Chapter 6
From Points to Habitat: Relating Environmental Information to GPS Positions

Ferdinando Urbano, Mathieu Basille and Pierre Racine

Abstract Animals move in and interact with complex environments that can be characterised by a set of spatial layers containing environmental data. Spatial databases can manage these different data sets in a unified framework, defining spatial and non-spatial relationships that simplify the analysis of the interaction between animals and their habitat. A large set of analyses can be performed directly in the database with no need for dedicated GIS or statistical software. Such an approach moves the information content managed in the database from a 'geographical space' to an 'animal's ecological space'. This more comprehensive model of the animals' movement ecology reduces the distance between physical reality and the way data are structured in the database, filling the semantic gap between the scientist's view of biological systems and its implementation in the information system. This chapter shows how vector and raster layers can be included in the database and how you can handle them using (spatial) SQL. The database built so far in Chaps. 2, 3, 4 and 5 is extended with environmental ancillary data sets and with an automated procedure to intersect these layers with GPS positions.

Keywords PostGIS · Raster data · Data management · Spatial database

F. Urbano (✉)
Università Iuav di Venezia, Santa Croce 191 Tolentini, 30135 Venice, Italy
e-mail: ferdi.urbano@gmail.com

M. Basille
Fort Lauderdale Research and Education Center, University of Florida,
3205 College Avenue, Fort Lauderdale, FL 33314, USA
e-mail: basille@ase-research.org

P. Racine
Centre for Forest Research, University Laval, Pavillon Abitibi-Price, 2405 de la Terrasse,
Bureau 1122, Quebec, QC G1V 0A6, Canada
e-mail: pierre.racine@sbf.ulaval.ca

F. Urbano and F. Cagnacci (eds.), *Spatial Database for GPS Wildlife Tracking Data*,
DOI: 10.1007/978-3-319-03743-1_6,

Introduction

Animals move in and interact with complex environments that can be characterised by a set of spatial layers containing environmental data. In traditional information systems for wildlife tracking data management, position data are stored in some file-based spatial format (e.g. shapefile). With a multi-steps process in a GIS environment, position data are associated with a set of environmental attributes through an analytical stage (e.g. intersection of GPS positions with vector and raster environmental layers). This process is usually time-consuming and prone to error, implies data replication and often has to be repeated for any new analysis. It also generally involves different tools for vector and raster data. An advanced data management system should achieve the same result with an efficient (and, if needed, automated) procedure, possibly performed as a real-time routine management task. To do so, the first step is to integrate both position data and spatial ancillary information on the environment in a unique framework. This is essential to exploring the animals' behaviour and understanding the ecological relationships that can be revealed by tracking data. Spatial databases can manage these different data sets in a unified framework, defining spatial and non-spatial relationships that simplify the analysis of the interaction between animals and their habitat. A large set of analyses can be performed directly in the database with no need for dedicated GIS or statistical software. This also affects performance, as databases are optimised to run simple processes on large data sets like the ones generated by GPS sensors. Database tools such as triggers and functions can be used, for example, to automatically intersect positions with the ancillary information stored as raster and vector layers. The result is that positions are transformed from a simple pair of numbers (coordinates) to complex multi-dimensional (spatial) objects that define the individual and its habitat in time and space, including their interactions and dependencies. In an additional step, position data can also be joined to activity data to define an even more complete picture of the animal's behaviour (see Chap. 12). Such an approach moves the information content managed in the database from a 'geographical space' to an 'animal's ecological space'. This more comprehensive model of the animal movement ecology reduces the distance between physical reality and the way data are structured in the database, filling the semantic gap between the scientist's view of biological systems and its implementation in the information system. This is not only interesting from a conceptual point of view, but also has deep practical implications. Scientists and wildlife managers can deal with data in the same way they model the object of their study as they can start their analyses from objects that represent the animals in their habitat (which previously was the result of a long and complex process). Moreover, users can directly query these objects using a simple and powerful language (SQL) that is close to their natural language. All these elements strengthen the opportunity provided by GPS data to move from mainly testing statistical hypotheses to focusing on biological hypotheses. Scientists can store, access and manipulate their data in a simple and quick way, which

allows them to formulate biological questions that previously were almost impossible to answer for technical reasons.

This chapter shows how vector and raster layers can be included in the database, how you can handle them using (spatial) SQL and how you can associate with the GPS locations. In the next chapter, we will focus on raster time series using remote sensing images.

Adding Ancillary Environmental Layers

In the exercise, you will see how to integrate a number of spatial features (see Fig. 6.1).

- Points: meteorological stations (derived from MeteoTrentino[1]).
- Linestrings: roads network (derived from OpenStreetMap[2]).
- Polygons: administrative units (derived from ISTAT[3]) and the study area.
- Rasters: land cover (source: Corine[4]) and digital elevation models (source: SRTM[5], see also Jarvis et al. 2008).

Each species and study have specific data sets required and available, so the goal of this example is to show a complete set of procedures that can be replicated and customised on different data sets. When layers are integrated into the database, you can visualise and explore them in a GIS environment (e.g. QGIS).

