireless and Mobile Access* (MobiDE), Vancouver, Canada, 2008.

32 MySQL Spatial Extension. http://dev.mysql.com/doc/refman/5.1/en/opengis-geometry-model.html (accessed on 9 July 2010).

33 Y. Ni, C. Ravishankar. Indexing Spatio-temporal Trajectories with Efficient Polynomial Approximations, *IEEE TKDE*, 19(5): 663-678, 2007.

34 Oracle Corp. *Oracle®* Spatial User's Guide and Reference. http://www.oracle.com/technology/products/spatial/spatial_doc_index.html (accessed on 9 July 2010).

35 Open Geospatial Consortium, Inc.® (OGC). http://www.opengeospatial.org/ (accessed on 9 July 2010).

36 R. Ortale, E. Ritacco, N. Pelekis, R. Trasarti, G. Costa, F. Giannotti, G. Manco, C. Renso, and Y. Theodoridis. The DAEDALUS Framework: Progressive Querying and Mining of Movement Data. *Proc. 16th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems* (ACMGIS), Irvine, CA, USA, 2008.

37 D. Peuquet. Making Space for Time: Issues in Spase-Time Data Representation. *GeoInformatica*, 5: 11-32, 2001.

38 N. Pelekis. STAU: A spatio-temporal extension to ORACLE DBMS. PhD Thesis, UMIST, Department of Computation, 2002.

39 N. Pelekis, G. Andrienko, N. Andrienko, I. Kopanakis, G. Marketos and Y. Theodoridis. Exploring Movement Data via Similarity-based Analysis, *Journal of Intelligent Information Systems*, submitted.

40 N. Pelekis, E. Frentzos, N. Giatrakos, Y. Theodoridis: HERMES: Aggregative LBS via a Trajectory DB Engine. *Proc. ACM SIGMOD Conference*, Vancouver, Canada, 2008.

41 N. Pelekis, I. Kopanakis, I. Ntoutsi, G. Marketos, G. Andrienko and Y. Theodoridis. Similarity Search in Trajectory Databases. *Proc. of the 14th IEEE International Symposium on Temporal Representation and Reasoning* (TIME 2007), Alicante, Spain, 2007.

42 N. Pelekis, Y. Theodoridis. Boosting Location-Based Services with a Moving Object Database Engine. *Proc. 5th Int'l ACM Workshop on Data Engineering for Wireless and Mobile Access* (MobiDE), 2006.

43 N. Pelekis, and Y. Theodoridis. An Oracle data cartridge for moving objects. Information Systems Laboratory, Department of Informatics, University of Piraeus, UNIPI-ISL-TR-2010-01. July 2010. http://isl.cs.unipi.gr/publications.html.

44 N. Pelekis, B. Theodoulidis, I. Kopanakis, Y. Theodoridis. Literature Review of Spatio-Temporal Database Models. *Knowledge Engineering Review*, 19(3), 235-274, June 2004.

45 N. Pelekis, Y. Theodoridis, S. Vosinakis, T. Panayiotopoulos. Hermes – A Framework for Location-Based Data Management. *Proc. 10th Int'l Conference on Extending Database Technology* (EDBT), Munich, Germany, 2006.

46 D. Pfoser, C. S. Jensen, Y. Theodoridis. Novel Approaches to the Indexing of Moving Object Trajectories. *Proceedings of VLDB*, 2000.

47 PostGIS. http://postgis.refractions.net/ (accessed on 9 July 2010).

48 A. Raffaetà, L. Leonardi, G. Marketos, G. Andrienko, N. Andrienko, E. Frentzos, N. Giatrakos, S. Orlando, N. Pelekis, A. Roncato, C. Silvestri. Visual Mobility Analysis using T-Warehouse. *International Journal of Data Warehousing & Mining*, to appear.

49 A. Renolen. Temporal Maps and Temporal Geographical Information Systems (Review of Research). Department of Surveying and Mapping, The Norwegian Institute of Technology, February 1997.

50 P. Sistla, O. Wolfson, S. Chamberlain, S.Dao. Modeling and Querying Moving Objects. *Proc. 13th Int'l Conf. on Data Engineering* (ICDE), Birmingham, UK, 1997.

51 SQL Server Spatial Data. http://www.microsoft.com/sqlserver/2008/en/us/spatial-data.aspx (accessed on 9 July 2010).

52 J. Su, H. Xu and O. Ibarra. Moving Objects: Logical Relationships and Queries. *Proc. 7th Int'l Symp. on Spatial and Temporal Databases* (SSTD), Redondo Beach, California, USA, 2001.

53 Tansel, A.U., J. Clifford, S. Gadia, S. Jajodia, A. Segev and R. Snodgrass. *Temporal Databases: Theory, Design and Implementation*. Benjamin/Cummings Publishing Company, 1993.

54 Y. Theodoridis. Ten Benchmark Database Queries for Location-based Services, *The Computer Journal*, 46 (6): 713-725, 2003.

55 Y. Theodoridis, M. Vazirgiannis, T. Sellis. Spatio-temporal Indexing for Large Multimedia Applications. *Proceedings of ICMCS*, 1996.

56 I. Theodoulidis and P. Loucopoulos. The Time Dimension in Conceptual Modeling. *Information Systems,* 16 (3): 273-300, 1991.

57 M. Vazirgiannis and O. Wolfson. A Spatiotemporal Model and Language for Moving Objects on Road Networks. *Proc. 7th Int'l Symp. on Spatial and Temporal Databases* (SSTD), Redondo Beach, California, USA, 2001.

58 G. Wiederhold, S. Jajodia and W. Litwin. Dealing with Granularity of Time in Temporal Databases. *Proc. 3rd Nordic Conf. on Advanced Information Systems Engineering*, Trondheim, Norway, May 1991.

59 M.F. Worboys. Unifying the Spatial and Temporal Components of Geographical Information. *Proc. Int'l Symp. on Spatial Data Handling* (SDH), 1994.

60 O. Wolfson, A. P. Sistla, S. Chamberlain and Y. Yesha. Updating and Querying Databases that Track Mobile Units. *Distributed and Parallel Databases*, 7 (3): 257-387, 1999.

61 O. Wolfson, B. Xu, S. Chamberlain, L. Jiang. Moving Objects Databases: Issues and Solutions. *Proc. 10th Int'l Conf. on Scientific and Statistical Database Management* (SSDBM), Capri, Italy, 1998.

62 P. Zhang. The Spatial Movement Extensions of STAU. MPhil Thesis, UMIST, Department of Computation, 2003.

63 Esteban Zimányi, personal communication, 2010.

1 Appendix A – the Temporal Data Model Adopted by HERMES-MDC

TAU Model augment the four temporal data types found in ODMG object model, *Date*, *Time*, *Timestamp* and *Interval*, with three new temporal data types: *Timepoint*, *Period* and *Temporal Element*. In the following sections, the semantics and the formal definitions of all the temporal literal types supported by *TAU Time Model* are given, as well as the formal specifications of the atomic literal types that are utilized in the definition of the temporal types, in terms of set theory.

Operations related with each temporal type fall into three categories namely, constructors, access methods and utilities.

Constructors are operations that create instances of a type and initialize their state.

Access Methods are operations used to retrieve values of built-in properties.

Utilities are operations that return general information regarding the instance.

1.1 Atomic Literal Types

The set of *Atomic Literal Types ALT* is defined as

$ALT = \Pi\, boolean T \cup \Pi\, char T \cup \Pi\, short T \cup \Pi\, ushort T \cup \Pi\, long T \cup \Pi\, ulong T \cup \Pi\, float T \cup \Pi\, double T \cup \Pi\, octet T \cup \Pi\, string T \cup \Pi\, enum T$, *where*

$\Pi\, boolean T = \{true, false\}$ $\quad$ $\Pi\, char T = \{x \mid x \in ASCII\}$

$\Pi\, short T = \{x: \wedge \mid s\_lb \leq x \leq s\_ub\}$ $\quad$ $\Pi\, ushort T = \{x: \subseteq \mid x \leq us\_ub\}$

$\Pi\, long = \{x: \wedge \mid l\_lb \leq x \leq l\_ub\}$ $\quad$ $\Pi\, ulong T = \{x: \subseteq \mid x \leq ul\_ub\}$

$\Pi\, float T = \{x: \nabla \mid f\_lb \leq x \leq f\_ub\}$ $\quad$ $\Pi\, double T = \{x: \nabla \mid d\_lb \leq x \leq d\_ub\}$

$\Pi\, bit T = \{0, 1\}$ $\quad$ $\Pi\, octet T = bit^8$

$\Pi\, string T = Char^n, n \in \subseteq^*$ $\quad$ $\Pi\, enum T = \{(s, n) \mid s \in string, n \in \text{any numerical type}\}$

s_lb, l_lb, f_lb, d_lb are the lower bounds and *s_ub, us_ub, l_ub, ul_ub, f_ub, d_ub* are the upper bounds of the corresponding numerical types. The representation, precision, ranges and operations of numerical types are implementation platform specific.

Further more, in order to formalize the definition of the temporal literal types we should first define the time divisions in the *Gregorian* calendar, which are,

$\Pi\, GrYear T = \{y: long \mid lb \leq y \leq ub \wedge y \neq 0\}$ $\quad$ $\Pi\, GrMonth T = \{m: ushort \mid 1 \leq m \leq 12\}$

$\Pi\,GrDay\,T = \{d\text{: ushort} \mid 1 \le d \le 31\}$ $\Pi\,GrHour\,T = \{h\text{: ushort} \mid 0 \le h \le 23\}$

$\Pi\,GrMinute\,T = \{m\text{: ushort} \mid 0 \le m \le 59\}$ $\Pi\,GrSecond\,T = \{s\text{: double} \mid 0 \le s \le 59\}$

as well as the set *granularity* that contains elements that represent time accuracy:

$$\Pi\,granularity\,T = \{YEAR, MONTH, DAY, HOUR, MINUTE, SECOND\}$$

As such the set of Temporal Literal Types *TLT* is defined as

$$TLT = \Pi\,date\,T \cup \Pi\,time\,T \cup \Pi\,timestamp\,T \cup \Pi\,timepoint\langle g\rangle T \cup \Pi\,interval\,T \cup \Pi\,period\langle g\rangle T \cup \Pi\,temporalElement\langle g\rangle T.$$

1.2 ODMG Temporal Data Types

The *ODMG* Standard [CB97] defines the following temporal data types:

Date: Instances of the *Date* type represent unique points in time. It supports the fields YEAR, MONTH and DAY.

$$date =_d \langle\langle year\text{: } GrYear,\ month\text{: } GrMonth,\ day\text{: } GrDay\rangle\rangle$$

Time: The *Time* data type supports the fields HOUR, MINUTE and SECOND. It either represents a unique point in time (for which the date is implicit) or it represents a recurring point of time. It is possible to specify a precision, i.e. the number of decimal places of accuracy to which the SECOND field will be kept. The default precision is zero (whole seconds only). The maximum precision is implementation defined (at least 6). The *Time* data type has a WITH TIME ZONE option. If the option is not specified the values of the data type are assumed to be always in the current default time zone of the user session. If the option is specified then the values of the data type include the TIMEZONE_HOUR and TIMEZONE_MINUTE fields, which specify the offset of the time zone of the rest of the value from Universal Coordinated Time.

$$time =_d \langle\langle hour\text{: } GrHour,\ minute\text{: } GrMinute,\ second\text{: } GrSecond\rangle\rangle$$

Timestamp: The *Timestamp* data type supports the fields YEAR, MONTH, DAY, HOUR, MINUTE and SECOND. It represents unique points in time. With *Timestamp* data type it is possible to specify a precision and WITH TIME ZONE option (see *Time* data type).

$$timestamp =_d date \parallel time$$

Interval: The *Interval* data type is used to represent an unanchored duration of time. Every interval data type consists of a contiguous subset of the fields: DAY, HOUR, MINUTE and SECOND.

$$interval =_d \langle\!\langle day: long,\ hour: GrHour,\ minute: GrMinute,\ second: GrSecond \rangle\!\rangle$$

1.3 Advanced Temporal Data Types

We augment the four temporal literal data types found in *ODMG* object model [CB97] with three new temporal object data types presented below:

Timepoint: *TAU Model* extends the *Timestamp* data type to include granularity. The new data type is a subtype of the *Timestamp* data type. It inherits all the properties and the operations that are defined for the *Timestamp* data type. It refines all the operations, which had as argument *Timestamp* to *Timepoint*.

$$timepoint\langle g\rangle =_d tp\langle g\rangle \cup STV \text{ where}$$

$$tp\langle year\rangle =_d \langle\!\langle year: GrYear\rangle\!\rangle,\ tp\langle month\rangle =_d tp\langle year\rangle \parallel \langle\!\langle month: GrMonth\rangle\!\rangle, \ldots,$$

$$tp\langle second\rangle =_d tp\langle minute\rangle \parallel \langle\!\langle second{:}GrSecond\rangle\!\rangle \text{ and } STV =_d \{beginning,\ forever,\ now\}$$

Beginning and *forever* are defined to be members of *timepoint* such as

$$\forall t \in timepoint\langle g\rangle \cdot beginning \le t \le forever$$

Period: The *Period* data type is used to represent an anchored duration of time, that is, duration of time with starting and ending points. A period has an associated granularity. The period is the composition of two timepoints with the constraint that the timepoint that starts the period equals or precedes the timepoint that terminates it. Without loss of generality, it is assumed that both timepoints have the same granularity.

$$period\langle g\rangle =_d \{\langle\!\langle start{:}Timepoint\langle g\rangle,\ end{:}Timepoint\langle g\rangle\rangle\!\rangle \mid start \le end\},\ g \in granularity$$

There are four categories of periods depending on whether they include their starting and/or their ending timepoints or not: [T_1, T_2] (closed-closed), [T_1, T_2) (closed-open), (T_1, T_2] (open-closed), and (T_1, T_2) (open-open). Without loss of generality, *TAU Model* supports only closed-open periods, with which it is possible to model any other category. For example, the period [T_1, T_2] is equivalent to the period [T_1, T_2+1 "granule"). The meaning of "1 granule" depends on the granularity of the period. For instance, if the granularity is day then the period [T_1, T_2] is equivalent to the period [T_1, T_2+1*DAY).

Temporal Element: The *Temporal Element* data type is used to represent a finite union of disjoint periods. Temporal elements are closed under the set theoretic operations of union, intersection and complementation.

$$temporalElement\langle g\rangle =_d \{te: set\langle period\langle g\rangle\rangle \mid \forall i, j \cdot i \ne j \Rightarrow te_i \cap te_j = \emptyset\}$$

2 Appendix B – the Spatial Data Model Adopted by HERMES-MDC

2.1 Description of Spatial Data Types

The spatial data model adopted by Oracle10g is a hierarchical structure consisting of elements, geometries, and layers, which correspond to representations of spatial data. Layers are composed of geometries, which in turn are made up of elements. For example, a point might represent a building location, a line string might represent a road or flight path, and a polygon might represent a state, city, or zoning district.

