Marios Vodas

Hermes

– Building an Efficient Moving Object Database Engine –

Department of Informatics
University of Piraeus

MSc Thesis

March 29, 2013

To my beloved parents – Thomas & Eleni

Preface

This thesis was prepared at the Department of Informatics of the University of Piraeus, in partial fulfillment of the requirements for acquiring the MSc degree in Advanced Information Systems. This study has been conducted from September 2012 to March 2013 under the supervision of Professor Yannis Theodoridis.

Piraeus, March 2013 *Marios Vodas*

Acknowledgements

I would like to thank Yannis Theodoridis and Nikos Pelekis for their insightful supervision during the writing of my thesis. I would also like to thank my colleagues Despina Kopanaki, Panagiotis Tampakis, Nikos Giatrakos, Stelios Sideridis, Giannis Kostis, and Michalis Basios for our cooperation in the lab. Last but not least, I would like to thank Maria Zogkou, Evgenios Vodas, and Giorgos Festas for all their support and understanding. Finally, my acknowledgements to IMIS Hellas (www.imishellas.gr) for kindly providing the AIS dataset which allowed me to test and demonstrate my work.

Abstract

This thesis describes “Hermes”, a MOD developed as an extension of PostgreSQL. Hermes architecture is presented to show its general context of use through an SQL interface. The data types that comprise the data model are presented in three categories spatio-temporal, temporal, and spatial. A spatio-temporal 3D-Rtree index structure is proposed along with a collection of operators that get support from it. Also, a showcase on an AIS dataset is presented to indicate some of the querying capabilities of Hermes. There are two clustering algorithms implemented on Hermes that provide advanced functionality to the framework. Finally, the maturity of Hermes is shown by the fact that it is used in a real-world web application that offers spatio-temporal querying functionality to its users.

Contents

1

Introduction

Mobility has become a daily concept for people who own a GPS enabled electronic device, particularly because of the universal adoption of smartphones and the advanced capabilities they offer to their users. The main factors that drive the spreading of Mobility are the providers of geo-web services. Companies like Google and Nokia have developed an extensive stack of geo-web services that tightly integrate with mobile devices making it simple for third-party developers and organizations to build custom applications for the public. Open source initiatives like OpenStreetMap enhance the data provided by these web services with the qualities of openness of the data, community driven development and open feedback contribution. Organizations and the public sector have already recognized the benefits of incorporating Mobility data, originating either directly from their members or from third party collaborators, into their information systems. It is worth to mention the "Digital Government" key initiative [2] of the "US Office of E-Government Information Technology". In this initiative it is stated how the US government will place Mobility in the heart of the next generation digital services it will provide to its citizens. The expected benefits from such a move will span from QoS to economical.

A moving object can be any geometry (point, line, area, etc.) in the geographical space that changes its positions or even its shape in time. An object's (might that be a pedestrian, car, ship, etc.) movement implies the unification of space and time dimensions under one domain. Because an object moves constantly and continuously it is impossible to capture its movement in every detail; thus we only keep samples of the movement. In practice, that means that we have GPS position reports at a varying sampling rate of seconds or even minutes. These recordings only provide the object's position at a specific timestamp, and it is our responsibility to decide what happens between two consecutive samples. A trajectory is defined by the kind of interpolation we assume that happens between two sample points and the way we choose to represent and store a trajectory. The most common interpolation method is linear interpolation where two consecutive sample points are connected with

a straight line. Another method is the Bezier-curves which requires the speed and direction at each point in order to draw a smoother not straight line between the points. The first method assumes constant speed between the points and the trajectory contains many abrupt changes both in speed and direction in contrast to the second one. The simplicity of the linear interpolation makes it very fast and easy to use and that is the main reason for its wide adoption.

Mobility, when looking at its technological aspects, spans from data modeling to indexing and from data mining to visualization. This MSc thesis introduces a MOD prototype called Hermes (http://hermes-mod.java.net - http://infolab.cs.unipi.gr/hermes), which was developed on top of PostgreSQL ORDBMS through utilizing its extension interfaces. Hermes has a long history of publications since 2006 [7,9,10]. This work resulted in a brand new and improved design and implementation of the Hermes specification that has the potential of real world application as a result of improvements in scalability and efficiency. This implementation is available for downloading for research and educational purposes under Hermes license at URL: http://hermes-mod.java.net/Installer/ThesisVersion/.

After a short presentation of related work (chapter 2), we start (in chapter 3) by explaining the architecture of Hermes, its connection with PostgreSQL and the general context it can be used with. Then (in chapter 4), we describe the components of Hermes categorized according to the aspects of Mobility (data modeling, indexing, visualization, data mining, etc.). The last part (chapter 5) includes a showcase in a real-world trajectory dataset and case studies that Hermes has been tested and deployed with success.[1]

[1] Throughout this text there are example code segments that will use sample data (ship trajectories) to illustrate the capabilities of Hermes. A detailed description of the dataset can be found in chapter 5.

2

Related Work

The HERMES system [10] was previously developed on Oracle DBMS and could query continuously moving objects. Oracle's spatial data types are used along with the TAU Temporal Literal Library Data Cartridge (TAU-TLL) types in order to construct the moving object data types (figure 2.1). In HERMES a trajectory is modelled as a sequence of segments called unit functions.

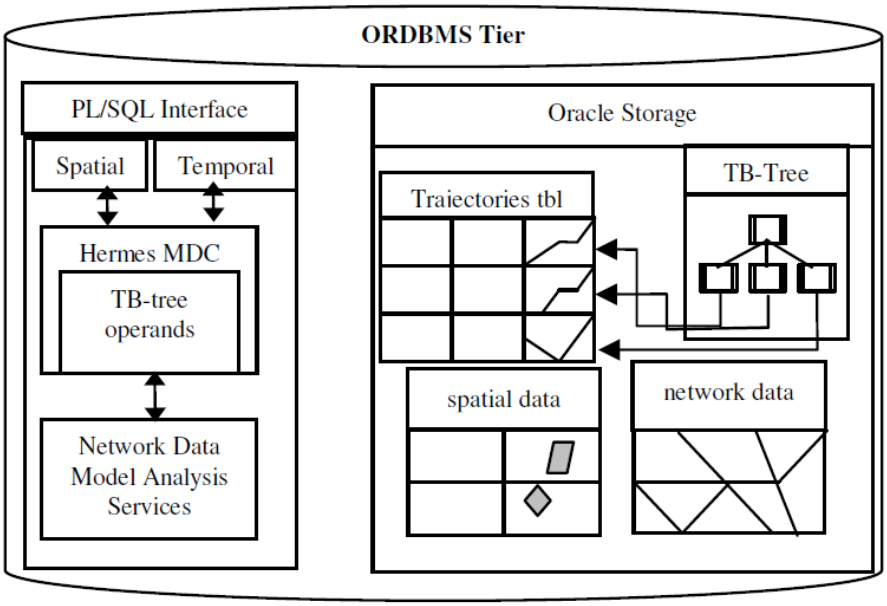


Fig. 2.1: HERMES architecture on Oracle [7]

The SECONDO system [5] is a state of the art moving objects database which represents trajectories using the sliced representation method where

the trajectory is segmented into fragments and each fragment can be viewed as a simple function of time over geographical space. SECONDO is based on Berkeley DB for storage management and is comprised of three main subsystems (figure 2.2) namely the kernel, optimizer, and graphical user interface (GUI). The kernel provides an extensible algebra and query processing on that. The optimizer supports an SQL-like language and is used for query optimization. The GUI is used to visualize the different data types and models that SECONDO supports. There are about thirty available algebras in SECONDO and the most important ones are the *Standard-Algebra* (basic data types like numbers and strings), the *Relational-Algebra*, the *Spatial-Algebra* (geometries such as points, lines, regions), and the *Temporal-Algebra* (moving objects).

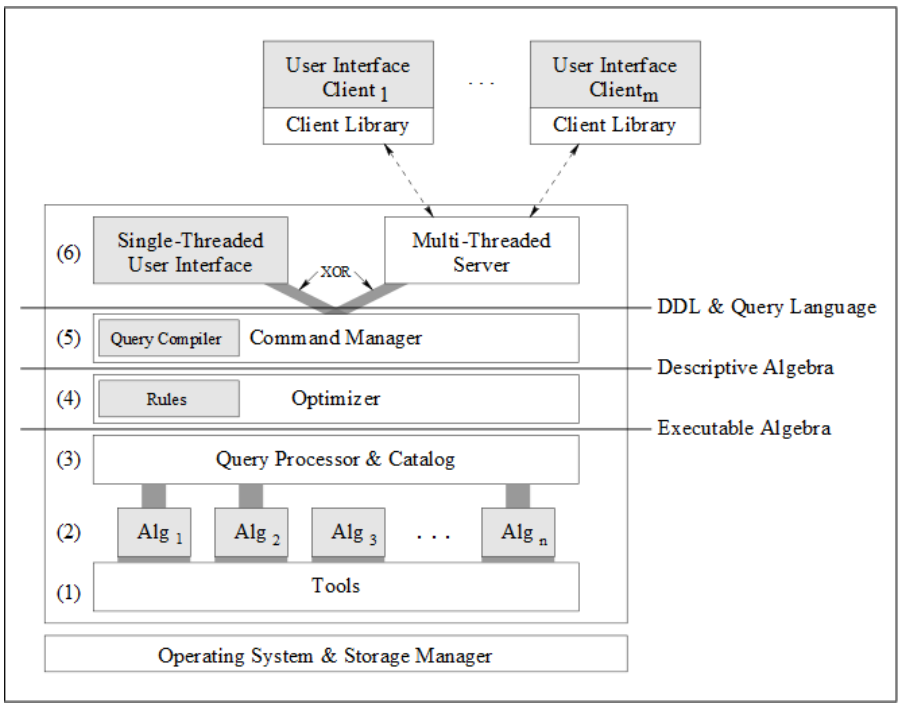
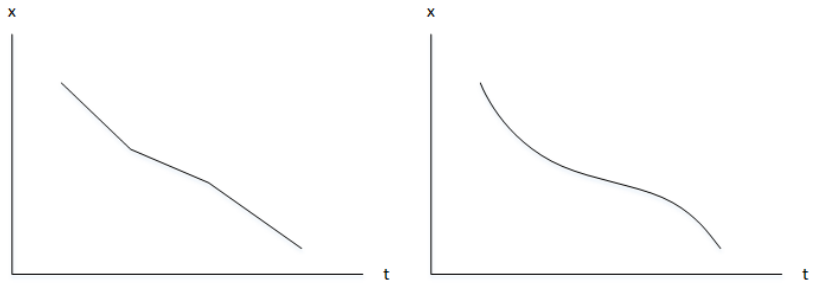


Fig. 2.2: SECONDO architecture [5]

3

Hermes MOD Architecture Principles

Hermes builds on PostgreSQL's underlying functionality and extends it to support trajectory data. In its core Hermes contains a few data types both spatial and temporal but also a unification of those (i.e. spatio-temporal data types). All of its functionality and features rely and utilize that model. In Hermes a trajectory is a sequence of sampled time-stamped locations (p_i, t_i) where p_i is a 2D point (x_i, y_i) and t_i is the recording timestamp of p_i. We can choose from two alternatives for interpolating the position of an object between two sampled points (figure 3.1). The first and most common one is to assume constant speed linear interpolation and the second one is to consider constant acceleration. The first option is the most popular within the spatio-temporal database community whilst the second is closer to a real-world model.



(a) Constant Speed (b) Constant Acceleration

Fig. 3.1: Alternative Interpolation Techniques

The constant speed option entails simple and low cost calculations to interpolate the position between two consecutive sample points and is preferable in datasets that the speed of the objects doesn't change often (e.g. the move-

ment of ships). On the other hand, the constant acceleration option entails a computational cost higher than that of the first though it is more suitable in datasets that contain an underlying road network, thus the speed of the objects changes frequently. Having a more accurate interpolation technique allows to do more advanced computations such as emissions and consumption or even collision detection.

$$\Delta s = v_i \cdot \Delta\tau, v_i \text{ is constant during } [\tau i - 1, \tau i] \tag{3.1}$$

$$\Delta s = {}^1\!/_2 a_i \cdot \Delta\tau^2, a_i \text{ is constant during } [\tau i - 1, \tau i] \tag{3.2}$$

Hermes provides an SQL interface comprised of types, functions and operators that the user can combine in order to construct data and perform calculations on them. This SQL interface is accessible through a series of protocols such as JDBC, ODBC and practically any other protocol that has the ability to connect to a standard PostgreSQL server. Keeping that in mind, Hermes can be utilized within web frameworks and web services in order to build applications that are backed by Hermes.

3.1 Data Types

In this section, the data types supported by Hermes are presented in three categories: temporal, spatial and spatio-temporal. In summary, they are listed in figure 3.2.

3.1.1 Temporal Data Types

Temporal data types are those types that model only the temporal dimension of Mobility.

The building block data type in this category is the **timestamp without time zone** (or just **Timestamp**) which is not really a data type introduced by Hermes but is built in to PostgreSQL and Hermes encapsulates it in its data model. An example of a timestamp is '2012-09-20 08:05:46'. Another encapsulated data type is the **Interval** which is used to store information like '1 second' or '05:30:5' (which means 5 hours 30 minutes and 5 seconds) hence contains a temporal quantity. One potential limitation of interval is that on cases where it was produced from a subtraction of two timestamps it won't keep the original timestamps from which it was computed. That limitation led to the development of some custom temporal types in Hermes.

