INFORMATIK
BERICHTE

340 - 12/2007

BerlinMOD: A Benchmark for Moving Object Databases

Christian Düntgen, Thomas Behr, Ralf Hartmut Güting

Fakultät für Mathematik und Informatik
Postfach 940
D-58084 Hagen

BerlinMOD: A Benchmark for Moving Object Databases

Christian Düntgen, Thomas Behr, and Ralf Hartmut Güting
Faculty of Mathematics and Computer Science
University of Hagen, D-58084 Hagen, Germany
{christian.duentgen, thomas.behr, rhg}@fernuni-hagen.de

December 4, 2007

Abstract

This document presents a method to design scalable and representative moving object data (MOD) and a set of queries for benchmarking spatio-temporal DBMS. Instead of programming a dedicated generator software, we use the existing Secondo DBMS to create benchmark data. The benchmark is based on a simulation scenario, where the positions of a sample of vehicles are observed for an arbitrary period of time within the street network of Berlin. We demonstrate the data generator's extensibility by showing how to achieve more natural movement generation patterns, and how to disturb the vehicles' positions to create noisy data. As an application and for reference, we also present first benchmarking results for the Secondo DBMS.

Such a benchmark is useful in several ways: It provides well-defined data sets and queries for experimental evaluations; it simplifies experimental repeatability; it emphasizes the development of complete systems; it points out weaknesses in existing systems motivating further research. Moreover, the BerlinMOD benchmark allows one to compare different representations of the same moving objects.

1 Introduction

Current database systems are able to store large sets of data. Besides standard data, also special kinds of data, e.g. multimedia, spatial, and spatio-temporal data can be stored. Whereas the handling and the access of standard data is well known, storing and efficient processing of non-standard data is a current challenge. To be able to compare different DBMS and their storage and access methods, benchmarks can be used.

In general, a benchmark consists of a well defined (scalable) data set and a set of problems. In the context of database systems these are mostly formulated as a set of SQL-queries. Benchmarks have been proven to be a proper tool to check the performance of DBMS. Though data structures, index structures and different operator implementations can be compared separately from other components, their impact on database performance becomes clearer when they are tested all together within in a real system. Also, benchmark evaluations make results from different researchers comparable in a straightforward way.

Benchmarks simplify the setup and description of experiments, because one can easily refer to a well-defined data set whose properties are described elsewhere in detail. Using predefined data sets and queries also reduces the risk of introducing bias in experiments.

In this paper, we present a benchmark for the field of moving objects databases. Such databases come in two flavors: (i) representing current movements, e.g. of a fleet of trucks, in real time, supporting questions about current and expected near future positions, and (ii) representing complete histories of movements, allowing for complex analyses of movements in the past. Our benchmark addresses databases of the second kind, sometimes called trajectory databases.

There has been a lot of research on moving object databases in the last ten years. Much of this research has focused on providing specialized index structures or efficient algorithms to support specific types of queries. However, the maturity of a field shows itself in the fact that complete systems are around that can handle a significant range of interesting queries. Up to now, very few such systems exist. A benchmark defining such a set of queries may help the community to focus more on integration, i.e. building complete systems.

A benchmark covering an interesting range of queries may show not only the strengths, but also the weaknesses of existing solutions, indicating where more work is needed. It may also show that gains in efficiency in specialized components are relatively irrelevant because the bottlenecks in a system context arise elsewhere.

Because database systems are used in practice, the best data set would be real data. But providing real data for moving objects will lead to some problems. The first one is the effort for capturing the data; e.g. to observe 1000 or more vehicles at the same time, each vehicle must be equipped with a GPS receiver and the data must be collected. Another problem is that real data are not scalable, e.g. it is impossible to extend the sample in retrospect. Data generators are a good method to cope with these problems. Their output can be varied in size just by changing some parameters. Some data generators create data "observing" objects within a simulated scenario; if the scenario is realistic, this approach promises to yield representative data.

Our proposed benchmark *BerlinMOD* addresses the handling of moving point data. Besides a set of queries, we also provide a tool generating the moving point data. To generate the data set, we simulate a number of cars driving on the road network of Berlin for a given period of time (e.g. a month) and capture their positions at least every two seconds.

Instead of implementing a stand-alone dedicated generator tool, we use the infrastructure of Secondo, a prototypical extensible DBMS. Benchmark data are created by sequences of Secondo commands collected in a Secondo *script*. To our knowledge, this is the first time, a DBMS is used in this way.

As far as we know, we also present the first data generator creating scalable representative network-based moving point data simulating long-term observation of objects. Long-term observation yields huge histories for moving objects, which is interesting for benchmarking indexes and spatio-temporal operators.

To model the impreciseness of real-world position tracking systems, we can optionally disturb resulting moving point data.

The benchmark is evaluated within the Secondo system. Both the evaluation of the queries and the script for creating the benchmark data demonstrate the capabilities of a state-of-the-art moving object database system.

The Secondo system as well as all tools needed to create the benchmark data and run the benchmark in Secondo are freely available on the Web [19, 1]. Hence anyone can repeat the experiments reported here. Of course, benchmark data can also be saved as text files and then be converted to the input formats of other systems.

2 Related Work

Moving objects are time dependent geometries, i.e. geometries described as a function of time. In contrast to earlier work on spatio-temporal databases, geometries may change continuously.