Once data are loaded into the database, you will extend the *gps_data_animals* table with the environmental attributes derived from the ancillary layers provided in the test data set. You will also modify the function *tools.new_gps_data_animals* to compute these values automatically. In addition, you are encouraged to develop your own (spatial) queries (e.g. detect how many times each animal crosses a road, calculate how many times two animals are in the same place at the same time).

It is a good practice to store your environmental layers in a dedicated schema in order to keep a clear database structure. Let us create the schema *env_data*:

```
CREATE SCHEMA env_data
  AUTHORIZATION postgres;
GRANT USAGE ON SCHEMA env_data TO basic_user;
COMMENT ON SCHEMA env_data
IS 'Schema that stores environmental ancillary information.';
ALTER DEFAULT PRIVILEGES IN SCHEMA env_data
  GRANT SELECT ON TABLES TO basic_user;
```

[1] Provincia autonoma di Trento—Servizio Prevenzione Rischi—Ufficio Previsioni e pianificazione, http://www.meteotrentino.it.

[2] http://www.openstreetmap.org.

[3] http://www.istat.it/it/strumenti/cartografia.

[4] http://www.eea.europa.eu/data-and-maps/data/corine-land-cover-2006-clc2006-100-m-version-12-2009.

[5] http://srtm.csi.cgiar.org/.

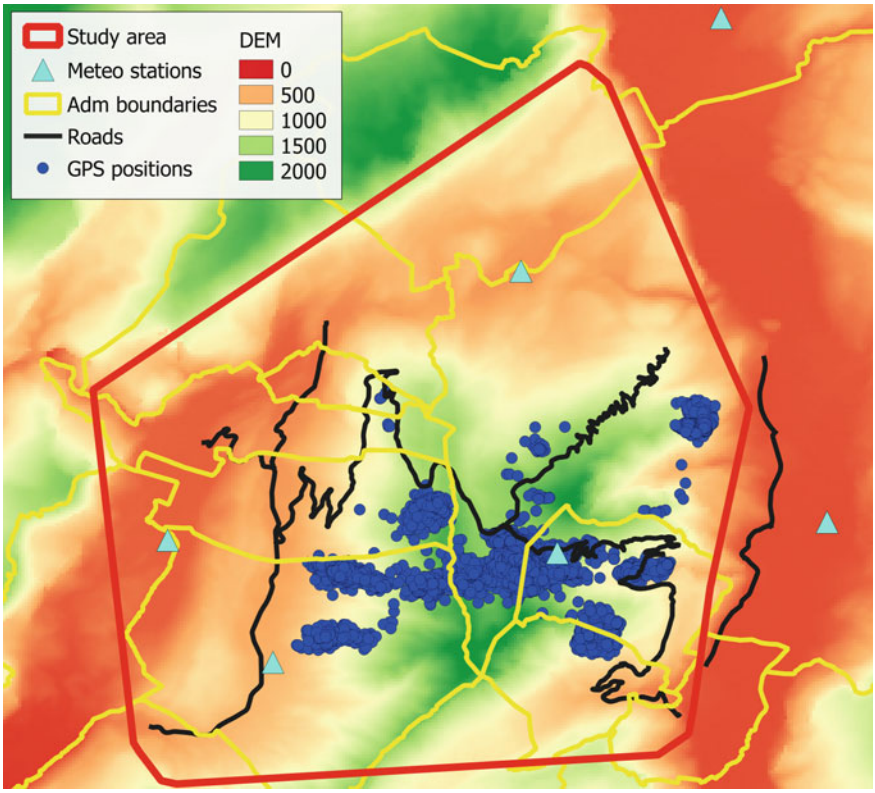

Fig. 6.1 Environmental layers that will be integrated into the database

Importing Shapefiles: Points, Lines and Polygons

Now you can start importing the shapefiles of the (vector) environmental layers included in the test data set. As discussed in Chap. 5, an option is to use the drag-and-drop function of 'DB Manager' (from QGIS Browser) plugin in QGIS (see Fig. 6.2).

Alternatively, a standard solution to import shapefiles (vector data) is the *shp2pgsql* tool. *shp2pgsql* is an external command-line tool, which cannot be run in an SQL interface as it can for a regular SQL command. The code below has to be run in a command-line interpreter (if you are using Windows as operating system, it is also called Command Prompt or MS-DOS shell, see Fig. 6.3). You will see other examples of external tools that are run in the same way, and it is very important to understand the difference between these and SQL commands. In this guide, this difference is represented graphically by white text boxes (see below) for shell commands, while the SQL code is shown in grey text boxes. Start with the meteorological stations:

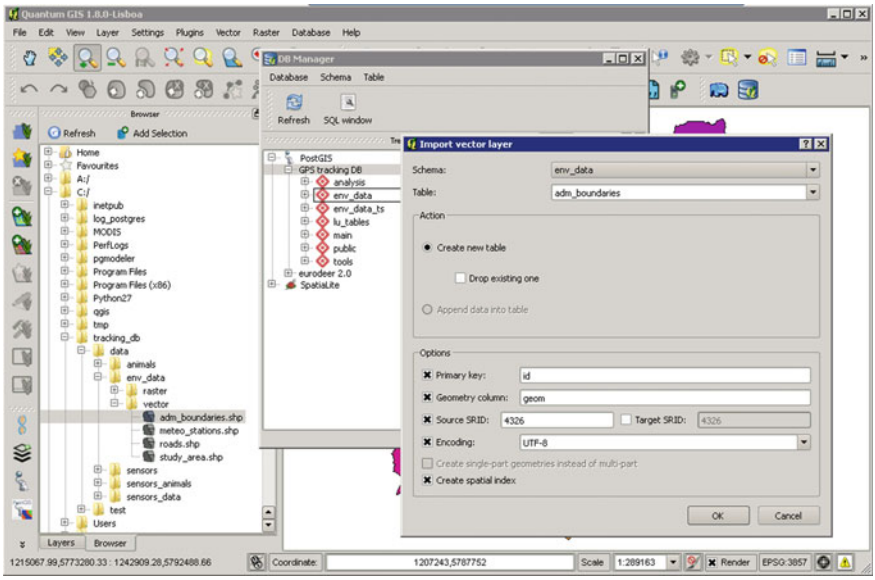

Fig. 6.2 Loading data into PostgreSQL using the drag-and-drop tool in QGIS

```
"C:\Program Files\PostgreSQL\9.2\bin\shp2pgsql.exe" -s 4326 -I
C:\tracking_db\data\env_data\vector\meteo_stations.shp
env_data.meteo_stations | "C:\Program Files\PostgreSQL\9.2\bin\psql.exe" -p
5432 -d gps_tracking_db -U postgres -h localhost
```

Note that the path to *shp2pgsql.exe* and *psql.exe* can be different according to the folder where you installed your version of PostgreSQL. If you connect with the database remotely, you also have to change the address of the server (*-h* option). In the parameters, set the reference system (option *-s*) and create a spatial index for the new table (option *-I*). The result of *shp2pgsql* is a text file with the SQL that generates and populates the table *env_data.meteo_stations*. With the symbol 'I' you 'pipe' (send directly) the SQL to the database (through the PostgreSQL interactive terminal *psql*[6]) where it is automatically executed. You have to set the port (*-p*), the name of the database (*-d*), the user (*-U*), and the password, if requested. In this way, you complete the whole process with a single command. You can refer to *shp2pgsql* documentation for more details. You might have to add the whole path to *psql* and *shp2pgsql*. This depends on the folder where you installed PostgreSQL. You can easily verify the path searching for these two files. You also have to check that the path of your shapefile (meteo_stations.shp) is properly defined.

You can repeat the same operation for the study area layer:

[6] http://www.postgresql.org/docs/9.2/static/app-psql.html.

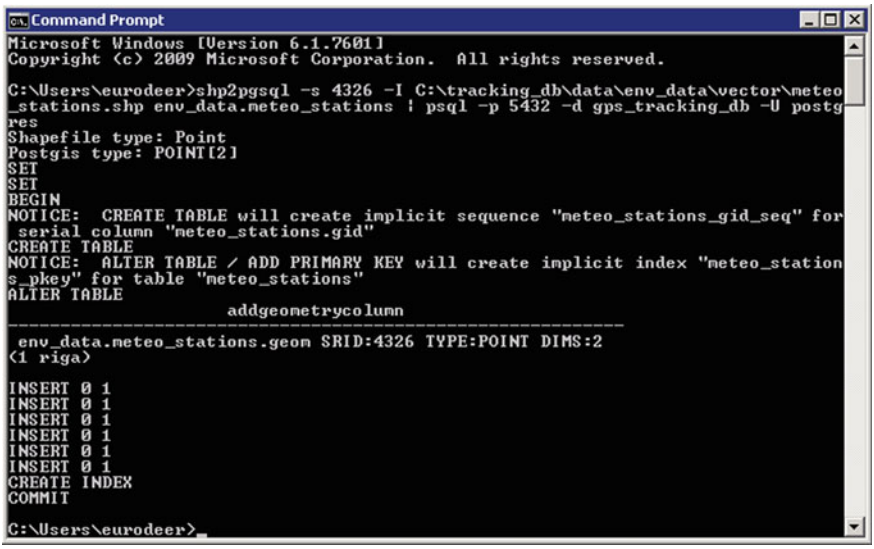

Fig. 6.3 The command shp2pgsql from Windows command-line interpreter

```
"C:\Program Files\PostgreSQL\9.2\bin\shp2pgsql.exe" -s 4326 -I
C:\tracking_db\data\env_data\vector\study_area.shp env_data.study_area |
"C:\Program Files\PostgreSQL\9.2\bin\psql.exe" -p 5432 -d gps_tracking_db -U
postgres -h localhost
```

Next for the roads layer

```
"C:\Program Files\PostgreSQL\9.2\bin\shp2pgsql.exe" -s 4326 -I
C:\tracking_db\data\env_data\vector\roads.shp env_data.roads | "C:\Program
Files\PostgreSQL\9.2\bin\psql.exe" -p 5432 -d gps_tracking_db -U postgres -h
localhost
```

And for the administrative boundaries

```
"C:\Program Files\PostgreSQL\9.2\bin\shp2pgsql.exe" -s 4326 -I
C:\tracking_db\data\env_data\vector\adm_boundaries.shp
env_data.adm_boundaries | "C:\Program Files\PostgreSQL\9.2\bin\psql.exe" -p
5432 -d gps_tracking_db -U postgres -h localhost
```

Now the shapefiles are in the database as new tables (one table for each shapefile). You can visualise them through a GIS interface (e.g. QGIS). You can also retrieve a summary of the information from all vector layers available in the database with the following command:

```
SELECT * FROM geometry_columns;
```

Importing Raster Files

The primary method to import a raster layer is the command-line tool *raster2pgsql*[7], the equivalent of *shp2pgsql*, but for raster files, that converts GDAL-supported rasters into SQL suitable for loading into PostGIS. It is also capable of loading folders of raster files.