Code Sample

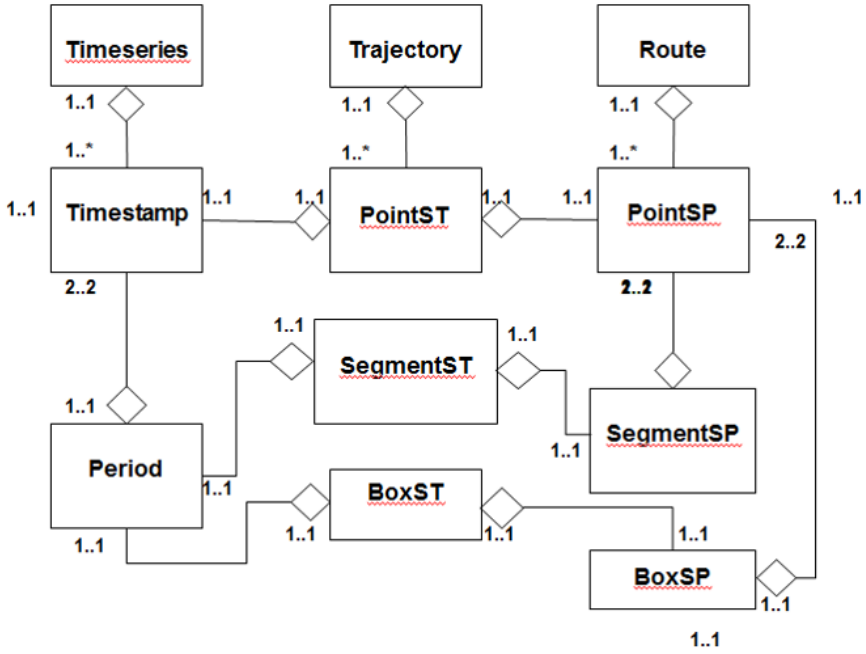


Fig. 3.2: Hermes Data Types

```
1  --1st Query--
2  SELECT '2008-12-31 19:29:30'::Timestamp;
3
4  --2nd Query--
5  SELECT '00:00:01'::Interval;
```

Code Explanation

These two queries have the potential to introduce some basic programming concepts of PostgreSQL. Notice that "SELECT" is used to run a query even if that doesn't involve a SQL table. It's like saying to PostgreSQL run this code. Each command is terminated by a ";" semicolon.

PostgreSQL can cast a string to any datatype. In the 1^{st} query a timestamp is given as a string and is cast using "::" to the timestamp datatype. This is also done for another datatype (i.e. interval) in the 2^{nd} query.

It is important to note that the string that we cast to a datatype must be conformant to the datatype. For example, if we cast the string in the 1^{st} query to an interval it will throw an error.

Hermes temporal types

Hermes introduces the **Period** temporal data type. A period is comprised of two timestamps (i, e), meaning i-nitial and e-nding, thus an interval can be computed from of a period.

Code Sample

```
1 SELECT ''(''2008-12-31 19:29:30'', ''2009-01-02 17:10:06'')''
    ::Period;
```

Code Explanation

What should be noted here is that when a single quote is needed in a string it is required to put one single quote before that. Other than that, a period object is formed by providing the two timestamps it needs and by enclosing them with parenthesis (notice that the first and last character of the string is a parenthesis).

3.1.2 Spatial Data Types

Hermes is designed to work with data in the Euclidean space, which means that in order to define a position we need x, y coordinates measured in meters. All the underlying mathematical procedures operate in the Euclidean space as well. In GIS-related fields Projected coordinate systems are widely used and since they essentially are a form of a Euclidean space they are fully compatible with Hermes approach.

Usually the data will be in the World Geodetic System (WGS) where a position is defined by a longitude and a latitude measured in decimal degrees. Later, it will be described how we can transform from (lon, lat) to (x, y), and the opposite, in order to feed Hermes with WGS data.

Spatial data types are those types that model only the spatial dimension of Mobility. The building block data type in this category is the **PointSP** and is comprised of (x, y) coordinates in meters. Another spatial data type is **SegmentSP** and comprises of two PointSP (i, e) components where “i” is the initial point and “e” is the ending point (figure 3.3).

The previous spatial types do not have a surface in contrast with the next one that has. This very important spatial type, with a surface is the **BoxSP** and is comprised of two PointSP (l, h) components where “l” is the low left point and “h” is the high right point as illustrated in Figure 3.4.

Fig. 3.3: Spatial Segment

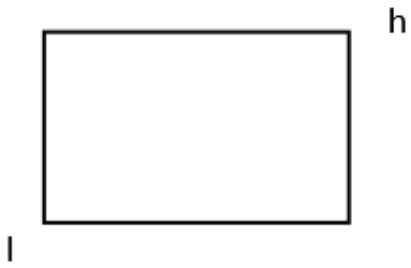


Fig. 3.4: Spatial Box

Code Sample

```
1  --1st Query--
2  SELECT '(2337709, 4163887)'::PointSP;
3
4  --2nd Query--
5  SELECT '((2337709, 4721671), (3228259, 4721671))'::
       SegmentSP;
6
7  --3rd Query--
8  SELECT '((2337709, 4163887), (3228259, 4721671))'::BoxSP;
```

Code Explanation

It should be noted here that coordinates are measured in meters for the reasons explained earlier.

3.1.3 Spatio-Temporal Data Types

Spatio-Temporal data types are those types that model both the temporal and spatial dimension of Mobility in a unified manner. In these types we distinguish the temporal and spatial dimension in the following way:

- A spatio-temporal point **PointST** is comprised of a Timestamp "t" and a PointSP "sp".
- A spatio-temporal segment **SegmentST** is comprised of two PointST components "i" and "e".
- A spatio-temporal box **BoxST** is comprised of a Period "t" and a BoxSP "sp". Consider a BoxST as a cube in 3D space.

Code Sample

```
1  --1st Query--
2  SELECT ''(''2008-12-31 19:29:30'', (2337709, 4163887))''::
       PointST;
3
4  --2nd Query--
5  SELECT ''((''2008-12-31 19:29:30'', (2337709, 4721671)), (''
       2009-01-02 17:10:06'', (3228259, 4721671))''::SegmentST;
6
7  --3rd Query--
8  SELECT ''((''2008-12-31 19:29:30'', ''2009-01-02 17:10:06'')
       , ((2337709, 4163887), (3228259, 4721671)))''::BoxST;
```

Code Explanation

Each component of the object is defined within parenthesis. The temporal component always comes first and the spatial comes second.

3.1.4 The "Trajectory" type

Hermes defines a trajectory through its **Trajectory** data type which is an object containing a sequence of spatio-temporal points ordered in time. It is a variable length type in contrast to previous types and is comprised of a sequence of PointST objects ordered by time. This type marks a different approach to storing/handling/manipulating Mobility data, meaning that here we look at the movement of an object as whole and not segmented in smaller parts i.e. segments.

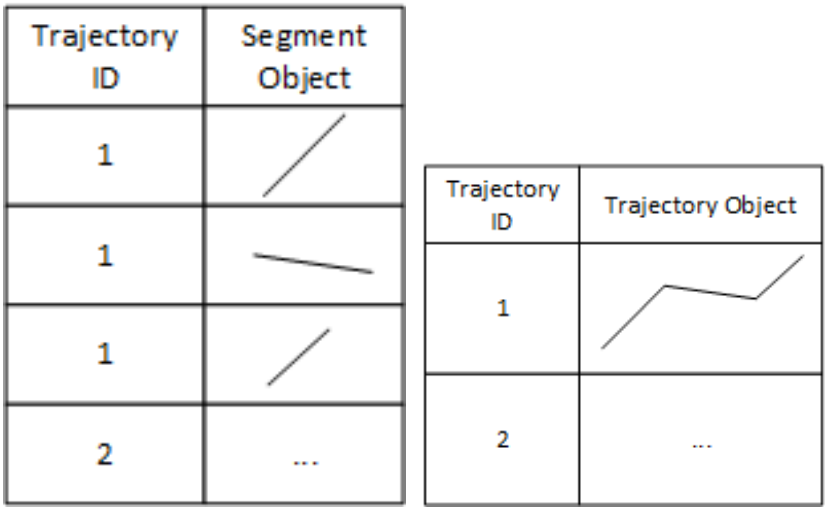
The distinction between the two alternative modeling proposals for storing a trajectory is illustrated in figure 3.5: in figure 3.5a a trajectory consists of a set of SegmentST objects; in figure 3.5b a trajectory consists of a single Trajectory object.

Code Sample

```
1  SELECT ''(''2008-12-31 19:29:30'', (2337709, 4163887))~(''
       2008-12-31 19:29:35'', (2337710, 4163890))~(''
       2008-12-31 19:29:41'', (2337715, 4163893))''::Trajectory
       ;
```

Code Explanation

Spatio-temporal points are delimited with " ".



(a) Segment based modeling (b) Trajectry based modeling

Fig. 3.5: Alternative Interpolation Techniques

3.2 Database Schema

This section describes the structure of the database that was designed to be able to host multiple datasets. Having defined the data types the main missing feature is methods for loading structured data into the database. This is solved by developing a metadata infrastructure, essentially a catalog, which will help us host multiple datasets with different characteristics. For example, it is possible to host two diverse datasets one of moving vehicles and another one of traveling vessels in the same database.

That metadata infrastructure takes the form of a table named **dataset**. So, each dataset in this table has a unique identifier and a unique short name e.g. [1, 'imis'] or [2, 'milan']. Most spatio-temporal datasets are in the form of [objectID, trajectoryID, t, lon, lat] where "objectID" is the identifier of the object, "trajectoryID" is the identifier of the trajectory for that object, "t" is the UTC (Coordinated Universal Time) timestamp at which it was recorded and "lon" and "lat" are degrees of longitude and latitude in WGS 84 Geographic Coordinate System. "objectID" and "trajectoryID", normally, are combined to form the unique identifier of a trajectory in a specific dataset.

3.2.1 Coordinate Transformation

Hermes works on the Euclidean space, meaning it needs degrees (lon, lat) to be transformed into meters (x, y). For this transformation the Geographic to/from Topocentric conversion (EPSG 9837) [4] was implemented. According to this specification, to do the transformation we only need a reference

point (lon, lat) which in (x, y) will be regarded as (0, 0), i.e. the Cartesian center. So, the closer a position is to this reference point the more accurate the transformation will be. Thus, a dataset must have a reference point for transformations.

Code Sample

```
1  --1st Query--
2  SELECT ll2xy('(20.999999, 35.000044)'::PointLL, '(23.63994,
       37.9453)'::PointLL);
3  --Result of 1st Query
4  '(-240909.991094767, -323271.482666732)'
5
6  --2nd Query--
7  SELECT xy2ll('(-240909.991094767, -323271.482666732)'::
       PointSP, '(23.63994, 37.9453)'::PointLL);
8  --Result of 2nd Query
9  '(20.999999000041, 35.0000440000481)'
```

Code Explanation

Here we notice one more data type, **PointLL** composed of (lon, lat), which is considered auxiliary because it is only used to represent points in longitude and latitude so that we can later transorm them to meters.

The function ll2xy(point, reference point) returns the point transformed to a PointSP.

The second query shows the opposite operation where it should be noticed that there is a slight loss of precision in the result w.r.t. the input in the 1^{st} query. This phenomenon is generally common in coordinate system transformations.

3.2.2 Metadata Catalog

The **dataset** table that was mentioned before is the metadata catalog and each of its row corresponds to a dataset. Its structure is as follows:

- **id** is an auto incremented integer column that is the primary key of the table. Each dataset hosted is given an id.
- **name** is a text column that contains a unique short name of the dataset.
- **name_long** is a text column that contains a human friendly name of the dataset.
- **parent_dataset** is a foreign key to another existing row in the table that, when it is not NULL, indicates a parent-child relationship between the datasets.
- **parent_dataset_notes** is a text column that contains notes on the parent-child relationship of the dataset.

- **local_ref_poi** is a PointLL column that is the reference point for coordinate transformation.
- **SRID** is an integer column that contains the EPSG code of the projected reference system in which the dataset is stored in Hermes. Note that if we give a value for local_ref_poi then SRID will have to be NULL and vice versa.
- There also some statistics about the dataset that can be kept in this table:
  - bounds of the dataset (tmin, tmax, lx, ly, hx, hy, llon, llat, hlon, hlat).
  - centroid of the dataset (centroid_x, centroid_y, centroid_lon, centroid_lat).
  - number of objects / trajectories / points / segments.
  - minimum /average / maximum number of points per trajectory / trajectory duration / trajectory length.
- **notes** is a text column that contains arbitrary notes on the dataset.

3.2.3 Loading a Dataset

Each dataset consists of three tables. Each table's name begins with the dataset's name followed by a suffix:

- **_obj**: this table hosts the objects that exist in the dataset and contains one column:
  - **obj_id** the unique identifier of the object.
- **_traj**: this table hosts the trajectories of the objects and contains three columns:
  - **obj_id** is a foreign key to _obj table.
  - **traj_id** is an identifier for the particular trajectory of the object.
  - **traj** contains an object of type Trajectory (optional, see below).
- **_seg**: this table will host the segments of the trajectories and contains four columns:
  - **obj_id** is a foreign key to _obj table.
  - **traj_id** is a foreign key to _traj table.
  - **seg_id** is an identifier for the particular segment of the trajectory.
  - **seg** contains a segment of a trajectory, an object on type SegmentST.

The objects table and the trajectories table must have data in contrast with segments table. The "traj" column in trajectories table could be empty if trajectories are stored in the segments table (that is why it is not bold in figure 3.6). It is always possible to build a trajectory object from its corresponding segments that we can find in the segments table on the fly using aggregate functions. That allows us to use advanced methods that Hermes provides for its "Trajectory" type.

The entire database schema presented earlier is illustrated in figure 3.6. The indexing illustrated in this image refers to a 3D-Rtree that can be built either on the segment or trajectory tables. Building the tree on trajectory objects is prone to the dead space that the minimum bounding box of the

trajectory usually introduces. Nonetheless, Hermes supports building a 3D-Rtree both on segment and trajectory objects.