Two views on moving object data have been established in the past years. The first one focuses on answering questions on the current positions of moving objects, and on their predicted temporal evolution in the (near) future. This approach is sometimes called *tracking*. To model moving object data in a way suitable to these classes of queries, the Moving Objects Spatio-Temporal (MOST) model and the Future Temporal Logic (FTL) language have been proposed [20, 25, 27, 26].

A second approach represents complete histories of moving objects, for moving point objects also called *trajectory databases*. This approach was pursued, for example, in [8, 12, 9]. In this work the complete evolution of a moving object can be represented as a single attribute within an object-relational or other data model.

In this article we focus on the second, the history-based approach.

As access methods are essential to efficient query processing, index structures for both flavors of moving object databases have been proposed – for an overview see [15]. To compare and evaluate indexes on trajectories, our benchmark can be used.

Whereas free movement is the general case for moving object data, also constrained moving objects have been investigated [23, 7, 14], especially with regard to real world applications. As an example, in [3] the authors describe transportation networks as an abstraction of constrained movement and present

a specialized index structure for them. We will propose to generate data for moving objects constrained by a transportation network.

Note that the data model implemented in the SECONDO system that is used below for executing the benchmark is the one from [12] based on free movement in the 2D plane. Because the benchmark data set provides network constrained data, it is suitable to be also represented in an implementation of the model of [14]. This implementation is underway in SECONDO.

For spatial DBMS (SDBMS), the most prominent benchmark is the Sequoia 2000 storage benchmark [21]. It comprises real spatial data and a set of queries for testing a SDBMS's performance. Both are claimed to be representative for geoscience tasks. The test database consists of various scales, granularities, and sizes of real geographic data. Covered types include raster, point, polygon, and directed graph data. The system is rated by the total response time generating the test data and processing queries involving spatial joins, recursive searches, point and range queries. Although some of the queries include selection and sorting by timestamps, Sequoia does not handle "real" spatio-temporal, i.e. moving object data.

In an attempt to generalize the Sequoia Benchmark, Werstein [24] introduces time as a third dimension. He proposes 36 queries based upon the original Sequoia queries, e.g. using "matrix data" as a 3D analogue of the original Sequoia raster data. Effectively, he saves data snapshots with time stamps and uses them in spatial, temporal and spatio-temporal point and range queries. He also considers temporal updates, extending the data histories (Queries 28-32), and performs "walks through time" (Queries 33-36). Nonetheless, several important aspects of spatio-temporal data processing are missing, such as spatio-temporal relations (predicates), computations based on spatio-temporal data, and computations creating spatio-temporal data. These features remain out of the benchmark's scope. For example, Query 5 calculates arithmetic functions only on the single "cells" of the matrix data — one by one. As a summary, the proposed benchmark is suitable for a temporal SDBMS, but not for a "real" spatio-temporal database system in the narrower sense.

One of the first generators for moving object data, GSTD [22], creates unconstrained moving point or rectangle data. A refinement [16] allows for more realistic trajectories by covering more agile, clustered, and obstructed movement. The refined algorithm has been used within other data generators, e.g. to create cellular network positioning data [10]. Since in the real world objects often follow a predefined network (cars, trains, etc.) and access methods are strongly influenced by this fact, such data are inappropriate to measure the performance of systems dealing with moving objects in networks.

Another generator for spatio-temporal data, called Oporto [18], simulates a fishing scenario to create scalable and representative moving object data. Shoals of fish follow fluctuating spots of plancton. Fishing boats travel to and from harbours to find fish swarms and try to evade changing storm areas. Oporto is capable of creating unrestricted moving point data and moving region data (with fixed center but moving shape and size, and fully moving ones, with changing location, shape and size). Observation of ships and shoals is possible for long periods of time. Since the movement of objects is (almost) unrestricted, we cannot get appropriate data for moving objects in networks.

A generator and visual interface for objects moving in networks is proposed by Brinkhoff in [2]. During the generation process, new objects appear and disappear when their destination is reached. Speed and route of a moving object depend on the load of network edges (current number of cars using it) and so-called external objects. Networks can be created from shape files, tiger line files and other sources. The raw behaviour of the generator is controlled by a set of parameters. More sophisticated changes require changes within the source code. Because an object only exists while it moves from its start node to its destination, long time observations of objects are not realizable.

In contrast, our approach allows both, the trip based representation (like Brinkhoff's generator) and the object based representation where an object is observed during a given time interval. Our approach does not consider the edge load within the street graph. Instead, we insert stops and decelerations depending on the shape of the street section represented by the edge. Additionally, we insert also stops at transitions between road sections.

The region based approach for the selection of the start and the end of a trip described in [2] is not implemented in Brinkhoff's generator. The generator described here supports this kind of selection. Furthermore, we are able to simulate measuring errors of the position detecting device.

In the remainder of the paper we first describe the scenario simulated during data generation in Section 3. Then we present in Section 4 two different relation schemas for the generated data. After identifying groups of queries we propose a set of interesting queries for benchmarking (Section 5). After this, we

explain the SECONDO-script implementing the data generator (Section 6) and create the benchmark to evaluate the SECONDO DBMS in Section 7. Finally, we conclude the paper in Section 8.