Special Topic: **GDAL**

GDAL[8] (Geospatial Data Abstraction Library) is a (free) library for reading, writing and processing raster geospatial data formats. It has a lot of simple but very powerful and fast command-line tools for raster data translation and processing. The related OGR library provides a similar capability for simple vector data features. GDAL is used by most of the spatial open source tools and by a large number of commercial software programs as well. You will probably benefit in particular from the tools gdalinfo[9] (get a layer's basic metadata), gdal_translate[10] (change data format, change data type, cut), gdalwarp[11] (mosaicing, reprojection and warping utility).

An interesting feature of *raster2pgsql* is its capability to store the rasters inside the database (in-db) or keep them as (out-db) files in the file system (with the *raster2pgsql -R* option). In the last case, only rasters as metadata are stored in the database, not pixel values themselves. Loading out-db rasters as metadata is much faster than loading them completely in the database. Most operations at the pixel values level (e.g. *ST_SummaryStats*) will have equivalent performance with out- and in-db rasters. Other functions, like *ST_Tile*, involving only the metadata, will be faster with out-db rasters. Another advantage of out-db rasters is that they stay accessible for external applications unable to query databases (with SQL). However, the administrator must make sure that the link between what is in the db (the path to the raster file in the file system) is not broken (e.g. by moving or renaming the files). On the other hand, only in-db rasters can be generated with *CREATE TABLE* and modified with *UPDATE* statements. Which is the best choice depends on the size of the data set and on considerations about performance and database management. A good practice is generally to load very large raster data sets as out-db and to load smaller ones as in-db to save time on loading and to avoid repeatedly backing up huge, static rasters.

The QGIS plugin 'Load Raster to PostGIS' can also be used to import raster data with a graphical interface. An important parameter to set when importing raster layers is the number of tiles (*-t* option). Tiles are small subsets of the image and correspond to a physical record in the table. This approach dramatically

[7] http://postgis.net/docs/manual-2.0/using_raster.xml.html#RT_Raster_Loader.

[8] http://www.gdal.org/.

[9] http://www.gdal.org/gdalinfo.html.

[10] http://www.gdal.org/gdal_translate.html.

[11] http://www.gdal.org/gdalwarp.html.

decreases the time required to retrieve information. The recommended values for the tile option range from 20 × 20 to 100 × 100. Here is the code (to be run in the Command Prompt) to transform a raster (the digital elevation model derived from SRTM) into the SQL code that is then used to physically load the raster into the database (as you did with *shp2pgsql* for vectors):

```
"C:\Program Files\PostgreSQL\9.2\bin\raster2pgsql.exe" -I -M -C -s 4326 -t
20x20 C:\tracking_db\data\env_data\raster\srtm_dem.tif env_data.srtm_dem |
"C:\Program Files\PostgreSQL\9.2\bin\psql.exe" -p 5432 -d gps_tracking_db -U
postgres -h localhost
```

You can repeat the same process on the land cover layer:

```
"C:\Program Files\PostgreSQL\9.2\bin\raster2pgsql.exe" -I -M -C -s 3035
-t 20x20 C:\tracking_db\data\env_data\raster\corine06.tif
env_data.corine_land_cover | "C:\Program Files\PostgreSQL\9.2\bin\psql.exe"
-p 5432 -d gps_tracking_db -U postgres -h localhost
```

The reference system of the Corine land cover data set is not geographic coordinates (SRID 4326), but ETRS89/ETRS-LAEA (SRID 3035), an equal-area projection over Europe. This must be specified with the -s option and kept in mind when this layer will be connected to other spatial layers stored in a different reference system. As with *shp2pgsql.exe*, the *-I* option will create a spatial index on the loaded tiles, speeding up many spatial operations, and the *-C* option will generate a set of constraints on the table, allowing it to be correctly listed in the *raster_columns* metadata table. The land cover raster identifies classes that are labelled by a code (an integer). To specify the meaning of the codes, you can add a table where they are described. In this example, the land cover layer is taken from the Corine project[12]. Classes are described by a hierarchical legend over three nested levels. The legend is provided in the test data set in the file 'corine_legend.csv'. You import the table of the legend (first creating an empty table, and then loading the data):

```
CREATE TABLE env_data.corine_land_cover_legend(
  grid_code integer NOT NULL,
  clc_l3_code character(3),
  label1 character varying,
  label2 character varying,
  label3 character varying,
  CONSTRAINT corine_land_cover_legend_pkey
    PRIMARY KEY (grid_code ));
COMMENT ON TABLE env_data.corine_land_cover_legend
IS 'Legend of Corine land cover, associating the numeric code to the three
nested levels.';
```

[12] http://www.eea.europa.eu/publications/COR0-landcover.