Element: An *element* is the basic building block of a geometry. The supported spatial element types in the object-relational model are points, simple, arc (circular arcs) and compound linestrings and polygons, as well as circles and rectangles as sub-cases of polygon geometries. Figure 69 illustrates the supported geometric primitive types. *Point* is the simplest geometry, which consists of one coordinate. Each coordinate in an element is stored as a *(x, y)* pair often corresponding to longitude and latitude. *LineStrings* are composed of one or more pairs of points that define line segments. *Polygons* are composed of connected linestrings that form a closed ring and the interior of the polygon is implied.

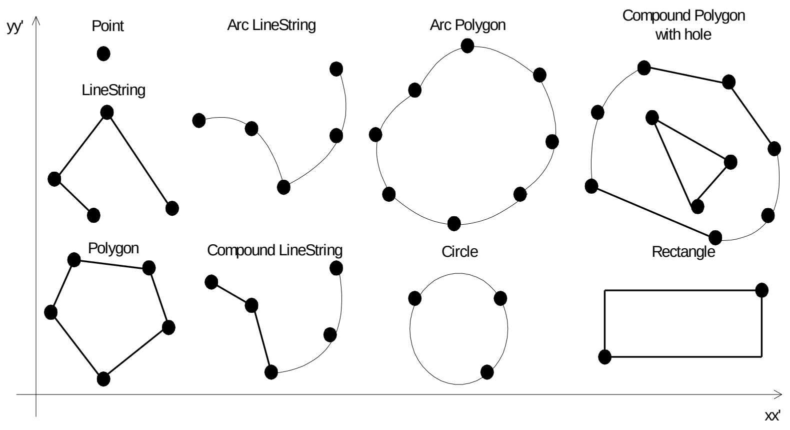


Figure 69 Primitive Geometry Types Supported by Oracle10g

As it is obvious in Figure 69, *arc* and *compound* types generalize the LineString and Polygon types, to represent geometries with arbitrary interpolations but the same topology. Self-crossing polygons are not supported although self-crossing linestrings are (see Figure 70). If a linestring crosses itself, it does not become a polygon. A self-crossing linestring does not have any implied interior. The exterior ring and the interior ring of a polygon with holes are considered as two distinct elements that together make up a complex polygon.

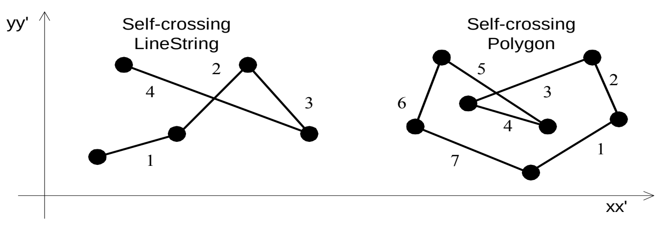


Figure 70 Self-crossing LineString & Polygons

Geometry: A *geometry* (or *geometry object*) is the representation of a spatial feature, modelled as an ordered set of primitive elements. A geometry can consist of a single element, which is an instance of one of the supported primitive types, a homogeneous or heterogeneous collection of elements. A multipolygon, such as one used to represent a set of islands, is a homogeneous collection. A heterogeneous collection is one in which the elements are of different types.

Layer: A *layer* is a heterogeneous collection of geometries having the same attribute set. For example, one layer in a Geographical Information System (GIS) might include topographical features, while another might describe population density, and a third describes the network of roads and bridges in the area (lines and points). Each layer's geometries are stored in the database in standard tables.

2.2 Object Orientation and Geometry Hierarchy

Until now we have clarified all the geometric types that our model supports. In Figure 71, one can see the geometry interface hierarchy adopted by the proposed spatial model and developed as an extension of the Open GIS geometry model [OGIS]. In the proposed model, a geometry object can be either a simple geometry or a geometry collection. A simple geometry is defined as previously, while a geometry collection is a heterogeneous collection of points, linestrings and polygons. More specific types like multipoint, multicurve and multisurface are introduced to represent homogeneous collections of points, linestrings and polygons respectively for easier geospatial analysis.

In Figure 71, the white blocks are helper interfaces i.e., Segment, LinearSegment, CircularArc, Spline and other potential interpolations of a Segment. The dark-shaded blocks are the Open GIS types i.e., Geometry, Point, Curve, Surface, LineString, Polygon, GeometryCollection, MultiPoint, MultiCurve, MultiSurface, MultiLineString and MultiPolygon. The light-shaded blocks are the extended types i.e., CurveString, CurvePolygon, MultiCurveString and MultiCurvePolygon.

The Curve, Surface, Multicurve and Multisurface are intermediate abstract types that make this model more flexible for expansion. A *curve* is an arbitrary topologically one-dimensional geometry object. A *surface* is an arbitrary topologically two-dimensional geometry object that may or may not be plane. A *multicurve* and a *multisurface* represent collections of curves and surfaces, respectively.

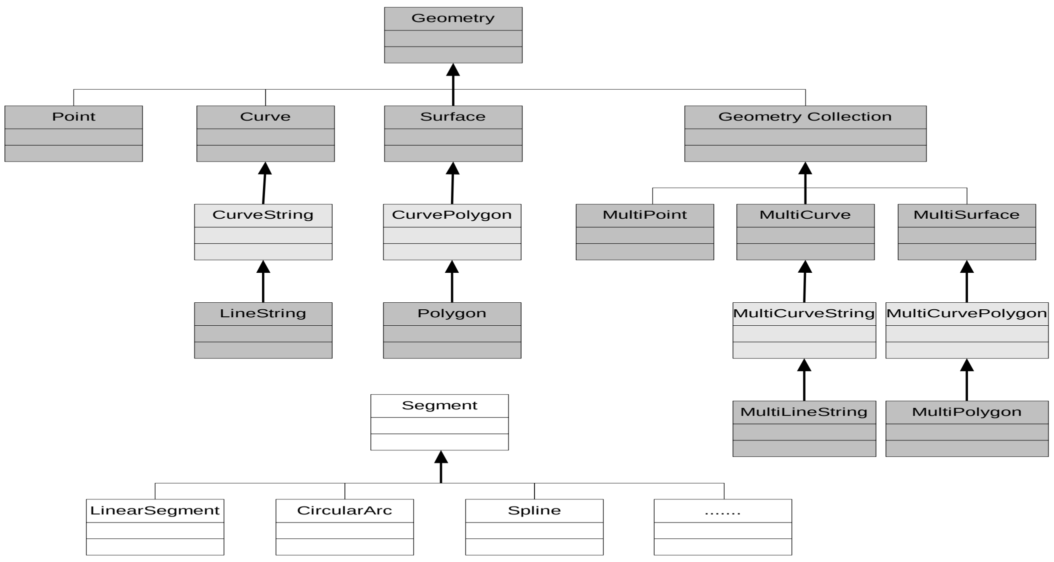


Figure 71 Geometry Interface Hierarchy

The proposed geometry interface hierarchy is fully compatible with the Open GIS model because the existence of the extended types does not affect the inheritance relationships when developers implement linear interfaces only. In fact, the new CurveString and CurvePolygon interfaces generalize the LineString and Polygon interfaces respectively, to represent geometries with arbitrary interpolations but the same topology as traditional Open GIS geometries.

2.3 Structures for Spatial Data Types

In the spatial object-relational model, the geometric description of a spatial object is stored in a single row, in a single column of object type *SDO_GEOMETRY* (defined under the *MDSYS* Oracle user) in a user-defined table. Any table that has a column of type *SDO_GEOMETRY* must have another column, or set of columns, that define a unique primary key for that table. This object type corresponds to the most general type defined in the interface hierarchy of Figure 71. Each subtype is declared and stored in a database table as an *SDO_GEOMETRY* object and the knowledge of which sub-type is or what its special characteristics are, are embodied in the structure of this generic object. Oracle Spatial defines *SDO_GEOMETRY* object type as:

CREATE TYPE SDO_GEOMETRY AS OBJECT (

SDO_GTYPE NUMBER,

SDO_SRID NUMBER,

SDO_POINT SDO_POINT_TYPE,

SDO_ELEM_INFO MDSYS.SDO_ELEM_INFO_ARRAY,

SDO_ORDINATES MDSYS.SDO_ORDINATE_ARRAY);

The sections that follow describe the semantics of each *SDO_GEOMETRY* attribute, and some usage considerations.

SDO_GTYPE indicates the type of the geometry. Valid geometry types correspond to those specified in the *geometry interface hierarchy*. The following table shows the valid *SDO_GTYPE* values and the correspondence between the names and semantics.

Value	*Geometry Type*	*Description*
*d*000	UNKNOWN_GEOMETRY	Spatial ignores this geometry
*d*001	POINT	Geometry contains one point
*d*002	LINESTRING	Geometry contains one line string
*d*003	POLYGON	Geometry contains one polygon with or without holes
*d*004	COLLECTION	Geometry is a heterogeneous collection of elements
*d*005	MULTIPOINT	Geometry has multiple points
*d*006	MULTILINESTRING	Geometry has multiple linestrings
*d*007	MULTIPOLYGON	Geometry has multiple, disjoint polygons (more than one exterior boundary)

Table 1 Valid SDO_GTYPE Values

For a polygon with holes, the user should enter the exterior boundary first, followed by any interior boundaries. In a multi-polygon all polygons in the collection must be disjoint. The *d* in the *Value* column of the previous table is the number of dimensions: 2, 3, or 4. For example, a value of 2003 indicates a 2-dimensional polygon. For the time only 2-dimensional geometries are supported. The number of dimensions reflects the number of coordinates used to represent each vertex (for example, *(x,y)* for 2-dimensional objects or *(x,y,z)* for 3-dimensional objects). Points and lines are considered to be 2-dimensional objects. In any given *layer* (column), all geometries must have the same number of dimensions. For example, we cannot mix 2-dimensional and 3-dimensional data in the same layer.

SDO_SRID is intended to be a foreign key in a spatial reference system definition table, in order to integrate support into Oracle10g for storing and manipulating *SDO_GEOMETRY* objects in a variety of coordinate systems.

SDO_POINT is defined using an object type with attributes *x*, *y* and *z* of type *NUMBER*. If the *SDO_ELEM_INFO* and *SDO_ORDINATES* arrays are both null, and the *SDO_POINT* attribute is non-null, then the *x* and *y* values are considered to be the coordinates for a point geometry. Otherwise the *SDO_POINT* attribute is ignored.

SDO_ELEM_INFO is defined using a varying length array of numbers. This attribute helps to interpret the ordinates stored in the *SDO_ORDINATES* attribute (see section 3.2.1.5). Each triplet set of numbers is interpreted as follows:

SDO_STARTING_OFFSET indicates the offset within the *SDO_ORDINATES* array where the first ordinate for this element is stored.

SDO_ETYPE indicates the type of the element. Valid values are 0 through 5, as well as the following: 1003 and 2003 (variants of 3), and 1005 and 2005 (variants of 5). *SDO_ETYPE* values 1, 2, and 3 concern *simple elements*. They are defined by a single triplet entry in the *SDO_ELEM_INFO* array. Moreover, the following are considered variants of type 3, with the first digit indicating *exterior* (1) or *interior* (2):

1003: exterior polygon ring (must be specified in counter-clockwise order)

2003: interior polygon ring (must be specified in clockwise order)

SDO_ETYPE values 4 and 5 concern *compound elements*. They contain at least one header triplet with a series of triplet values that belong to the compound element. The elements of a compound element are contiguous. The last point of a subelement in a compound element is the first point of the next subelement. The point is not repeated.

SDO_ETYPE	*SDO_INTERPRETATION*	*Meaning*
0	0	Unsupported element type. Ignored by the Spatial functions and procedures.
1	1	Point type.
1	*n* >1	Point cluster with *n* points.
2	1	Line string whose vertices are connected by

		straight-line segments.
2	2	Line string made up of a connected sequence of circular arcs.
3	1	Simple polygon whose vertices are connected by straight-line segments.
3	2	Polygon made up of a connected sequence of circular arcs that closes on itself.
3	3	Rectangle type. A bounding rectangle such that only two points, the lower-left and the upper-right, are required to describe it.
3	4	Circle type. Described by three points, all on the circumference of the circle.
4	$n > 1$	Line string with some vertices connected by straight-line segments and some by circular arcs.
5	$n > 1$	Compound polygon with some vertices connected by straight-line segments and some by circular arcs.

Table 2 Values and Semantics of SDO_ELEM_INFO

SDO_INTERPRETATION can mean one of two things, depending on whether or not *SDO_ETYPE* is a compound element. If the *SDO_ETYPE* is a compound element (4 or 5), this field specifies how many subsequent triplet values are parts of the element. If the *SDO_ETYPE* is not a compound element (1, 2, or 3), the interpretation attribute determines how the sequence of ordinates for this element is interpreted. For example, a line string or polygon boundary may be made up of a sequence of connected straight-line segments or circular arcs. If a geometry consists of more than one element, then the last ordinate for an element is always one less than the starting offset for the next element. The last element in the geometry is described by the ordinates from its starting offset to the end of the *SDO_ORDINATES* varying length array. The semantics of each *SDO_ETYPE* element and the relationship between the *SDO_ELEM_INFO* and *SDO_ORDINATES* varying length arrays for each of these *SDO_ETYPE* elements are given in the following table.

Each circular arc in the geometries is described using three coordinates: the arc's starting point, any point on the arc, and the arc's end point. The coordinates for a point designating the end of one arc and the start of the next arc are not repeated. For example, five coordinates are used to describe a linestring made up of two connected circular arcs. Points 1, 2, and 3 define the first arc, and points 3, 4, and 5 define the second arc, where point 3 is only stored once.

For polygon geometries the user needs to specify a point for each vertex, and the last point specified must be identical to the first (to "close" the polygon). For example, for a 4-sided polygon, we need to specify 5 points, with point 5 the same as point 1.

For compound elements the value, *n*, in the interpretation column specifies the number of contiguous subelements that make up the geometry. The next *n* triplets in the *SDO_ELEM_INFO* array describe each of these subelements. The subelements can only be of *SDO_ETYPE* 2. The end point of a subelement is the start point of the next subelement, and it must not be repeated.

SDO_ORDINATES is defined using a varying length array of *NUMBER* type that stores the coordinate values that make up the boundary of a spatial object. This array must always be used in conjunction with the *SDO_ELEM_INFO* varying length array. The values in the array are ordered by dimension. For example, a polygon whose boundary has four 2-dimensional points is stored as $\{x_1,y_1, x_2,y_2, x_3,y_3, x_4,y_4, x_1,y_1\}$.

The values in the *SDO_ORDINATES* array must all be valid and non-null. There are no special values used to delimit elements in a multi-element geometry. The start and end points for the sequence describing a specific element are determined by the *STARTING_OFFSET* values for that element and the next element in the *SDO_ELEM_INFO* array as explained previously.

Usage considerations: The *Spatial Data Cartridge* user should use the *SDO_GTYPE* values as shown in table 1. The *Spatial* component enforces some geometry consistency constraints and more specifically, the following:

For *SDO_GTYPE* values *d*001 and *d*005, any subelement not of *SDO_ETYPE* 1 is ignored.

