Example: Loading Tracks and Displaying Them With a Map Background

Ralf Hartmut Güting, September 2011

SECONDO was built by the *SECONDO team*.

In this document we briefly show how one can load tracking data collected by a GPS device, create moving objects from them, and display them in front of a tiled map background obtained from OpenStreetMap or GoogleMaps.

We assume a Secondo installation of version 3.2 or higher to be present. We also assume the reader is (somewhat) familiar with the documents "Short Guide to Using Secondo" and "Short Guide to Querying at the Executable Level" available at the Secondo Downloads page.

In the secondo/bin directory is a file "Trk110731.csv". We open the file with a text editor (e.g. gedit) and find that it has the following structure:

```
#GPS Track File generated by KDR GPS Tracker Version 2.69
#Name:Trk110713_1
#track param.: t=015,v=010,c=010,d=200,NAOAO
#each data set holds:
#Latitude in Deg.Decimals, (neg.=south)
#Longitude in Deg.Decimals, (neg.=west)
#UTC(hh:mm:ss),Altitude(m),Distance(km),Speed(km/h),Date(yyyy-mm-
dd),Name,Sat
51.472242,7.444085,07:11:18,170,671.259,48,2011-07-31,07506,07
51.472772,7.447040,07:11:36,166,671.475,44,2011-07-31,07507,07
51.473322,7.447972,07:11:44,166,671.564,40,2011-07-31,07508,07
51.473870,7.448530,07:11:51,163,671.636,36,2011-07-31,07509,07
51.473870,7.448530,07:11:51,163,671.636,24,2011-07-31,07510,07
51.473933,7.448605,07:11:53,164,671.645,14,2011-07-31,07511,07
...
```

It contains the GPS tracks of the movement of a single person. We will convert this into a series of trips as values of type *mpoint*. After starting SECONDO (Monitor and Javagui) we first create a database and open it:

```
create database mvtrips

open database mvtrips
```

In the next step we load the lines of the file into a relation "Raw", using the *csvimport* operator:

```
# load raw data
```

```
let KDRSchema = [const rel(tuple([
  Latitude: real,
  Longitude: real,
  UTC: string,
  Altitude: int,
  Distance: real,
  Speed: int,
  Date: string,
  Name: string,
  Sat: int
]))
value ()]
```

This command creates an empty relation “KDRSchema” with a schema corresponding to the fields in the csv file.

```
let Raw = KDRSchema csvimport['../bin/Trk110731.csv', 7, "#", ","] consume
```

The *cvsimport* operator takes as a first argument a relation of which the schema (or type) is used. Second argument is the path to the csv file, third the number of lines at the start of the file that should be ignored, fourth is a symbol designating comment lines that should be ignored, fifth is the symbol separating fields.

After executing the command, with

```
query Raw count
```

we find that reading data from the file was successful and relation Raw has 7243 tuples.

The UTC time measurements in the file have to be adjusted by adding one hour to obtain central European time, so we define:

```
let onehour = [const duration value (0 3600000)]
```

We now create one trip for each day:

```
# create Trip for each day

let DayTrips =  Raw feed
  extend[
    I: str2instant(.Date + "-" + .UTC) + onehour,
    P: makepoint(.Longitude, .Latitude)]
  sortby[Date asc, I asc]
  groupby[Date
  ; DayTrip: group feed approximate[I, P, [const duration value (0
300000)]] ]
consume
```

Here a tuple stream is created from the Raw relation by *feed*. The *extend* operator adds to each tuple two new attributes *I* and *P* representing the instant of time and the location of the observation. The resulting tuples are sorted by date and time and then grouped by date. The tuples in each group are processed by the *approximate* operator which connects two successive observations by an assumed linear movement. The fourth, optional parameter (in this case a duration value of 300 seconds) specifies that two successive observations with a gap of more than this duration should not be connected any more. The result returned from the *approximate* operator is an *mpoint* value

which is stored in attribute *DayTrip*. Hence the resulting tuples have two attributes *Date* and *DayTrip*. They are collected into a relation by *consume* which is stored as *DayTrips*.

