Chapter 2
Storing Tracking Data in an Advanced Database Platform (PostgreSQL)

Ferdinando Urbano and Holger Dettki

Abstract The state-of-the-art technical tool for effectively and efficiently managing tracking data is the spatial relational database. Using databases to manage tracking data implies a considerable effort for those who are not already familiar with these tools, but this is necessary to be able to deal with the data coming from the new sensors. Moreover, the time spent to learn databases will be largely paid back with the time saved for the management and processing of the data. In this chapter, you are guided through how to set up a new database in which you will create a table to accommodate the test GPS data sets. You create a new table in a dedicated schema. We describe how to upload the raw GPS data coming from five sensors deployed on roe deer in the Italian Alps into the database and provide further insight into time-related database data types and into the creation of additional database users. The reference software platform used is the open source PostgreSQL with its spatial extension PostGIS. This tool is introduced with its graphical interface pgAdmin. All the examples provided (SQL code) and technical solutions proposed are tuned on this software, although most of the code can be easily adapted for other platforms. The book is focused on the applications of spatial databases to the specific domain of movement ecology: to properly understand the content of this guide and to replicate the proposed database structure and tools, you will need a general familiarity with GIS, wildlife tracking and database programming.

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

H. Dettki
Umeå Center for Wireless Remote Animal Monitoring (UC-WRAM),
Department of Wildlife, Fish, and Environmental Studies,
Swedish University of Agricultural Sciences (SLU), Skogsmarksgränd,
SE-901 83 Umeå, Sweden
e-mail: holger.dettki@slu.se

Introduction

The first step in the data management process is the acquisition of data recorded by GPS sensors deployed on animals. This can be a complex process that can also change depending on the GPS sensor provider and exceeds the scope of this book. The procedure presented in this chapter assumes that the raw information recorded by the GPS sensors is already available in the form of plain text (*.txt) or comma delimited (*.csv) files. In the Special Topic below, we give some insight on how this can be accomplished.

Special Topic: **Data acquisition from GPS sensors**

Depending on the GPS sensor provider and the technical solutions for data telemetry present in the GPS sensor units deployed on the animals, the data can be obtained in different ways. For near real-time applications it is important to automate the acquisition process as much as possible to fully exploit the information that the continuous data flow potentially provides and to avoid delay or even data loss. Other applications may be satisfied by eventual downloads, which can be handled manually, as long as not too many sensors are involved over a longer period. Most GPS sensor providers offer a so-called 'Push'[1] solution in combination with proprietary communication software to parse the incoming data from an encrypted format into a text format. For this, some providers allow the users to set up their own receiving stations to receive, e.g. data-SMS or data through a mobile data link, using a local GSM- or UMTS-modem together with the proprietary software to decode and export the data into a text file, while other providers use a company-based receiving system and forward the data by simply sending emails with the data in the email body or as email attachment to the user. The same routine is often chosen by providers when the original raw data are received through satellite systems (e.g. ARGOS, Iridium, Globalstar). Many of these communication software tools can be configured easily to automatically receive data in regular intervals and export it as text files into a pre-defined directory, without any user intervention. The procedure is slightly more complicated when emails are used. However, using simple programming languages like Python or Visual-Basic, any IT programmer can quickly write a program to extract data from an email body or extract an email attachment containing the data. There are also a number of free programs on the Internet which can accomplish this task. One of the most elegant ways to push data from the provider's database into the tracking database is to enable a direct link between the two databases and use triggers to move or copy new data. Although this is rarely a technical problem, unfortunately it often fails due to security considerations on the provider or user side. Other tracking applications do not need near real-time access to the data and may therefore handle the data download manually in regular or irregular intervals. The classical situation is the use of 'store-on-board' GPS units, where all data are stored in internal memory on-board the GPS devices until they are removed from the animal after a pre-defined period. The user then downloads the stored data using a cable connection between the GPS unit and the providers' proprietary communication software to parse and export a text file for each unit. This is called a typical 'Pull'[2] solution, which is offered by nearly every GPS unit provider. When the GPS sensor units are equipped with short-range radio signal telemetry transmitters (UHF/VHF), the user can pull the data manually from

---

[1] The information is 'pushed' from the unit or the provider to the user without user intervention.