Then, you load the data:

```
COPY env_data.corine_land_cover_legend
FROM
  'C:\tracking_db\data\env_data\raster\corine_legend.csv'
  WITH (FORMAT csv, HEADER, DELIMITER ';');
```

You can retrieve a summary of the information from all raster layers available in the database with the following command:

```
SELECT * FROM raster_columns;
```

To keep a well-documented database, add comments to describe all the spatial layers that you have added:

```
COMMENT ON TABLE env_data.adm_boundaries
IS 'Layer (polygons) of administrative boundaries (comuni).';
COMMENT ON TABLE env_data.corine_land_cover
IS 'Layer (raster) of land cover (from Corine project).';
COMMENT ON TABLE env_data.meteo_stations
IS 'Layer (points) of meteo stations.';
COMMENT ON TABLE env_data.roads
IS 'Layer (lines) of roads network.';
COMMENT ON TABLE env_data.srtm_dem
IS 'Layer (raster) of digital elevation model (from SRTM project).';
COMMENT ON TABLE env_data.study_area
IS 'Layer (polygons) of the boundaries of the study area.';
```

Querying Spatial Environmental Data

As the set of ancillary (spatial) information is now loaded into the database, you can start playing with this information using spatial SQL queries. In fact, it is possible with spatial SQL to run queries that explicitly handle the spatial relationships among the different spatial tables that you have stored in the database. In the following examples, SQL statements will show you how to take advantage of PostGIS features to manage, explore, and analyse spatial objects, with optimised performances and no need for specific GIS interfaces. You start by asking for the name of the administrative unit (‘comune’, Italian commune) in which the point at coordinates (11, 46) (longitude, latitude) is located. There are two commands that are used when it comes to intersection of spatial elements: *ST_Intersects* and *ST_Intersection*. The former returns *true* if two features intersect, while the latter returns the geometry produced by the intersection of the objects. In this case, *ST_Intersects* is used to select the right commune:

```
SELECT
  nome_com
FROM
  env_data.adm_boundaries
WHERE
  ST_Intersects((ST_SetSRID(ST_MakePoint(11,46), 4326)), geom);
```

The result is

```
nome_com
----------
Cavedine
```

In the second example, you compute the distance (rounded to the metre) from the point at coordinates (11, 46) to all the meteorological stations (ordered by distance) in the table *env_data.meteo_stations*. This information could be used, for example, to derive the precipitation and temperature for a GPS position at the given acquisition time, weighting the measurement from each station according to the distance from the point. In this case, *ST_Distance_Spheroid* is used. Alternatively, you could use *ST_Distance* and cast your geometries as *geography* data types.

```
SELECT
  station_id, ST_Distance_Spheroid((ST_SetSRID(ST_MakePoint(11,46), 4326)),
  geom, 'SPHEROID["WGS 84",6378137,298.257223563]')::integer AS distance
FROM
  env_data.meteo_stations
ORDER BY
  distance;
```

The result is

```
station_id | distance
-----------+----------
         1 |     2224
         2 |     4080
         5 |     4569
         4 |    10085
         3 |    10374
         6 |    18755
```

In the third example, you compute the distance to the closest road:

```
SELECT
  ST_Distance((ST_SetSRID(ST_MakePoint(11,46), 4326))::geography,
  geom::geography)::integer AS distance
FROM
  env_data.roads
ORDER BY
  distance
LIMIT 1;
```

The result is

```
 distance
----------
     1560
```

For users, the data type (vector, raster) used to store spatial information is not so relevant when they query their data: queries should transparently use any kind of spatial data as input. Users can then focus on the environmental model instead of worrying about the data model. In the next example, you intersect a point with two raster layers (altitude and land cover) in the same way you do for vector layers. In the case of land cover, the point must first be projected into the Corine reference system (SRID 3035). In the raster layer, just the Corine code class (integer) is stored while the legend is stored in the table *env_data.corine_land_cover_legend*. In the query, the code class is joined to the legend table and the code description is returned. This is an example of integration of both spatial and non-spatial elements in the same query.

```
SELECT
  ST_Value(srtm_dem.rast,
  (ST_SetSRID(ST_MakePoint(11,46), 4326))) AS altitude,
  ST_value(corine_land_cover.rast,
  ST_transform((ST_SetSRID(ST_MakePoint(11,46), 4326)), 3035)) AS land_cover,
  label2,label3
FROM
  env_data.corine_land_cover,
  env_data.srtm_dem,
  env_data.corine_land_cover_legend
WHERE
  ST_Intersects(corine_land_cover.rast,
    ST_Transform((ST_SetSRID(ST_MakePoint(11,46), 4326)), 3035)) AND
  ST_Intersects(srtm_dem.rast,(ST_SetSRID(ST_MakePoint(11,46), 4326))) AND
  grid_code = ST_Value(corine_land_cover.rast,
    ST_Transform((ST_SetSRID(ST_MakePoint(11,46), 4326)), 3035));
```