Note that, if a csv file with observations of many distinct moving objects is given where objects are distinguished by some identifier field, one can in a similar way group by identifiers and so create many moving objects moving simultaneously.

We further split the trips per day (each trip given as a single moving point) into several distinct trips:

```
# alternative representation: split by trips

let Trips = DayTrips feed
  projectextendstream[Date
  ; Trip: .DayTrip sim_trips[ [const duration value (0 300000)]] ]
  consume
```

Here for each tuple, the *sim_trips* operator is applied to the *DayTrip* attribute of type *mpoint*. Using the second parameter of type *duration* (again 5 minutes = 300000 ms), the moving point is split into pieces at time gaps of more than the given duration. The operator returns a stream of *mpoint* values, one for each such piece. The *projectextendstream* operator creates one tuple for each element in this stream, copying the mentioned attributes from the input tuple (*Date* in this case) and putting the given stream element into a new attribute (*Trip* in this case). The resulting stream of tuples is stored in a *Trips* relation.

We can now display the trips with a map background. First, use the menu *File->Load categories* to load *BerlinU.cat,* a set of display styles. Then, use *Settings->Projections* to select the *OSM-Mercator* projection. Finally, use *Settings->Background->TiledMap* and select a map style (default is OpenStreetMap Mapnik style) and press *accept*. A map of the world appears (provided your computer has internet access).

```
query Trips
```

The *Trips* relation is loaded. The map section shown in the graphics window is reduced. If you select one of the *Trip* attributes in the table on the left, a blue point appears showing the position of the starting point of this trip. Dragging a rectangle with the right mouse button, one can zoom into the map to see the precise location of this point. Using the animation buttons (top left of the viewer window), one can see the animated movement. At this point, one should have a GUI window like the one shown in Figure 1.

We can also show the trajectories of the trips, i.e., the paths taken by the traveller.

```
query Trips feed projectextend[Date; Traj: trajectory(.Trip)] consume
```

In the upcoming dialog, select *QueryLine* as a *View Category* (display style) to see the paths taken as red lines. Finally, let us add the starting points and ending point of trips as interesting “stop” positions.