[2] The user 'Pulls' the information manually from the unit or the providers' database without provider intervention.

the units using a custom receiver in the field after approaching the animals. Again, the providers' proprietary communication software can then be used to manually download, parse and export a text file for each unit from the custom receiver. When the GPS units are equipped for automatic data transfer via telemetry into the provider's database, some providers offer a manual download option from their database through, e.g. the Telnet protocol, or from their website. Here usually a manual login is required, after which data for some or all animals can be downloaded as a text file. While any of the 'Pull' options are fine as long as few GPS units are deployed for a relatively short time period, it is advisable to try in any tracking application to use 'Push' services and automation from the beginning, as building solutions around manual 'Pull' services can become very costly in human resources when the amount of deployed units increases over time.

As discussed in the previous chapter, the state-of-the-art technical tool to effectively and efficiently manage tracking data is the spatial relational database (Urbano et al. 2010). In this chapter's exercise, you will be guided through how to set up a new database in which you will create a table to accommodate the test GPS data sets. You will see how to upload data from source files using specific database tools.

The reference software platform used in this guide is PostgreSQL, along with its spatial extension PostGIS. All the examples provided and the technical solutions proposed are tuned on this software, although most of the code can be easily adapted for other platforms. There are many reasons that support the choice of PostgreSQL/PostGIS:

- both are free and open source, so any available financial resources can be used for customisation instead of software licences, and they can be used by research groups working with limited funds;
- PostgreSQL is an advanced and widely used database system and offers many features useful for animal movement data management;
- PostGIS is currently one of the most, if not the most, advanced database spatial extensions available and its development by the IT community is very fast;
- PostGIS includes support for raster data, a dedicated geography spatial data type, topology and networks, and has a very large library of spatial functions;
- there is a wide, active and very collaborative community for both PostgreSQL and PostGIS;
- there is very good documentation for both PostgreSQL and PostGIS (manuals, tutorials, books, wiki, blogs);
- PostgreSQL and PostGIS widely implement standards, which make them highly interoperable with a large set of other tools for data management, analysis, visualisation and dissemination;
- they are available for all the most common operating systems and CPU architectures, notably x86 and x86_64.

Although we recommend using PostgreSQL/PostGIS to develop your data management system, a valid open source alternative is SpatiaLite[3] if the number of sensors and researchers involved is small and basic functionality is needed.

---

[3] http://ww.gaia-gis.it/spatialite/.

SpatiaLite has a more limited set of functions compared with PostgreSQL/PostGIS and is not designed for concurrent access for multiple users, but it has the advantage of being a single and portable file with no need for any software installation. Another option is to use existing e-infrastructures dedicated to wildlife tracking data, such as WRAM[4] or Movebank[5]. These have the advantage of offering a Web-based 'ready to use' management system, with a lot of additional features for data sharing. Data are hosted in an external server managed by these projects, so no database maintenance is needed. Movebank offers also a large set of tools to support collaborations among scientists with both local and global perspectives. On the other hand, these systems are designed for large numbers of users and have limited support for specific customisation. It is also easy to foresee that in the future data management will be increasingly provided by GPS sensor vendors as a basic service.

As mentioned earlier, technical explanations about the basics of GIS, database programming and wildlife tracking are not covered in these exercises. When needed, references to specific documentation (e.g. Web pages, books, articles, tutorials) are given. We recommend that readers become familiar with PostgreSQL and PostGIS before reading this text. In particular, whenever you need any information on these two software tools, we suggest you begin by consulting the official documentation[6].

This book is focused on the applications of these technical tools to the specific domain of movement ecology. We also encourage you to use the database management interface pgAdmin[7] to build and manage PostgreSQL databases. It is installed automatically with PostgreSQL. It has a user-friendly graphical interface and a set of tools to facilitate interaction with the database. On the pgAdmin website, you can find the necessary documentation. In addition, the Web-based tool phpPgAdmin[8] can be used to manage a PostgreSQL database and retrieve data remotely without installing any software on the client side.

Create a New Database

The first step to create the database is the installation of PostgreSQL. Once you have downloaded[9] and installed[10] the software (the release used to test the code in this guide is 9.2), you can use the 'Application Stack Builder' (included with

---

[4] http://www.slu.se/wram/.

[5] http://www.movebank.org/.

[6] http://www.postgresql.org/docs/, http://postgis.refractions.net/documentation/.

[7] http://www.pgadmin.org/.