For *SDO_GTYPE* values *d*002 and *d*006, any subelement not of *SDO_ETYPE* 2 or 4 is ignored.

For *SDO_GTYPE* values *d*003 and *d*007, any subelement not of *SDO_ETYPE* 3 or 5 is ignored. (This includes *SDO_ETYPE* variants 1003, 2003, 1005, and 2005).

2.4 Formal Definition of Pure Spatial Types

This section describes formally in terms of set theory the unique object type that represents all the different geometric constructs adopted in our spatial model.

$$ST = \Pi\, SDO\_GEOMETRY\, T$$

Before introducing the constraints and the interdependencies between the element types that compose the *SDO_GEOMETRY* object, let us first define these components, in order to associate conceptually their formal description with their linguistic one in Section 12.3. The reader should have in mind that even though the spatial model supports geometries of higher dimension than two, we are interested in 2-Dimensional spatial objects only.

$\Pi SDO\_GTYPE\_TYPE\, T = \{gt: ushort \mid 2000 \le gt \le 2007\}$

$\Pi SDO\_POINT\_TYPE\, T = \{(x, y) \mid x, y \in double\}$

$\Pi SDO\_ELEM\_INFO\_ARRAY\, T = \{set\langle (so, et, ip) \rangle \mid so \in SDO\_STARTING\_OFFSET \wedge et \in SDO\_ETYPE \wedge ip \in SDO\_INTERPRETATION \wedge \forall i, j \cdot i < j \Rightarrow so_i < so_j \}$

where

$\Pi SDO\_STARTING\_OFFSET\, T = \{so: ulong \mid 1 \le so < LAST(ORD)\}$

$\Pi\, SDO\_ETYPE\, T = \{et: ushort \mid 0 \le et \le 5 \vee et \in \{1003, 2003, 1005, 2005\}\}$

$\Pi\, SDO\_INTERPRETATION\, T = \{ip: ushort\}$

$\Pi SDO\_ORDINATES\, T = \{set\langle x \rangle \mid x \in double, |set\langle x \rangle| = 2k, k \ge 0, k \in ulong \}$

As such *SDO_GEOMETRY* is defined as follows:

$SDO\_GEOMETRY =_d \{ \langle\langle\ SDO\_GTYPE: SDO\_GTYPE\_TYPE,$

$SDO\_SRID: ushort,$

$SDO\_POINT: SDO\_POINT\_TYPE,$

$SDO\_ELEM\_INFO: SDO\_ELEM\_INFO\_ARRAY,$

$SDO\_ORDINATES: SDO\_ORDINATES\_ARRAY \rangle\rangle \mid$

/ Due to space limitations we use the following abbreviations:*

SDO_GTYPE_TYPE:=GTYPE

SDO_POINT_TYPE:=PTYPE

SDO_ELEM_INFO_ARRAY:=ELEM

*SDO_ORDINATES_ARRAY:=ORD */*

$(\forall gt \in GTYPE \cdot gt \in \{2001, 2005\} \subset GTYPE \Leftrightarrow \forall (so, et, ip) \in ELEM: et=1) \wedge$

$(\forall gt \in GTYPE \cdot gt \in \{2002, 2006\} \subset GTYPE \Leftrightarrow \forall (so, et, ip) \in ELEM: et=2 \vee et=4) \wedge$

$(\forall gt \in GTYPE \cdot gt \in \{2003, 2007\} \subset GTYPE \Leftrightarrow \forall (so, et, ip) \in ELEM: et=3 \vee et=5) \wedge$

$(\forall gt \in GTYPE \cdot gt=2001 \Rightarrow (PTYPE \neq \emptyset \wedge (ELEM = \emptyset \wedge ORD = \emptyset)) \vee (PTYPE = \emptyset \wedge (ELEM = (1, 1, 1) \wedge |ORD|=2))) \wedge$

$(\forall gt \in GTYPE \cdot gt=2002 \Rightarrow |ORD| \geq 4 \wedge (ORD_1 \neq ORD_{LAST(ORD)-1} \wedge ORD_2 \neq ORD_{LAST(ORD)}) \wedge (\forall j=2k+1, k \geq 0, k \in ulong: point\ (ORD_j, ORD_{j+1}) \neq point\ (ORD_{j+2}, ORD_{j+3})) \wedge (\forall j=2k+1, k \geq 0, k \in ulong: collinear\ (segment\ (point\ (ORD_j, ORD_{j+1}), point\ (ORD_{j+2}, ORD_{j+3})), segment\ (point\ (ORD_{j+2}, ORD_{j+3}), point\ (ORD_{j+4}, ORD_{j+5})) \Rightarrow \neg overlap(segment\ (point\ (ORD_j, ORD_{j+1}), point\ (ORD_{j+2}, ORD_{j+3})), segment\ (point\ (ORD_{j+2}, ORD_{j+3}), point\ (ORD_{j+4}, ORD_{j+5})))) \wedge (\forall j=2k+1, k \geq 0, k \in ulong: arcline\ ((point\ (ORD_j, ORD_{j+1}), point\ (ORD_{j+2}, ORD_{j+3}), point\ (ORD_{j+4}, ORD_{j+5})) \Rightarrow \neg collinear\ ((point\ (ORD_j, ORD_{j+1}), point\ (ORD_{j+2}, ORD_{j+3}), point\ (ORD_{j+4}, ORD_{j+5})))) \wedge$

$(\forall gt \in GTYPE \cdot gt=2003 \Rightarrow (\exists\ (so, et, ip) \in ELEM: ip=3 \Rightarrow \neg parallel\ (segment\ (point\ (ORD_{so}, ORD_{so+1}), point\ (ORD_{so+2}, ORD_{so+3})), xx') \wedge \neg parallel\ (segment\ (point\ (ORD_{so}, ORD_{so+1}), point\ (ORD_{so+2}, ORD_{so+3})), yy')) \wedge (\exists\ (so, et, ip) \in ELEM: ip=4 \Rightarrow \neg collinear\ ((point\ (ORD_{so}, ORD_{so+1}), point\ (ORD_{so+2}, ORD_{so+3}), point\ (ORD_{so+4}, ORD_{so+5}))) \wedge (\forall j, 1 \leq j \leq |ELEM| DIV3, j \in ulong \cdot (so_j, et_j, ip_j) \in ELEM: et_j=3 \vee et_j=5 \Rightarrow Polygon_j =_d \{ORDso_j, \ldots, ORDso_{j+1}\text{-}1\} \subset ORD \cdot linestringsOfPolygon_j =_d \{set\langle\ linestring: \langle ORD_k, \ldots, ORD_l \rangle\rangle \mid so_j \leq k < l \leq so_{j+1}\text{-}1 \wedge \forall l, m \in linestring: l=next(m), l \neq m \Rightarrow meet\ (l, m) \wedge \neg intersect\ (l, m) \wedge \neg touch\ (l, m) \wedge (ORDso_j = ORDso_{j+1}\text{-}2 \wedge ORDso_j+1 = ORDso_{j+1}\text{-}1) \wedge \forall\ linestring:$ (v) $rules\ applied\ to\ \langle ORD_k, \ldots, ORD_l \rangle\ instead\ to\ all\ ORD\} \cdot \forall m, 1 < m: inside(Polygon_m, Polygon_1) \wedge counter\text{-}clockwise(Polygon_1) \wedge clockwise(Polygon_m) \wedge \forall m_1, m_2\ 1 < m_1 \wedge 1 < m_2: m_1 \neq m_2 \Rightarrow disjoint(m_1, m_2))) \wedge$

$(\forall gt \in GTYPE \cdot gt=2004 \Rightarrow \forall j, 1 \leq j \leq |ELEM| DIV3, j \in ulong: (et_j=2 \vee et_j=4 \Rightarrow geometry_j(ORDso_j, \ldots, ORDso_{j+1}\text{-}1)\ follows$ (v) $rules) \wedge (et_j=3 \vee et_j=5 \Rightarrow geometry_j(ORDso_j, \ldots, ORDso_{j+1}\text{-}1)\ follows$ (vi) $rules \wedge unique(geometry_j(ORDso_j, \ldots, ORDso_{j+1}\text{-}1))) \wedge$

$(\forall gt \in GTYPE \cdot gt=2005 \Rightarrow \forall j, 1 \leq j \leq |ELEM| DIV3, j \in ulong: et_j=1 \wedge ip_j=1 \wedge so_j=2*j\text{-}1 \wedge |ORD|=2*|ELEM| DIV3 \wedge unique(point(ORDso_j, ORD\ so_{j+1}\text{-}1))) \wedge$

$(\forall gt \in GTYPE \cdot gt=2006 \Rightarrow \forall j, 1 \leq j \leq |ELEM| DIV3, j \in ulong: linestring_j(ORDso_j, \ldots, ORDso_{j+1}\text{-}1)\ follows$ (v) $rules \wedge unique(linestring_j\ (ORDso_j, \ldots, ORDso_{j+1}\text{-}1))) \wedge$

$(\forall gt \in GTYPE \cdot gt=2007 \Rightarrow \forall i, j, 1 \leq i, j \leq |ELEM| DIV3, i, j \in ulong: i \neq j \Rightarrow disjoint(polygon_i, polygon_j) \wedge polygon_j(ORDso_j, \ldots, ORDso_{j+1}\text{-}1)\ follows$ (vi) $rules \wedge unique(polygon_j(ORDso_j, \ldots, ORDso_{j+1}\text{-}1)))$

}

The constraints *(i)* to *(iii)* describe formally some usage considerations, while the rules from *(iv)* to *(x)* illustrate possible interrelations between the constituent types of the

SDO_GEOMETRY object for each one of the geometries that can be represented by this object. More specifically, the *(iv)* rule depicts the two possible ways to define a point geometry and *(v)* exemplifies that in order to construct a valid linestring the size of the ordinates array should be at least four (namely two points) and the first point must not coincide with the last point, as this is the case that differentiates a simple polygon from a linestring. What is more, each pair of sequential points must be different and for each triplet of points, if these points are intended to describe two sequential linear segments then we require that there are no such co-linear overlapping segments, otherwise if these points describe just an arc-segment then we require that these points are not co-linear.

The *(vi)* set theory proposition describes the constraints that should stand in case the *SDO_GEOMETRY* object models a polygon geometry. Firstly ensures the validity of rectangular and circle geometries, which are special cases of a polygon. This is accomplished by not permitting parallelism between the segment that is formed by the lower left and upper right point that define a rectangular and the *xx'* or the *yy'* axis; and by forbidding co-linearity between the three points needed to define a circle.

For simple polygons the model requires that the first polygon-element described in the elements-info array must be the exterior boundary that will include one or more possible disjoint hole-polygons. The points that form the exterior boundary in the ordinates array must be specified in counter-clockwise order, while points composing hole-polygons must be specified in clockwise order. Furthermore, the linestring subelements that describe complex interpolations of the boundary of a polygon must meet (the end point of a linestring is the same with the starting point of the next linestring), must not intersect in their interior (a point other than an end point), must not touch (the end point lies in the interior of the other linestring) and the starting point of the first linestring must be equal with the end point of the last linestring. Finally, each of these linestring sub-elements must fulfil the constraints imposed for linestring geometries in *(v)*.

The rules described in *(vii)* impose that, for each distinct geometry object that is integrated in a heterogeneous collection, the corresponding constraints must stand depending on the kind of geometry. For example, if the collection has a linestring, then the *(v)* constraints must stand for this linestring. Similarly, propositions from *(viii)* to *(x)* require unique representation and existence of a geometry object inside a homogeneous collection and validity of each of them as this is implied by the rules that conform to its type. An additional rule that is enforced in the case of a multi-polygon is that either the exterior boundaries of the polygons composing the collection are disjoint or the exterior boundary of a polygon is inside a hole of another polygon.

3 Appendix C – Formal Definition of Spatio-Temporal Moving Types

In order to formally define in terms of set theory the moving types introduced in this paper we follow a down-top approach, meaning that we first describe the simpler data types and subsequently we define the more complex data types. In this section we just present the formal definitions while their linguistic explanations have been given in section.

MT = Π Moving_Point Τ ∪ Π Moving_LineString Τ ∪ Π Moving_Rectangle Τ ∪ Π Moving_Circle Τ ∪ Π Moving_Polygon Τ ∪ Π Moving_Collection Τ ∪ Π Moving_Object Τ.

First of all let us define the unit moving types, which are the basic building components of the spatio-temporal data types. The simpler of the unit moving types, the *Unit_Moving_Point*, upon which, is based the definition of all the others, needs for its construction two kind of objects, namely the *D_Period_Sec* and the *Unit_Function*. The D_Period_Sec is formally described in Appendix B as the *period⟨SECOND⟩* type. The *Unit_Function* has been defined earlier, as such,

Unit_Moving_Point $=_d$ *《p: period⟨SECOND⟩,* m: Unit_Function 》

Unit_Moving_Rectangle $=_d$ *{ 《 ll: Unit_Moving_Point, ur: Unit_Moving_Point 》 | equal (ll.p, ur.p) }*

Unit_Moving_Circle $=_d$ *{ 《 f: Unit_Moving_Point, s: Unit_Moving_Point, t: Unit_Moving_Point 》 | equal (f.p, s.p, t.p) }*

Unit_Moving_Segment $=_d$ *{ 《 b: Unit_Moving_Point, e: Unit_Moving_Point, m: Unit_Moving_Point, kind:TypeOfSegment 》 | (kind=SEG ⇒ equal (b.p, e.p)) ∧ (kind =ARC ⇒ equal (b.p, e.p, m.p)) }, where Π TypeOfSegment Τ = { SEG, ARC }*

Unit_Moving_Linestring $=_d$ *{ l: set⟨Unit_Moving_Segment⟩ | ∀ i, j ∈ ulong: i≠j ⇒ equal* (l_i*.b.p,* l_j*.e.p) }*

Unit_Moving_Polygon $=_d$ *{《 l: set⟨Unit_Moving_Segment⟩, hole:boolean 》 | ∀ i, j ∈ ulong: i≠j ⇒ equal* (l_i*.b.p,* l_j*.e.p) }*

Having defined all the unit-moving types we are now ready to formalize the description of our moving types:

Moving_Point $=_d$ *{ p: set⟨Unit_Moving_Point⟩ | ∀ i, j ∈ ulong, 1≤ i, j≤ |set⟨Unit_Moving_Point⟩|: j= i+1 ⇒* p_i*.p <* p_j*.p ∧ ¬overlaps(*p_i*.p,* p_j*.p) ∧ ∀ t ∈ double: inside(t,* p_i*.p) ⇒ at_instant(p, t) ∈* $\mathit{SDO\_GEOMETRY}_{SDO\_GTYPE=2001}$ */*point geometry*/}*