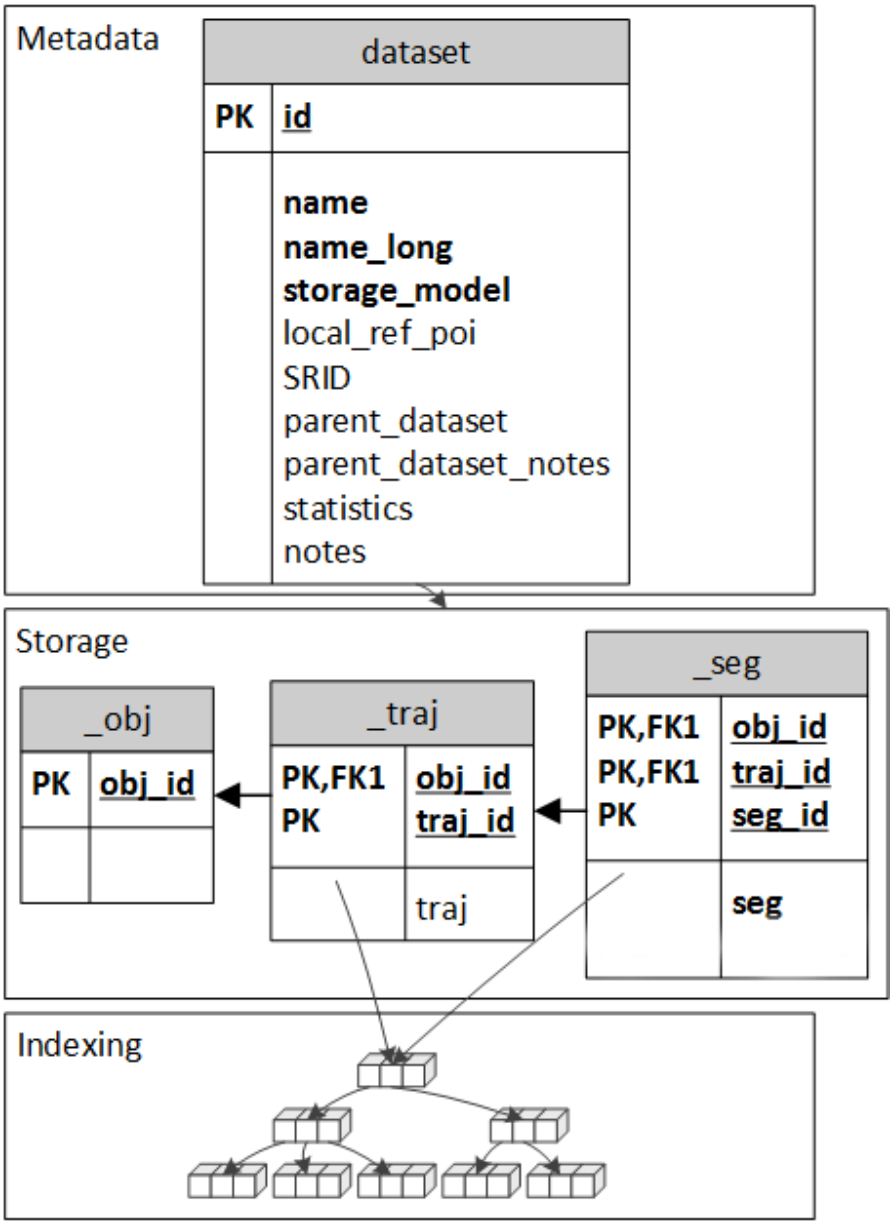


Fig. 3.6: Database Schema

Hermes Loader

In Hermes, a function named Loader can be called, with some parameters of the dataset, in order to fill the above tables. The most common format for Loader is the CSV. In this format the loader is fed with a csv file that has the structure: objectID, trajectoryID, t, lon, lat. The file has to contain a header. An example of such a CSV file is illustrated here:

```
1  objectID,trajectoryID,t,lon,lat
2  201100024,1,2009-01-02
       08:54:07,24.609728324369,38.013503319816
3  201100024,1,2009-01-02
       08:54:25,24.6094016577037,38.0127699864845
4  201100024,1,2009-01-02
       08:55:06,24.6086749910399,38.011116653155
5  201100024,1,2009-01-02
       08:55:56,24.6076299910435,38.0092066531597
6  201100024,1,2009-01-02
       08:56:16,24.6071983243782,38.0084733198281
7  201100034,1,2009-01-02
       04:19:26,23.1092366579214,38.5853616531322
8  201100034,1,2009-01-02
       04:19:36,22.9272199909328,38.8922416526431
9  201100034,1,2009-01-02
       04:19:45,23.0359933243564,38.7788549861265
10 201100034,1,2009-01-02
       04:19:55,22.9355449909622,38.868204986019
11 201100034,1,2009-01-02
       04:20:05,23.0638616578755,38.6383849863914
12 ...
```

```
1  INPUT: CSV file
2  OUTPUT: Hermes tables
3
4  1. Bulk load the csv file in a table ‘‘pos’’.
5  2. Order table ‘‘pos’’ by obj\_id ASC, traj\_id ASC, t ASC
6  3. Iterate through the ordered result and for each position
7      a) Check if this position belongs to the previous
           trajectory
8          i. If yes then form a segment with the previous
               position and this position and insert it into
               the segments table. At the same time insert
               into the trajectories table if the trajectory
               is not already there.
9          ii. If not then start a new trajectory and continue
               to the next position.
```

Listing 3.1: Steps of the Hermes CSV loader

Code Sample

```
1 SELECT HLoader('imis', 'IMIS 3 Days');
2 SELECT HLoaderCSV_II('imis', 'imis3days.txt');
3 SELECT HDatasetsOfflineStatistics('imis');
4 CREATE INDEX ON imis_seg USING gist (seg);
```

Code Explanation

In this sample code we notice that there is a function and a table with the same name "HLoader". The function loads the dataset by taking into account the information / parameters that we pass to the function but also the ones that are present in the table. Because loader can be extended to support more formats beyond CSV this is why "HLoader" table exists to hold the specific parameters for that extension. Every loader though must have the parameters that are passed in the function since they are common to any dataset and loader combination.

3.3 Indexing with pg3D-Rtree

Indexing mechanisms in DBMS's for Spatio-Temporal data lack a unified structure for space and time dimensions [12]. This leads to maintaining multiple indexes, one for each dimension, and adds complexity to queries.

This section presents the Rtree-like pg3D-Rtree indexing mechanism that was developed on top of GiST (Generalized Search Tree) interface of PostgreSQL and can be applied on SegmentST and Trajectory types. GiST [6] is a balanced, tree-structured access method, which acts as a base template in which to implement arbitrary indexing schemes [1]. B-trees, R-trees and many other indexing schemes can be implemented in GiST.

pg3D-Rtree implements 8 functions as specified by GiST so that Spatio-Temporal data can be indexed in a unified Rtree-like structure.

In the next paragraphs the most important functions that define our indexing mechanism are described. To facilitate the discussion, let's think of BoxST as a Spatio-Temporal Minimum Bounding Box MBB(X_l, Y_l, T_i, X_h, Y_h, T_e) where (X_l, Y_l, T_i) is the low-left and (X_h, Y_h, T_e) is the high-right 3-dimensional point of MBB.

Consistent(E, q) given an entry E and a query q this function supports a variety of operators.

- Spatial operators:
  - Overlaps
  - Contains
  - On the Left/Right/Above/Below
- Temporal operators:

  - Overlaps
  - Contains
  - Before/After
- Spatio-Temporal operators:
  - Overlaps
  - Contains

Compress(seg/traj) is used to transform the next to insert segment or trajectory in the structure to an MBB.

Union(E1, , En) aggregates the MBB's of the input entries into one single MBB that contains all of them. This new MBB will become the parent entry of those entries in the tree.

Penalty(E1, E2) given entry E1 and a new entry E2 this function calculates the "cost" of inserting E2 under E1. The equation to compute this "cost" is Size(Union(E1, E2)) - Size(E1), where size is the volume of the box. The lower the cost the more possible it will be that E2 is inserted under E1.

Picksplit is responsible for splitting the entries of a node in the tree that has overflowed. Hermes implements the "New Linear Node Splitting Algorithm for R-trees" algorithm, proposed by Anf and Tan [3].

GiST is not suitable for all variations of R-Trees. For example, TB-Tree [11] cannot be implemented using GiST because it requires either a linked-list between leaf nodes or a leaf node to contain segments from only one trajectory. This is because GiST doesn't support custom interconnections between nodes (apart from parent-child) and penalty method cannot guarantee that a segment will be inserted to a specific node.

4

Hermes MOD Functionality

The index mechanism that was described earlier is utilized through a set of spatio-temporal operators listed in table 4.1. In the rest of this paragraph we elaborate on the functionality of Hermes in terms of SQL functions and the algorithms they implement.

4.1 Methods

The following methods can be used interchangeably either on segment or trajectory objects (recall the discussion about alternative models of storage in 3.1.4), thus each time they assume a different interpolation model. In the case of segments a uniform linear motion model is assumed, in fact it is the only assumption we can make about the segment since we have no other information about its previous state. On the other hand, when the function is called on a trajectory object the non-uniform linear motion with constant non-zero acceleration between two points is used. An assumption is made on the initial speed of the object: the speed of the object at the first point of the trajectory is considered equal to the speed at the second point, in other words, the acceleration at the first segment of the trajectory is zero.

In the following, there is example code segments for the segment model mainly.

4.1.1 average speed

This function takes a segment or a trajectory as a parameter and returns the average speed.

Code Sample

```
1 --1st Query--
2 SELECT averageSpeed('(''1970-1-1 0:0:0'', ''1970-1-1 0:0:4
    ''), ((0, 0), (0, 4)))'::SegmentST);
3 --Result of 1st Query
4 "1" -m/s
5
6 --2nd Query--
7 SELECT averageSpeed('(''1970-01-01 00:00:00'', (0, 0))~(''
    1970-01-01 00:00:01'', (0, 1))~(''1970-01-01 00:00:02''
    , (0, 2))~(''1970-01-01 00:00:03'', (0, 4))'::
    Trajectory);
8 --Result of 2nd Query
9 "1.33" -m/s
```

Code Explanation

The result is measured in meters per second. The 2^{nd} query will calculate the average speed of the trajectory by looking only at the first and last point and since it looks only at two points it assumes a zero acceleration between them.

4.1.2 at instant

This function takes a segment and a timestamp as parameters and returns the point where the object was found at the given timestamp.

Code Sample

```
1 --1st Query--
2 SELECT atInstant('(''1970-1-1 0:0:0'', ''1970-1-1 2:0:0'')
    , ((0, 0), (2, 2)))'::SegmentST, '1970-1-1 1:0:0'::
    Timestamp);
3 --Result of 1st Query
4 "(1, 1)" -Of type PointSP
```

Code Explanation

Notice that the result is the middle of the segment.

4.1.3 at point

This function takes a segment and a point as parameters and returns the timestamp at which the object was found at the given point.

Code Sample

```
1 --1st Query--
2 SELECT atPoint('(''1970-1-1 0:0:0'', ''1970-1-1 2:0:0''),
      ((0, 0), (2, 2)))'::SegmentST, '(1, 1)'::PointSP);
3 --Result of 1st Query
4 "1970-01-01 01:00:00" -0f type Timestamp
```

Code Explanation

The point has to be on the segment, otherwise the function returns NULL.

4.1.4 at period

This function takes a segment and a period as parameters and returns the part of the segment that corresponds to the given period.

Code Sample

```
1 --1st Query--
2 SELECT n, s, p FROM atPeriod('(''1970-1-1 0:0:0'', ''
      1970-1-1 4:0:0''), ((0, 0), (4, 4)))'::SegmentST, '(''
      1970-1-1 1:0:0'', ''1970-1-1 2:0:0'')'::Period);
3 --Result of 1st Query
4 2, "((1, 1), (2, 2))", NULL
```

Code Explanation

The segment might have only one timestamp in common with the period so in that case the function returns a point instead of a segment. This is why the function returns three columns (n, s, p) where n is the number of common points, s is the segment within the period (if n is 2) and p is the point that the segment was within the period (if n is 1).

4.1.5 at box

This function takes a segment and a box as parameters and returns the part of the segment that resides within the box.

Code Sample

```
1 --1st Query--
2 SELECT n, s, p FROM atBox('(''1970-1-1 0:0:0'', ''1970-1-1
       4:0:0''), ((0, 0), (4, 4)))'::SegmentST, '((1, 1), (2,
       2))'::BoxSP);
3 --Result of 1st Query
4 2, "(''1970-1-1 1:0:0'', ''1970-1-1 2:0:0'')", NULL
```

Code Explanation

The n, s, and p have the same meaning as in atPeriod.

4.1.6 intersection

This function takes a spatial segment and a spatial box as parameters and returns the intersection of the segment with the box. There is also a third optional parameter, called "solid", that when is set to false the function returns NULL when the segment is fully contained within the box without touching the perimeter.

Code Sample

```
1 --1st Query--
2 SELECT n, s, p FROM intersection('((0, 0), (4, 4))'::
      SegmentSP, '((1, 1), (2, 2))'::BoxSP);
3 --Result of 1st Query
4 2, "((1, 1), (2, 2))", NULL
```

Code Explanation

The n, s, and p have the same meaning as in atPeriod and atBox.

4.1.7 enter-leave points

The enter_leave function finds the points where the object entered or left a specific region. It takes an array of segments and a box as parameters.

Code Sample

```
1 --1st Query--
2 SELECT enterPoint, leavePoint FROM enter_leave(
      array_of_segments[], box_area);
```

Code Explanation

The function returns two columns one for the enter and one for the leave point. If one of them doesn't exist then it returns NULL to the corresponding column.

4.1.8 trajectory (aggregate function)

This is an aggregate function (meaning that it is used with a GROUP BY clause) that takes segments ordered by time as input and returns a trajectory object.