3 The BerlinMOD Scenario

BerlinMOD uses moving object (vehicle) positions, which are generated using the following design:

We assume that a graph representation of the street network exists. *Nodes* represent street crossings and dead ends, while *edges* represent parts of streets between these nodes. For each edge, a simple (non-branching and connected) *line* object describes its 2D-geometry. Edges are labeled with a cost value being the time needed to travel it at maximum allowed speed (a *real* value), and with a street identifier. Nodes are labeled with a node identifier and its spatial 2D-position (a *point* value).

3.1 Home Node, Work Node, and Neighbourhood

We wish to model a person's trips to and from work during the week as well as some additional trips at evenings or weekends.

Hence each object (vehicle) has a *HomeNode*, representing its holder's residence. It is chosen randomly from the set of nodes, using uniform distribution. This selection mechanism corresponds to the "network-based approach" defined in [2]. By small changes it is also possible to partition the set of all nodes (for example by predefined regions) and to use a different probability for the selection of each part. This would implement the "region based approach" from [2].

Also, each object has a *WorkNode* representing the owner's working place. Again, this node is chosen randomly among all nodes of Berlin or using the region based approach.

Additionally, a set of nodes called *Neighbourhood* is defined by all nodes within a 3 km line of sight distance around the *HomeNode*. Nodes in this area will be used more frequently for the additional trips than all other nodes.

3.2 Temporal Layout of Trips

The goal of the following design is to model a person's behaviour in a natural way. At the same time it should be possible to create trips independently, with only a minimal risk of temporal overlapping.

We assume that a person works Monday to Friday and commutes between her home node and her work node: On each such working day, she leaves her *HomeNode* in the morning (at 8 am + T_1), drives to her *WorkNode*, stays there until 4 pm + T_2 in the afternoon, and then returns to her *HomeNode* (T_1, T_2 are bounded Gaussian distributed durations[1] within an interval of -2 to 2 hours). We call these the *labour trips*. Because a trip on the street network of Berlin will take less than two hours (we have never observed a trip longer than 1:30 h; also see Figure 1 for a typical distribution of trip durations), we can be sure that the person has arrived at home before 8 pm.

At home, the person stays until 8 pm, then a 4h block of spare time begins.

On the weekend, the person will just have two 5h blocks of spare time per day, the first starting at 9 am and the second one at 7 pm.

For each block of spare time, there is a probability of 0.4, that the person will do an *additional trip* which may have 1 to 3 intermediate stops and ends at home.

If the last trip for a day ends at 6:00 am or later on the following day, there is a risk of temporally overlapping trips. Therefore, we invalidate all trips of that day. The probability for this to happen is below 0.000138 per vehicle and day. There are various natural explanations for such "days off", as illness or vacations.

3.3 Trip Creation Algorithm

For the generation of a trip we have the following assumptions:

[1] Note that, although data are generated in a probabilistic way using various distributions, the underlying pseudorandom number generators use seed values and therefore produce deterministic results. Hence the data set provided by the benchmark will always be the same, except possibly for very small numeric differences on different platforms and operating systems.

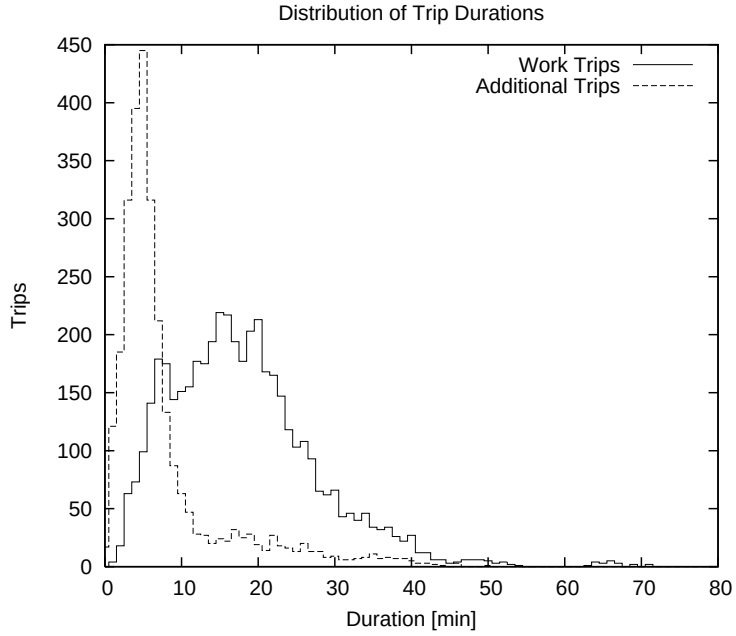

Figure 1: Typical Distribution of Single Trip Durations

A *trip* is parameterized by a triple (*Start, Destination, Time*), where *Start* and *Destination* are nodes, and *Time* is the instant, when the trip starts. The trip is created following the shortest path from *Start* to *Destination* through the street graph. Sample points of the movement are created tracing the object's movement along the segments of the line geometry corresponding to the path.

All streets are classified into one category of {freeway (70), main road (50), side road (30), closed road}. Closed roads are removed from the graph. For all other roads, the maximum allowed velocity v_{max} is given in km/h. Drivers always respect speed limits.