Moving_Rectangle $=_d$ *{ r: set⟨Unit_Moving_Rectangle⟩ | ∀ i, j ∈ ulong, 1≤ i, j≤ |set⟨Unit_Moving_Rectangle⟩|: j= i+1 ⇒* r_i*.ll.p <* r_j*.ur.p ∧ ¬overlaps(*r_i*.ll.p,* r_j*.ur.p) ∧ ∀ t ∈ double: inside(t,* r_i*.ll.p) ⇒ at_instant(r, t) ∈* $\mathit{SDO\_GEOMETRY}_{SDO\_GTYPE=2003 \wedge SDO\_ELEM\_INFO=(1, 3, 3)}$ */*rectangle geometry*/ }*

Moving_Circle $=_d$ *{ c: set⟨Unit_Moving_Circle⟩* | $\forall$ *i, j* $\in$ *ulong,* $1\leq i, j\leq$ *|set⟨Unit_Moving_Circle⟩|: j= i+1* $\Rightarrow$ $c_i.f.p < c_j.s.p \land \neg$*overlaps*($c_i.f.p$, $c_j.s.p$) $\land$ $\forall$ *t* $\in$ *double: inside(t,* $c_i.f.p$) $\Rightarrow$ *at_instant(c, t)* $\in$ $SDO\_GEOMETRY_{SDO\_GTYPE=2003 \land SDO\_ELEM\_INFO=(1, 3, 4)}$ */*circle geometry*/ }*

Moving_LineString $=_d$ *{ line: set⟨Unit_Moving_LineString⟩* | $\forall$ *i, j* $\in$ *ulong,* $1\leq i, j\leq$ *|set⟨Unit_Moving_LineString⟩|: j= i+1* $\Rightarrow$ $line_i.l_1.b.p < line_j.l_1.e.p \land \neg$*overlaps*($line_i.l_1.b.p$, $line_j.l_1.e.p$) $\land$ $\forall$ *t* $\in$ *double: inside(t,* $line_i.l_1.b.p$) $\Rightarrow$ *at_instant(line, t)* $\in$ $SDO\_GEOMETRY_{SDO\_GTYPE=2002}$ */*linestring geometry*/ }*

Moving_Polygon $=_d$ *{ pol: set⟨Unit_Moving_Polygon⟩* | $\forall$ *i, j* $\in$ *ulong,* $1\leq i, j\leq$ *|set⟨Unit_Moving_Polygon⟩|: j= i+1* $\Rightarrow$ $pol_i.l_1.b.p < pol_j.l_1.e.p \land \neg$*overlaps*($pol_i.l_1.b.p$, $pol_j.l_1.e.p$) $\land$ $\forall$ *t* $\in$ *double: inside(t,* $pol_i.l_1.b.p$) $\Rightarrow$ *at_instant(pol, t)* $\in$ $SDO\_GEOMETRY_{SDO\_GTYPE=2003}$ */*polygon geometry*/ }*

In order to define the Moving_Collection and subsequently the Moving_Object data types, we first need to describe formally the multi object types for each one of the moving types:

Multi_Moving_Point $=_d$ *{ multi_mpoint: set⟨ Moving_Point⟩* | $\forall$ *i, j* $\in$ *ulong* $\land$ $\forall$ *t* $\in$ *double: inside(t,* $multi\_mpoint_i.p_j.p$) $\Rightarrow$ $\cup_i$ *(at_instant(*$multi\_mpoint_i$*, t))* $\in$ $SDO\_GEOMETRY_{SDO\_GTYPE=2005}$ */*multi-point geometry*/ }*

Multi_Moving_LineString $=_d$ *{ multi_mline: set⟨ Moving_LineString⟩* | $\forall$ *i, j* $\in$ *ulong* $\land$ $\forall$ *t* $\in$ *double: inside(t,* $multi\_mline_i.line_j.l_1.b.p$) $\Rightarrow$ $\cup_i$ *(at_instant(*$multi\_mline_i$*, t))* $\in$ $SDO\_GEOMETRY_{SDO\_GTYPE=2006}$ */*multi-linestring geometry*/ }*

Multi_Moving_Circle $=_d$ *{ multi_mcircle: set⟨ Moving_Circle⟩* | $\forall$ *i, j* $\in$ *ulong* $\land$ $\forall$ *t* $\in$ *double: inside(t,* $multi\_mcircle_i.c_j.f.p$) $\Rightarrow$ $\cup_i$ *(at_instant(*$multi\_mcircle_i$*, t))* $\in$ $SDO\_GEOMETRY_{SDO\_GTYPE=2007}$ */*multi-polygon geometry*/ }*

Multi_Moving_Rectangle $=_d$ *{ multi_mrectangle: set⟨ Moving_Rectangle⟩* | $\forall$ *i, j* $\in$ *ulong* $\land$ $\forall$ *t* $\in$ *double: inside(t,* $multi\_mrectangle_i.r_j.ll.p$) $\Rightarrow$ $\cup_i$ *(at_instant(*$multi\_mrectangle_i$*, t))* $\in$ $SDO\_GEOMETRY_{SDO\_GTYPE=2007}$ */*multi-polygon geometry*/ }*

Multi_Moving_Polygon $=_d$ *{ multi_mpolygon: set⟨ Moving_Polygon⟩* | $\forall$ *i, j* $\in$ *ulong* $\land$ $\forall$ *t* $\in$ *double: inside(t,* $multi\_mpolygon_i.pol_j.l_1.b.p$) $\Rightarrow$ $\cup_i$ *(at_instant(*$multi\_mpolygon_i$*, t))* $\in$ $SDO\_GEOMETRY_{SDO\_GTYPE=2007}$ */*multi-polygon geometry*/ }*

As such, *Moving_Collection* $=_d$ *{ 《 multi_mpoint: Multi_Moving_Point,*

multi_mline: Multi_Moving_LineString,

multi_mcircle: Multi_Moving_Circle,

multi_mrectangle: Multi_Moving_Rectangle,

multi_mpolygon: Multi_Moving_Polygon 》 |

$\forall$ *i, j* $\in$ *ulong* $\land$ $\forall$ *t* $\in$ *double: inside(t,* $multi\_mpoint_i.p_j.p$) $\land$ *inside(t,* $multi\_mline_i.line_j.l_1.b.p$) $\land$ *inside(t,* $multi\_mcircle_i.c_j.f.p$) $\land$ *inside(t,* $multi\_mrectangle_i.r_j.ll.p$) $\land$ *inside(t,* $multi\_mpolygon_i.pol_j.l_1.b.p$)

*$\Rightarrow$ [($\cup_i$ (at_instant(multi_mpoint$_i$, t))) $\cup$($\cup_i$ (at_instant(multi_mline$_i$, t))) $\cup$($\cup_i$ (at_instant(multi_mcircle$_i$, t))) $\cup$($\cup_i$ (at_instant(multi_mrectangle$_i$, t))) $\cup$ ($\cup_i$ (at_instant(multi_mpolygon$_i$, t)))] $\in$ SDO_GEOMETRY$_{SDO\_GTYPE=2004}$ /*collection geometry*/ }*

Moving_Object $=_d$ {《 mobject: Moving_Object,

mpoint: Moving_Point,

mline: Moving_LineString,

mcircle: Moving_Circle,

mrectangle: Moving_Rectangle,

mpolygon: Moving_Polygon,

mcolection: Moving_Collection,

geometry: SDO_GEOMETRY,

gtype: GeometryType,

optype: string,

arg1: ushort,

arg2: ushort,

input: Union_Input 》 | section<> }》

where

Π GeometryType T = { MOBJECT, MPOINT, MLINE, MCIRCLE, MRECTANGLE, MPOLYGON, MCOLLECTION }

Union_Input $=_d$ 《mask: string, tolerance: double, distance: double》

4 Appendix D – Description of HERMES-MDC's operations

4.1 Maintaining the Database Consistent

The two subsequent sections present how *HERMES-MDC* facilitates a user checking the construction data of two moving objects and as such maintaining the database in a consistent state:

4.1.1 Validation of a Moving LineString

In the following figure we demonstrate a series of visual transformations (virtual movements) of a moving linestring. The corresponding linguistic description of the figure as well as events raised by *HERMES-MDC* is given in Table 3. By this way we show special features, interesting and degenerated cases as well as rules and constraints that we impose in this type. HERMES-MDC identifies and reports such phenomena in order to maintain the consistency of the database. In the figure below the black solid lines represents snapshots of the moving objects. The various spatio-temporal transformations T_i that are of interest are depicted by the grey dashed arrows from some initial positions of the unit-moving points to some others (that are differently coloured).

Spatio-Temporal Transformations	*HERMES-MDC Events*
T_1: $u\_m\_p_2 \rightarrow (4, 4)$;	*Raises an application error because $u\_m\_p_2$, $u\_m\_p_3$ & $u\_m\_p_4$ that define an arc segment are becoming co-linear.*
T_2: $u\_m\_p_6 \rightarrow (7.5, 6.5)$;	*Raises an application error because $u\_m\_p_6$ is lying on an interior point of the segment defined by $u\_m\_p_4$ & $u\_m\_p_5$. In other words, segments ($u\_m\_p_4$, $u\_m\_p_5$) & ($u\_m\_p_5$, $u\_m\_p_6$) overlap.*
T_3: $u\_m\_p_6 \rightarrow (12, 8)$;	*Despite that $u\_m\_p_4$, $u\_m\_p_5$ & $u\_m\_p_6$ are becoming co-linear and as such two consequent linear segments could be replaced by only one, this is an acceptable case.*
T_4 & T_5: $u\_m\_p_6$ & $u\_m\_p_7 \rightarrow (10, 6)$;	*Raises an application error due to that $u\_m\_p_6$ & $u\_m\_p_7$ that define a line segment are becoming the same point and as such the segment is degenerated to a point.*
T_6: $u\_m\_p_7 \rightarrow (7.5, 6.5)$;	*Even though $u\_m\_p_7$, which is the last unit-moving point for the moving linestring, touches the interior of another segment, this is an acceptable case.*
T_7: $u\_m\_p_7 \rightarrow (4, 6)$;	*Despite the fact that $u\_m\_p_7$, which is the last unit-moving point for the*

	moving linestring, crosses another segment, this is an acceptable case.
T_8: u_m_p_7 → (14, 7); & u_m_p_6 → (15, 5);	*Raises an application error due to that the trajectories of u_m_p_6 & u_m_p_7, which are sequential unit-moving points, are intersecting, and this is a criterion that the corresponding unit-moving segment is rotating, something we do not accept.*
T_{10} & T_{11}: u_m_p_1 & u_m_p_7 → (7, 2);	*Raises an application error because u_m_p_1 & u_m_p_7, which are the first and last (respectively) unit-moving points for the moving linestring, are moving to the same point and as such they form a closed polygon.*

Table 3 Spatial Validation of a Moving LineString

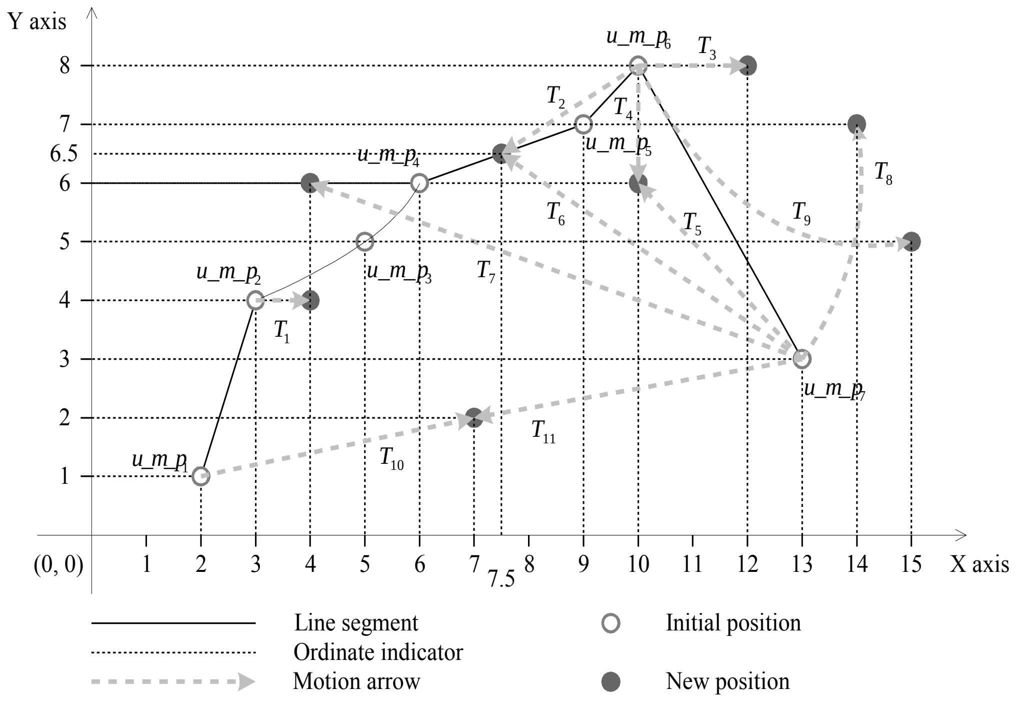


Figure 72 Spatial Validation of a Moving LineString

4.1.2 Validation of a Moving Polygon

As previously in the case of a Moving_LineString, here we follow exactly the same technique to demonstrate the *check_degeneracies* and *validate_geometry* operations that perform a spatial consistency check upon a Moving_Polygon.