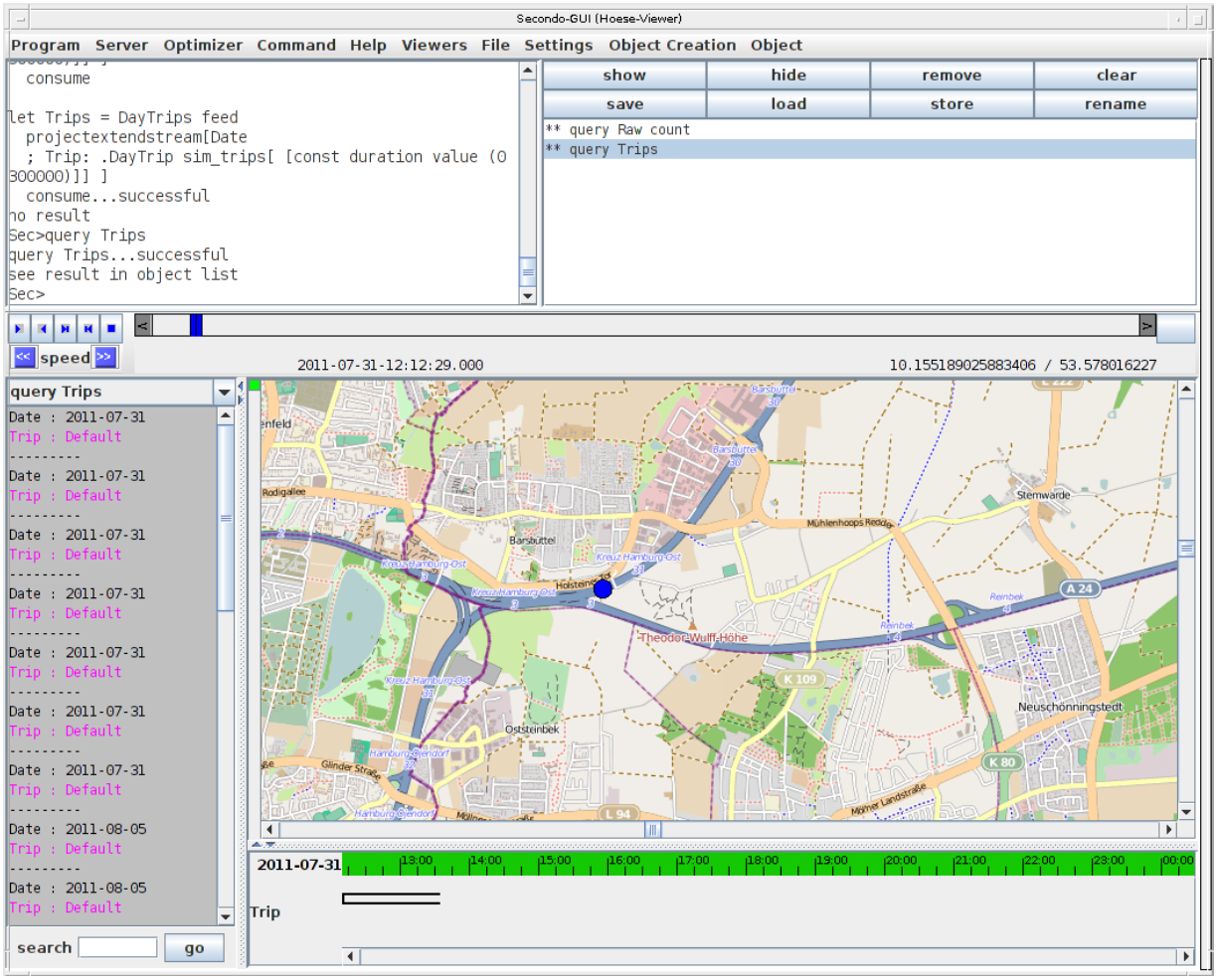

Figure 1: The GUI after loading trips, selecting one, and zooming in.

```
# finding stops
# starting and ending points for each trip

let Stops =
  Trips feed projectextend[Date
  ; Time: inst(initial(.Trip)),
    Stop: val(initial(.Trip)),
    Kind: "start"]
  Trips feed projectextend[Date
  ; Time: inst(final(.Trip)),
    Stop: val(final(.Trip)),
    Kind: "end"]
  concat
  sortby[Date, Time]
  consume
```

We display the stops:

```
query Stops
```

After zooming into the island of Usedom and then into the village of Heringsdorf, we may see GUI windows like those shown in Figures 2 and 3.