Drivers will always try to travel at the maximum allowed velocity. They will slow down only (or even stop), if they are required to do so, e.g. by red traffic lights, narrow curves, playing children, respecting other drivers' way of right, etc.

We use three kinds of "event" to characterize this behaviour:

Acceleration event — When the vehicle's current velocity v_c is below the allowed maximum speed v_{max}, it will automatically accelerate at a constant rate of 12 m/s^2.

Deceleration event — The vehicle's current velocity v_c is reduced to $v_c \cdot X/20$, where v_c is the current speed and $X \sim B(20, 0.5)$ a binomially distributed random variable.[2] Hence the expected new speed is half the current one.

Stop event — When a car must stop (e.g. at traffic lights), it will stay immobile ($v_c := 0.0$) for a duration of $t_w \sim \text{Exp}(15/86400)$ milliseconds.[3] The choosen mean results in an expected waiting time of 15 seconds.

Acceleration events occur automatically; events of other kinds may occur with a certain probability p_{event} every 5 travelled metres, where p_{event} depends on the current maximum allowed speed, $p_{event} = \frac{1}{v_{max}}$ (v_{max} in units of km/h). *Curves* are defined by sequential line segments on the vehicle's trajectory, that enclose an angle $\phi < 180°$. They will reduce the effective v_{max} depending on their enclosed angle, $v_{max} := v_{max} \cdot \frac{\phi}{180.0°}$. *Crossings* are all points, where at least two streets meet. When passing a crossing, a stop event is created with a probalibity depending on the type of transition in road types (Table 1).

[2] $B(n, p)$ means the Binomial distribution with parameters n – the number of experiments, and p – the probability for a "positive" outcome for each single experiment.

[3] $\text{Exp}(\mu)$ denotes the exponential distribution with mean μ.

Transition	p(Stop)	Transition	p(Stop)	Transition	p(Stop)	Transition	p(Stop)
S → S	0.33	M → S	0.33	S → M	0.66	M → M	0.50
S → F	1.00	M → F	0.66	F → F	0.05	F → M	0.33
F → S	0.10						

Table 1: Stop Probability by Street Type Transition. F = freeway, M = main street, S = side street.

Algorithm 1 *CreateTrip*

INPUT: *Start* - the start node, *Dest* the destination node, *Time* the starting time
OUTPUT: a trip from *Start* to *Dest*

```
let P be the shortest path from Start to Dest;
for each edge e = (p_i, p_{i+1}) ∈ P do
  Access v_max and segs, the geodata for e;
  for each seg = (s, t) ∈ segs do
    pos := s;
    while pos ≠ t do
      if distance(pos,t) > 50 m then
        if current speed < v_max then
          Apply an acceleration event to the trip;
        else
          Randomly choose either evt := deceleration event (p=90%) or evt := stop event (p=10%);
          With a probability proportional to 1/v_max: Apply evt;
        end if
      else
        Reduce velocity to α/180° · v_max where α is the angle between seg and the next segment in P;
      end if
      Move pos 5m towards t (or to t if it is closer than 5m);
    end while
    With a propability p(Stop) depending on the street type of the current egde and the street type
    of the next edge in P and according to Table 1, apply a stop event;
  end for
end for
```

Trips are created using the described behaviour, which is implemented by Algorithm 1.

Whereas work trips the can be created easily using Algorithm 1 passing *HomeNode*, *WorkNode*, and a starting instant, additional trips are more complicated. They are created using Algorithm 2 that determines random destinations and delay times and employs Algorithm 1 to create its sub trips.

4 Database Model

In BerlinMOD, we use two "kinds" of MOD: object-based and trip-based data. In the object-based approach (OBA), the complete history is kept together. In the trip-based approach (TBA), the motion of objects is recorded and stored as a sequence of single trips, which are kept separately in an additional relation. The licence plate number is used as a reference from the base relation to the relation containing the trips and vice versa.

In the trip-based design, each period of waiting time is represented as a separate stationary trip, i.e. a trip, where the vehicle doesn't move, but keeps its position all the time. Since we do not explicitly differentiate between single trips in the OBA, between each pair of subsequent trips, a vehicle will simply keep immobile at its positon, which is the final position of the earlier and the initial position of the latter trip.

The amount of spatio-temporal data generated is determined by parameter SCALEFACTOR. It scales the amount of simulated vehicles (SCALEFCARS) and the number of days (SCALEFDAYS) they are observed. With SCALEFACTOR = 1.0, 2,000 vehicles are observed for 28 days, starting on a Monday. With

Algorithm 2 *AdditionalTrip*

INPUT: *Home*: *node*, *BlockStart*: *instant* (the begin of the spare time block),
nbh: *setofnodes* (neighboorhood of the vehicle's home node)
OUTPUT: a trip: *mpoint* with up to 3 destinations
Select a number N of destinations: 1 (p=50%), 2 (p=25%), or 3 (p=25%);
Let *Start* := *Home*; $i := 0$; *Trip* := empty;
Select *Time* uniformly distributed within two hours after *BlockStart*;
for $i \in \{0, 1, 2, 3\}$ **do**
  if $i < N$ **then**
    Select destination node *Dest* within *nbh* (p=80%) or from the complete graph (p=20%);
  else
    Dest := *Home*;
  end if
  Trip := *Trip* + **createTrip**(*Start*, *Dest*, *Time*);
  if $i < N$ **then**
    Determine a delay time $dt \in [0, 120]$ min using a bounded Gaussian distribution;
    Append a break of length *dt* to *Trip*;
    Start := *Dest*;
    Time := **endtime**(*Trip*);
  end if
end for
return *Trip*

any other scaling factor, these default values are scaled by the square root of the choosen SCALEFACTOR.