[8] http://phppgadmin.sourceforge.net/.

[9] http://www.postgresql.org/download/.

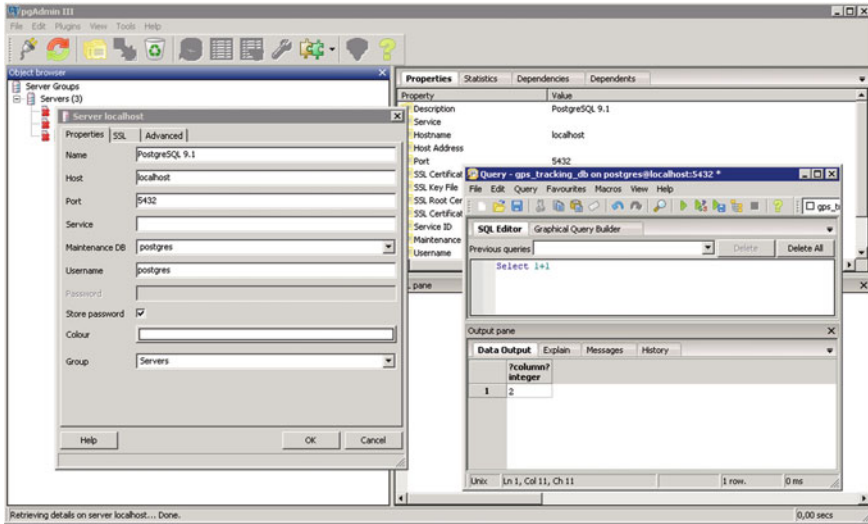[10] http://wiki.postgresql.org/wiki/Detailed_installation_guide/.

Fig. 2.1 pgAdmin GUI to PostgreSQL. On the *left* is the interface to introduce the connection parameters, and on the *right* is the SQL editor window

PostgreSQL) to get other useful tools, in particular the spatial extension PostGIS (the release used as reference here is 2.0).

During the installation process, you will be able to create the database super-user (the default 'postgres' will be used throughout this guide; don't forget the password!) and set the value for the port (default '5432'). You will need this information together with the IP address of your computer to connect to the database. In case you work directly on the computer where the database is installed, the IP address is also aliased as 'localhost'. If you want your database to be remotely accessible, you must verify that the port is open for external connections. The exercises in this guide use a test data set that includes information from five GPS Vectronic Aerospace GmbH sensors deployed on five roe deer monitored in the Italian Alps (Monte Bondone, Trento), kindly provided by Fondazione Edmund Mach, Trento, Italy. This information is complemented with a variety of (free) environmental layers (locations of meteorological stations, road networks, administrative units, land cover, DEM and NDVI time series). The test data set is available at http://extras.springer.com.

Once you have your PostgreSQL system up and running, you can start using it with the help of SQL queries. Note that you can run SQL code from a PSQL command-line interface[11] or from a graphical SQL interface. The PostgreSQL graphical user interface pgAdmin makes it possible to create all the database objects with user-friendly tools that drive users to define all the required information. Figure 2.1 shows an example of the pgAdmin graphical interface.

---

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

The very first thing to do, even before importing the raw sensor data, is to create a new database with the SQL code[12]

```
CREATE DATABASE gps_tracking_db
  ENCODING = 'UTF8'
  TEMPLATE = template0
  LC_COLLATE = 'C'
  LC_CTYPE = 'C';
```

You could create the database using just the first line of the code[13]. The other lines are added just to be sure that the database will use UTF8 as encoding system and will not be based on any local setting regarding, e.g. alphabets, sorting, or number formatting. This is very important when you work in an international environment where different languages (and therefore characters) can potentially be used.

Create a New Table and Import Raw GPS Data

Now you connect to the database (in pgAdmin you have to double-click on the icon of your database) in order to create a schema (a kind of 'folder' where you store a set of information[14]). By default, a database comes with the 'public' schema; it is good practice, however, to use different schemas to store user data. Here, you create a new schema called *main*:

```
CREATE SCHEMA main;
```

And then you can add a comment to describe the schema:

```
COMMENT ON SCHEMA main IS 'Schema that stores all the GPS tracking core
data.';
```

Comments are stored into the database. They are not strictly needed, but adding a description to every object that you create is a good practice and an important element of effective documentation for your system.

