The Berlintest Demo

Mahmoud A.Sakr [#1,*2], Ralf H.Güting [#1]

[#1]*Database Systems for New Applications, FernUniversität in Hagen*
58084 Hagen, Germany
[*2]*Faculty of Computer and Information Sciences, University of Ain Shams*
Cairo, Egypt
[1]`mahmoud.sakr@fernuni-hagen.de`
[2]`rhg@fernuni-hagen.de`

January 13, 2010

1 Introduction

This document is a demonstration for our new approach for spatiotemporal pattern (STP) queries [1] [2]. It is intended to guide the reader to execute the *Berlintest* application example within the SECONDO platform. Similar text is found in [1], yet here more details are given on how to execute the example queries and what are the expected results.

The *Berlintest* example consists of five STP query examples. The queries are based on the *berlintest* database that is available within the SECONDO distribution. The queries are not linked to a single scenario. They are intended to demonstrate STP queries that involve moving points, moving regions, and several kinds of lifted operations.

2 Preparing for the Demo

The implementation of our approach for STP queries is made available as a Plugin for the SECONDO system [3]. It can be downloaded from the Plugin website [4]. The *User Manual* (also available on the Plugin website) describes how to install and run the Plugin. Within the Plugin, we have also made available a script for creating the relations needed for the *Berlintest* example.

First you need to install:

1. The SECONDO system version 2.9.1. A brief installation guide is given in the *Plugin User Manual* on [4], and a detailed guide is given in the SECONDO *User Manual* [5].
2. The Spatiotemporal Pattern Queries Plugin as described in [4].

To execute the queries in the *Berlintest* example, you need first to run the script *BerlintestScript.sec* from the SecondoTTYNT prompt in order to create the necessary database objects. The script is copied to the *$SECONDO_BUILD_DIR/bin* directory during the installation of the Plugin.

1. Start SecondoTTYNT. In a shell, go to $SECONDO_BUILD_DIR/bin and type
```
SecondoTTYNT
```
2. Make sure that the *berlintest* database is restored. Type
```
list databases
```
If *berlintest* is not in the displayed database list, you'll need to restore it. Type
```
restore database berlintest from berlintest
```

3. Execute the *BerlintestScript.sec* script. Type
   `@BerlintestScript.sec`

After running the *BerlintestScript.sec* script, use the *Javagui* to execute the queries. It is the graphical user interface for SECONDO. To launch it:

1. Start the SECONDO kernel in server mode, the optimizer server, and the GUI:
   In a new shell, go to $SECONDO_BUILD_DIR/bin, and type
   `SecondoMonitor -s`
   In a new shell, go to $SECONDO_BUILD_DIR/Optimizer, and type
   `StartOptServer`
   In a new shell, go to $SECONDO_BUILD_DIR/Javagui, and type
   `sgui`
   The Javagui will start and connect to both the kernel and the optimization server.

2. Open the database. In the Javagui type:
   `open database berlintest`

3. Set the optimizer options. The SECONDO optimizer maintains a list of options that controls the optimization. The examples in this paper require the options *improvedcosts*, *determinePredSig*, *autoSamples*, *rewriteInference*, *rtreeIndexRules*, and *autosave*. To set each of these options, type in the Javagui:
   `optimizer setOption(`*`option`*`)`

4. View the underlying network. Type:
   `select * from ubahn` to display the underground trains network.
   `select * from trains` to display the moving trains. Use the slider to view the results.
   Select the last query in the top-right panel and press hide to hide the trains.
   `select * from snowstorms` to display the moving snow storms.
   hide the snow storms.

5. Type the example queries as in Section 3, and make sure to type everything in lower case.

3 The Berlintest Example

In this example, we use the database *berlintest*, more specifically, the *Trains* relation and three newly added relations. The schemas are as follows:

Trains[Id :$\underline{int}$, Line :$\underline{int}$, Up: $\underline{bool}$, Trip: $\underline{mpoint}$]
SnowStorms[Serial: $\underline{int}$, Storm: $\underline{mregion}$]
TrainsMeet[Line: $\underline{int}$, Uptrip: $\underline{mpoint}$, Downtrip: $\underline{mpoint}$, Stations: $\underline{points}$]
TrainsDelay[Id: $\underline{int}$, Line: $\underline{int}$, Actual: $\underline{mpoint}$, Schedule: $\underline{mpoint}$]
msnow:$\underline{mregion}$

The *SnowStorms* relation contains 72 tuples, each of which contains a moving region, representing a snow storm that moves over Berlin. The *TrainsMeet* relation is generated from the *Trains* relation. The tuples contain all possible combinations of two trains that belong to the same line and move in opposite directions. The *Stations* attribute represents the train stations of the associated line. The *TrainsDelay* relation is also generated from the *Trains* relation. Each tuple contains the original *Trip* attribute (renamed into *Schedule*), and a delayed copy of it with delays of around 30 minutes. The scripts for creating the three relations and for executing the example queries are available for download as will be explained in Section 4.