The result is

```
 altitude | land_cover | label2  |      label3
----------+------------+---------+----------------
      956 |         24 | Forests | Coniferous forest
```

Now, combine roads and administrative boundaries to compute how many metres of roads there are in each administrative unit. You first have to intersect the two layers (*ST_Intersection*), then compute the length (*ST_Length*) and summarise per administrative unit (*sum* associated with *GROUP BY* clause).

```
SELECT
  nome_com,
  sum(ST_Length(
    (ST_Intersection(roads.geom, adm_boundaries.geom))::geography))::integer
    AS total_length
FROM
  env_data.roads,
  env_data.adm_boundaries
WHERE
  ST_Intersects(roads.geom, adm_boundaries.geom)
GROUP BY
  nome_com
ORDER BY
  total_length desc;
```

The result of the query is

```
   nome_com    | total_length
---------------+--------------
 Trento        |        24552
 Lasino        |        15298
 Garniga Terme |        12653
 Calavino      |         6185
 Cavedine      |         5802
 Cimone        |         5142
 Padergnone    |         4510
 Vezzano       |         1618
 Aldeno        |         1367
```

The last examples are about the interaction between rasters and polygons. In this case, you compute some statistics (minimum, maximum, mean and standard deviation) for the altitude within the study area:

```
SELECT
  (sum(ST_Area(((gv).geom)::geography)))/1000000 area,
  min((gv).val) alt_min,
  max((gv).val) alt_max,
  avg((gv).val) alt_avg,
  stddev((gv).val) alt_stddev
FROM
  (SELECT
    ST_intersection(rast, geom) AS gv
  FROM
    env_data.srtm_dem,
    env_data.study_area
  WHERE
    ST_intersects(rast, geom)
) foo;
```

The result, from which it is possible to appreciate the large variability of altitude across the study area, is

```
       area        | alt_min | alt_max |      alt_avg       |    alt_stddev
-------------------+---------+---------+--------------------+------------------
 199.018552456188 |     180 |    2133 | 879.286157704969 | 422.56622698974
```

You might also be interested in the number of pixels of each land cover type within the study area. As with the previous example, you first intersect the study area with the raster of interest, but in this case, you need to reproject the study area polygon into the coordinate system of the Corine land cover raster (SRID 3035). With the following query, you can see the dominance of mixed forests in the study area:

```
SELECT (pvc).value, SUM((pvc).count) AS total, label3
FROM
  (SELECT ST_ValueCount(rast) AS pvc
  FROM env_data.corine_land_cover, env_data.study_area
  WHERE ST_Intersects(rast, ST_Transform(geom, 3035))) AS cnts,
  env_data.corine_land_cover_legend
WHERE grid_code = (pvc).value
GROUP BY (pvc).value, label3
ORDER BY (pvc).value;
```

The result is

```
 lc_class | total |                    label3
----------+-------+-----------------------------------------------
        1 |   114 | Continuous urban fabric
        2 |   817 | Discontinuous urban fabric
        3 |   324 | Industrial or commercial units
        7 |   125 | Mineral extraction sites
       16 |   324 | Fruit trees and berry plantations
       18 |   760 | Pastures
       19 |   237 | Annual crops associated with permanent crops
       20 |  1967 | Complex cultivation patterns
       21 |  2700 | Land principally occupied by agriculture
       23 |  4473 | Broad-leaved forest
       24 |  2867 | Coniferous forest
       25 |  8762 | Mixed forest
       26 |   600 | Natural grasslands
       27 |   586 | Moors and heathland
       29 |  1524 | Transitional woodland-shrub
       31 |   188 | Bare rocks
       32 |   611 | Sparsely vegetated areas
       41 |   221 | Water bodies
```

The previous query can be modified to return the percentage of each class over the total number of pixels. This can be achieved using window functions[13]:

[13] http://www.postgresql.org/docs/9.2/static/tutorial-window.html.

```
SELECT
  (pvc).value,
  (SUM((pvc).count)*100/ SUM(SUM((pvc).count)) over ())::numeric(4,2)
   AS total_perc, label3
FROM
  (SELECT ST_ValueCount(rast) AS pvc
   FROM env_data.corine_land_cover, env_data.study_area
   WHERE ST_Intersects(rast, ST_Transform(geom, 3035))) AS cnts,
   env_data.corine_land_cover_legend
WHERE grid_code = (pvc).value
GROUP BY (pvc).value, label3
ORDER BY (pvc).value;
```

The result is

```
 value | total_perc |                    label3
-------+------------+-----------------------------------------------
     1 |       0.42 | Continuous urban fabric
     2 |       3.00 | Discontinuous urban fabric
     3 |       1.19 | Industrial or commercial units
     7 |       0.46 | Mineral extraction sites
    16 |       1.19 | Fruit trees and berry plantations
    18 |       2.79 | Pastures
    19 |       0.87 | Annual crops associated with permanent crops
    20 |       7.23 | Complex cultivation patterns
    21 |       9.93 | Land principally occupied by agriculture
    23 |      16.44 | Broad-leaved forest
    24 |      10.54 | Coniferous forest
    25 |      32.21 | Mixed forest
    26 |       2.21 | Natural grasslands
    27 |       2.15 | Moors and heathland
    29 |       5.60 | Transitional woodland-shrub
    31 |       0.69 | Bare rocks
    32 |       2.25 | Sparsely vegetated areas
    41 |       0.81 | Water bodies
```