Before importing the GPS data sets into the database, it is recommended that you examine the source data (usually .dbf, .csv, or .txt files) with a spreadsheet or a text editor to see what information is contained. Every GPS brand/model can produce

---

[12] You can also find a plain text file with the SQL code proposed in the book in the trackingDB_code.zip file available at http://extras.springer.com.

[13] http://www.postgresql.org/docs/9.2/static/sql-createdatabase.html.

[14] http://www.postgresql.org/docs/9.2/static/ddl-schemas.html.

different information, or at least organise this information in a different way, as unfortunately no consolidated standards exist yet (see Chap. 13). The idea is to import raw data (as they are when received from the sensors) into the database and then process them to transform data into information. Once you identify which attributes are stored in the original files, you can create the structure of a table with the same columns, with the correct data types. The list of the main data types available in PostgreSQL/PostGIS is available in the official PostgreSQL documentation[15]. You can find the GPS data sets in .csv files included in the trackingDB_datasets.zip file with test data in the sub-folder \tracking_db\data\sensors_data.

The SQL code that generates the same table structure of the source files within the database, which is called here *main.gps_data*[16], is

```
CREATE TABLE main.gps_data(
  gps_data_id serial,
  gps_sensors_code character varying,
  line_no integer,
  utc_date date,
  utc_time time without time zone,
  lmt_date date,
  lmt_time time without time zone,
  ecef_x integer,
  ecef_y integer,
  ecef_z integer,
  latitude double precision,
  longitude double precision,
  height double precision,
  dop double precision,
  nav character varying(2),
  validated character varying(3),
  sats_used integer,
  ch01_sat_id integer,
  ch01_sat_cnr integer,
  ch02_sat_id integer,
  ch02_sat_cnr integer,
  ch03_sat_id integer,
  ch03_sat_cnr integer,
  ch04_sat_id integer,
  ch04_sat_cnr integer,
  ch05_sat_id integer,
  ch05_sat_cnr integer,
  ch06_sat_id integer,
  ch06_sat_cnr integer,
  ch07_sat_id integer,
  ch07_sat_cnr integer,
  ch08_sat_id integer,
  ch08_sat_cnr integer,
```

---

[15] http://www.postgresql.org/docs/9.2/static/datatype.html.

[16] 'Main' is the name of the schema where the table will be created, while 'gps_data' is the name of the table. Any object in the database is referred by combining the name of the schema and the name of the object (e.g. table) separated by a dot ('.').

```
  ch09_sat_id integer,
  ch09_sat_cnr integer,
  ch10_sat_id integer,
  ch10_sat_cnr integer,
  ch11_sat_id integer,
  ch11_sat_cnr integer,
  ch12_sat_id integer,
  ch12_sat_cnr integer,
  main_vol double precision,
  bu_vol double precision,
  temp double precision,
  easting integer,
  northing integer,
  remarks character varying
);
COMMENT ON TABLE main.gps_data
IS 'Table that stores raw data as they come from the sensors (plus the ID of
the sensor).';
```

In a relational database, each table should have a primary key: a field (or combination of fields) that uniquely identifies each record. In this case, you added a serial[17] field (*gps_data_id*) not present in the original file. As a serial data type, it is managed by the database and will be unique for each record. You set this field as the primary key of the table:

```
ALTER TABLE main.gps_data
   ADD CONSTRAINT gps_data_pkey
   PRIMARY KEY(gps_data_id);
```

To keep track of database changes, it is useful to add another field to store the time when each record was inserted in the table. The default for this field can be automatically set using the current time using the function *now*()[18]:

```
ALTER TABLE main.gps_data
   ADD COLUMN insert_timestamp timestamp with time zone
   DEFAULT now();
```

If you want to prevent the same record from being imported twice, you can add a unique constraint on the combination of the fields *gps_sensors_code* and *line_no*:

```
ALTER TABLE main.gps_data
   ADD CONSTRAINT unique_gps_data_record
   UNIQUE(gps_sensors_code, line_no);
```

---

[17] http://www.postgresql.org/docs/9.2/static/datatype-numeric.html#DATATYPE-SERIAL.

[18] http://www.postgresql.org/docs/9.2/static/functions-datetime.html.

In case a duplicated record is imported, the whole import procedure fails. You must verify if the above condition on the two fields is reasonable in your case (e.g. the GPS sensor might produce two records with the same values for GPS sensor code and line number). This check also implies some additional time in the import stage.