Table 1 lists the lifted operations used within the queries. We have designed the queries so that they illustrate the expressive power of our approach by using various lifted operations to compose complex pattern queries. The table shows only the operator signatures that are used in the queries. The complete list of valid signatures is in [6]. For the following queries, the used temporal connectors (e.g. *together*, *meanwhile*, etc.) have already been defined in the database by the *BerlintestScript.sec* script.

Table 1: Lifted Operations

Operation	**Signature**	**Type**	**Meaning**
at	$\underline{mregion} \times \underline{point} \rightarrow \underline{mpoint}$	topological operation	computes a moving point that exists whenever the point argument is inside the moving region argument.
isempty	$\underline{mpoint} \rightarrow \underline{mbool}$	set operation	true whenever the argument is defined.
not	$\underline{mbool} \rightarrow \underline{mbool}$	boolean operation	logical negation.
rough_center	$\underline{mregion} \rightarrow \underline{mpoint}$	aggregation	aggregates the moving region into a moving point that represents its center of gravity.
speed	$\underline{mpoint} \rightarrow \underline{mreal}$	metric property	the metric speed of the moving point.
distancetraversed	$\underline{mpoint} \rightarrow \underline{mreal}$	metric property	the distance that the moving point traversed since the start of its definition time.
area	$\underline{mregion} \rightarrow \underline{mreal}$	metric property	the area of the moving region.
intersection	$\underline{mpoint} \times \underline{mpoint} \rightarrow \underline{mpoint}$	set operation	computes the common parts of the two arguments.
inside	$\underline{mpoint} \times \underline{mregion} \rightarrow \underline{mbool}$ $\underline{mpoint} \times \underline{points} \rightarrow \underline{mbool}$	spatial range predicate	true whenever the $\underline{mpoint}$ is contained in the $\underline{mregion}$, or passes some of the $\underline{points}$.
delay	$\underline{mpoint} \times \underline{mpoint} \rightarrow \underline{mreal}$	metric operation	considers the first argument *actual*, and the second *schedule movement* and computes the delay of the actual movement in seconds.
=	$\underline{mpoint} \times \underline{point} \rightarrow \underline{mbool}$	spatial range predicate	true whenever the moving point passes the point.
xangle	$\underline{mpoint} \rightarrow \underline{mreal}$	direction	the angle (in degrees) between x-axis and the tangent of the moving point.
and	$\underline{mbool} \times \underline{mbool} \rightarrow \underline{mbool}$	boolean operation	logical and.
<, <=, >, >=	$\underline{mreal} \times \underline{real} \rightarrow \underline{mbool}$	left/right range predicate	true in the time intervals during which the comparison holds.

3.1 Find the snow storms that passed over the train station *mehringdamm* with speed greater than 40 km/h.

```
SELECT *
FROM   snowstorms
WHERE  pattern([not(isempty(storm at mehringdamm)) as pred1,
         speed(rough_center(storm)) > 40.0 as pred2],
       [stconstraint("pred1","pred2", together)])
```

where *together* is a vector temporal connector that yields true if the two predicates happen simultaneously.
Result: the query yields one tuple with *Serial = 2*.

3.2 Find the snow storms that could increase their area over 1/4 square km during the first traversed 5 km.

```
SELECT *
FROM   snowstorms
WHERE  pattern(
       [distancetraversed(rough_center(storm)) <= 5000.0 as pred1,
         area(storm) > 250000.0 as pred2],
       [stconstraint("pred1","pred2", meanwhile)])
```

Result: the query yields 16 tuples.

3.3 Find the trains whose up and down trips meet inside one of the train stations.

```
SELECT   *
FROM     trainsmeet
WHERE    pattern(
         [not(isempty(intersection(uptrip, downtrip))) as pred1,
           uptrip inside stations as pred2 ],
         [stconstraint("pred1","pred2", together)])
ORDERBY  line
```

Result: the query yields 57 tuples, all having *Line = 5*.

3.4 Find the trains that encountered a delay of more than 30 minutes after passing through the snow storm *msnow*.

```
SELECT *
FROM   trainsdelay
WHERE  pattern([not(delay(actual, schedule) > 1800.0) as pred1,
         actual inside msnow as pred2,
         delay(actual, schedule) > 1800.0 as pred3 ],
       [stconstraint("pred1", "pred2", vec("abab", "aba.b", "abba")),
       stconstraint("pred2", "pred3",
          vec("abab", "aba.b", "abba", "aa.bb", "aabb"))])
```

Result: the number of tuples that this query yields is not fixed. This is because the delay between the *Actual* and the *Schedule* attributes of the *TrainsDelay* relation is randomly assigned during its creation. In one of our experiments, for example, the query yields 54 tuples.