In addition to the represented vehicle movements, we also define six relations containing random spatial and temporal data to build query points and ranges within the benchmark queries. The size of these samples is determined by the SAMPLESIZE parameter. We use a value of 100 for the benchmark.

Rather than using simple spatial ranges, like MBR-like rectangles, we use more complex polygon-shaped regions. This implies, that simple index-selections won't suffice for selections in most cases. Instead, candidates selected by an index must additionally qualify by passing a more complex spatial selection predicate.

In our database, we distinguish between objects used in both approaches (the object-based and the trip-based), and those used in only one of them.

Common Database Objects:

`Nodes`: **relation**{*NodeId*: *int*, *Pos*: *point*} — relation of all nodes.

`QueryPoints`: **relation**{*Id*: *int*, *Pos*: *point*} — relation of SAMPLESIZE query points, randomly chosen from `Nodes`.*Pos*, *Id* is a generated key for this relation.

`QueryRegions`: **relation**{*Id*: *int*, *Region*: *region*} — relation of SAMPLESIZE query regions, where *Id* is a key. The regions are regular n-gones with center point p and height h, with $n \sim F(1, 100, 100)$[4], p is a uniformly randomly chosen *Pos* from `Nodes`, and $h \sim F(3, 1000, 998)$.

`QueryInstants`: **relation**{*Id*: *int*, *Instant*: *instant*} — relation of SAMPLESIZE query instants, where *Id* is a key. The instants are uniformly distributed within the observation period.

`QueryPeriods`: **relation**{*Id*: *int*, *Period*: *periods*} — relation of SAMPLESIZE query periods, where *Id* is a key. The starting instants are sampled uniformly from the complete observation period. The periods' durations d are calculated to be $d = abs(x)$ days, were x is sampled from a Gaussian distribution ($x \sim N(0, 1)$[5]).

`QueryLicences`: **relation**{*Id*: *int*, *Licence*: *string*} — relation of SAMPLESIZE query licence plate numbers, where *Id* is a key. Licence plate numbers are uniformly sampled from all vehicles' licence plate numbers.

[4] $F(a, b, m)$ denotes the discrete uniform distribution of the m integer events from the interval $[a, b]$.
[5] $N(\mu, \sigma^2)$ is the normal (Gaussian) distribution with mean μ and variance σ^2.

Object-Based Approach (OBA) only:

`dataScar`: **relation**{*Licence*: *string*, *Model*: *string*, *Type*: *string*, *Trip*: *mpoint*} — relation of vehicle descriptions (car type, car model and licence plate number), including the complete position history as a single *mpoint* value per vehicle, where *Licence* is a key.

Trip-Based Approach (TBA) only:

`dataMcar`: **relation**{*Licence*: *string*, *Model*: *string*, *Type*: *string*} — relation of all vehicle descriptions (without position history).

`dataMtrip`: **relation**{*Licence*: *string*, *Trip*: *mpoint*} — relation containing all vehicles' movements and pauses as single trips (*mpoint* values).

Here, {*Licence*} is a key/foreign key for `dataMcar` and {*Licence*, *Trip*} is a key for `dataMtrip`.

5 Benchmark Queries

In this section, we present a set of queries for the BerlinMOD benchmark, working on the data set described before. First, we will determine, which kinds of queries may arise. To do so, we define **query types**. From the large amount of possible types, we select the most interesting ones and formulate queries for each of them.

5.1 Query Types

The first query type contains queries whose predicates do not rely on moving object type attributes, i.e. they are restricted to standard attributes. This is important because some storage models for spatio-temporal data require copying of standard types or introduction of synthetic key attributes which again require additional joins within queries. For example, in the object-based approach (OBA), the licence plate number *licence* is a key attribute. When switching to the trip based representation (TBA), this is no longer a key (a car with a given licence plate number will make several trips), or it is a key outside the relation containing the trips. In this case, the licence plate number must be connected with the trips using a join operation.

If a spatio-temporal object is involved, we can identify the following query properties:

1. **Object Identity** (known / unknown)
   Here we distinguish between queries starting with a known object "Does object X ..." and queries where we do not know any concerned object in advance "Which objects do ...", respectively.

2. **Dimension** (temporal / spatial / spatio-temporal)
   This criterion refers to the dimension(s) used in the query.

3. **Query Interval** (point / range / unbounded)
   This property determines the presence/size of the query interval.

4. **Condition Type** (single object / object relations)
   If relations between objects are subject of the query, joins are part of the query plan and we call this "object relation".

5. **Aggregation** (aggregation / no aggregation)
   This attribute indicates whether the result is computed by some kind of aggregation. Sometimes, this property will depend on whether the object based or trip based approach is used (no aggregation is needed in the first case, but it is required in the latter one). Such cases will be noted by "(no) aggregation".

By combining each possible value of the different attributes, we can identify 72 query types if a spatio-temporal object is part of the query. Not all types of queries are realizable. For example, if the object identity is known within a point query on a single object, this will exclude any kind of aggregation.