You are now ready to import the GPS data sets. There are many ways to do it. The main one is to use the *COPY*[19] (*FROM*) command setting the appropriate parameters (*COPY* plus the name of the target table, with the list of column names in the same order as they are in the source file, then *FROM* with the path to the file and *WITH* followed by additional parameters; in this case, 'csv' specifies the format of the source file, *HEADER* means that the first line in the source file is a header and not a record and *DELIMITER* ';' defines the field separator in the source file). Do not forget to change the *FROM* argument to match the actual location of the file on your computer:

```
COPY main.gps_data(
  gps_sensors_code, line_no, utc_date, utc_time, lmt_date, lmt_time, ecef_x,
  ecef_y, ecef_z, latitude, longitude, height, dop, nav, validated, sats_used,
  ch01_sat_id, ch01_sat_cnr, ch02_sat_id, ch02_sat_cnr, ch03_sat_id,
  ch03_sat_cnr, ch04_sat_id, ch04_sat_cnr, ch05_sat_id, ch05_sat_cnr,
  ch06_sat_id, ch06_sat_cnr, ch07_sat_id, ch07_sat_cnr, ch08_sat_id,
  ch08_sat_cnr, ch09_sat_id, ch09_sat_cnr, ch10_sat_id, ch10_sat_cnr,
  ch11_sat_id, ch11_sat_cnr, ch12_sat_id, ch12_sat_cnr, main_vol, bu_vol,
  temp, easting, northing, remarks)
FROM
  'C:\tracking_db\data\sensors_data\GSM01438.csv'
  WITH (FORMAT csv, HEADER, DELIMITER ';')
```

If PostgreSQL complains that date is out of range, check the standard date format used by your database:

```
SHOW datestyle;
```

If it is not 'ISO, DMY' (Day, Month, Year), then you have to set the date format in the same session of the *COPY* statement:

```
SET SESSION datestyle = "ISO, DMY";
```

If the original files are in .dbf format, you can use the pgAdmin tool 'Shapefile and .dbf importer'. In this case, you do not have to create the structure of the table before importing because it is done automatically by the tool, that tries to guess the

---

[19] See http://www.postgresql.org/docs/9.2/static/sql-copy.html or http://wiki.postgresql.org/wiki/COPY. If you want to upload on a server a file that is located in another machine, you have to use the command '\COPY'; see http://www.postgresql.org/docs/9.2/static/app-psql.html for more information. pgAdmin offers a user-friendly interface for data upload from text files.

right data type for each attribute. This might save some time but you lose control over the definition of data types (e.g. time can be stored as a text value).

Let us also import three other (sensor GSM02927 will be imported in Chap. 4) GPS data sets using the same *COPY* command:

```
COPY main.gps_data(
  gps_sensors_code, line_no, utc_date, utc_time, lmt_date, lmt_time, ecef_x,
  ecef_y, ecef_z, latitude, longitude, height, dop, nav, validated, sats_used,
  ch01_sat_id, ch01_sat_cnr, ch02_sat_id, ch02_sat_cnr, ch03_sat_id,
  ch03_sat_cnr, ch04_sat_id, ch04_sat_cnr, ch05_sat_id, ch05_sat_cnr,
  ch06_sat_id, ch06_sat_cnr, ch07_sat_id, ch07_sat_cnr, ch08_sat_id,
  ch08_sat_cnr, ch09_sat_id, ch09_sat_cnr, ch10_sat_id, ch10_sat_cnr,
  ch11_sat_id, ch11_sat_cnr, ch12_sat_id, ch12_sat_cnr, main_vol, bu_vol,
  temp, easting, northing, remarks)
FROM
  'C:\tracking_db\data\sensors_data\GSM01508.csv'
  WITH (FORMAT csv, HEADER, DELIMITER ';');
```

```
COPY main.gps_data(
  gps_sensors_code, line_no, utc_date, utc_time, lmt_date, lmt_time, ecef_x,
  ecef_y, ecef_z, latitude, longitude, height, dop, nav, validated, sats_used,
  ch01_sat_id, ch01_sat_cnr, ch02_sat_id, ch02_sat_cnr, ch03_sat_id,
  ch03_sat_cnr, ch04_sat_id, ch04_sat_cnr, ch05_sat_id, ch05_sat_cnr,
  ch06_sat_id, ch06_sat_cnr, ch07_sat_id, ch07_sat_cnr, ch08_sat_id,
  ch08_sat_cnr, ch09_sat_id, ch09_sat_cnr, ch10_sat_id, ch10_sat_cnr,
  ch11_sat_id, ch11_sat_cnr, ch12_sat_id, ch12_sat_cnr, main_vol, bu_vol,
  temp, easting, northing, remarks)
FROM
  'C:\tracking_db\data\sensors_data\GSM01511.csv'
  WITH (FORMAT csv, HEADER, DELIMITER ';');
```

```
COPY main.gps_data(
  gps_sensors_code, line_no, utc_date, utc_time, lmt_date, lmt_time, ecef_x,
  ecef_y, ecef_z, latitude, longitude, height, dop, nav, validated, sats_used,
  ch01_sat_id, ch01_sat_cnr, ch02_sat_id, ch02_sat_cnr, ch03_sat_id,
  ch03_sat_cnr, ch04_sat_id, ch04_sat_cnr, ch05_sat_id, ch05_sat_cnr,
  ch06_sat_id, ch06_sat_cnr, ch07_sat_id, ch07_sat_cnr, ch08_sat_id,
  ch08_sat_cnr, ch09_sat_id, ch09_sat_cnr, ch10_sat_id, ch10_sat_cnr,
  ch11_sat_id, ch11_sat_cnr, ch12_sat_id, ch12_sat_cnr, main_vol, bu_vol,
  temp, easting, northing, remarks)
FROM
  'C:\tracking_db\data\sensors_data\GSM01512.csv'
  WITH (FORMAT csv, HEADER, DELIMITER ';');
```