3.5 Find the trains that are always heading north-west after passing *mehringdamm*.

```
SELECT *
FROM   trains
WHERE  patternex([trip = mehringdamm as pred1,
         ndefunit(((xangle(trip) >= 90.0) and
           (xangle(trip) <=180.0)), int2bool(1)) as pred2],
       [stconstraint("pred1","pred2",then)],
       (((start("pred2")- end("pred1")) < create_duration(0, 120000))
       and
       ((inst(final(trip)) - end("pred2")) < create_duration(0, 15000))))
```

where the *ndefunit* operator replaces the undefined periods within an <u>*mbool*</u> by defined units having the constant <u>*bool*</u> value that is provided in the last argument. In this query, we replace the undefined periods by *true* units, as indicated by the *int2bool(1)* operator, which yields the <u>*bool*</u> value *true*. This is because

the *xangle* [1] operator yields undefined during the train stops in the stations. In other words, *pred2* is true whenever the train is not heading other than north-west. The query restricts the results to the trains which started heading north at most 2 minutes after passing *mehringdamm* and remained so till at least 15 seconds before the end of the trip. These time margins are used to cut out small noisy parts in the data, so that the query yields results.
Result: the query yields 21 tuples, all having *Line = 6*.

4 The *BerlintestScript.sec* script

To execute the queries in the berlintest example, you need first to run the script *BerlintestScript.sec* from the SecondoTTYNT prompt. The script is installed within the STPattern Plugin. You also need to have the berlintest database restored in your system. The script file creates the required database objects. Following we explain the contents of the script. It first defines some temporal connectors to be used in the queries:

```
close database;
open database berlintest;
let later= vec("aabb", "a.abb", "aab.b", "a.ab.b");
let follows= vec(...
let immediately= vec(...
let meanwhile= vec(...
let then= vec(...
let together= vec(...
```

Then it restores the *SnowStorms* relation from the *SnowStorms* file in the SECONDO/bin directory, which is installed with the Plugin.

```
restore SnowStorms from SnowStorms;
```

The following command creates the relation *TrainsMeet*, that is used in the example in Section 3.3. Every tuple in the relation is a different combination of an up train, down train of the same line, and the stations where the train line stops.

```
let TrainsMeet =
  Trains feedproject[Line, Trip, Up] {t2}  filter[.Up_t2 = FALSE]
  Trains feedproject[Line, Trip, Up] {t1}  filter[.Up_t1 = TRUE]
  hashjoin[Line_t2 , Line_t1 , 99997]
  extend[Line: .Line_t1, Uptrip: .Trip_t1, Downtrip: .Trip_t2,
    Stations: ((breakpoints(.Trip_t1, create_duration(0,5000) )
      union val(initial(.Trip_t1)))
      union val(final(.Trip_t1)))]
  project[Line, Uptrip, Downtrip, Stations]
  consume;
```

where the *breakpoints* operator accepts an <u>*mpoint*</u> and a <u>*duration*</u>, and yields the <u>*points*</u> where the <u>*mpoint*</u> argument stops for at least the period indicated by the <u>*duration*</u> argument. The *create_duration* operator is used in this query to create a <u>*duration*</u> of zero days and 5 seconds. In reality, a 5 seconds stop is not a reasonable duration for indicating a train station. In our experiments, however, we've found that it reports the correct train stations for the *Trains* relation.

Next we create the relation *TrainsDelay*, used in the example in Section 3.4. Every tuple has a *schedule* and an *actual* moving point. The *schedule* movement is a copy from the *Trip* attribute in the

[1]The *xangle* operator is a corrected copy of the SECONDO *mdirection* operator. It is presented only for the sake of this example. In the SECONDO versions newer than 2.9.1, the *mdirection* operator works fine.

Trains relation. The actual movement should have delays of about half an hour. We shift the *Trip* 1795 seconds forward, and apply a random positive or negative delay up to 10 seconds to the result. This creates actual movements with random delays between 29:45 and 30:05 minutes.

```
let TrainsDelay=
  Trains feed
  extend[Schedule: .Trip,
    Actual: randomdelay(
      .Trip translate[create_duration(0, 1795000), 0.0, 0.0],
      create_duration(0, 10000) ) ]
  project[Id, Line, Actual, Schedule]
  consume;
```

References

[1] M. Sakr and R. H. Güting, "Spatiotemporal pattern queries," FernUniversität Hagen, Tech. Rep. Informatik-Report 355, November 2009.

[2] M. Sakr and R. H. Güting, "A new approach for spatiotemporal pattern queries in trajectory databases," 2009, submitted to MDM 2010.

[3] SECONDO web site. [Online]. Available: http://dna.fernuni-hagen.de/Secondo.html/

[4] SECONDO plugins. [Online]. Available: http://dna.fernuni-hagen.de/secondo.html/start_content_plugins.html

[5] R. H. Güting, D. Ansorge, C. Düntgen, S. Jandt, T. Behr, and M. Spiekermann. (2009, September) SECONDO user manual. [Online]. Available: http://dna.fernuni-hagen.de/Secondo.html/files/SecondoManual.pdf

[6] R. H. Güting, M. H. Böhlen, M. Erwig, C. S. Jensen, N. A. Lorentzos, M. Schneider, and M. Vazirgiannis, "A foundation for representing and querying moving objects," *ACM Trans. Database Syst.*, 2000.