Type	description
bool	usual boolean data type
instant	a point in time
int	integer numbers
ipoint	a pair of an *instant* and a *point* value
line	data type describing a complex line as a set of segments
mbool	a time dependent boolean value
mpoint	moving point, i.e. a mapping from time into space
mreal	a time dependent real number
periods	a set of disjoint and non-connected time intervals
point	a geometric 2D position (x,y)
real	a real number
region	data type for spatial 2D regions

Table 2: Used Data Types

5.2 Queries

We present a set of seventeen queries of different types, which may be interesting. Together, these queries form our corpus of benchmark queries. First, we formulate the query in common English and then in a more formal, SQL-like notation. Table 2 shows an overview of the data types used. In Table 3 the signatures and a short description of the operators used in the queries are given. Operators and data types are formally specified in [12].

As the performance of the queries strongly depends on the point/range values and the objects' identity, we do not run queries for just a single combination of parameters (parameters involved are query points, regions, instants, periods and vehicle identities/licences), but choose a set of 100 parameter combinations, wherever this is applicable.

To this end, we have sampled the universe of our database for query parameters and saved them to relations `QueryPoints`, `QueryRegions`, `QueryInstants`, `QueryPeriods`, and `QueryLicences`, as described in Section 4. For the sake of simplicity during formulation of queries, we save the first ten tuples of each such relation to a relation with the same name and a suffix "1", the second ten tuples to a relation with that name, but with suffix "2".

From these sample relations, we create 100 combinations of query parameters, either by using the full sample relation (if there is only one parameter), or by combining the smaller "1" relations of distinct types, or the "1" and "2"-relation, where two parameters have the same type.

Due to the two representations (object or trip based), some queries must be formulated differently. If only the query for the object based representation is given below, then the query for the trip based representation can be derived by just changing the names of the used relations properly.

As usual, index structures should be created to allow for performant data access. The choice of index types and index keys is left to the database administrator. As indexes have strong influence on the benchmark results, we will explain which indexes could help with the different queries.

non spatio-temporal

Query 1 What are the models of the vehicles with licence plate numbers from `QueryLicence`?

```
SELECT DISTINCT LL.Licence AS Licence, Model
FROM dataScar, QueryLicence LL
WHERE Licence = LL.Licence;
```

Name	Signature / Description	Name	Signature / Description
val:	*ipoint* → *point* Extracts the position from the argument.	**atinstant**:	*mpoint* × *instant* → *ipoint* Computes the state of the *mpoint* for the given *instant*.
create_instant:	*string* → *instant* Converts the argument to an *instant*.	**circle**:	*point* × *real* × *int* → *region* Constructs a regular n-gon around the given position and diameter, n is given by the third argument.
present:	*mpoint* × *instant* → *bool* Checks whether the given *instant* is part of the definition time of the *mpoint*.	**passes**:	*mpoint* × *point* → *bool* *mpoint* × *region* → *bool* Test if the moving point will be on the given position in any instant.
trajectory:	*mpoint* → *line* Projects the moving point into the space.	**distance**:	*line* × *line* → *real* *mpoint* × *mpoint* → *mreal* Computes the minimum distance between the arguments.
inst:	*ipoint* → *instant* Extract the instant from the argument.	**initial**:	*mpoint* → *ipoint* Computes the initial state of the argument.
intersection:	*mpoint* × *mpoint* → *mpoint* Computes the spatio-temporal intersection of its arguments.	**at**:	*mbool* × *bool* → *mbool* *mpoint* × *point* → *mpoint* *mpoint* × *region* → *mpoint* Restricts the first argument to be inside or equal to the second argument.
length:	*mpoint* → *real* Computes the driving distance of the moving point.	**atperiods**:	*mpoint* × *periods* → *mpoint* Restricts the *mpoint* to the given time intervals.
deftime:	*mpoint* → *periods* Returns the set of time intervals where the *mpoint* is defined.	**inside**:	*point* × *region* → *bool* Tests, whether the *point* is inside the *region*.
sometimes:	*mbool* → *bool* Returns 'TRUE', if the *mpoint* is 'TRUE' at last once.	**isempty**:	X → *bool* Returns 'TRUE', if the argument is undefined or empty (for moving type and period values).
<= :	*mreal* × *real* → *mbool* Returns a time dependent boolean indicating when the first argument is smaller than the second.	**concat**:	*mpoint* × *mpoint* → *mpoint* Concatenates two moving points with distinct time intervals.

Table 3: Operator Descriptions

The operators listed correspond to those described in [12]. Operators **circle**, **create_instant** and **concat** have been added. Operators **sometimes** and **length** have been added for the sake of simplicity, as e.g. **sometimes**(X) could also be expressed as (**not**(**isempty**(**deftime**(X **at** TRUE)))).

This query will test the performance on standard types and indexes. An index on *Licence* might be useful. Some DBMS might partially access *Trip* data, while this is not needed, resulting in performance losses. This can be avoided in the TBA.