Code Sample

```
1  --1st Query--
2  SELECT trajectory(seg ORDER BY t_i(seg))
3  FROM
4      (
5          SELECT '(('1970-1-1 0:0:0'', ''1970-1-1 0:0:1''),
               ((0, 0), (1, 1)))'::SegmentST AS seg
6          UNION
7          SELECT '(('1970-1-1 0:0:1'', ''1970-1-1 0:0:2''),
               ((1, 1), (4, 4)))'::SegmentST
8          UNION
9          SELECT '(('1970-1-1 0:0:2'', ''1970-1-1 0:0:4''),
               ((4, 4), (4, 6)))'::SegmentST
10     ) AS segs;
11 --Result of 1st Query
12 "('1970-01-01 00:00:00', (0, 0))~('1970-01-01 00:00:01',
       (1, 1))~('1970-01-01 00:00:02', (4, 4))~('1970-01-01
       00:00:04', (4, 6))" -Of type Trajectory
```

Code Explanation

The query uses UNION keyword in order to build a result consisting of three rows. That result is named "segs" and is considered a table of segments. The function trajectory receives all the segments one by one in ascending time order and builds the trajectory object. This trajectory can be passed as an argument to more advanced methods that are discussed in the next paragraph.

4.2 Basic Operators

In this section, we present the index-supported operators of Hermes. The operators rely on the methods of the previous section in order to be implemented.

The && (overlaps) operator checks if the segment has any common points (or common timespan, in the case of Period) with the object in the right of the operator. When the object in the right is of spatio-temporal type interpolate is used to find if both the spatial and temporal components interact.

The $\sim$ (contains) operator checks if the segment contains the object in the right argument. When the right argument is PointST then interpolation takes place in order to find the position the segment was at the timestamp that PointST contains and then if the position is the same as the position that PointST contains the operator returns true.

The @ (contained) operator checks whether the segment is contained within a BoxSP (or Period, when we only check time) allowing it to touch

Table 4.1: 3D-Rtree operators

Symbol Operation	Returns	Left Argument	Right Argument
&& overlaps	boolean	SegmentST	Period, BoxSP, SegmentSP, BoxST, SegmentST
~ contains	boolean	SegmentST	Timestamp, Period, PointSP, PointST
@ contained	boolean	SegmentST	Period, BoxSP, BoxST
@! contained properly	boolean	SegmentST	BoxST
-< within distance	boolean	SegmentST	RangeSP, RangeST
<-> distance	number	SegmentST	Timestamp, Period, PointSP, SegmentSP, BoxSP

the perimeter of the box. The @! (contained properly) operator differentiates in that it doesn't allow the segment to touch the perimeter (thus fully contained).

The -<(within distance) operator checks whether the distance of the segment from the center of the RangeSP is less than the radius of the RangeSP object. In the case where the right argument is a RangeST interpolation takes place before evaluating the spatial distance. Specifically, atPeriod method is called on the segment and the Period (Period is the temporal quantity that is represented in the RangeST object).

The <->(distance) operator returns a number, in contrast to the previous operators that return a boolean value, and shows the distance in seconds or meters from the SegmentST to the right argument. If the right argument is a temporal type the operator returns distance in seconds whereas if the argumment is a spatial type it returns in meters.

4.3 Similarity Library

Measuring the similarity/distance between trajectories is not straightforward mainly because we need to take into account the temporal dimension. There are many proposed measures in the bibliography and Hermes implements some of the state of the art methods in its similarity measures library.

The trajectory similarity functions that are implemented in Hermes include: Manhattan, Euclidean, Tchebycheff, DISSIM, DTW, LCSS, EDR, ERP.

5

Showcase on IMIS AIS Dataset

This chapter is a showcase on AIS data provided by IMIS Hellas and exploits the capabilities of Hermes to efficiently query the data.

5.1 AIS Dataset Description

The "IMIS 3 Days" dataset spawns from "2008-12-31 19:29:30" to "2009-01-02 17:10:06" and contains positions reports for 933 ships. It is spatially constrained in the Aegean Sea and covers an area of 496736 km^2, from (21, 35)-lowest to (29, 39)-highest longitude-latitude point.

In figure 5.1 there is an overview of the dataset with the blue lines representing the trajectories of the ships, the green circles that contain a number indicate how many trajectories start around that area and the red ones how many end.

5.2 Querying AIS Dataset

Find the position of all ships on the second day of 2009, midnight.

```
1  SELECT DISTINCT ON (obj_id, traj_id) obj_id, traj_id,
2      atInstant(seg, '2009-01-02 00:00:00') AS position
3  FROM imis_seg
4  WHERE seg ~ '2009-01-02 00:00:00'::timestamp;
```

The index-supported "contains" operator ( ) filters the database. Then atInstant() method finds the exact location.

So, we can clearly see that greek seas are busy.

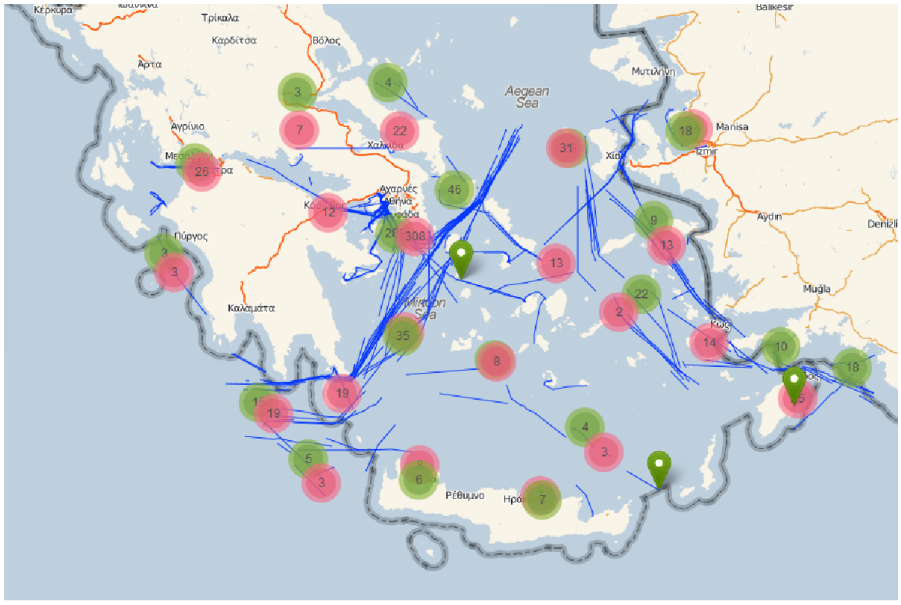


Fig. 5.1: Overview of the "IMIS 3 Days" dataset

What was the movement of ships in Heraklion port on New Years Eve 2009?

```
1  WITH TO_METERS AS (
2  SELECT
3      PointSP(PointLL(25.1325, 35.3407), HDatasetID('imis'))
           AS low,
4      PointSP(PointLL(25.1569, 35.3527), HDatasetID('imis'))
           AS high
5  ), SPT_WINDOW AS (
6  SELECT BoxST(
7          Period('2008-12-31 23:00:00', '2009-01-01 01:00:00'
               ),
8          BoxSP((SELECT low FROM TO_METERS), (SELECT high
               FROM TO_METERS))
9      ) AS box
10 )
11 SELECT obj_id, traj_id, (atBox(seg, (SELECT box FROM
       SPT_WINDOW))).s AS seg
12 FROM imis_seg
13 WHERE seg && (SELECT box FROM SPT_WINDOW)
14     AND (atBox(seg, (SELECT box FROM SPT_WINDOW))).n = 2;
```

The index-supported "intersects" operator (&&) filters the database. Then atBox() method finds the sub-trajectory in the range. This method could

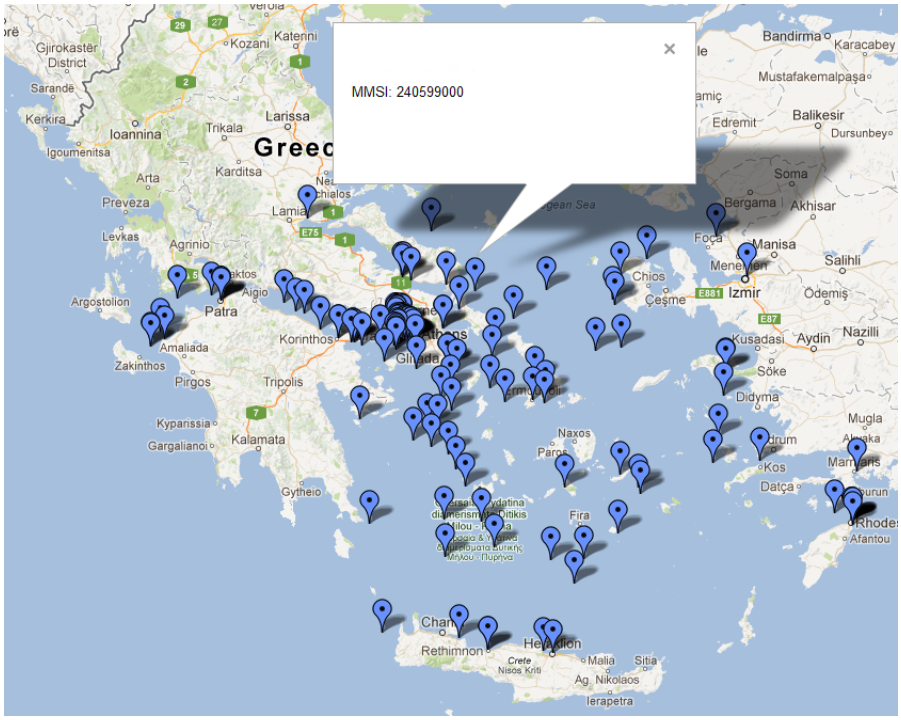


Fig. 5.2: TimeSlice Result

return a point instead of a segment under certain circumstance such as when the intersection between segment and the box is a point or when the segment and the period have only one common timestamp. This is why the method returns three properties the first of them is "n" which informs if the result is a point in which case the value would be 1 or if it is a segment in which case the value is 2. In case it is 0 then there is no intersection between the segment and the box. To get the point use the "p" property and for the segment the "s".

There is no movement since the year is about to change.

Find the ships that came closer than half nautical mile from an old lighthouse in Patrai.

```
1  SELECT DISTINCT obj_id, traj_id
2  FROM imis_seg
3  WHERE seg -< RangeSP(
4          round(nm2metres(0.5))::integer,
5          PointSP(PointLL(21.72565, 38.24513), HDatasetID('
               imis'))
```

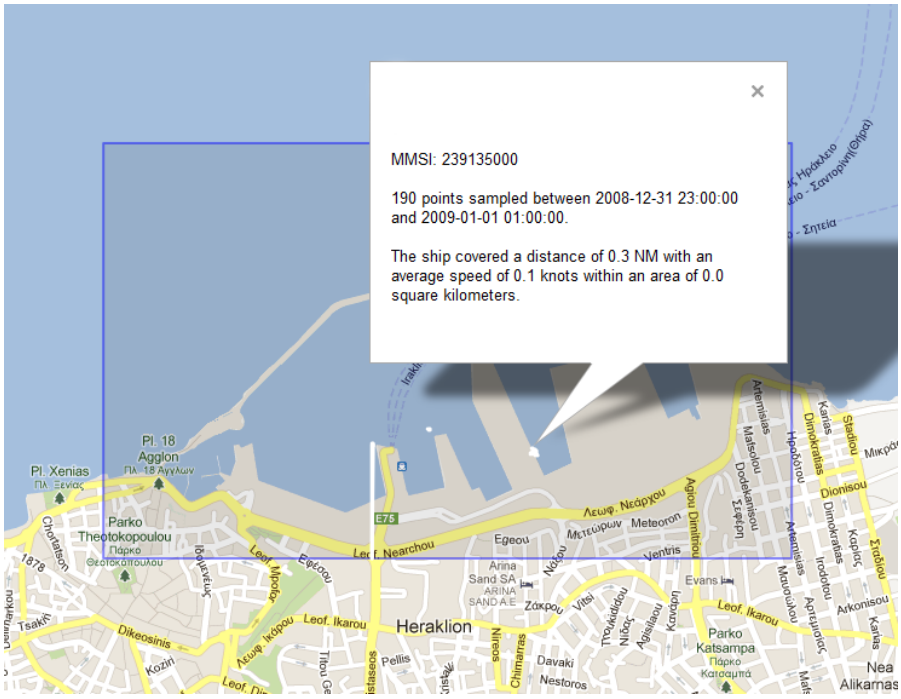


Fig. 5.3: Range Result

```
6       );
```

The index-supported "within distance" operator (-<) filters the database.

Notice that "nm2metres" function is used to transform nautical miles to meters. Since Hermes uses 1 meter accuracy the number is rounded to the nearest integer.

In figure 5.4 we notice a few ships passing close to the lighthouse. The lighthouse is at the same location with the port of Patrai so we expect a lot of ships passing very close to it.