Associate Environmental Characteristics with GPS Locations

After this general introduction to the use of spatial SQL to explore spatial layers, you can now use these tools to associate environmental characteristics with GPS positions. You can find a more extended introduction to spatial SQL in Obe and Hsu (2011). The goal here is to automatically transform position data from simple points to objects holding information about the habitat and conditions where the animals were located at a certain moment in time. You will use the points to automatically extract, by the mean of an SQL trigger, this information from other ecological layers. The first step is to add the new fields of information into the

main.gps_data_animals table. You will add columns for the name of the administrative unit to which the GPS position belongs, the code for the land cover it is located in, the altitude from the digital elevation model (which can then be used as the third dimension of the point), the id of the closest meteorological station and the distance to the closest road:

```
ALTER TABLE main.gps_data_animals
  ADD COLUMN pro_com integer;
ALTER TABLE main.gps_data_animals
  ADD COLUMN corine_land_cover_code integer;
ALTER TABLE main.gps_data_animals
  ADD COLUMN altitude_srtm integer;
ALTER TABLE main.gps_data_animals
  ADD COLUMN station_id integer;
ALTER TABLE main.gps_data_animals
  ADD COLUMN roads_dist integer;
```

These are several common examples of environmental information that can be associated with GPS positions, and others can be implemented according to specific needs. It is important to keep in mind that these spatial relationships are implicitly determined by the coordinates of the elements involved; you do not necessarily have to store these values in a table as you can compute them on the fly whenever you need. Moreover, you might need different information according to the specific study (e.g. the land cover composition in an area of 1 km around each GPS position instead of the value of the pixel where the point is located). Computing these spatial relationships on the fly can require significant time, so in some cases, it is preferable to run the query just once and permanently store the most relevant parameters for your specific study (think about what you will most likely use often). Another advantage of making the relations explicit within tables is that you can then create indexes on columns of these tables. This is not possible with on-the-fly sub-queries. Making many small queries and hence creating many tables and indexing them along the way is generally more efficient in terms of processing time than trying to do everything in a long and complex query. This is not necessarily true when the data set is small enough, as indexes are mostly efficient on large tables. Sometimes, the time necessary to write many SQL statements and the associated indexes exceed the time necessary to execute them. In that case, it might be more efficient to write a single, long and complex statement and forget about the indexes. This does not apply to the following trigger function, as all the ecological layers were well indexed at load time and it does not rely on intermediate sub-queries of those layers.

The next step is to implement the computation of these parameters inside the automated process of associating GPS positions with animals (from *gps_data* to *gps_data_animals*). To achieve this goal, you have to modify the trigger function *tools.new_gps_data_animals*. In fact, the function *tools.new_gps_data_animals* is activated whenever a new location is inserted into *gps_data_animals* (from *gps_data*). It adds new information (i.e. fills additional fields) to the incoming

record (e.g. creates the geometry object from latitude and longitude values) before it is uploaded into the *gps_data_animals* table in the code, *NEW* is used to reference the new record not yet inserted). The SQL code that does this is below. The drawback of this function is that it will slow down the import of a large set of positions at once (e.g. millions or more), but it has little impact when you manage a continuous data flow from sensors, even for a large number of sensors deployed at the same time.

```
CREATE OR REPLACE FUNCTION tools.new_gps_data_animals()
RETURNS trigger AS
$BODY$
DECLARE
  thegeom geometry;
BEGIN

IF NEW.longitude IS NOT NULL AND NEW.latitude IS NOT NULL THEN
  thegeom = ST_SetSRID(ST_MakePoint(NEW.longitude, NEW.latitude), 4326);
  NEW.geom =thegeom;
  NEW.pro_com =
    (SELECT pro_com::integer
     FROM env_data.adm_boundaries
     WHERE ST_Intersects(geom,thegeom));
  NEW.corine_land_cover_code =
    (SELECT ST_Value(rast,ST_Transform(thegeom,3035))
     FROM env_data.corine_land_cover
     WHERE ST_Intersects(ST_Transform(thegeom,3035), rast));
  NEW.altitude_srtm =
    (SELECT ST_Value(rast,thegeom)
     FROM env_data.srtm_dem
     WHERE ST_Intersects(thegeom, rast));
  NEW.station_id =
    (SELECT station_id::integer
     FROM env_data.meteo_stations
     ORDER BY ST_Distance_Spheroid(thegeom, geom, 'SPHEROID["WGS 84",
     6378137,298.257223563]')
     LIMIT 1);
  NEW.roads_dist =
    (SELECT ST_Distance(thegeom::geography, geom::geography)::integer
     FROM env_data.roads
     ORDER BY ST_distance(thegeom::geography, geom::geography)
     LIMIT 1);
END IF;

RETURN NEW;
END;$BODY$
LANGUAGE plpgsql VOLATILE
COST 100;
COMMENT ON FUNCTION tools.new_gps_data_animals()
IS 'When called by the trigger insert_gps_positions (raised whenever a new
position is uploaded into gps_data_animals) this function gets the longitude
and latitude values and sets the geometry field accordingly, computing a set
of derived environmental information calculated intersecting or relating the
position with the environmental ancillary layers.';
```

As the trigger function is run during GPS data import, the function only works on the records that are imported after it was created, and not on data imported previously. To see the effects, you have to add new positions or delete and reload the GPS positions stored in *gps_data_animals.* You can do this by saving the records in *gps_sensors_animals* in an external .csv file, and then deleting the records from the table (which also deletes the records in *gps_data_animals* in a cascade effect). When you reload them, the new function will be activated by the trigger that was just defined, and the new attributes will be calculated. You can perform these steps with the following commands.