Spatio-Temporal Transformations	*HERMES-MDC Events*
T_1: u_m_p_1 → (3, 1.5);	*Raises an application error due to that the moving polygon is not "closed", meaning that the first unit-moving point is not the same with the last one.*
T_2: u_m_p_2 → (5, 5);	*Despite that u_m_p_1, u_m_p_2 & u_m_p_3 are becoming co-linear and as such two consequent linear segments (u_m_p_1, u_m_p_2) & (u_m_p_2, u_m_p_3)*

	could be replaced by only one ($u\_m\_p_1$, $u\_m\_p_3$), this is an acceptable case.
T_3: $u\_m\_p_4 \rightarrow (9.5, 7)$;	*Raises an application error because $u\_m\_p_3$, $u\_m\_p_4$ & $u\_m\_p_5$ that define an arc segment are becoming co-linear.*
T_4: $u\_m\_p_7 \rightarrow (13, 7)$;	*Raises an application error due to that $u\_m\_p_7$ crosses another segment. Self-intersection of unit-moving segments is forbidden in a moving polygon.*
T_5: $u\_m\_p_7 \rightarrow (13, 6)$;	*Raises an application error due to that $u\_m\_p_7$ touches the interior of another segment. Similar situation as the previous one.*
T_6: $u\_m\_p_7 \rightarrow u\_m\_p_8$;	*Raises an application error due to that $u\_m\_p_7$ & $u\_m\_p_8$ that are components of an arc segment are becoming the same point and as such there are not three different unit-moving points to define the arc. The same case can be noticed when a moving segment is degenerated to a point.*
T_7: $u\_m\_p_9 \rightarrow (11/3, 4)$;	*Raises an application error because $u\_m\_p_9$ that is one of the vertices of a hole; touches the exterior boundary of the polygon.*
T_8: $u\_m\_p_9 \rightarrow (3, 6)$;	*Raises an application error because $u\_m\_p_9$ crosses the boundary of the polygon and as such the hole that belongs to, intersects with the exterior polygon.*
T_9: $u\_m\_p_{10} \rightarrow (11, 4)$; T_{10}: $u\_m\_p_{10} \rightarrow u\_m\_p_{15}$	*Both transformations raise an application error because the first implies that two hole-polygons are intersecting, while the second that these interior polygons are touching.*
T_{11}: $u\_m\_p_{12} \rightarrow (15, 6)$; T_{12}: $u\_m\_p_{13} \rightarrow (16, 6)$; T_{13}: $u\_m\_p_{14} \rightarrow (15, 4)$; T_{14}: $u\_m\_p_{15} \rightarrow (16, 3)$;	*These transformations also raise an application error because they are transferring an interior hole-polygon outside the exterior boundary.*

Table 4 Spatial Validation of a Moving Polygon

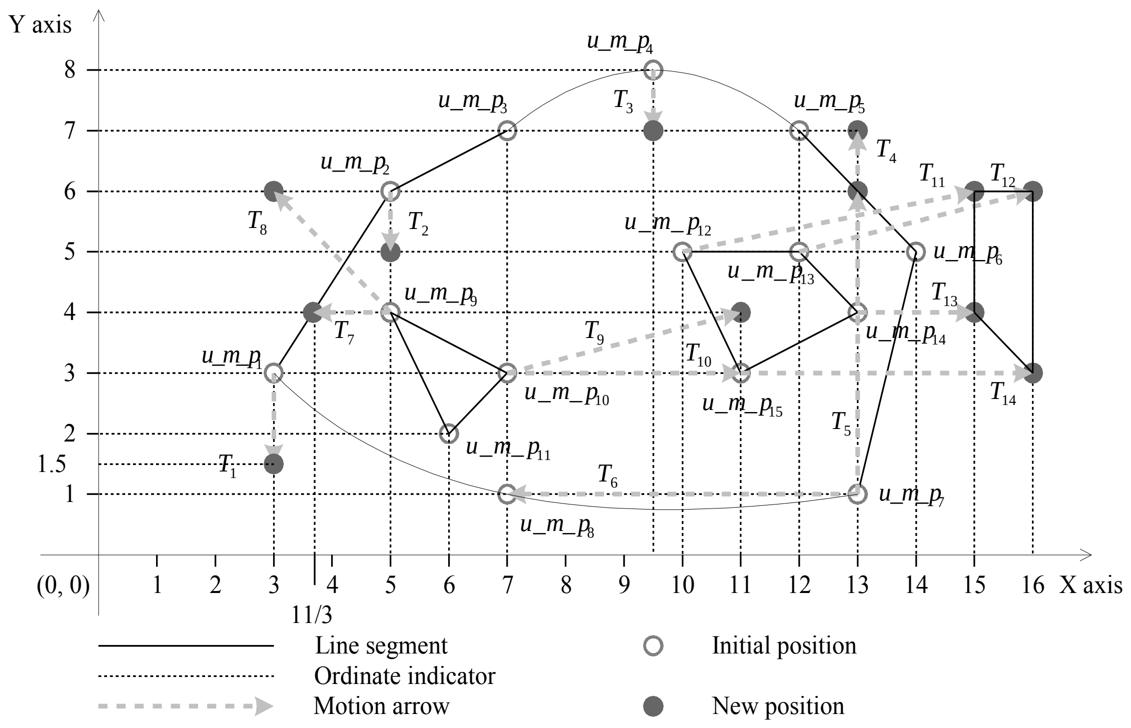


Figure 73 Spatial Validation of a Moving Polygon

4.2 Predicates Modeling Topological Relationships

The user can specify the kind of topological relationships that he requires to check via a *mask* parameter. The following *mask* relationships can be tested in HERMES-MDC:

ANYINTERACT - Returns *TRUE* if the objects are not disjoint.

CONTAINS - Returns *CONTAINS* if the argument moving object is entirely within the caller object and the object boundaries do not touch, at the given instance of time; otherwise, returns *FALSE*.

COVEREDBY - Returns *COVEREDBY* if the parameter object is entirely within the caller object and the object boundaries touch at one or more points; otherwise, returns *FALSE*.

COVERS - Returns *COVERS* if the argument object is entirely within the caller object and the boundaries touch in one or more places; otherwise, returns *FALSE*.

DISJOINT - Returns *DISJOINT* if the objects have no common boundary or interior points; otherwise, returns *FALSE*.

EQUAL - Returns *EQUAL* if the objects share every point of their boundaries and interior, including any holes in the objects; otherwise, returns *FALSE*.

INSIDE - Returns *INSIDE* if the argument object is entirely within the caller object and the object boundaries do not touch; otherwise, returns *FALSE*.

OVERLAPBDYDISJOINT - Returns *OVERLAPBDYDISJOINT* if the objects overlap, but their boundaries do not interact; otherwise, returns *FALSE*.

OVERLAPBDYINTERSECT - Returns *OVERLAPBDYINTERSECT* if the objects overlap, and their boundaries intersect in one or more places; otherwise, returns *FALSE*.

TOUCH - Returns *TOUCH* if the two objects share a common boundary point, but no interior points; otherwise, returns *FALSE*.

Values for *mask* can be combined using a logical boolean operator. For example, *'INSIDE + TOUCH'* returns *'INSIDE + TOUCH'* or *'FALSE'* depending on the outcome of the test.

Generally, the *"relate"* function can return the following types of answers:

If we pass a *mask* listing one or more relationships, the function returns the name of the relationship if it is true for the pair of geometries. If all of the relationships are false, the procedure returns *FALSE*.

If we pass the *DETERMINE* keyword in *mask*, the function returns the one relationship keyword that best matches the geometries. *DETERMINE* can only be used when the *relate* predicate is in the SELECT clause of the SQL statement.

If we pass the *ANYINTERACT* keyword in *mask*, the function returns *TRUE* if the two geometries are not disjoint.

4.3 Projection and Interaction to Temporal and/or Spatial Domain

The signatures of the object methods as these are defined for the Moving_Object type that HERMES-MDC provides for handling the projection and interaction of moving types to temporal and/or spatial domain are given in Section 4.4. Here we present the algorithms of an interesting as well as representative subset of these methods.

Below the reader can find an abstract description of the algorithm of the *at_instant* operation for a Moving_Object in the form of PL/SQL-like pseudo-code. Due to space limitations we present only the parts of the algorithm that have to do with Moving_Polygon and Moving_Collection objects as these are more interesting. Also, we do not present the algorithms but just the function calls for all the time-specific operations developed in *TAU TLL Data Cartridge*. The reader interested in these operations is referred to [Pel02].

```
FUNCTION at_instant(tp TAU_TLL.D_Timepoint_Sec) return Union_Output is
result Union_Output;
geom MDSYS.SDO_GEOMETRY;
geom1 MDSYS.SDO_GEOMETRY;
geom2 MDSYS.SDO_GEOMETRY;
BEGIN
      IF m_object.gtype IS NOT NULL THEN
        SWITCH (m_object.gtype)
        CASE 'MOBJECT':
            result :=  m_object.at_instant(tp);
        CASE 'MPOINT':
            geom := m_object.m_point.at_instant(tp);
        CASE 'MLINE':
            geom := m_object.m_line.at_instant(tp);
        CASE 'MCIRCLE':
            geom := m_object.m_circle.at_instant(tp);
        CASE 'MRECTANGLE':
            geom := m_object.m_rectangle.at_instant(tp);
        CASE 'MPOLYGON':
            geom := m_object.m_polygon.at_instant(tp);
        CASE 'MCOLLECTION':
            geom := m_object.m_collection.at_instant(tp);
        END SWITCH;
         result := Construct Union_Output from geom or result.geom;
     ELSE
        IF m_object.optype is unary THEN
            SWITCH (m_obj.arg1)
                CASE 1: result :=  m_object.at_instant(tp);
                CASE 2: geom := m_object.m_point.at_instant(tp);
                ...
                CASE 7: geom := m_object.m_collection.at_instant(tp);
                CASE 8: geom := geometry;
            END SWITCH;
            result := invoke_unary_operation(m_object.optype, geom1 or result.geom, m_object.input);
        ELSIF m_object.optype is binary THEN
            SWITCH (m_object.arg1)
                CASE 1: result := m_object.at_instant(tp);
                CASE 2: geom1  := m_object.m_point.at_instant(tp);
                ...
                CASE 7: geom1  := m_object.m_collection.at_instant(tp);
                CASE 8: geom1  := m_object.geometry;
            END SWITCH;
            SWITCH (m_object.arg2)
                CASE 1: result := m_object.at_instant(tp);
                CASE 2: geom2  := m_object.m_point.at_instant(tp);
                ...
                CASE 7: geom2  := m_object.m_collection.at_instant(tp);
                CASE 8: geom2  := m_object.geometry;
            END SWITCH;
            result := invoke_binary_operation(m_object.optype, geom1, geom2, m_object.input);
        ELSE
            raise_application_error('At_Instant operation is invalid for this kind of Moving_Object');
        END IF;
     END IF;
     return result;
END;
```

Figure 74 The *at_instant* algorithm for a Moving_Object

The pseudo-code of the *at_instant* operation for a Moving Polygon is given below:

```
FUNCTION at_instant(tp TAU_TLL.D_Timepoint_Sec) return Sdo_Geometry is
 t double;
 BEGIN
   IF check_periods_equality() <> TRUE THEN
raise_application_error('Periods in at least one entry of the nested table of type Unit_Moving_Polygon are NOT equal');
   END IF;

   IF check_sorting() <> TRUE THEN
raise_application_error('Periods in the nested table of type Unit_Moving_Polygon are NOT sorted');
   END IF;

   IF check_disjoint() <> TRUE THEN
raise_application_error('Periods in the nested table of type Unit_Moving_Polygon are NOT disjoint');
   END IF;

   /* OPTIONAL -   IF check_meet() <> TRUE THEN
raise_application_error('Periods in the nested table of type Unit_Moving_Polygon do NOT meet');
   END IF; */

   i := pol.FIRST;  -- get subscript of first unit moving polygon
   WHILE i IS NOT NULL LOOP
       contain_flag := pol(i).b.p.f_contains(pol(i).b.p, tp); --Check if tp is "inside" the period of pol(i)
       IF contain_flag = TRUE THEN
           t := tp.get_Abs_Date(); -- Map Timepoint object to real number (instant on time-line).
           result := merge_polygons(i, t);
           // The merge_polygons algorithm is given in Figure 76
          exit;
```

```
        END IF;


        i := pol.NEXT(i);  -- get subscript of next unit moving polygon
    END LOOP;


    IF result is not null THEN
        err_msg := VALIDATE_GEOMETRY(result);
        IF err_msg = 'TRUE' THEN
          return result;
        ELSE
          raise_application_error('Geometry validation failed'||err_msg);
        END IF;
    ELSE
raise_application_error('The Timepoint is NOT contained in any of the Periods in the nested table of type Unit_Moving_Polygon');
    END IF;


  END;
```

Figure 75 The *at_instant* algorithm for a Moving_Polygon

The algorithm *merge_polygons* invoked in the *at_instant* method of a *Moving_Polygon* is given in Figure 76:

```
FUNCTION merge_polygons (i, integer, t double) return Sdo_Geometry is
BEGIN
 LOOP FOREVER
    j := pol(i).l.FIRST;  -- get subscript of first moving segment
    WHILE j IS NOT NULL LOOP
      Interpolate the Unit_Moving_Points of current moving segment at instance t
```

```
          Add the description of the linestring element in the Elem_Info_Array
          Add the corresponding co-ordinates in the Ordinates_Array
          Check for degenerecies in the linestring element
          j := pol(i).l.NEXT(j);  -- get subscript of next moving segment
        END LOOP;

        result := Construct the polygon formed by the Elem_Info_Array & Ordinates_Array;
        Check for degenerecies in the polygon geometry;

        IF i <> pol.LAST THEN
          i := i + 1;
          Initialize flags & local variables;
          Extend Elem_Info_Array & Ordinates_Array for probable addition of holes;

          IF pol(i).hole = FALSE THEN
              return result;
          END IF;
        ELSE
          return result;
        END IF;
      END LOOP;
    END;
```

Figure 76 The *merge_polygons* algorithm

The pseudo-code of the *at_instant* operation for a *Moving_Collection* is given below:

```
    FUNCTION at_instant(tp TAU_TLL.D_Timepoint_Sec) return Sdo_Geometry is
    BEGIN
FOR each m_collection.multi_moving_type in {multi_mpoint, multi_mline, multi_mcircle, multi_mrectangle, multi_mpolygon}
            i := m_collection.multi_moving_type.FIRST;  -- get subscript of first element
```

```
        WHILE i IS NOT NULL LOOP
          mtype := m_collection.multi_moving_type(i);
          IF i = 1 AND result IS NULL THEN
            current_homogeneous_collection := mtype.at_instant(tp);
          ELSE
            current_geom := mtype.at_instant(tp);
current_homogeneous_collection:=ADD(current_homogeneous_collection, current_geom);
          END IF;

           i := m_collection.multi_moving_type.NEXT(i);  -- get subscript of next element
         END LOOP;

heterogeneous_collection:=ADD(heterogeneous_collection, current_homogeneous_collection);
      END FOR;

     return heterogeneous_collection;
   END;
```

Figure 77 The *at_instant* algorithm for a Moving_Collection

Figure 78 depicts the algorithm of the *at_period* operation for the case of a *Moving_LineString* object.

```
FUNCTION at_period(p TAU_TLL.D_Period_Sec) return Moving_LineString is
new_line set<Unit_Moving_LineString>;
new_p TAU_TLL.D_Period_Sec;
BEGIN
    i := line.FIRST;  -- get subscript of first Unit_Moving_LineString
    WHILE i IS NOT NULL LOOP
      /* Check if period that characterizes the current Unit
      Moving LineString overlaps with the argument period */
      overlaps_flag := line(i).b.p.f_overlaps(line(i).b.p, p); --Check if  p "overlaps" the period of line(i)


      /* If YES take the period formed as the intersection of the two
      overlapped periods and update every Unit_Moving_Point */
      IF overlaps_flag = TRUE THEN
        new_p := line(i)b.p.intersects(line(i).b.p, p);

        j := line(i).l.FIRST;  -- get subscript of first element
        WHILE j IS NOT NULL LOOP
          line(i).l(j).b.p := new_p;
          line(i).l(j).e.p := new_p;
          line(i).l(j).m.p := new_p;

          j := line(i).l.NEXT(j);
        END LOOP;

        new_line(i) := line(i);
      END IF;

      i := line.NEXT(i);  -- get subscript of next Unit_Moving_LineString
    END LOOP;

    return  Moving_LineString(new_line);
END;
```