Find the ship that was the closest to the lighthouse in Patrai.

```
1  WITH TO_METERS AS (
2  SELECT PointSP(PointLL(21.72565, 38.24513), HDatasetID('
       imis')) AS lighthouse
3  )
4  SELECT obj_id, traj_id, atPoint(seg, cp, false) cp,
5      distance(cp, (SELECT lighthouse FROM TO_METERS)) AS
           dist
6  FROM (
```

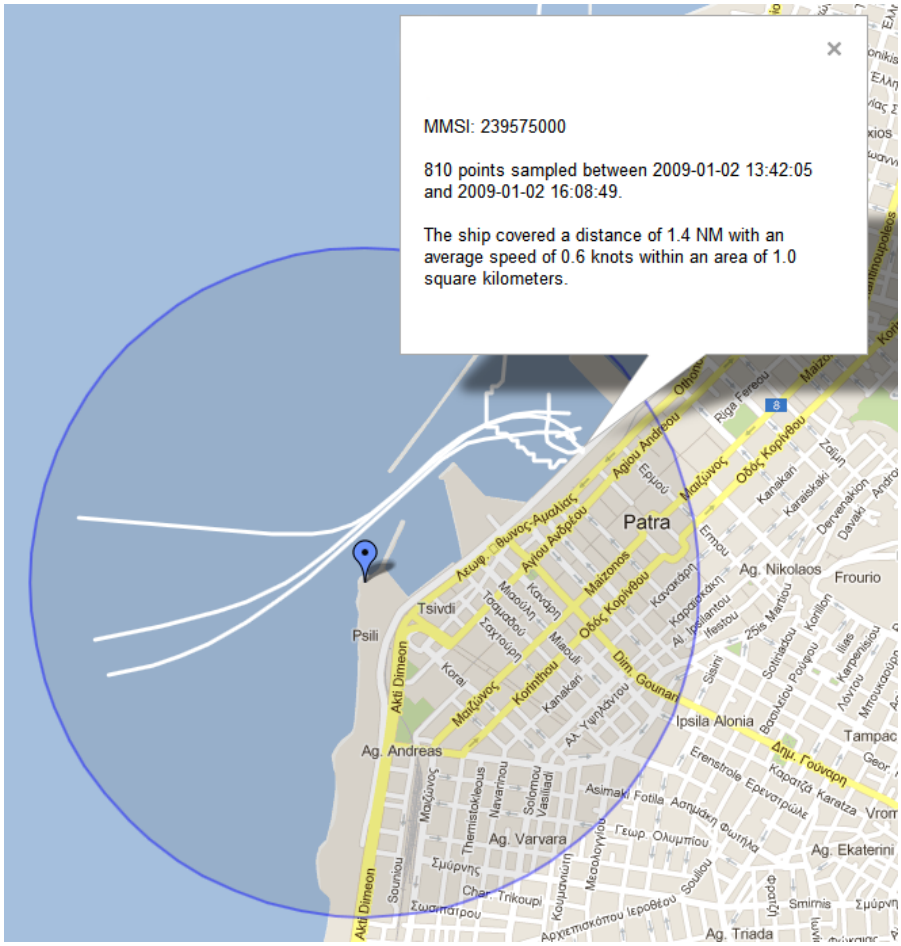


Fig. 5.4: Distance Result

```
7       SELECT obj_id, traj_id, seg,
8           closestPoint(getSp(seg), (SELECT lighthouse FROM
                TO_METERS)) AS cp
9       FROM imis_seg
10      ORDER BY seg <-> (SELECT lighthouse FROM TO_METERS)
11      LIMIT 1
12  ) AS tmp;
```

The index-supported distance operator (<->) searches the database for the k-NN segments w.r.t. a stationery object. Here k is limited to 1, so we saw a 1-NN case.

Notice that we use “closestPoint” function to find the point in the trajectory segment that is the closest to the lighthouse. After that we use “atPoint”

function to find the timestamp for that point. We command the function not to check for containment by adding "false" as the last parameter since we already know the point is contained on the segment.

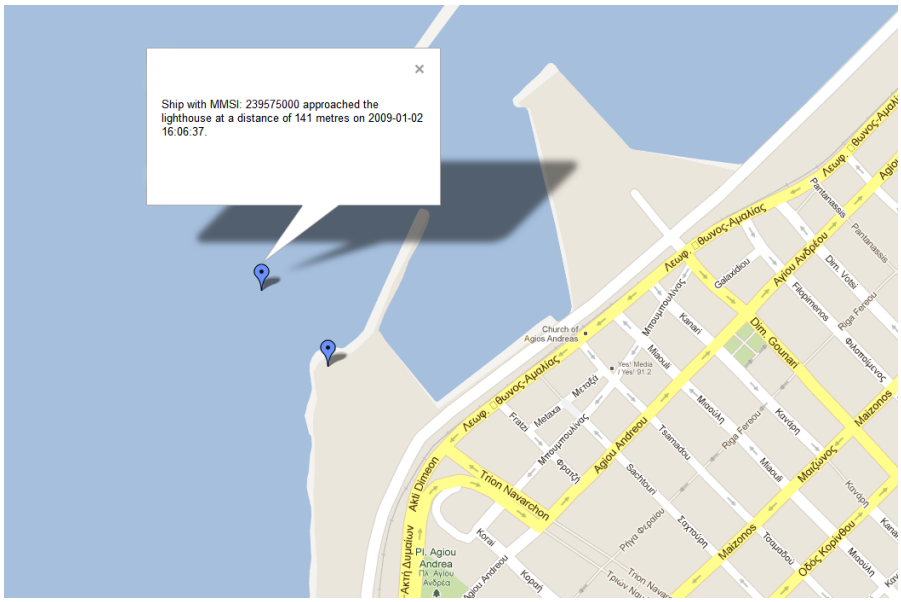


Fig. 5.5: 1-NN Result

This query gives us one of the ships of the result of the previous query, see figure 5.5. This time we know that the ship with MMSI 239575000 was the one that came closest to the lighthouse and at what point exactly.

For every ship, find which other ships were in its 1 n.m. vicinity 5 minutes around 2009-01-02 11:00:00. Also, let's give an estimate on how close each ship was to the other.

```
1  SELECT r.obj_id, db.obj_id,
2      intersection(HUnion(getT(db.seg)),
3          Period('2009-01-02 10:55:00', '2009-01-02 11:05:00'
               )) AS common_period,
4      avg(metres2nm(distance(getSp(db.seg), getSpc(r.range)))
           ) AS avg_dist
5  FROM imis_seg AS db INNER JOIN (
6      SELECT obj_id, RangeST('00:05:00', getT(position),
7              round(nm2metres(1))::integer, getX(position),
                   getY(position)
8          ) AS range
```

```
9       FROM (
10          SELECT DISTINCT ON (obj_id) obj_id,
11              atInstant(seg, '2009-01-02 11:00:00') AS
                    position
12          FROM imis_seg
13          WHERE seg ~ '2009-01-02 11:00:00'::timestamp
14      ) AS timeslice
15  ) AS r ON db.seg -< r.range
16  WHERE r.obj_id <> db.obj_id
17  GROUP BY r.obj_id, db.obj_id
18  ORDER BY r.obj_id ASC, avg_dist ASC;
```

In this query we first execute a timeslice query and then use that result to execute a distance query. Also, we use an aggregate function "HUnion", along with a "GROUP BY" clause of course, on the period component of the segments of a trajectory to find their union.

Output pane

Data Output | Explain | Messages | History

	obj_id integer	obj_id integer	common_period period	avg_dist double precision
1	207060000	271001030	('2009-01-02 10:55:00', '2009-01-02 11:03:54')	0.0529427489186636
2	207060000	271000814	('2009-01-02 11:01:44', '2009-01-02 11:05:00')	0.825825787000513
3	209000409	355318000	('2009-01-02 10:55:00', '2009-01-02 11:05:00')	0.736592964339636
4	209000409	239778000	('2009-01-02 10:55:00', '2009-01-02 11:05:00')	0.819759306759323
5	209000409	237096100	('2009-01-02 11:03:52', '2009-01-02 11:05:00')	0.888930361514072
6	209000409	636013395	('2009-01-02 10:55:00', '2009-01-02 11:05:00')	0.891594634471942
7	209000409	240689000	('2009-01-02 10:55:00', '2009-01-02 11:05:00')	0.911194717027903
8	210276000	239513000	('2009-01-02 10:55:00', '2009-01-02 11:05:00')	0.14504519173728
9	210276000	239919000	('2009-01-02 10:55:00', '2009-01-02 11:05:00')	0.180290333479663
10	210276000	240732000	('2009-01-02 10:55:00', '2009-01-02 11:05:00')	0.212356086996474
11	210276000	239575000	('2009-01-02 10:55:00', '2009-01-02 11:05:00')	0.216239086141727
12	210276000	240790000	('2009-01-02 10:55:00', '2009-01-02 11:05:00')	0.327370439171995
13	210276000	240580000	('2009-01-02 10:55:00', '2009-01-02 11:05:00')	0.526691464910599
14	210578000	239245000	('2009-01-02 10:55:00', '2009-01-02 11:01:40')	0.340101472404443
15	210578000	214180508	('2009-01-02 10:55:00', '2009-01-02 11:05:00')	0.351253346797302
16	210578000	214180202	('2009-01-02 10:55:00', '2009-01-02 11:05:00')	0.455882174800386
17	210578000	514694000	('2009-01-02 10:55:00', '2009-01-02 11:05:00')	0.461300596759244
18	210578000	232109000	('2009-01-02 10:55:00', '2009-01-02 11:05:00')	0.556618114118193
19	210578000	271002099	('2009-01-02 10:55:00', '2009-01-02 11:05:00')	0.70560164557695
20	210578000	376519000	('2009-01-02 10:55:00', '2009-01-02 11:05:00')	0.749717030282366
21	210578000	256276000	('2009-01-02 10:55:00', '2009-01-02 11:05:00')	0.887284039928957
22	210578000	215593000	('2009-01-02 10:55:00', '2009-01-02 11:05:00')	0.906579121599075
23	212023000	240860000	('2009-01-02 10:55:00', '2009-01-02 11:05:00')	0.038364246280891
24	212023000	354291000	('2009-01-02 10:55:00', '2009-01-02 11:05:00')	0.0473532557881682
25	212023000	352566000	('2009-01-02 10:55:00', '2009-01-02 11:05:00')	0.0584376965993939

Fig. 5.6: Join Result

Find the ships that entered Patras port area. (Irrespective of time)

```
WITH TO_METERS AS (
SELECT
    PointSP(PointLL(21.7223, 38.2448), HDatasetID('imis'))
        AS low,
    PointSP(PointLL(21.7394, 38.2630), HDatasetID('imis'))
        AS high
), PORT_AREA AS (
SELECT BoxSP((SELECT low FROM TO_METERS), (SELECT high FROM
    TO_METERS)) AS box
)
SELECT obj_id, (el).enterPoint
FROM (
    SELECT obj_id,
        enter_leave(array_agg(seg), (SELECT box FROM
            PORT_AREA)) AS el
    FROM imis_seg
    WHERE seg && (SELECT box FROM PORT_AREA)
    GROUP BY obj_id
) AS tmp
WHERE (el).enterPoint IS NOT NULL;
```

Notice the "enter_leave" function which takes an array of segments of the same trajectory and returns an enter and a leave points of that trajectory in the area specified in the second argument. If there is no enter and/or leave point then the corresponding property in the result of the function will be NULL.

We can clearly see the main entrance points in the figure 5.7.

Find the ships that crossed Evvoia - Andros narrow passage. (Irrespective of time)

```
WITH TO_METERS AS (
SELECT
    PointSP(PointLL(24.528, 37.920), HDatasetID('imis')) AS
        low,
    PointSP(PointLL(24.810, 38.010), HDatasetID('imis')) AS
        high
), PORT_AREA AS (
SELECT BoxSP((SELECT low FROM TO_METERS), (SELECT high FROM
    TO_METERS)) AS box
)
SELECT obj_id, (el).enterPoint, (el).leavePoint
FROM (
    SELECT obj_id,
        enter_leave(array_agg(seg), (SELECT box FROM
            PORT_AREA)) AS el
    FROM imis_seg
```

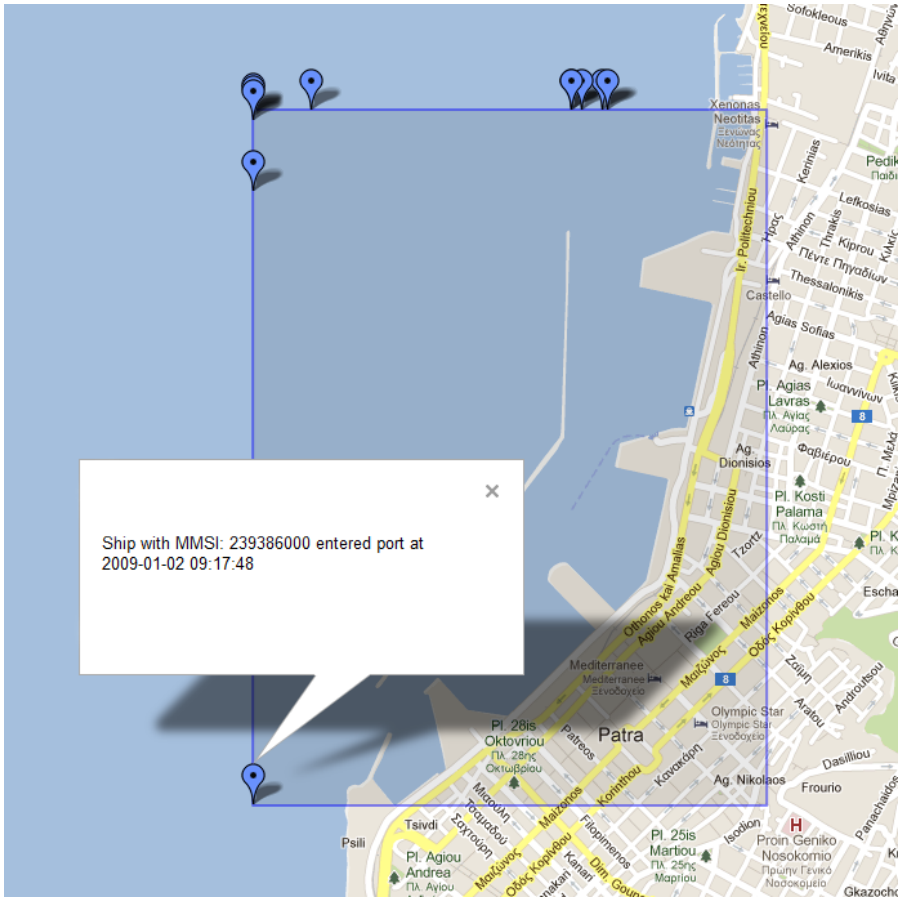


Fig. 5.7: Enter Result

```
13      WHERE seg && (SELECT box FROM PORT_AREA)
14      GROUP BY obj_id
15  ) AS tmp
16  WHERE (el).enterPoint IS NOT NULL AND (el).leavePoint IS
        NOT NULL;
```

We utilize the "enter_leave" function again to find if a ship crossed the area.