Special Topic: **Time and date data type in PostgreSQL**

The management of time and date is more complicated that it may seem. Often, time and date are recorded and stored by the sensors as two separate elements, but the information

that identifies the moment when the GPS position is registered is made of the combination of the two (a so-called 'timestamp'). Moreover, when you have a timestamp, it always refers to a specific time zone. The same moment has a different local time according to the place where you are located. If you do not specify the correct time zone, the database assumes that it is the same as the database setting (usually derived from the local computer setting). This can potentially generate ambiguities and errors. PostgreSQL offers a lot of tools to manage time and date[20]. We strongly suggest using the data type *timestamp with time zone* instead of the simpler but more prone to errors *timestamp without time zone*.

To determine the time zone set in your database you can run

```
SHOW time zone;
```

You can run the following SQL codes to explore how the database manages different specifications of the time and date types:

```
SELECT
  '2012-09-01'::DATE AS date1,
  '12:30:29'::TIME AS time1,
  ('2012-09-01' || ' ' || '12:30:29') AS timetext;
```

In the result below, the data type returned by PostgreSQL are respectively *date*, *time without time zone* and *text*.

```
  date1    |  time1   |      timetext
------------+----------+--------------------
 2012-09-01 | 12:30:29 | 2012-09-01 12:30:29
```

Here you have some examples of how to create a timestamp data type in PostgreSQL:

```
SELECT
  '2012-09-01'::DATE + '12:30:29'::TIME AS timestamp1,
  ('2012-09-01' || ' ' || '12:30:29')::TIMESTAMP WITHOUT TIME ZONE AS
  timestamp2,
  '2012-09-01 12:30:29+00'::TIMESTAMP WITH TIME ZONE AS timestamp3;
```

In this case, the data type of the first two fields returned is *timestamp without time zone*, while the third one is *timestamp with time zone* (the output can vary according to the default time zone of your database server):

```
     timestamp1      |      timestamp2      |        timestamp3
---------------------+---------------------+------------------------
 2012-09-01 12:30:29 | 2012-09-01 12:30:29 | 2012-09-01 12:30:29+00
```

You can see what happens when you specify the time zone and when you ask for the *timestamp with time zone* from a *timestamp without time zone* (the result will depend on the default time zone of your database server):

```
SELECT
  '2012-09-01 12:30:29 +0'::TIMESTAMP WITH TIME ZONE AS timestamp1,
  ('2012-09-01'::DATE + '12:30:29'::TIME) AT TIME ZONE 'utc' AS timestamp2,
  ('2012-09-01 12:30:29'::TIMESTAMP WITHOUT TIME ZONE)::TIMESTAMP WITH TIME
  ZONE AS timestamp3;
```