First, check how many records you have per animal:

```
SELECT animals_id, count(animals_id)
FROM main.gps_data_animals
GROUP BY animals_id;
```

The result is

```
 animals_id | count
------------+-------
          4 |  2869
          5 |  2924
          2 |  2624
          1 |  2114
          3 |  2106
```

Then, copy the table *main.gps_sensors_animals* into an external file.

```
COPY
  (SELECT animals_id, gps_sensors_id, start_time, end_time, notes
   FROM main.gps_sensors_animals)
TO
  'c:/tracking_db/test/gps_sensors_animals.csv'
  WITH (FORMAT csv, DELIMITER ';');
```

You then delete all the records in *main.gps_sensors_animals*, which will delete (in a cascade) all the records in *main.gps_data_animals.*

```
DELETE FROM main.gps_sensors_animals;
```

You can verify that there are now no records in *main.gps_data_animals* (the query should return 0 rows).

```
SELECT * FROM main.gps_data_animals;
```

The final step is to reload the .csv file into *main.gps_sensors_animals*. This will launch the trigger functions that recreate all the records in *main.gps_data_animals*,

in which the fields related to environmental attributes are also automatically updated. Note that, due to the different triggers that imply massive computations, the query can take several minutes to execute[14].

```
COPY main.gps_sensors_animals
  (animals_id, gps_sensors_id, start_time, end_time, notes)
FROM
  'c:/tracking_db/test/gps_sensors_animals.csv'
  WITH (FORMAT csv, DELIMITER ';');
```

You can verify that all the fields are updated:

```
SELECT
  gps_data_animals_id AS id, acquisition_time, pro_com,
  corine_land_cover_code AS lc_code, altitude_srtm AS alt, station_id AS
  meteo, roads_dist AS dist
FROM
  main.gps_data_animals
WHERE
  geom IS NOT NULL
LIMIT 10;
```

The result is

```
 id    |    acquisition_time    | pro_com | lc_code | alt  | meteo | dist
-------+------------------------+---------+---------+------+-------+------
15275 | 2005-10-23 22:00:53+02 |   22091 |      18 | 1536 |     5 |  812
15276 | 2005-10-24 02:00:55+02 |   22091 |      18 | 1519 |     5 |  740
15277 | 2005-10-24 06:00:55+02 |   22091 |      18 | 1531 |     5 |  598
15280 | 2005-10-24 18:02:57+02 |   22091 |      23 | 1198 |     5 |  586
15281 | 2005-10-24 22:01:49+02 |   22091 |      25 | 1480 |     5 |  319
15282 | 2005-10-25 02:01:23+02 |   22091 |      18 | 1531 |     5 |  678
15283 | 2005-10-25 06:00:53+02 |   22091 |      18 | 1521 |     5 |  678
15284 | 2005-10-25 10:01:10+02 |   22091 |      23 | 1469 |     5 |  546
15285 | 2005-10-25 14:01:26+02 |   22091 |      23 | 1412 |     5 |  571
15286 | 2005-10-25 18:02:29+02 |   22091 |      23 | 1435 |     5 |  465
```

You can also check that all the records of every animal are in *main.gps_data_animals*:

```
SELECT animals_id, count(*) FROM main.gps_data_animals GROUP BY animals_id;
```

[14] You can skip this step and speedup the process by simply calculating the environmental attributes with an update query.

The result is

```
 animals_id | count
------------+-------
          4 |  2869
          5 |  2924
          2 |  2624
          1 |  2114
          3 |  2106
```

As you can see, the whole process can take a few minutes, as you are calculating the environmental attributes for the whole data set at once. As discussed in the previous chapters, the use of triggers and indexes to automatise data flow and speedup analyses might imply processing times that are not sustainable when large data sets are imported at once. In this case, it might be preferable to update environmental attributes and calculate indexes in a later stage to speed up the import process. In this book, we assume that in the operational environment where the database is developed, the data flow is continuous, with large but still limited data sets imported at intervals. You can compare this processing time with what is generally required to achieve the same result in a classic GIS environment based on flat files (e.g. shapefiles, .tif). Do not forget to consider that you can use these minutes for a coffee break, while the database does the job for you, instead of clicking here and there in your favourite GIS application!

References

Jarvis A, Reuter HI, Nelson A, Guevara E (2008) Hole-filled seamless SRTM data V4. International centre for tropical agriculture (CIAT)

Obe OR, Hsu LS (2011) PostGIS in action. Manning Publications Company, Greenwich