Figure 78 The *at_period* algorithm for a Moving_LineString

In Figure 79, we provide the reader with the pseudo-code of the *at_temp_element* operation for the case of a *Moving_Point* object, where it is obvious the different strategy of restricting the temporal domain with a temporal element, rather than with a period.

```
FUNCTION at_temp_element(te TAU_TLL.D_Temp_Element_Sec) return Moving_Point is
new_point set<Unit_Moving_Point>;
intersection_te TAU_TLL.D_Temp_Element_Sec;
new_period TAU_TLL.D_Period_Sec;
BEGIN
    new_point := p;

    /* First find the temporal element object that is the intersection of the argument
    temporal element with the temporal element returned by f_temp_element function */
    intersection_te := intersection(f_temp_element(), te);

    /* For each period <new_period> composing the previous resulted temporal element,
    update those periods of the Unit Moving Points that "contain" the <new_period>. */
    k := intersection_te.te.FIRST; -- get the subscript of first period of the temporal element
    WHILE k IS NOT NULL LOOP
      new_period := intersection_te.te(k);

      i := new_point(i).FIRST;
      WHILE i IS NOT NULL LOOP
        contain_flag := new_point(i).p.f_contains(new_point(i).p, new_period);
        IF contain_flag = 1 THEN
          new_point(i).p := new_period;
        END IF;

      i := new_point.NEXT(i);
      END LOOP;

      k := intersection_te.te.NEXT(k); -- get the subscript of next period of the temporal element
    END LOOP;
    return Moving_Point(new_point);
END;
```

Figure 79 The *at_temp_element* algorithm for a Moving_Point

In Figure 80 we provide the reader with the pseudo-code of the *f_traversed* operation for the case of a Moving_LineString object. We should mention that the current implementation supports only time-changing geometries whose vertices move linearly. What is more, the soundness of the algorithm presumes that during the period associated with the linear functions describing the motion of the vertices, rotation of the segments is forbidden by a condition of the model.

```
FUNCTION f_traversed return MDSYS.SDO_GEOMETRY is
result, prev_result, line_1, line_2 MDSYS.SDO_GEOMETRY;
tp_curr TAU_TLL.D_Timepoint_Sec;
BEGIN
    i := line.FIRST; -- get subscript of first unit moving linestring;
    WHILE i IS NOT NULL LOOP
tp_curr := line(i).l(1).b.p.b; -- get the first instant of the period of the current unit moving linestring;
line_1 := at_instant(tp_curr); -- Project the Moving LineString at the spatial domain at this time point;
      -- Access the Elem_Info_Array & Ordinates_Array of line_1;
      tp_curr := f_decr( line(i).l(1).b.p.e );  -- get the last instant of the period of the current unit moving linestring;
line_2 := at_instant(tp_curr); -- Project the Moving LineString at the spatial domain at this time point;
      -- Access the Elem_Info_Array & an Ordinates_Array of line_2;
-- Initialize an Elem_Info_Array & an Ordinates_Array object for constructing the traversed polygon;
-- Depending on the type of the projected linestrings (1 & 2) construct an Elem_Info_Array that will represent a
    polygon geometry with elements the union of the elements of the Elem_Info_Arrays of line_1 & line_2;
-- Similarly, construct an Ordinates_Array that will represent a polygon geometry, whose boundary will be composed by the two projected lines connected at their end points by linear segments;
      For example...
    IF line_1 & line_2 are linearly interpolated THEN
      -- Construct Elem_Info_Array for a linerly interpolated polygon;
-- Extend the Ordinates_Array as the size of the Ordinates_Array of line_1 and transfer all the ordinates from the second to the first array;
      ordinates.EXTEND(ordinates_1.LAST);
      WHILE ordinates_offset_1 IS NOT NULL LOOP
        ordinates(ordinates_offset_1) := ordinates_1(ordinates_offset_1);
        ordinates_offset_1 := ordinates_1.NEXT(ordinates_offset_1);
      END LOOP;

-- Extend the Ordinates_Array as the size of the Ordinates_Array of line_2 and transfer all the ordinates from the second to the end of the first array;
      ordinates.EXTEND(ordinates_2.LAST);
      WHILE ordinates_offset_2 IS NOT NULL LOOP
        ordinates(ordinates_1.LAST + ordinates_offset_2) := ordinates_2(ordinates_offset_2);
        ordinates_offset_2 := ordinates_2.NEXT(ordinates_offset_2);
      END LOOP;

      ordinates.EXTEND(2); -- Connect first point to last point to form a polygon
      ordinates(ordinates_1.LAST + ordinates_2.LAST + 1) := ordinates_1(1);
      ordinates(ordinates_1.LAST + ordinates_2.LAST + 2) := ordinates_1(2);

      result := Construct the traversed polygon formed by the Elem_Info_Array & Ordinates_Array;
    ELSIF line_1 & line_2 are arc-interpolated OR are compound linestrings THEN
      Similarly...
    END IF;

-- The final traversed polygon is the union of the traversed areas at all time periods for which the Moving LineString is defined;
    IF i <> line.FIRST THEN
      result := UNION(prev_result, result);
```

```
        END IF;


        prev_result := result;

        i := line.NEXT(i); -- get subscript of last unit moving linestring

      END LOOP;

      return result;

END;
```

Figure 80 The *f_traversed* algorithm for a Moving_LineString

4.4 Signatures of operations

The signatures of the UTILITIES package:

```
PACKAGE utilities AS TYPE CursorType IS REF CURSOR;

    -- Checks if three point are co-linear
    FUNCTION check_colinear (x1 NUMBER, y1 NUMBER, x2 NUMBER, y2 NUMBER, x3
NUMBER, y3 NUMBER) RETURN BOOLEAN;
    -- Checks if the segment defined by the first two points overlaps with
the segment defined by the last two points
    FUNCTION check_overlap (x1 NUMBER, y1 NUMBER, x2 NUMBER, y2 NUMBER, x3
NUMBER, y3 NUMBER) RETURN BOOLEAN;
    -- Prints a MDSYS.SDO_GEOMETRY
    PROCEDURE print_geometry (geom MDSYS.SDO_GEOMETRY, descr VARCHAR2);
    -- Adds two angles
    FUNCTION add_angles (angle1 NUMBER, angle2 NUMBER) RETURN NUMBER;
    -- Adds two angles
    FUNCTION is_angle_between (min_angle NUMBER, angle NUMBER, max_angle
NUMBER) RETURN BOOLEAN;
    -- Returns the angle (in degrees) between the segment defined by the
two points (arguments) and the xx' axis
    FUNCTION direction (x1 NUMBER, y1 NUMBER, x2 NUMBER, y2 NUMBER) RETURN
NUMBER;
    -- Returns the angle (in degrees) between the segment defined by the
two points (arguments) and the xx' axis
    FUNCTION direction (geom1 MDSYS.SDO_GEOMETRY, geom2 MDSYS.SDO_GEOMETRY)
RETURN NUMBER;
    -- Returns the angle (in degrees) between the segment defined by the
two points (arguments) and the xx' axis
    FUNCTION get_tan (geom1 MDSYS.SDO_GEOMETRY, geom2 MDSYS.SDO_GEOMETRY)
RETURN NUMBER;
    -- Returns the angle (in degrees 0-180) between the segment defined by
the points Q_start -> Q_end and the segment defined by the points S_start -
> S_end
    FUNCTION angle (q_start MDSYS.SDO_GEOMETRY, q_end MDSYS.SDO_GEOMETRY,
s_start MDSYS.SDO_GEOMETRY, s_end MDSYS.SDO_GEOMETRY) RETURN NUMBER;
    -- Returns the angle (in degrees) between the segment defined by the
points Q_start -> Q_end and the S_angle
    FUNCTION angle2(Q_angle number, S_angle number) return number;
    -- Returns the angle (in degrees 0-360) between the segment defined by
the points Q_start -> Q_end and the segment defined by the points S_start -
> S_end
    FUNCTION angle3 (q_start MDSYS.SDO_GEOMETRY, q_end MDSYS.SDO_GEOMETRY,
s_start MDSYS.SDO_GEOMETRY, s_end MDSYS.SDO_GEOMETRY) RETURN NUMBER;
    -- Returns the distance between two points
    FUNCTION distance (x1 NUMBER, y1 NUMBER, x2 NUMBER, y2 NUMBER) RETURN
NUMBER;
    -- Sorts the multi-point argument geometry according to the direction
of a single linestring (segment)
    FUNCTION f_sort (mpoint IN OUT MDSYS.SDO_GEOMETRY, line
MDSYS.SDO_GEOMETRY) RETURN MDSYS.SDO_GEOMETRY;
    -- Returns the points being at odd positions 1,3,5 etc
    FUNCTION get_odd_points (multipoint MDSYS.SDO_GEOMETRY) RETURN
MDSYS.SDO_GEOMETRY;
    -- Returns the points being at odd positions 2,4,6, etc
    FUNCTION get_even_points (multipoint MDSYS.SDO_GEOMETRY) RETURN
MDSYS.SDO_GEOMETRY;
    -- Transfers linestring S according to the first point of linestring Q.
    FUNCTION transfer(Q MDSYS.SDO_GEOMETRY, S IN OUT MDSYS.SDO_GEOMETRY)
return MDSYS.SDO_GEOMETRY;
    FUNCTION transfer2(Q MDSYS.SDO_GEOMETRY, S IN OUT MDSYS.SDO_GEOMETRY)
return MDSYS.SDO_GEOMETRY;
    -- Computes the cost (area in m^2) for transfering segment Q towards S.
    FUNCTION transfer_cost(Q MDSYS.SDO_GEOMETRY, S MDSYS.SDO_GEOMETRY, dir
number) return number;
    -- Constructs a segment (single linestring) from the two argument
points
    FUNCTION f_segment(xi number, yi number, xe number, ye number) return
MDSYS.SDO_GEOMETRY;
    -- Returns the number of the segment of the linestring where the point
resides. The algorithm starts from from segment with number "old_pos"
    FUNCTION position(line MDSYS.SDO_GEOMETRY, x number, y number, old_pos
pls_integer) return pls_integer;
    -- Checks if Q's (PQ) or S's (PS) point "sees" the last segment of
Q_line or S_line without intersecting the previous segments of the latter
    FUNCTION BadSegment(Q_line MDSYS.SDO_GEOMETRY, S_line
MDSYS.SDO_GEOMETRY, PQx number, PQy number, PSx number, PSy number) return
boolean;
    -- Smooth linestring
    PROCEDURE SmoothLine(L IN OUT MDSYS.SDO_GEOMETRY);
    -- Spatial Similarity
    FUNCTION LIP(Q MDSYS.SDO_GEOMETRY, S IN OUT MDSYS.SDO_GEOMETRY, trans
boolean, Q_LEN number, S_LEN  number) return number;
    -- Integrates LIP
    FUNCTION FindBadSegments(Q IN OUT MDSYS.SDO_GEOMETRY, S IN OUT
MDSYS.SDO_GEOMETRY, trans boolean, policy pls_integer, Q_LEN number, S_LEN
number) return number;
    -- Second policy
    FUNCTION GenLIP(Q IN OUT MDSYS.SDO_GEOMETRY, S IN OUT
MDSYS.SDO_GEOMETRY, trans boolean, policy pls_integer, Q_LEN number, S_LEN
number) return number; --, avg_sim  IN OUT number, NoLIPgrams IN OUT
pls_integer
    -- Direction Distance
    FUNCTION DDIST(Q IN OUT MDSYS.SDO_GEOMETRY, S IN OUT
MDSYS.SDO_GEOMETRY, policy pls_integer) return number;
    -- Computes MDI
    FUNCTION compute_MDI (startQ_tp TAU_TLL.D_Timepoint_Sec, endQ_tp
TAU_TLL.D_Timepoint_Sec, startS_tp TAU_TLL.D_Timepoint_Sec, endS_tp
TAU_TLL.D_Timepoint_Sec, delta TAU_TLL.D_Interval) return number;
```

END;

The signatures of the Unit_Moving_Point object methods:

```
TYPE unit_moving_point AS OBJECT (
   -- Time period with granularity second where Unit function is valid
   p   tau_tll.d_period_sec,
   -- Motion during period p
   m   unit_function,
   -- ###### MEMBER FUNCTIONS #####
   -- Polynomial of first degree
   MEMBER FUNCTION f_plnml_1 (tp tau_tll.d_timepoint_sec) RETURN coords,
   -- Polynomial of first degree
   MEMBER FUNCTION r_f_plnml_1 (x NUMBER, y NUMBER) RETURN
tau_tll.d_timepoint_sec,
   --
   MEMBER FUNCTION f_plnml_3_1 (tp tau_tll.d_timepoint_sec) RETURN coords,
   --
   MEMBER FUNCTION f_plnml_3_2 (tp tau_tll.d_timepoint_sec) RETURN coords,
   --
   MEMBER FUNCTION r_f_plnml_3_x (x NUMBER, y NUMBER) RETURN
tau_tll.d_timepoint_sec,
   -- Depending on the "descr" of the Unit_Function invokes the appropriate
function
   MEMBER FUNCTION f_interpolate (tp tau_tll.d_timepoint_sec) RETURN
coords,
   -- Returns the timepoint that corresponds to a specific xy coords
   MEMBER FUNCTION get_time_point (x NUMBER, y NUMBER) RETURN
tau_tll.d_timepoint_sec,
   -- Checks if this unit_moving_point contains the given (x, y)
   MEMBER FUNCTION f_contains (x NUMBER, y NUMBER) RETURN BOOLEAN,
   -- Gets the speed at the given timepoint
   MEMBER FUNCTION get_speed (tp tau_tll.d_timepoint_sec) RETURN NUMBER,
   -- Get (x, y) of the
   MEMBER FUNCTION get_midle_point RETURN coords
)
```