We notice that the passage is heavily used and there a lot of congestion, see figure 5.8.

Find the Origin-Destination Matrix between 4 large areas of the Greek territory.

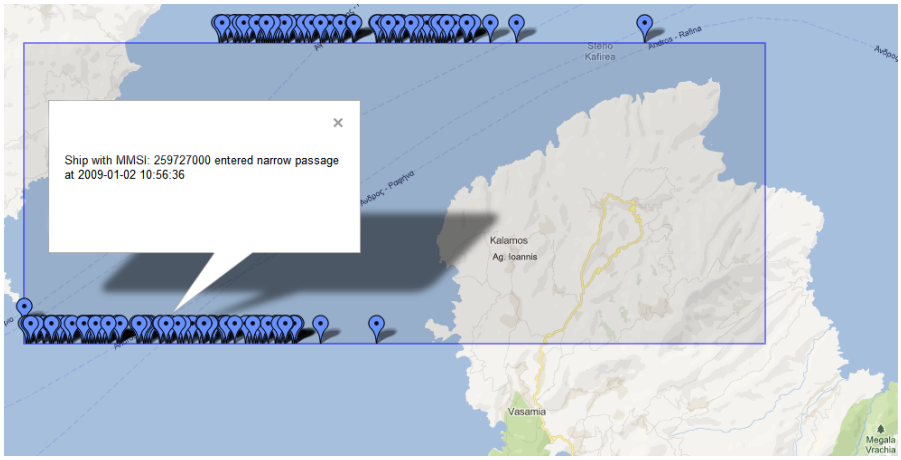


Fig. 5.8: Cross Result

```
1  WITH AREAS AS (
2  SELECT 'North Aegean' AS name, BoxSP(PointSP(PointLL(24.84,
       37.43), HDatasetID('imis')),
3      PointSP(PointLL(27.10, 40.06), HDatasetID('imis'))) AS
         area
4  UNION SELECT 'Piraeus', BoxSP(PointSP(PointLL(23.19, 37.50)
       , HDatasetID('imis')),
5      PointSP(PointLL(23.90, 38.10), HDatasetID('imis')))
6  UNION SELECT 'Ionian-Cretan', BoxSP(PointSP(PointLL(21.55,
       35.28), HDatasetID('imis')),
7      PointSP(PointLL(23.65, 36.68), HDatasetID('imis')))
8  UNION SELECT 'Dodecanese', BoxSP(PointSP(PointLL(26.39,
       35.05), HDatasetID('imis')),
9      PointSP(PointLL(28.57, 37.32), HDatasetID('imis')))
10 ), OD AS (
11 SELECT origin.name AS o_name, origin.area AS o_area,
12     destination.name AS d_name, destination.area AS d_area
13 FROM AREAS AS origin INNER JOIN AREAS AS destination
14     ON origin.name <> destination.name
15 ), START_END AS (
16 SELECT obj_id, minT(i(seg)) AS start, maxT(e(seg)) AS end
17 FROM imis_seg
18 GROUP BY obj_id
19 )
20 SELECT OD.o_name, OD.d_name, count(DISTINCT START_END.
       obj_id) AS nof_ships
21 FROM OD LEFT JOIN START_END
22     ON contains(OD.o_area, getSp(START_END.start))
23         AND contains(OD.d_area, getSp(START_END.end))
```

```
24  GROUP BY OD.o_name, OD.d_name
25  HAVING count(DISTINCT START_END.obj_id) > 0
26  ORDER BY OD.o_name ASC, OD.d_name ASC;
```

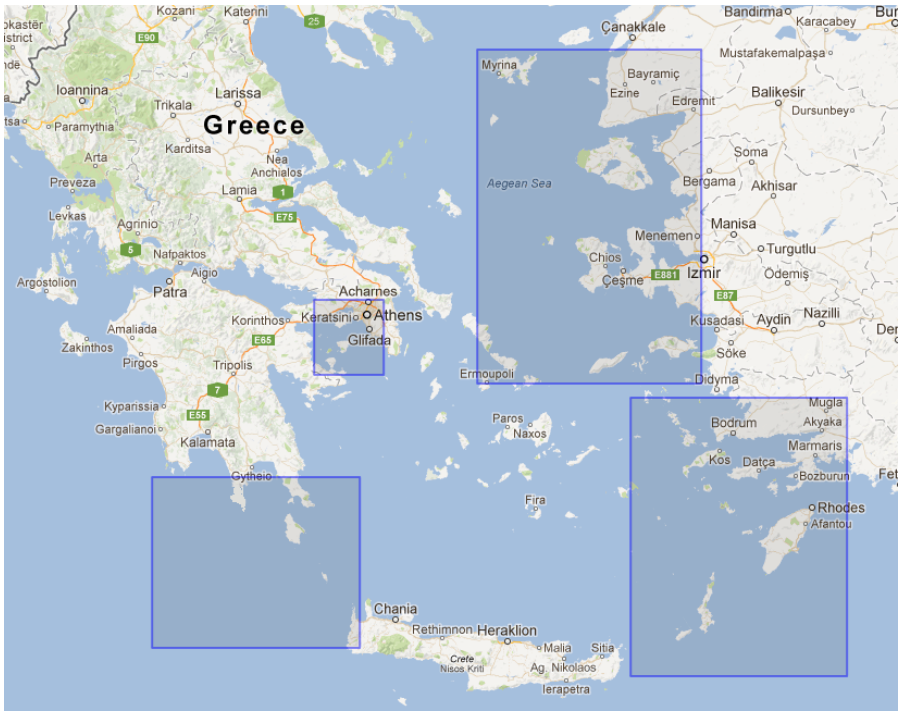


Fig. 5.9: Entrance and exit areas in Greek territory. See table 5.1 for an OD-Matrix between these areas.

Table 5.1: Origin-Destination Matrix between 4 large areas of the Greek territory

Origin / Destination	Dodecanese	Ionian-Cretan	North Aegean	Piraeus
Dodecanese	-	2	30	4
Ionian-Cretan	6	-	47	11
North Aegean	29	35	-	9
Piraeus	2	1	7	-

As shown by the OD-Matrix (table 5.1) the two main entrance and exit routes in Greek Seas are the most heavily used routes. Meaning that Greek Seas are a congested crossroad for travelling ships.

5.3 Visualization tips

Hermes provides a set of functions that allow to construct a KML document within a query in steps. An example of how we visualized the Timeslice query is the following.

```
1  COPY (
2
3  WITH TABULAR_RESULT AS (
4  ------------------------------- Core Query
       -------------------------------------
5  SELECT DISTINCT ON (obj_id, traj_id) obj_id, traj_id,
6      atInstant(seg, '2009-01-02 00:00:00') AS position
7  FROM imis_seg
8  WHERE seg ~ '2009-01-02 00:00:00'::timestamp
9  ------------------------------- End of Core Query
       ------------------------------
10 )
11 SELECT KMLDocument(KMLFolder('2009-01-02 00:00:00',
       string_agg(
12     KMLPoint('MMSI: ' || obj_id, getSp(position),
           HDatasetID('imis'))
13 , '')))
14 FROM TABULAR_RESULT
15
16 ) TO 'C:\Program Files\PostgreSQL\9.2\data\Timeslice.kml';
```

Function "KMLPoint" returns a string that gives a KML point placemark element (each point will have in its description the object and trajectory id it belongs to). Then we aggregate all points using "string_agg" function and pass that result to "KMLFolder" which will enclose the points under one KML folder element (the folder's name is the timestamp we gave for the query). Finally, we enclose that folder element in a KML document element. So, our KML file is now all in one row as a string and using "COPY" command we write it to a system file.

One more example that shows how to visualize trajectories.

```
1  COPY (
2
3  WITH TABULAR_RESULT AS (
4  ------------------------------- Core Query
       -------------------------------------
5  WITH TO_METERS AS (
6  SELECT
7      PointSP(PointLL(25.1325, 35.3407), HDatasetID('imis'))
           AS low,
8      PointSP(PointLL(25.1569, 35.3527), HDatasetID('imis'))
           AS high
9  ), SPT_WINDOW AS (
```

```
10 SELECT BoxST(
11         Period('2008-12-31 23:00:00', '2009-01-01 01:00:00'
               ),
12         BoxSP((SELECT low FROM TO_METERS), (SELECT high
               FROM TO_METERS))
13     ) AS box
14 )
15 SELECT obj_id, traj_id, (atBox(seg, (SELECT box FROM
       SPT_WINDOW))).s AS seg
16 FROM imis_seg
17 WHERE seg && (SELECT box FROM SPT_WINDOW)
18     AND (atBox(seg, (SELECT box FROM SPT_WINDOW))).n = 2
19 ------------------------------ End of Core Query
       ------------------------------
20 )
21 SELECT KMLDocument(KMLFolder('Input area', KMLPolygon('
       Heraklion port area',
22         BoxSP(PointSP(PointLL(25.1325, 35.3407), HDatasetID
               ('imis')),
23             PointSP(PointLL(25.1569, 35.3527), HDatasetID('
                   imis')))
24         , HDatasetID('imis'))) || string_agg(tracksFolder,
               ''))
25 FROM (
26     SELECT obj_id, KMLFolder('MMSI: ' || obj_id,
27         string_agg(trackPlacemark, '')) AS tracksFolder
28     FROM (
29         SELECT obj_id, traj_id, KMLTrack(
30             ------------------- Balloon Info
                   -----------------------------------
31             'MMSI: ' || obj_id || '<br/><br/>' ||
32             count(*) - 1 || ' points sampled between ' ||
                   min(getTi(seg)) ||
33             ' and ' || max(getTe(seg)) || '.<br/><br/>' ||
34             'The ship covered a distance of ' ||
35             trunc(metres2nm(sum(length(getSp(seg))))::
                   numeric, 1) ||
36             ' NM with an average speed of ' ||
37             trunc(mps2knots(sum(length(getSp(seg))) /
38                 extract(epoch from max(getTe(seg)) - min(
                       getTi(seg))))::numeric
39                 , 1) ||
40             ' knots within an area of ' ||
41             trunc(area(BoxSP(min(min(getIx(seg), getEx(seg)
                   )),
42                 min(min(getIy(seg), getEy(seg))),
43             max(max(getIx(seg), getEx(seg))),
44                 max(max(getIy(seg), getEy(seg))))) /
                       1000000, 1) ||
```

```
45              ' square kilometers.'
46              ------------------- End of Balloon Info
                    ----------------------------
47              , array_agg(seg ORDER BY getTi(seg) ASC),
                    HDatasetID('imis')
48              ) AS trackPlacemark
49          FROM TABULAR_RESULT
50          GROUP BY obj_id, traj_id
51      ) AS tracks
52      GROUP BY obj_id
53  ) AS folders
54
55  ) TO 'C:\Program Files\PostgreSQL\9.2\data\Range.kml';
```

6

Case Study: ChoroChronos Archive

ChoroChronos Archive (http://www.chorochronos.org) is a web portal and a collection of moving object databases and related algorithms that are used by the mobility data management and mining community for the empirical analysis and evaluation of mobility-centric query processing and mining algorithms [8]. Hermes was used to support ChoroChronos data layer so that guests can pose queries to a Moving Objects Database (MOD). The architecture of ChoroChronos is illustrated in figure 6.1. It is a classic 3-tier architecture (presentation-business-data) and Hermes is a part of the data layer. Spatio-temporal datasets are hosted in Hermes thus allowing efficient query execution. In figure 6.2 you can see an example execution of a window query on "IMIS 3 Days" dataset, indexed with pg3D-RTree.

The Window Query of ChoroChoronos utilizes atPeriod and atBox methods to compute the result that is visualized on the map. It also uses && (overlaps) operator in the same query to take advantage of the index. The TimeSlice Query (figure 6.3) uses atInstant to find the place where the ships were at a specific timestamp and takes advantage of the 3D-Rtree through the $\sim$ (contains) operator.