Query 2 How many vehicles exist that are "passenger" cars?

```
SELECT COUNT(Licence)
FROM dataScar
WHERE Type = "passenger";
```

Similar to the the first query, but having a more selective predicate.

known - temporal - point - single - no aggr

Query 3 Where have the vehicles with licences from `QueryLicences1` been at each of the instants from `QueryInstants1`?

```
SELECT LL.Licence AS Licence, II.Instant AS Instant,
       val(C.Trip atinstant II.Instant) AS Pos
FROM dataScar C, QueryLicences1 LL, QueryInstants1 II
WHERE C.Licence = LL.Licence AND C.Trip present II.Instant;
```

This query restricts histories to single instants. A temporal index on *Trip* might be useful only in the TBA. Both representations will benefit from indexes on *Licence*.

unknown - spatial - point - single - no aggr

Query 4 Which licence plate numbers belong to vehicles that have passed the points from `QueryPoints`?

```
SELECT PP.Pos AS Pos, C.Licence AS Licence
FROM dataScar C, QueryPoints PP
WHERE C.Trip passes PP.Pos;
```

In the TBA, *Licence* is no longer a key attribute in the relation used. Therefore we need to use `SELECT DISTINCT`. A spatial index on *Trips* is advantageous to lookup all motions passing *PP*.

known - spatial - unbounded - relation - (no) aggr

Query 5 What is the minimum distance between places, where a vehicle with a licence from `QueryLicences1` and a vehicle with a licence from `QueryLicences2` have been?

```
SELECT LL1.Licence AS Licence1, LL2.Licence AS Licence2,
       distance(trajectory(V1.Trip),trajectory(V2.Trip))
       AS Dist
FROM dataScar V1, dataScar V2, QueryLicences1 LL1,
     QueryLicences2 LL2
WHERE V1.Licence = LL1.Licence AND V2.Licence = LL2.Licence
      AND V1.Licence <> V2.Licence;
```

For the OBA, no aggregation is needed for this query, as the global distance is calculated using a combination of **trajectory** and **distance**.

For the TBA, we need an explicit aggregation:

```
SELECT LL1.Licence AS Licence1, LL2.Licence AS Licence2,
       MIN (distance(trajectory(T1.Trip),trajectory(T2.Trip)))
       AS Dist,
FROM dataMtrip T1, dataMtrip T2, QueryLicences1 LL1,
     QueryLicences2 LL2
WHERE T1.Licence = LL1.Licence AND T2.Licence = LL2.Licence
      AND T1.Licence <> T2.Licence
GROUP BY LL1.Licence, LL2.Licence;
```

Here, indexes on *Licence* seem to be most helpful. Differences between applying the spatial **distance** and the projection (**trajectory**) to small (TBA) and large (OBA) data will be of significance and show up differences in the DBMS's performance.

unknown - spatio-temporal - unbounded - relation - no aggr

Query 6 What are the pairs of licence plate numbers of "trucks", that have ever been as close as 10m or less to each other?

```
SELECT V1.Licence AS Licence1, V2.Licence AS Licence2
FROM dataScar V1, dataScar V2
WHERE V1.Licence < V2.Licence AND V1.Type = "truck"
      AND V2.Type = "truck"
      AND sometimes(distance(V1.Trip,V2.Trip) <= 10.0);
```

In the TBA:

```
SELECT DISTINCT T1.Licence AS Licence1, T2.Licence AS Licence2
FROM dataMtrip T1, dataMcar C1, dataMtrip T2, dataMcar C2
WHERE T1.Licence < T2.Licence AND T1.Licence = C1.Licence
      AND T2.Licence = C2.Licence AND C1.Type = "truck"
      AND C2.Type = "truck"
      AND sometimes(distance(T1.Trip, T2.Trip) <= 10.0);
```

A spatial (or spatio-temporal) index join on *Trip* can be used to select candidates from all vehicles. This query could be used to compare the performance of different index structures, e.g. indexes with different granularities for the indexed keys, as proposed in [4].

unknown - spatial - unbounded - relation - no aggr

Query 7 What are the licence plate numbers of the "passenger" cars that have reached the points from `QueryPoints` first of all "passenger" cars during the complete observation period?

```
SELECT PP.Pos AS Pos, V1.Licence AS Licence
FROM dataScar V1, QueryPoints PP
WHERE V1.Trip passes PP.Pos AND V1.Type = "passenger"
      AND inst(initial(V1.Trip at PP.Pos)) <= ALL
      ( SELECT inst(initial(V2.Trip at PP2.Pos)) AS FirstTime
        FROM dataScar V2
        WHERE V2.Trip passes PP.Pos AND V2.Type = "passenger"
      );
```

For the TBA:

```
SELECT DISTINCT V1.Licence AS Licence, PP.Pos AS Pos
FROM dataMtrip T1, dataMcar C1, QueryPoints1 PP
WHERE T1.Trip passes PP.Pos
      AND C1.Type = "passenger" AND C1.Licence = T1.Licence
      AND inst(initial(T1.Trip at PP.Pos)) <= ALL
      ( SELECT inst(initial(T2.Trip at PP.Pos)) AS FirstTime
        FROM dataMtrip T2, dataMcar C2
        WHERE V2.Trip passes PP.Pos
              AND T2.Licence = C2.Licence
              AND C2.Type = "passenger"
      );
```

This query is a classical example for using a spatial index.