The signatures of the Moving_Point object methods:

```
TYPE moving_point AS OBJECT (
   -- A series of Unit_Moving_Point defining the consequent parts of a
Moving_Point
   u_tab   moving_point_tab,        -- previous name of the attribute was
"p"
   --The trajectory id should be placed in the moving object so as to be
retrieved by
   --ODCIIndexUpdate and ODCIIndexInsert
   traj_id Integer,
   -- Returns moving point as a CLOB
   -- ###### MEMBER FUNCTIONS #####
   MEMBER FUNCTION to_clob RETURN CLOB,
   -- Returns moving point as a string
   MEMBER FUNCTION to_string RETURN VARCHAR2,
   -- Prints moving point to standard output
   MEMBER PROCEDURE print_moving_point,
   -- Add a unit_moving_point
   MEMBER PROCEDURE add_unit (new_unit unit_moving_point),
   -- Merge two Moving_Points
   MEMBER FUNCTION merge_moving_points (mp1 moving_point, mp2 moving_point)
RETURN moving_point,
   --Checks if there is an ascending sorting of the periods in the nested
table
   MEMBER FUNCTION check_sorting RETURN BOOLEAN,
   --Checks if even one period in the nested table overlaps with the next
in order period...THEN returns FALSE
   MEMBER FUNCTION check_disjoint RETURN BOOLEAN,
   --Checks if even one period in the nested table does NOT meets with the
next in order period...THEN returns FALSE
   MEMBER FUNCTION check_meet RETURN BOOLEAN,
   -- Returns that Unit_Moving_Point that corresponds to a specific
timepoint
   MEMBER FUNCTION unit_type (tp tau_tll.d_timepoint_sec) RETURN
unit_moving_point,
   -- Sorts the multi-point argument geometry by time
   MEMBER FUNCTION sort_by_time (mpoint IN OUT MDSYS.SDO_GEOMETRY) RETURN
MDSYS.SDO_GEOMETRY,
   -- Return the enter and leave points of the moving point for a given
geometry
   MEMBER FUNCTION get_enter_leave_points (geom MDSYS.SDO_GEOMETRY) RETURN
MDSYS.SDO_GEOMETRY,
   -- Returns a MDSYS.SDO_GEOMETRY of Point type as the result of
Mapping/Projecting the Moving_Point at a specific timepoint
   MEMBER FUNCTION at_instant (tp tau_tll.d_timepoint_sec) RETURN
MDSYS.SDO_GEOMETRY,
   -- Returns a moving point restricted at a specific period
   MEMBER FUNCTION at_period (per tau_tll.d_period_sec) RETURN
moving_point,
   -- Returns a moving point restricted at a specific temporal element
   MEMBER FUNCTION at_temp_element (te tau_tll.d_temp_element_sec) RETURN
moving_point,
   -- Restricts the moving point at the space specified by the linestring
parameter which is supposed to be part of his route
   MEMBER FUNCTION at_linestring (line MDSYS.SDO_GEOMETRY) RETURN
moving_point,
   -- Returns tha last valid timepoint of the lifespan of the moving point
   MEMBER FUNCTION f_final_timepoint RETURN tau_tll.d_timepoint_sec,
   -- Returns tha first valid timepoint of the lifespan of the moving point
   MEMBER FUNCTION f_initial_timepoint RETURN tau_tll.d_timepoint_sec,
   -- Returns the timepoint that corresponds to a specific xy coords
   MEMBER FUNCTION get_time_point (x NUMBER, y NUMBER) RETURN
tau_tll.d_timepoint_sec,
   -- Returns a linestring geometry representing the points that this
moving point traverses! NOTE: For linear motions use "f_trajectory2"
   MEMBER FUNCTION f_trajectory RETURN MDSYS.SDO_GEOMETRY,
   MEMBER FUNCTION f_trajectory2 RETURN MDSYS.SDO_GEOMETRY,
   -- Returns a temporal element constructed by the union of the periods
for which the moving point is defined
   MEMBER FUNCTION f_temp_element RETURN tau_tll.d_temp_element_sec,
   -- Returns the instanced point as this is defined at the first valid
second
   MEMBER FUNCTION f_initial RETURN MDSYS.SDO_GEOMETRY,
   -- Returns the instanced point as this is defined at the last valid
second
   MEMBER FUNCTION f_final RETURN MDSYS.SDO_GEOMETRY,
   -- Returns the angle of the moving point' s direction
   MEMBER FUNCTION f_direction (tp tau_tll.d_timepoint_sec) RETURN NUMBER,
   -- Returns TRUE for objects being in front of moving point at the given
timepoint
```

```
   MEMBER FUNCTION f_front (geom MDSYS.SDO_GEOMETRY, tp
tau_tll.d_timepoint_sec, angle_min   NUMBER, angle_max   NUMBER) RETURN
NUMBER,
   -- Returns TRUE for objects being behind of moving point at the given
timepoint
   MEMBER FUNCTION f_behind (geom MDSYS.SDO_GEOMETRY, tp
tau_tll.d_timepoint_sec, angle_min NUMBER, angle_max NUMBER) RETURN NUMBER,
   -- Returns TRUE for objects being left of moving point at the given
timepoint
   MEMBER FUNCTION f_left (geom MDSYS.SDO_GEOMETRY, tp
tau_tll.d_timepoint_sec, angle_min NUMBER, angle_max NUMBER) RETURN NUMBER,
   -- Returns TRUE for objects being right of moving point at the given
timepoint
   MEMBER FUNCTION f_right (geom MDSYS.SDO_GEOMETRY, tp
tau_tll.d_timepoint_sec, angle_min NUMBER, angle_max NUMBER) RETURN NUMBER,
   -- Returns TRUE for objects being north of moving point at the given
timepoint
   MEMBER FUNCTION f_north (geom MDSYS.SDO_GEOMETRY, tp
tau_tll.d_timepoint_sec, angle_min NUMBER, angle_max NUMBER) RETURN NUMBER,
   -- Returns TRUE for objects being south of moving point at the given
timepoint
   MEMBER FUNCTION f_south (geom MDSYS.SDO_GEOMETRY, tp
tau_tll.d_timepoint_sec, angle_min NUMBER, angle_max NUMBER) RETURN NUMBER,
   -- Returns TRUE for objects being east of moving point at the given
timepoint
   MEMBER FUNCTION f_east (geom MDSYS.SDO_GEOMETRY, tp
tau_tll.d_timepoint_sec, angle_min NUMBER, angle_max NUMBER) RETURN NUMBER,
   -- Returns TRUE for objects being west of moving point at the given
timepoint
   MEMBER FUNCTION f_west (geom MDSYS.SDO_GEOMETRY, tp
tau_tll.d_timepoint_sec, angle_min NUMBER, angle_max NUMBER) RETURN NUMBER,
   -- Returns TRUE when the moving point is between the multi-geometry at
the given timepoint
   MEMBER FUNCTION f_between (geom MDSYS.SDO_GEOMETRY, tp
tau_tll.d_timepoint_sec) RETURN NUMBER,
    -- Returns the rate of change of the Euclidean distance (speed) that
the moving point traverses at a specific time point
   MEMBER FUNCTION f_speed (tp tau_tll.d_timepoint_sec) RETURN NUMBER,
    -- Generates a buffer polygon around an instanced point at a specific
timepoint
   MEMBER FUNCTION f_buffer (distance NUMBER, tolerance NUMBER, tp
tau_tll.d_timepoint_sec) RETURN MDSYS.SDO_GEOMETRY,
    -- Computes the distance between two moving points instanced at a
specific timepoint.
    -- The distance between two geometry objects is the distance between
the closest pair of points or segments of the two objects
   MEMBER FUNCTION f_distance (moving_point moving_point, tolerance NUMBER,
tp tau_tll.d_timepoint_sec) RETURN NUMBER,
    -- Computes the distance between a moving point instanced at a specific
timepoint and another geometry type.
    -- The distance between two geometry objects is the distance between
the closest pair of points or segments of the two objects
   MEMBER FUNCTION f_distance (geom MDSYS.SDO_GEOMETRY, tolerance NUMBER,
tp tau_tll.d_timepoint_sec) RETURN NUMBER,
    -- Determines if this moving point is within some specified Euclidean
distance from other moving objects at  a specific timepoint
   MEMBER FUNCTION f_within_distance (distance NUMBER, moving_point
moving_point, tolerance NUMBER, tp tau_tll.d_timepoint_sec) RETURN
VARCHAR2,
    -- Determines if this moving point is within some specified Euclidean
distance from other geometry objects at a specific timepoint
```

```
   MEMBER FUNCTION f_within_distance (distance NUMBER, geom
MDSYS.SDO_GEOMETRY, tolerance NUMBER, tp tau_tll.d_timepoint_sec) RETURN
VARCHAR2,
    -- Examines current Moving_Point to determine its spatial relationship
with another moving point
   MEMBER FUNCTION f_relate (MASK VARCHAR2, moving_point moving_point,
tolerance NUMBER, tp tau_tll.d_timepoint_sec) RETURN VARCHAR2,
    -- Examines current Moving_Point to determine its spatial relationship
with other geometry objects
   MEMBER FUNCTION f_relate (MASK VARCHAR2, geom MDSYS.SDO_GEOMETRY,
tolerance NUMBER, tp tau_tll.d_timepoint_sec) RETURN VARCHAR2,
    -- Returns a geometry object that is the topological intersection (AND
operation) of an instanced point with another moving point at a specific
timepoint
   MEMBER FUNCTION f_intersection (moving_point moving_point, tolerance
NUMBER, tp tau_tll.d_timepoint_sec) RETURN MDSYS.SDO_GEOMETRY,
    -- Returns a geometry object that is the topological intersection (AND
operation) of an instanced point at a specific timepoint with another
geometry object
   MEMBER FUNCTION f_intersection (geom MDSYS.SDO_GEOMETRY, tolerance
NUMBER, tp tau_tll.d_timepoint_sec) RETURN MDSYS.SDO_GEOMETRY,
    -- Returns a moving point that is the restriction (intersection) of the
calling moving point inside the polygon argument
   MEMBER FUNCTION f_intersection (geom MDSYS.SDO_GEOMETRY, tolerance
NUMBER) RETURN moving_point,
    -- Returns a moving point that is the restriction (intersection) of the
calling moving point inside the polygon argument
   MEMBER FUNCTION f_intersection2 (geom MDSYS.SDO_GEOMETRY, tolerance
NUMBER) RETURN moving_point,
    -- Computes the linestring and the period that is the restriction
(intersection) of the calling moving point inside the polygon argument
   MEMBER PROCEDURE f_intersection (geom MDSYS.SDO_GEOMETRY, line_inside
OUT MDSYS.SDO_GEOMETRY, period_inside OUT tau_tll.d_period_sec, tolerance
NUMBER),
    -- Returns a moving point (and the corresponding linestring and period)
that is the restriction (intersection) of the calling moving point inside
the polygon argument
   MEMBER FUNCTION f_intersection (geom MDSYS.SDO_GEOMETRY, line_inside OUT
MDSYS.SDO_GEOMETRY, period_inside OUT tau_tll.d_period_sec, tolerance
NUMBER) RETURN moving_point,
    -- Returns a geometry object that is the topological union (OR
operation) of an instanced point with this moving point at a specific
timepoint
   MEMBER FUNCTION f_union (moving_point moving_point, tolerance NUMBER, tp
tau_tll.d_timepoint_sec) RETURN MDSYS.SDO_GEOMETRY,
    -- Returns a geometry object that is the topological union (OR
operation) of an instanced point at a specific timepoint with another
geometry object
   MEMBER FUNCTION f_union (geom MDSYS.SDO_GEOMETRY, tolerance NUMBER, tp
tau_tll.d_timepoint_sec) RETURN MDSYS.SDO_GEOMETRY,
    -- Returns a geometry object that is the topological symmetric
difference (XOR operation) of an instanced point with this moving point at
a specific timepoint
   MEMBER FUNCTION f_xor (moving_point moving_point, tolerance NUMBER, tp
tau_tll.d_timepoint_sec) RETURN MDSYS.SDO_GEOMETRY,
   -- Returns a geometry object that is the topological symmetric
difference (XOR operation) of an instanced point at a specific timepoint
with another geometry object
   MEMBER FUNCTION f_xor (geom MDSYS.SDO_GEOMETRY, tolerance NUMBER, tp
tau_tll.d_timepoint_sec) RETURN MDSYS.SDO_GEOMETRY,
    -- Returns the points(sorted by time) that the moving point enters
inside the area of the polygon argument
   MEMBER FUNCTION f_enterpoints (geom MDSYS.SDO_GEOMETRY) RETURN
MDSYS.SDO_GEOMETRY,
    -- Returns the points(sorted by time) that the moving point leaves the
area of the polygon argument
   MEMBER FUNCTION f_leavepoints (geom MDSYS.SDO_GEOMETRY) RETURN
MDSYS.SDO_GEOMETRY,
    -- Returns the timepoint that the moving point entered the given
polygonal geometry
   MEMBER FUNCTION f_enter (geom MDSYS.SDO_GEOMETRY) RETURN
tau_tll.d_timepoint_sec,
    -- Returns the timepoint that the moving point left the given polygonal
geometry
   MEMBER FUNCTION f_leave (geom MDSYS.SDO_GEOMETRY) RETURN
tau_tll.d_timepoint_sec,
    -- Returns the average speed of a moving point during its lifespan
    MEMBER FUNCTION f_avg_speed RETURN NUMBER,
    -- Returns the average acceleration of a moving point during its
lifespan
    MEMBER FUNCTION f_avg_acceleration RETURN NUMBER,
    -- Returns the average direction of a moving point during its lifespan
    MEMBER FUNCTION f_avg_direction RETURN NUMBER,
    -- Transfers moving point to the starting point of Sm. The translation
is dx on XX' and dy in YY'
    MEMBER FUNCTION transfer2(Qm moving_point, Sm IN OUT moving_point)
return moving_point,
    -- Returns the timepoint when the moving point passes from (x,y). The
algorithm starts looking from "old_pos"
    MEMBER FUNCTION f_timepoint(line MDSYS.SDO_GEOMETRY, x number, y
number, old_pos pls_integer,  new_pos OUT pls_integer) return
TAU_TLL.D_Timepoint_Sec,
    -- Returns the Locality In-between Polylines=projections of the two
moving points
    MEMBER FUNCTION LIP(m_point Moving_Point, trans boolean) return number,
    -- Returns the Spatio-Temporal Distance between two moving points
    MEMBER FUNCTION STLIP(S IN OUT Moving_Point, trans boolean, t
TAU_TLL.D_Interval, Q_LEN number, S_LEN    number, kapa number) return
number,
    -- Returns the Speed-Pattern STLIP between two moving points following
arbitrary trajectories
    MEMBER FUNCTION SPSTLIP(S IN OUT Moving_Point, trans boolean, t
TAU_TLL.D_Interval, Q_LEN number, S_LEN number) return number,
    -- Returns the Direction Distance between the spatial projections of
two moving points
    MEMBER FUNCTION DDIST(m_point Moving_Point, policy pls_integer) return
number,
    -- Returns the Direction Distance between two moving points
    MEMBER FUNCTION TDDIST(S IN OUT Moving_Point, policy pls_integer)
return number,
    -- Integrates STLIP
    MEMBER FUNCTION GenSTLIP_OSP(S_M IN OUT Moving_Point, trans boolean,
policy pls_integer, Q_LEN number, S_LEN number, kapa number, delta number)
return number

                                   )
```

5 Appendix E – Description of HERMES TB–TREE PL/SQL Implementation Building Blocks

Hereafter, we discuss the basic principles that have been fostered in our effort to embody the TB-Tree index in the Oracle ORDBMS. We will present in detail the primitive data types (objects) that have been defined to serve as the building blocks and primary storage elements of the underlying structure. Moreover, we proceed by discussing the OR database tables involving the index and the way they relate to the primary table where moving object trajectories are stored.