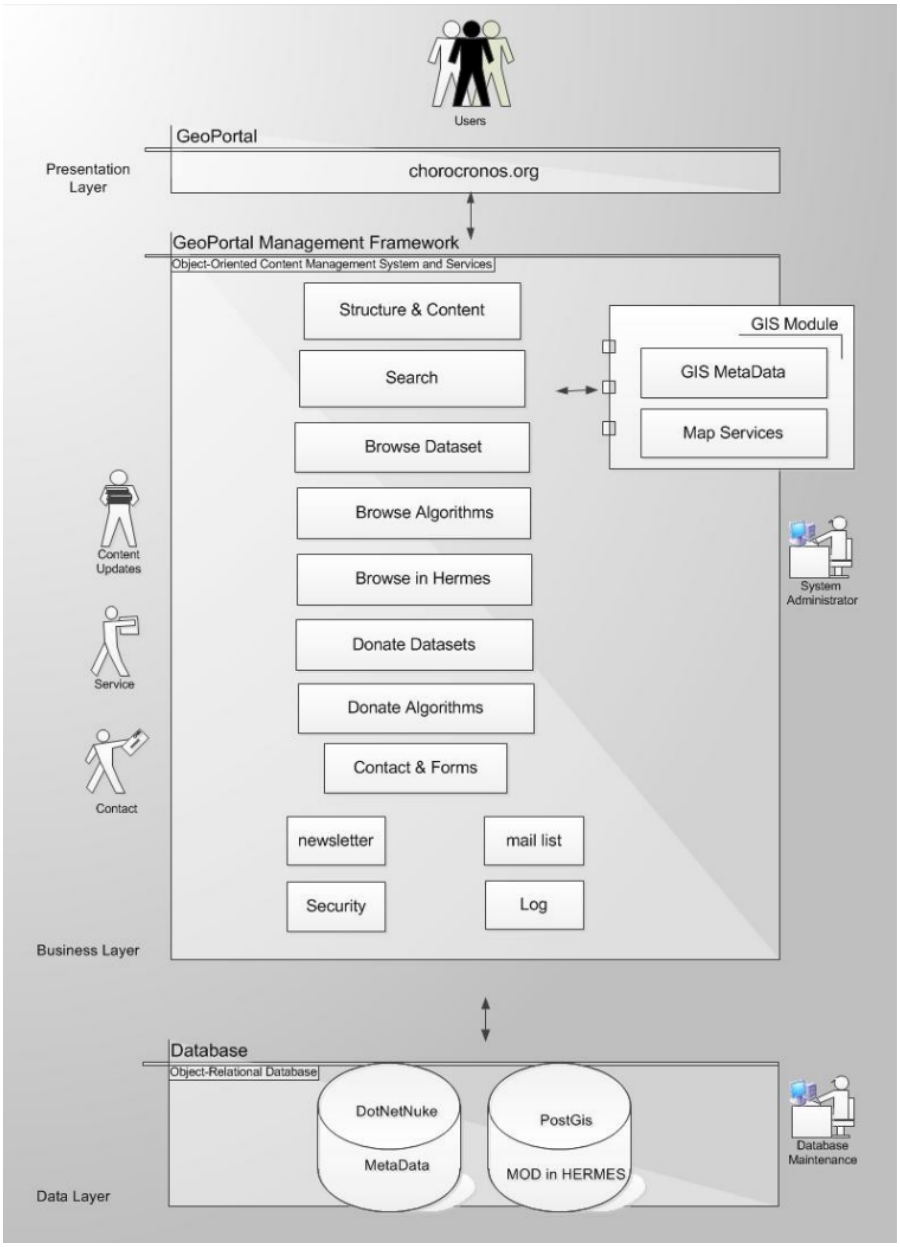


Fig. 6.1: The architecture of ChoroChronos.org [8]

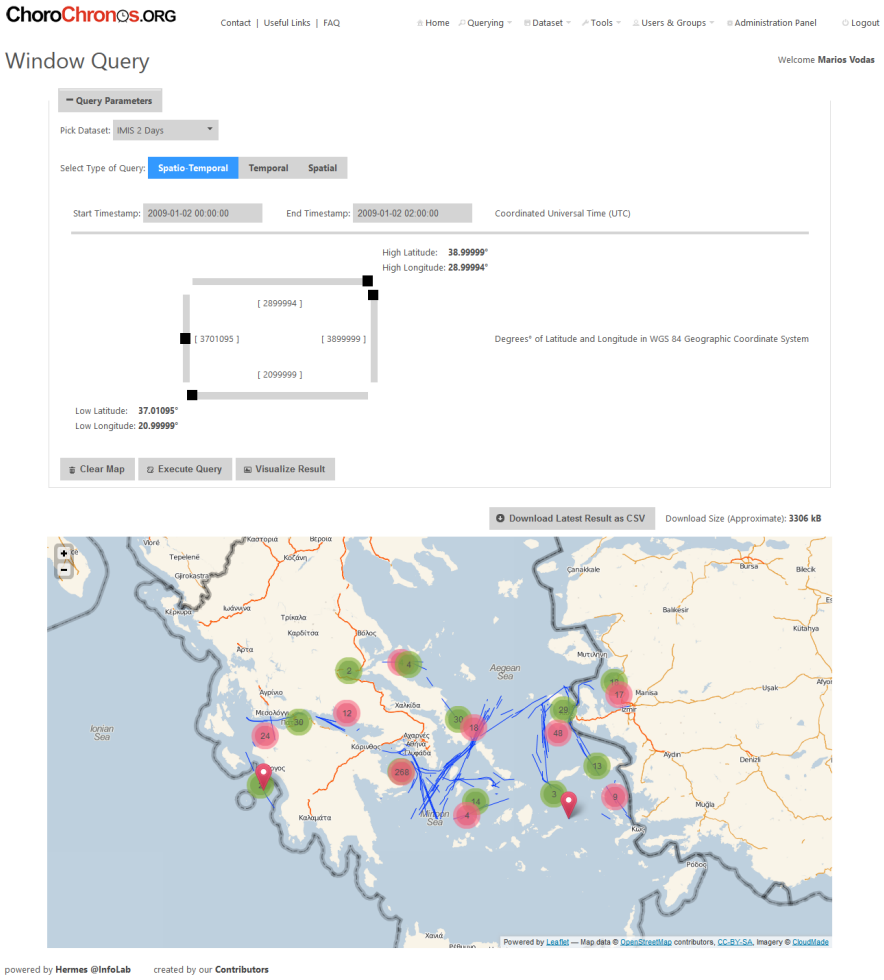


Fig. 6.2: ChoroChronos.org Window Query

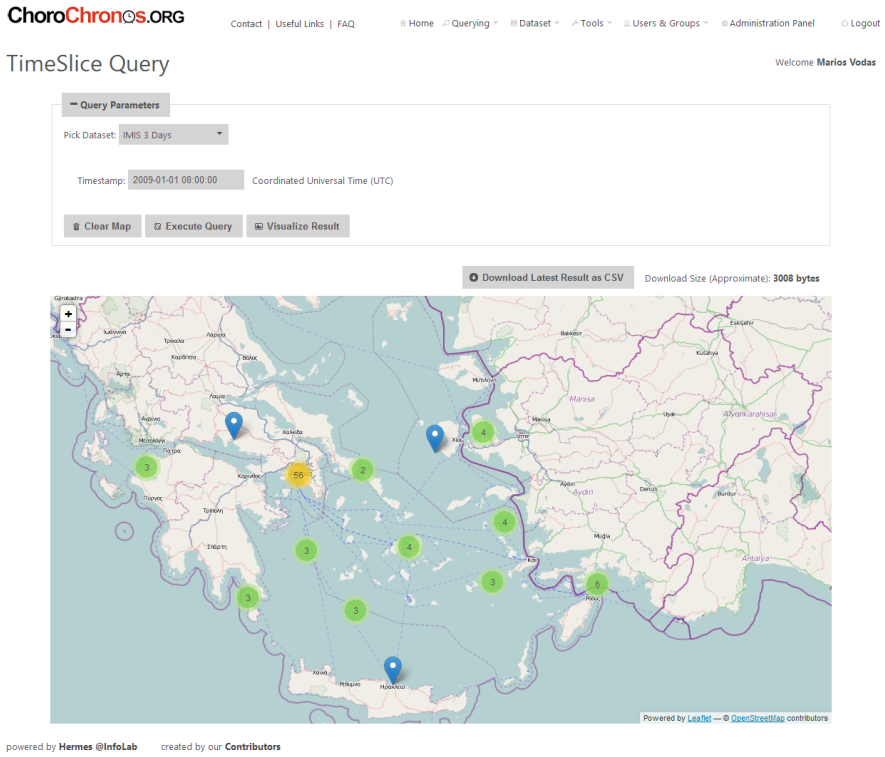


Fig. 6.3: ChoroChronos.org TimeSlice Query

7

Summary

The spread of the concept of Mobility to basic users in conjunction with the lack of software frameworks that would be able to handle spatio-temporal data and methods lead to the development of Hermes. Hermes provides a clear SQL interface to its data types, functions and operators that make it easy to learn and use when it comes to managing spatio-temporal data. We explained its components and demonstrated its capabilities on a real world dataset. We also showed its maturity by using it to support a real-world web application.

Next Steps

There is always room for improvement on a framework like Hermes and some of the areas this can be done are:

- spatio-temporal indexing: developing an indexing library on top of GiST with the state of the art indexing algorithms for spatio-temporal data
- advanced spatio-temporal processing (e.g. computational geometry algorithms)
- semantic trajectories management: integrating text and spatio-temporal data
- management of mobility data from cellular networks such as GSM: ways to store and query very sparsely sampled trajectories with high position uncertainty

An immediate feature that can be added to Hermes is a way to do Map-Matching over a network. This can adopt the network representation of pgRouting and it could be embedded into the Loader for automatic execution of the procedure during loading.

A

Installation Instructions

There are two ways to install Hermes on PostgreSQL. The first one is to use the installer and the second one is to run the install scripts manually thus provides a solution in cases where custom installation is required. The recommended way is the installer.

The installer supports downloading the latest version and updating a database to the latest version of Hermes.

Use the Installer

The latest version of the installer can be downloaded from this link. The .zip file is to be used on Windows and the .tar.gz on Linux.

***Important**: On 64-bit Linux you will need to install the following packages "ia32-libs" and "ia32-libs-gtk" for the installer to work.

The recommended first thing to do is extract the contents of the compressed file and read the "ReadMe.txt" file.

Below there are screenshots demonstrating the steps of the installer on Windows. The steps are the same on Linux.

Installation Step 1: Run installer as administrator. On Linux as root.

Installation Step 2: Choose "Install Hermes" to begin a fresh installation.

Installation Step 3: Choose "Install prerequisites for me" if you want the installer to install PostgreSQL, PostGIS, Python, and VCRedist later on another step. If you choose not to, then the installer assumes you have already installed the prerequisites.

Installation Step 4: You will have to accept the license terms in order to continue.

Installation Step 5: "Install as" field should NOT be changed unless you need to have multiple versions of Hermes in one PostgreSQL instance.

Installation Step 6: Now the installer will begin the installation of the prerequisites, provided that in a previous step you chose that option. The first prerequisite is "Microsoft Visual C++ 2012 Redistributable Package".

Installation Step 7: The second prerequisite is Python.

Installation Step 8: The third prerequisite is PostgreSQL.

Installation Step 9: When PostgreSQL installation finishes you will have to install the fourth prerequisite, PostGIS, usually via the Stack Builder application of PostgreSQL.

Installation Step 10: In this step you will have to provide the port your PostgreSQL instance runs on, as well as the password for the "postgres" superuser.

Installation Step 11: If everything works as it should the installer will show you the above screen. At this point you have installed Hermes on your PostgreSQL instance!

Manual Steps

Before following the installation steps, check the prerequisites of Hermes described below.

Hermes works both on 64-bit and 32-bit operating systems, and supports Linux and Windows Vista SP2 or later.

Hermes requires:

1. PostgreSQL version 9.2 or later
2. PostGIS version 2.0 or later (Optional in Hermes architecture, but highly recommended because it enhances user experience)
3. Python version 3.2 or later
4. Windows only: Microsoft Visual C++ 2012 Redistributable Package

Installation steps:

1. Install PostgreSQL and then PostGIS (usually via StackBuilder).
2. Install Python, 32-bit or 64-bit version depending on the architecture of your PostgreSQL (not of the operating system).
3. Only if you are using Windows, install "Microsoft Visual C++ 2012 Redistributable Package", 32-bit or 64-bit version depending on the architecture of your PostgreSQL (not of Windows).
4. Copy the shared library Hermes.so (Linux)/Hermes.dll (Windows) under $PostgreSQL_install_dir/lib folder. The shared library that you should copy also depends on whether the version of your PostgreSQL installation is 32-bit or 64-bit.
5. Run the SQL scripts found in "Hermes SQL Framework" folder.