The result for a server located in Italy (time zone +02 in summer time) is

```
      timestamp1       |        timestamp2       |        timestamp3
-----------------------+-------------------------+------------------------
 2012-09-01 14:30:29+02 | 2012-09-01 14:30:29+02 | 2012-09-01 12:30:29+02
```

You can easily extract part of the timestamp, including *epoch* (number of seconds from 1st January 1970, a format that in some cases can be convenient as it expresses a timestamp as an integer):

```
SELECT
  EXTRACT (MONTH FROM '2012-09-01 12:30:29 +0'::TIMESTAMP WITH TIME ZONE) AS
  month1,
  EXTRACT (EPOCH FROM '2012-09-01 12:30:29 +0'::TIMESTAMP WITH TIME ZONE) AS
  epoch1;
```

The expected result is

```
month1 |    epoch1
-------+------------
     9 | 1346502629
```

In this last example, you set a specific time zone (EST—Eastern Standard Time, which has an offset of -5 h compared to UTC[21]) for the current session:

```
SET timezone TO 'EST';
SELECT now() AS time_in_EST_zone;
```

Here you do the same using UTC as reference zone:

```
SET timezone TO 'UTC';
SELECT now() AS time_in_UTC_zone;
```

---

[21] Coordinated Universal Time (UTC) is the primary time standard by which the world regulates clocks and time. For most purposes, UTC is synonymous with GMT, but GMT is no longer precisely defined by the scientific community.

You can compare the results of the two queries to see the difference. To permanently change the reference time zone to UTC, you have to edit the file postgresql.conf[22]. In most of the applications related to movement ecology, this is probably the best option as GPS uses this reference.

Finalise the Database: Defining GPS Acquisition Timestamps, Indexes and Permissions

In the original GPS data file, no timestamp field is present. Although the table *main.gps_data* is designed to store data as they come from the sensors, it is convenient to have an additional field where date and time are combined and where the correct time zone is set (in this case UTC). To do so, you first add a field of *timestamp with time zone* type. Then, you fill it (with an *UPDATE* statement) from the time and date fields. In a later exercise, you will see how to automatise this step using triggers.

```
ALTER TABLE main.gps_data
  ADD COLUMN acquisition_time timestamp with time zone;
UPDATE main.gps_data
  SET acquisition_time = (utc_date + utc_time) AT TIME ZONE 'UTC';
```

Now, the table is ready. Next you can add some indexes[23], which are data structures that improve the speed of data retrieval operations on a database table at the cost of slower writes and the use of more storage space. Database indexes work in a similar way to a book's table of contents: you have to add an extra page and update it whenever new content is added, but then searching for specific sections will be much faster. You have to decide on which fields you create indexes for by considering what kind of query will be performed most often in the database. Here, you add indexes on the *acquisition_time* and the *sensor_id* fields, which are probably two key attributes in the retrieval of data from this table:

```
CREATE INDEX acquisition_time_index
  ON main.gps_data
  USING btree (acquisition_time );
CREATE INDEX gps_sensors_code_index
  ON main.gps_data
  USING btree (gps_sensors_code);
```

---

[22] http://www.postgresql.org/docs/9.2/static/config-setting.html.

[23] http://www.postgresql.org/docs/9.2/static/sql-createindex.html.

As a simple example, you can now retrieve data using specific selection criteria (GPS positions in May from the sensor GSM01438). Let us retrieve data from the collar 'GSM01512' during the month of May (whatever the year), and order them by their acquisition time:

```
SELECT
  gps_data_id AS id, gps_sensors_code AS sensor_id, latitude, longitude,
  acquisition_time
FROM
  main.gps_data
WHERE
  gps_sensors_code = 'GSM01512' and EXTRACT(MONTH FROM acquisition_time) = 5
ORDER BY
  acquisition_time
LIMIT 10;
```