Types	Description
tbX	The tbX collection type is a varray of size 3 used to hold triplets (x,y,t) of the points taking part in a moving object's trajectory formation
tbPoint	The tbPoint is defined as an object with x of tbX type as its only attribute
tbMBB	The tbMBB is an object that represents the Minimum Bounding Box of a tree node. Its attributes (MinPoint, MaxPoint) are both of tbPoint type and represent the lower left and upper right of the box
tbTreeLeafEntry	The tbTreeLeafEntry is an object involving the entries of leaf nodes. Each such entry has two attributes: MBB of tbMBB type and Orientation of integer type. The first involves the box defined by a moving object's trajectory segment while the latter denotes the orientation of the segment in the MBB
tbTreeNodeEntry	The tbTreeNodeEntry is an object involving the entries of internal (non-leaf) nodes. Each such entry has two attributes: MBB of tbMBB type and Ptr of integer type. The first involves the box defined by the MBB of all the entries of its child node, while the second denotes the identifier of the current entry's child
NodeEntries	The NodeEntries collection type is a varray of size 155 (hard coded for block size=8192 Bytes) used to hold an internal node's entries. The size of the varray was defined so as to ensure that the entries (plus any additional attributes) of the node will fit in one disk block
LeafEntries	The LeafEntries collection type is a varray of size 155 (hard coded for block size=8192 Bytes) used to hold a leaf node's entries. The size of the varray was defined so as to ensure that the entries (plus any additional attributes) of the node will fit in one disk block
LeafEntries2	Same as LeafEntries but this time the collection type is defined as a nested table. The reason is that when processing leaf entries in memory (i.e during their insertion in the leaf node) we need a structure the size of which is not known in hand. Furthermore, the design of certain operators where those entries are processed does not make any assumption about a fixed disk block size. Eventually, we would need a varray to store leaf entries since nested tables do not guarantee the insertion and storage of the entries in the sorted form that have been arranged during insertion algorithm execution. Obviously, this gives us the advantage that a change of the page size will only need the modification of the varray's dimension in LeafEntries, NodeEntries
tbTreeNode	The tbTreeNode is an object representing the internal node itself. Its attributes involve: 1) ptrParentNode of integer type which is a pointer to the parent of the current node used to ascend the tree when necessary, 2) ptrCurrentNode of integer type which is the current node's identifier encapsulated in the object to facilitate implementation issues, 3) counter of integer type to hold the number of the current node entries. This is extremely useful since we are able to know in hand the number of entries in the node instead of using the .COUNT collection operator to count the number of entries every time the node is used. 4) tbTreeNodeEnties of NodeEntries type which involves the entries of the node as were previously

	described
tbTreeLeaf	The tbTreeLeaf is an object representing the internal node itself. Its attributes involve: 1) MoID of type integer which is the global identifier of a trajectory 2) the rowid (varchar2!) of the moving object whose partial trajectory is contained in the leaf. This is used by the ODCIINDEXFETCH to return batches of base table rows, 3) ptrParentNode of integer type which is a pointer to the parent of the current node used to ascend the tree when necessary, 4) ptrCurrentNode of integer type which is the current node's identifier encapsulated in the object to facilitate implementation issues, 5) PtrPreviousNode of type integer which is a pointer to the node with the parts of the trajectory preceding those of the current node 6) PtrNextNode of type integer which is a pointer to the node with the next parts of the trajectory, 7) counter of integer type to hold the number of the current node's entries. The utility of the attribute has already been discussed, 8) tbTreeLeafEntries of LeafEntry type which involves the entries of the leaf as were previously described
tbTreeLeaf2	Same as before, but this time we use LeafEntries2
tbMovingObject	The tbMovingObject is an object with 2 attributes. The first one –ID- denotes the identifier of a stored trajectory while the second – ptrLastleaf - involves the identifier of the node where the last entries of the trajectory are kept
tbMovingObjectsCollection	A nested table of tbMovingobject used in function and operator implementations
tbMovingObjectEntry	Type tbMovingObjectEntry is a 3 - dimensional moving object line segment containing the object`s id and the points
tbMovingObjectEntries	A nested table of tbMovingobjectEntry
IDS	A nested table of integers used to hold IDs of moving objects
mp_Array	A nested table of hermes.moving_object type. This type is used as the return type of query functions. For instance, performing a range query, we expect parts of moving objects as the answer. These trajectory parts are stored in an array of mp_array type and are returned to the user in that form
Geom_tbl	Same as above but this time we use a nested table of geometries. The utility is the same as in the mp_Array case. The only difference is the transformation of moving_object into geometry so that it can be projected on a map
PriorityQueueNode	This is an object that constitutes the building block of a priority queue data structure. Such kind of node should be modeled as hybrid since it can hold features involving a tbTreeNode or TbTreeLeaf object. Its attributes are as follows: 1) Ptr integer, for tbtreenodeentry this is a pointer to the child leaf, 2) MBB tbMBB, for tbtreenodeentry this is the MBB of the entry, 3) Id number, the id of the moving object, 4) P1 tbPoint, the first point of a moving object entry, 5) P2 tbPoint, the last point of a moving object entry, 6) EType varchar2(20), the type of the entry in {x for null, tbMovingobjectEntry, tbTreenodeEntry}, 7) Dist number, the distance of the queue entry calculated by the MinDistLine2D function, and PtrNext integer, PtrPrevious integer, PtrCurrent integer, Trajectory tbMovingObjectEntries
QueueEntries	A nested table of PriorityQueueNode type used to keep the entries of the respective structure
PriorityQueue	The priority queue is intended to hold line segments of trajectories, ordered based on their 2D distance from a given trajectory or point. It is therefore useful to note here that its utility involves the implementation of IncPointNNSearch and IncTrajectoryNNSearch as well as their variations, namely mv_IncPointNNSearch, mv_ IncTrajectoryNNSearch. The attributes of the queue are as follows: 1) Entries of QueueEntries type to hold the actual entries of the queue, 2) counter of integer type which is used to hold the current number of entries in the queue as nodes are inserted or extracted (deleted), 3) Last of integer type which is a pointer to the tail of

	the queue, 4) while top is a pointer to the first element of the queue. The object is equipped with member functions used for initializing, enqueueing and dequeueing entries
TBTree_IdxType_Im	This object is the actual interface based on which our custom index type is built. Based on the extensible indexing capabilities provided by the oracle ordbms each such object should own the following functions: 1) ODCIIndexCreate which is a function that creates the index tables (i.e tbtreeidx_leaf, tbtreeidx_non_leaf) and populates the data already inserted in the table (mpoints) on which the index is created, 2) ODCIIndexInsert which a function that performs insertions in the tree triggered by the insertion of a new trajectory on the main table, 3) ODCIIndexUpdate which is a function that updates the tree every time a new trajectory segment (i.e unit_moving_point) is inserted, 4) ODCIIndexDelete to adjust the tree upon the occurrence of a deletion (not implemented), 5) The ODCIGetInterfaces function returns the list of names of the interfaces implemented by the type. To specify the current version of these interfaces, the ODCIGetInterfaces routine must return'SYS.ODCIINDEX2' in the OUT parameter, 6) The ODCIIndexDrop function drops the tables that store the index data. This method is called when a DROP INDEX statement is issued, 7) The ODCIIndexStart is a function that prepares the execution of an operator by determining the rowids of the base table that need to be fetched. Note that we need a new ODCIIndexStart for each operator that will be later defined. 8) The ODCIIndexFetch function fetches the base table rows as they are determined by a corresponding ODCIIndexStart function, 9) The ODCIIndexClose function completes the execution of a custom operator

Table 5: Defined types and corresponding description

Having described the basic data types defined to serve the implementation purposes of the index structure as well as the operations and functions built upon it, we will proceed by referring to the basic tables the tuples of which constitute the primary storage elements of trajectory and index data. Note beforehand that it is crucial for the table names to remain as are, since they are hard coded in the index implementation packages, functions and operands.

Table Name	Description
mpoints	This table is used to store the trajectories of moving objects. Its attributes involve the object and trajectory id and the actual trajectory of hermes.moving_point type. Note that the trajectory id (traj_id) should be unique and practically is attributed as a sequence of increasing integers based on the trajectory insertion order. Mpoints should be manually defined and constitutes the table on which the tb-tree index will be created
movingobjects	The movingobjects table is an auxiliary one which is used to store a pointer to the index leaf where the last part of a trajectory is stored. As such it aparts from 2 columns for ID, pointer integer values. Note that this table is automatically created/dropped upon the index creation/drop
Tbtreeidx_non_leaf	This is the table where the internal nodes of the tree are stored. Any additional column is just a copy of a certain object attribute and it is used for fast access. Note that this table is automatically created/dropped upon the index creation/drop
tbTreeidx_leaf	This is the table where the leaf nodes of the tree are stored. Any additional column is just a copy of a certain object attribute and it is used for fast access. Note that this table is automatically created/dropped upon the index creation/drop
mv_tbl	This table is an auxiliary one since it is used to store the geometries returned after the execution of functions such as mv_query_window, mv_IncTrajectoryNNSearch, mv_IncPointNNSearch, mv_query_window2. The aforementioned functions will execure the posed query, store the results in mv_tbl and the mapviewer interface will then issue a select * statement to visualize them. After the visualization they should be deleted so that the mv_tbl will be empty for

	the next query answers.

5.1 TB – TREE PL/SQL Implementation Packages

tbFunctions package

Function/ Procedure	Parameters	Return type	Description
MoAdd	IMO tbMovingObject		A procedure which adds or updates the MovingObjects Table
UpdateTreeHeight	—	—	A procedure that updates the tree height (a system parameter that is stored in movingobjects table) when the root of the tree is split
Savenode	Node tbtreeleaf, leaftab varchar2, existence boolean	—	A procedure used to save or update an internal node of the tree to the corresponding table
Saveleaf	Node tbtreeleaf, leaftab varchar2, existence boolean	—	A procedure used to save or update a leaf node of the tree to the corresponding table
TBINSERT	POINT1 TBPOINT, POINT2 TBPOINT, MOVINGOBJECTID INTEGER, RID varCHAR2, leaftab VARCHAR2, nodetab VARCHAR2	—	The Insertion Method of the TB-Tree
ChooseLastLeaf	nodetab varchar2, leaftab varchar2	tbTreeLeaf	A function that descents the TB-tree until it finds the last (right-most) leaf node which is finally returned
Includes	SourceMBR tbMBB, InsertedMBR tbMBB, Dimensions integer	Boolean	A function that returns true if a SourceMBR includes an InsertedMBR
AdjustTree	L tbTreeLeaf, LL tbTreeLeaf, nodetab varchar2, leaftab varchar2	tbTreeNode	Algorithm AdjustTree by Antonin Guttman
LCoveringMBB	Node tbTreeLeaf	tbMBB	A function that calculates the covering rectangle of a rTree Leaf Node
NCoveringMBB	Node tbTreeNode	tbMBB	A function that calculates the covering rectangle of a rTree internal Node
OVERLAPS1D	SourceMin Number, SourceMax Number, InsertedMin Number, InsertedMax Number	boolean	A function that returns true if the SourceMBR overlaps the InsertedMBR in 1 dimension
TBMAX	A1 NUMBER, A2 NUMBER	NUMBER	A function that finds the maximum between 2 numbers
TBMIN	A1 NUMBER, A2 NUMBER	NUMBER	A function that finds the minimum between 2 numbers
Equals	P1 tbPoint, P2 tbPoint	Boolean	A function that returns true if tbPoints P1 and P2 are very close (are equal..)
HFINDNODE	IDD integer, P1	tbTreeLeaf	A function which uses the hashed structure

	TBPOINT, tab varchar2		containing each trajectory's last position (moving objects table) and returns the appropriate leaf
READLEAFNODE	PTRNODE varchar2, tab varchar2	TBTREELEAF	A function used to read a LEAF node from the corresponding table
READNODE	PTRNODE varchar2, tab varchar2	TBTREENOD E	A function used to read an internal tree node from the corresponding table
ConstructEntry	Ent tbTreeLeafEntr, Id integer	tbMovingObject Entry	A function to convert a 3D R-tree entry to a 3D entry with starting point the (X1,Y1,T1) and ending point the (X2,Y2,T2)
Overlapss	sourceMBb TBMBB, insertedMBb TBMBB, Dimensions integer	boolean	General overlap function. It checks if overlap does exist in 1 to 3 dimensions based on a given (dimension) parameter
leafentry_to_unit_mov ing_point	tble tbtreeleafentry	hermes.unit_mo ving_point	A function that transforms a tb tree leaf entry to the corresponding hermes.unit_moving_point
tb_mp_in_spatiotemp oral_window	geom MDSYS.SDO_GEO METRY, tp tau_tll.D_period_sec	hermes.mp_Arra y	A function that returns the partial trajectories of all moving points restricted in a certain spatiotemporal window (Note: This is the operator used to extract statistics for the cuboids during the ETL procedure)
mv_query_window	geom MDSYS.SDO_GEO METRY,tp tau_tll.D_period_sec	—	Same as tb_mp_in_Spatiotemp_Wind but returns an array of SDO_GEOMETRIES to be used in mapviewer
ConstructMBB	Ent tbMovingObjectEntr y	tbMBB	A function that returns the MBB of a given tbMovingObjectEntry
Distance2D	P1 tbPoint, P2 tbPoint	integer	A function that calculates the squared distance between two points
MinDist2D	Point tbPoint, MBB tbMBB	integer	A function that returns the minimum distance between a point and an MBB
ActualDist2D	Point tbPoint, P1 tbPoint, P2 tbPoint	integer	A function that calculates the actual distance of a point from a straight line
Intersects2D	Line1 tbMovingObjectEntr y, Line2 tbMovingObjectEntr y	Boolean	A function that returns true if two line segments intersect
MinDistLine2D	Line tbMovingObjectEntr y, MBB tbMBB	integer	A function that returns the minimum distance between a line and an MBB
ActualLineDist2D	Line1 tbMovingObjectEntr y, Line2 tbMovingObjectEntr y	number	A function that calculates the minimum horizontal distance between two 3d lines
IncPointNNSearch Error! Reference source not found.	QueryPoint tbMovingObjectEntr y, k integer	tbMovingObject Entries	This is a function (actually operator) that acts as follows: given a static point, it returns the k trajectory segments that are closer to it
GetTrajectoryPart	Trajectory tbMovingObjectEntr ies, iMBB	tbMovingObject Entries	Algorithm GetTrajectoryPart retrieves the part of the trajectory temporaly contained inside the temporal component of iMBR

	tbMBB, traj_size integer		
MinDistTrajectory2D	Trajectory tbMovingObjectEntries, MBB tbMBB, traj_size integer	number	A function that returns the minimum distance between a trajectory and a MBB
IncTrajectoryNNSearch Error! Reference source not found.	QueryTrajectory hermes.moving_point, k number	tbMovingObject Entries	This is a function (actually operator) that acts as follows: given a trajevtory segments, it returns the k trajectory segments that are closer to it
mv_IncTrajectoryNNSearch	t_id integer, k number	—	Same as IncTrajectoryNNSearch but this time the results are stored in the mv_tbl to be later visualized on the map
mv_IncPointNNSearch	x number, y number, t1 tau_tll.d_timepoint_sec, t2 tau_tll.d_timepoint_sec, k integer	—	Same as IncPointNNSearch but this time the results are stored in the mv_tbl to be later visualized on the map
tb_Topological_Query	geom MDSYS.SDO_GEOMETRY, tp tau_tll.D_period_sec, mask varchar2	IDS	A function that returns the trajectory IDs that (enter, leave, enter/leave) a certain region within a given time period