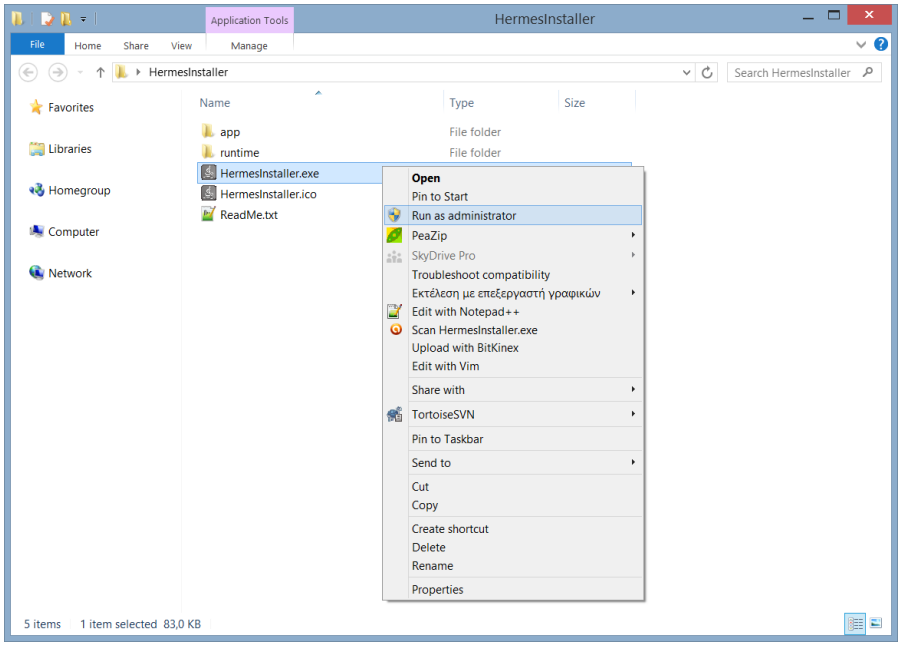


Fig. A.1: Installation Step 1

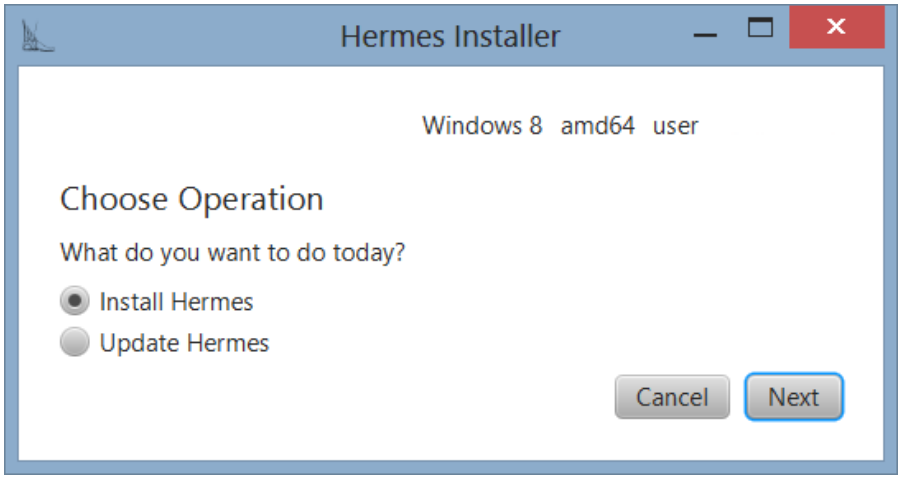


Fig. A.2: Installation Step 2

Fig. A.3: Installation Step 3

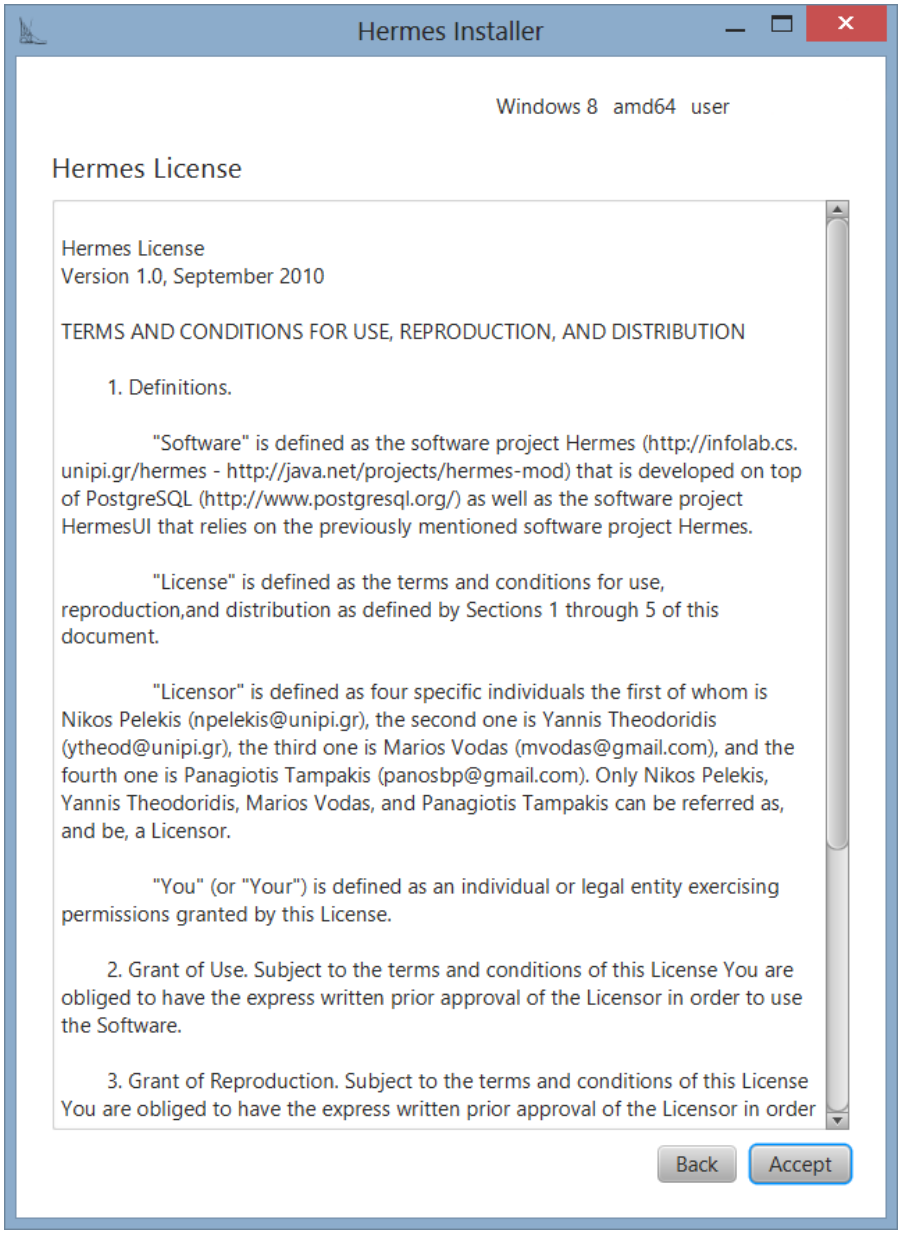


Fig. A.4: Installation Step 4

Fig. A.5: Installation Step 5

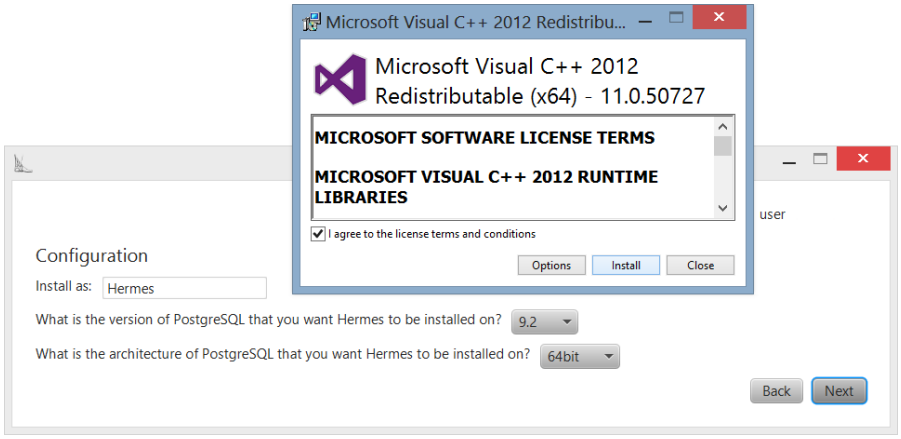


Fig. A.6: Installation Step 6

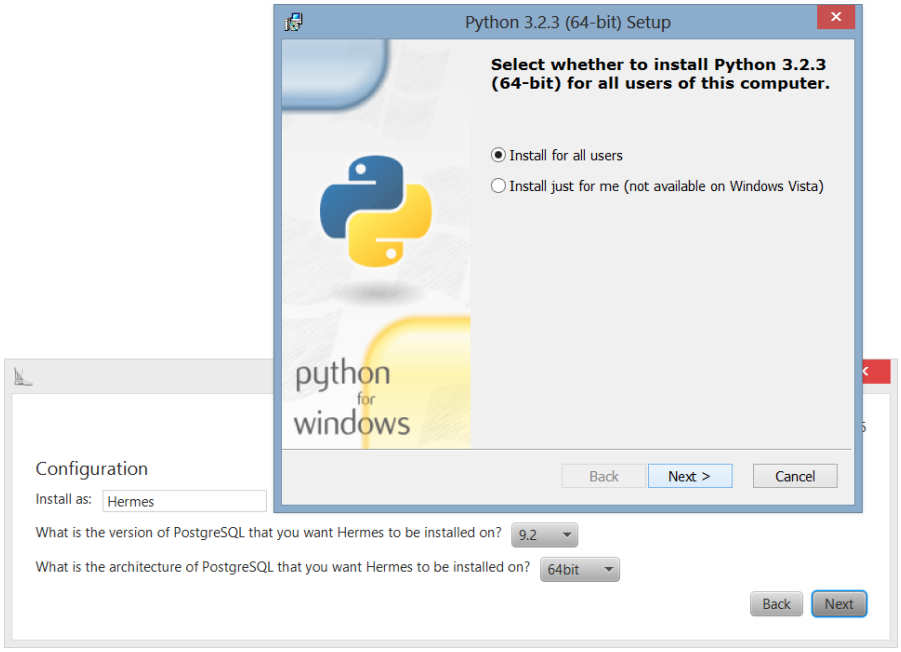


Fig. A.7: Installation Step 7

Fig. A.8: Installation Step 8

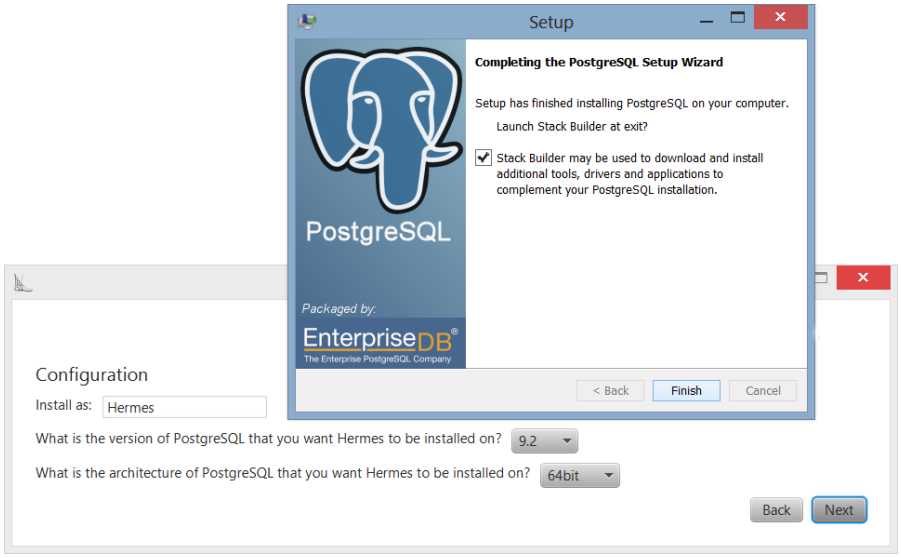


Fig. A.9: Installation Step 9

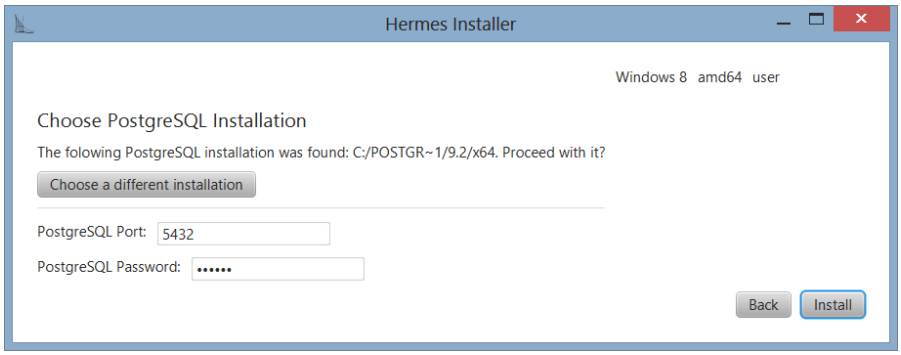


Fig. A.10: Installation Step 10

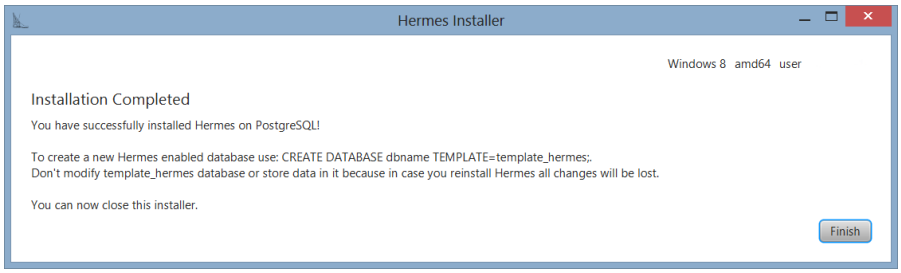


Fig. A.11: Installation Step 11

References

1. Postgresql documentation. http://www.postgresql.org/docs/current/static.
2. Building a 21st century platform to better serve the american people. http://www.whitehouse.gov/sites/default/files/omb/egov/digital-government/digital-government.html, May 2012.
3. C.-H. Ang and T. C. Tan. New linear node splitting algorithm for r-trees. In *Proceedings of the 5th International Symposium on Advances in Spatial Databases*, SSD '97, pages 339–349, London, UK, UK, 1997. Springer-Verlag.
4. O. G. Committee. Guidance notes. http://info.ogp.org.uk/geodesy/guides.
5. S. Dieker and R. H. Güting. Plug and play with query algebras: Secondo-a generic dbms development environment. In *Proceedings of the 2000 International Symposium on Database Engineering & Applications*, IDEAS '00, pages 380–392, Washington, DC, USA, 2000. IEEE Computer Society.
6. J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized search trees for database systems. In *Proceedings of the 21th International Conference on Very Large Data Bases*, VLDB '95, pages 562–573, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.
7. N. Pelekis, E. Frentzos, N. Giatrakos, and Y. Theodoridis. Hermes: aggregative lbs via a trajectory db engine. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 1255–1258, New York, NY, USA, 2008. ACM.
8. N. Pelekis, E. Stefanakis, I. Kopanakis, C. Zotali, M. Vodas, and Y. Theodoridis. Chorochronos.org: A geoportal for movement data and processes. In *Proceedings of the 10th International Conference on Spatial Information Theory*, COSIT '11, 2011.
9. N. Pelekis and Y. Theodoridis. Boosting location-based services with a moving object database engine. In *Proceedings of the 5th ACM international workshop on Data engineering for wireless and mobile access*, MobiDE '06, pages 3–10, New York, NY, USA, 2006. ACM.
10. N. Pelekis, Y. Theodoridis, S. Vosinakis, and T. Panayiotopoulos. Hermes - a framework for location-based data management. In *Proceedings of the 10th international conference on Advances in Database Technology*, EDBT'06, pages 1130–1134, Berlin, Heidelberg, 2006. Springer-Verlag.
11. D. Pfoser, C. S. Jensen, and Y. Theodoridis. Novel approaches in query processing for moving object trajectories. In *Proceedings of the 26th International*

Conference on Very Large Data Bases, VLDB '00, pages 395–406, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

12. Y. Theodoridis, M. Vazirgiannis, and T. Sellis. Spatio-temporal indexing for large multimedia applications. In *Multimedia Computing and Systems, 1996., Proceedings of the Third IEEE International Conference on*, pages 441–448, 1996.

List of Figures

List of Tables