The first records (*LIMIT 10* returns just the first 10 records; you can remove this condition to have the full list of records) of the result of this query are

```
  id   | sensor_id | latitude | longitude |    acquisition_time
-------+-----------+----------+-----------+------------------------
 11906 | GSM01512  | 46.00563 |  11.05291 | 2006-05-01 00:01:01+00
 11907 | GSM01512  | 46.00630 |  11.05352 | 2006-05-01 04:02:54+00
 11908 | GSM01512  | 46.00652 |  11.05326 | 2006-05-01 08:01:03+00
 11909 | GSM01512  | 46.00437 |  11.05536 | 2006-05-01 12:02:40+00
 11910 | GSM01512  | 46.00720 |  11.05297 | 2006-05-01 16:01:23+00
 11911 | GSM01512  | 46.00709 |  11.05339 | 2006-05-01 20:00:53+00
 11912 | GSM01512  | 46.00723 |  11.05346 | 2006-05-02 00:00:54+00
 11913 | GSM01512  | 46.00649 |  11.05251 | 2006-05-02 04:01:54+00
 11914 | GSM01512  |          |           | 2006-05-02 08:03:06+00
 11915 | GSM01512  | 46.00687 |  11.05386 | 2006-05-02 12:01:24+00
```

One of the main advantages of an advanced database management system like PostgreSQL is that the database can be accessed by a number of users at the same time, keeping the data always in a single version with a proper management of concurrency. This ensures that the database maintains the ACID (atomicity, consistency, isolation, durability) principles in an efficient manner. Users can be different, with different permissions. Most commonly, you have a single administrator that can change the database, and a set of users that can just read the data. As an example, you create here a user[24] (login *basic_user*, password *basic_user*) and grant *read* permission for the *main.gps_data* table and all the objects that will be created in the *main* schema in the future:

---

[24] http://www.postgresql.org/docs/9.2/static/sql-createrole.html.

```
CREATE ROLE basic_user LOGIN
   PASSWORD 'basic_user'
   NOSUPERUSER INHERIT NOCREATEDB NOCREATEROLE NOREPLICATION;
GRANT SELECT ON ALL TABLES
   IN SCHEMA main
   TO basic_user;
ALTER DEFAULT PRIVILEGES
   IN SCHEMA main
   GRANT SELECT ON TABLES
   TO basic_user;
```

Permissions can also be associated with user groups, in which case new users can be added to each group and will inherit all the related permissions on database objects. Setting a permission policy in a complex multi-user environment requires an appropriate definition of data access at different levels and it is out of the scope of this guide. You can find more information on the official PostgreSQL documentation[25].

Export Data and Backup

There are different ways to export a table or the results of a query to an external file. One is to use the command *COPY* (*TO*)[26]. An example is

```
COPY (
  SELECT gps_data_id, gps_sensors_code, latitude, longitude, acquisition_time,
  insert_timestamp
  FROM main.gps_data)
TO
   'C:\tracking_db\test\export_test1.csv'
   WITH (FORMAT csv, HEADER, DELIMITER ';');
```

Another possibility is to use the pgAdmin interface: in the SQL console select 'Query'/'Execute to file'. Other database interfaces have similar tools.

A proper backup policy for a database is important to securing all your valuable data and the information that you have derived through data processing. In general, it is recommended to have frequent (scheduled) backups (e.g. once a day) for schemas that change often and less frequent backups (e.g. once a week) for schemas (if any) that occupy a larger disk size and do not change often. In case of

---

[25] http://www.postgresql.org/docs/9.2/static/user-manag.html.

[26] http://www.postgresql.org/docs/9.2/static/sql-copy.html, http://wiki.postgresql.org/wiki/COPY.

ad hoc updates of the database, you can run specific backups. PostgreSQL offers very good tools for database backup and recovery[27]. The two main tools to backup are as follows:

- pg_dump.exe: extracts a PostgreSQL database or part of the database into a script file or other archive file (pg_restore.exe is used to restore the database);
- pg_dumpall.exe: extracts a PostgreSQL database cluster (i.e. all the databases created inside the same installation of PostgreSQL) into a script file (e.g. including database setting, roles).

These are not SQL commands but executable commands that must run from a command-line interpreter (with Windows, the default command-line interpreter is the program cmd.exe, also called Command Prompt). pgAdmin also offers a graphic interface for backing up and restoring the database. Moreover, it is also important to keep a copy of the original raw data files, particularly those generated by sensors.

Reference

Urbano F, Cagnacci F, Calenge C, Dettki H, Cameron A, Neteler M (2010) Wildlife tracking data management: a new vision. Philos Trans R Soc B 365:2177–2185. doi:10.1098/rstb.2010. 0081

---

[27] http://www.postgresql.org/docs/9.2/static/backup.html.