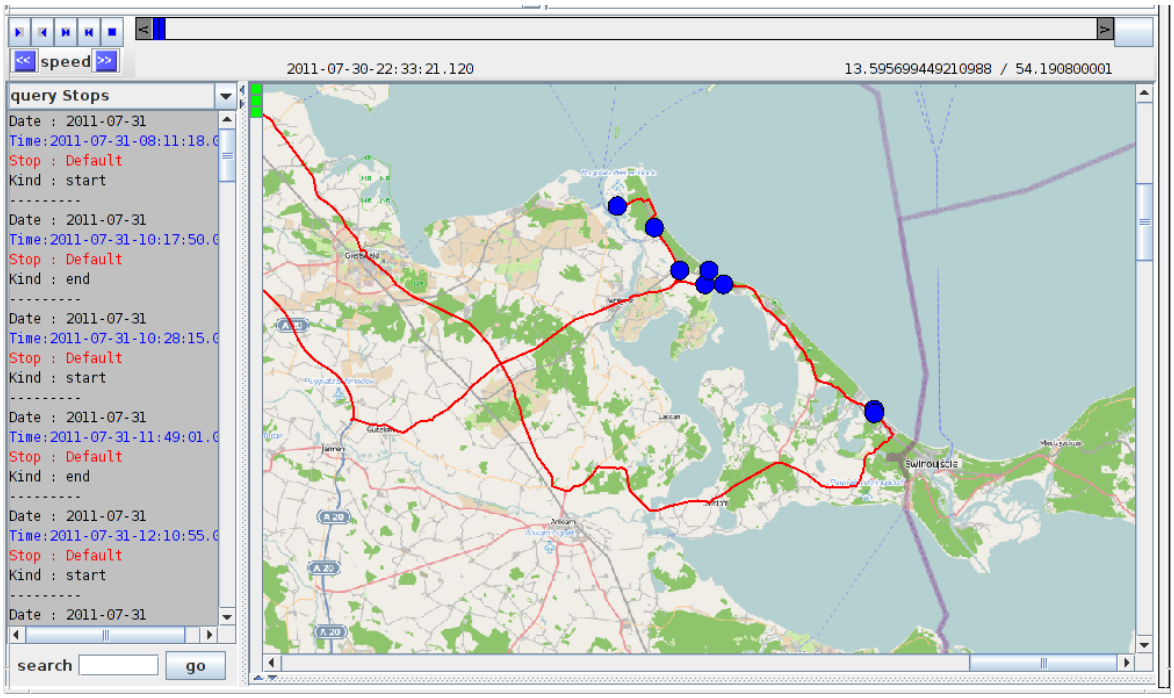

Figure 2: Trajectories and stops at Usedom

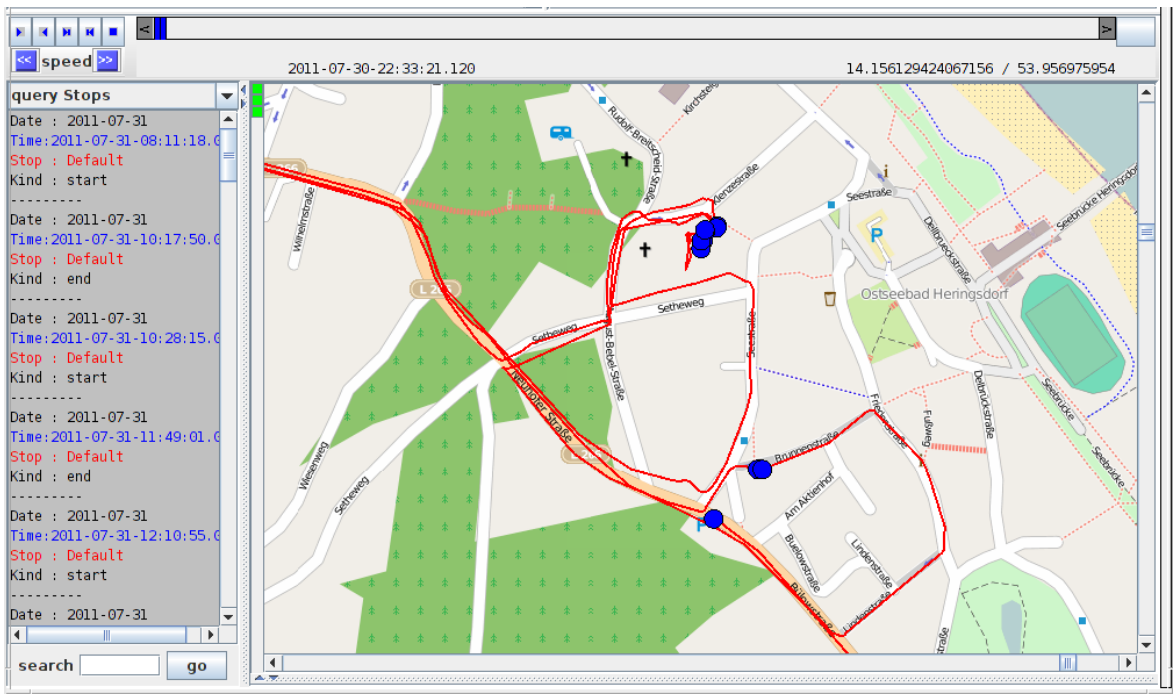

Figure 3: Trajectories and stops at Heringsdorf

The commands shown in this document are also available in a script file “createmvtrips.sec” in the secondo/bin directory. Running the script creates and closes the database *mvtrips*, e.g. using the command (at the SECONDO prompt):

```
@createmvtrips.sec
```