known - temporal - range - single - (no) aggr

Query 8 What are the overall travelled distances of the vehicles with licence plate numbers from `QueryLicences1` during the periods from `QueryPeriods1`?

```
SELECT V1.Licence AS Licence, PP.Period AS Period
       length(V1.Trip atperiods PP.Period) AS Dist
FROM dataScar V1, QueryPeriods1 PP, QueryLicences1 LL
WHERE V1.Licence = LL.Licence AND V1.Trip present PP.Period;
```

Within the TBA, we need an aggregation to sum up the travelled distances.

```
SELECT V1.Licence AS Licence, PP.Period AS Period,
       SUM(length(V1.Trip atperiods PP.Period)) AS Dist
FROM dataMtrip V1, QueryPeriods1 PP, QueryLicences1 LL
WHERE V1.Licence = LL.Licence AND V1.Trip present PP.Period
GROUP BY V1.Licence, PP.Period;
```

A temporal index on *Trip* is helpful in the TBA, while in the OBA, it will be useless unless some sophisticated approach like double indexing [4] is used.

unknown - temporal - range - single - aggr

Query 9 What is the longest distance that was travelled by a vehicle during each of the periods from `QueryPeriods`?

```
SELECT PP.Period AS Period,
       MAX(length(V1.Trip atperiods PP)) AS Dist
FROM dataScar V1, QueryPeriods PP
WHERE V1.Trip present PP.Period
GROUP BY PP.Period;
```

Again, in the TBA we need to sum up the lengths of all restricted trips. Therefore, we create a view first:

```
CREATE VIEW Distances AS
  SELECT SUM(length(V1.Trip atperiods PP)) AS Length,
         PP.Period AS Period, V1.Licence AS Licence
  FROM dataMtrip V1, QueryPeriods PP
  WHERE V1.Trip present PP.Period
  GROUP BY PP.Period, V1.Licence;

SELECT Period, MAX(Length) AS Dist
FROM Distances
GROUP BY Period;
```

A temporal index on *Trip* can be of use in the TBA to select candidates for the aggregation.

known - spatio-temporal - range - relation - no aggr

Query 10 When and where did the vehicles with licence plate numbers from `QueryLicences1` meet other vehicles (distance < 3m) and what are the latters' licences?

```
SELECT V1.Licence AS QueryLicence, V2.Licence AS OtherLicence,
       (V1.Trip atperiods(deftime((distance(V1.Trip, V2.Trip)
        <= 3.0) at TRUE))) AS Pos
FROM dataScar V1, dataScar V2, QueryLicences1 LL
WHERE V1.Licence = LL.Licence AND V2.Licence <> V1.Licence
      AND sometimes(distance(V1.Trip, V2.Trip) <= 3.0);
```

Since this will be a rather complex query (the temporal distances to all other vehicles need to be calculated), we restrict our parameter set to 10 licences.

In the TBA, we need to group by *Licence* and aggregate (using concat) the intermediate results to one single *mpoint* value per QueryLicence:

```
SELECT V1.Licence AS Licence, V2.Licence AS OtherLicence,
       AGGR(concat, V1.Trip atperiods (deftime((distance(
            V1.Trip, V2.Trip) <= 3.0) at TRUE)), emptympoint)
       AS Pos
FROM dataMtrip V1, dataMtrip V2, QueryLicences1 LL
WHERE V1.Licence = LL.Licence AND V2.Licence <> V1.Licence
      AND sometimes(distance(V1.Trip, V2.Trip) <= 3.0)
GROUP BY V1.Licence, V2.Licence;
```

The construct **AGGR**(aggr_op, val, defaultval) is a notation for a universal aggregation operator that applies an operator **aggr_op**: $X \times X \rightarrow X$ to aggregate over all instantiations of *val*. If no *val* is provided, i.e. the input relation/stream is empty, *defaultval* (which must also be of type X) is returned. *emptympoint* is an *mpoint* with an empty history (i.e. the *mpoint* is defined, but has no recorded position ever).

This query features an expensive nonequal condition. Smart optimizers may translate the last condition into a range query using a spatial or spatio-temporal index on *Trip*.

unknown - spatio-temporal - point - single - no aggr

Query 11 Which vehicles passed a point from `QueryPoints1` at one of the instants from `QueryInstants1`?

```
SELECT C.Licence AS Licence, PP.Pos AS Pos,
       II.Instant AS Instant
FROM dataScar C, QueryPoints1 PP, QueryInstants1 II
WHERE val(C.Trip atinstant II.Instant) = PP.Pos;
```

Though the condition seems to be a spatial range rather than a point condition, the spatio-temporal semantics require that for attribute *Trip* the position is a function of time, and hence this condition actually is a spatio-temporal point condition. In the TBA, this query performs best with a spatio-temporal index, but also a temporal or spatial index may help here. For the OBA, the spatio-temporal index might be inferior to the spatial index, and the temporal index will be of no worth.

unknown - spatio-temporal - point - relation - no aggr

Query 12 Which vehicles met at a point from `QueryPoints1` at an instant from `QueryInstants1`?

```
SELECT PP.Pos AS Pos, II.Instant AS Instant,
       C1.Licence AS Licence1, C2.Licence AS Licence2
FROM dataScar C1, dataScar C2,
     QueryPoints1 PP, QueryInstants1 II
WHERE val(C1.Trip atinstant II.Instant) = PP.Pos
      AND val(C2.Trip atinstant II.Instant) = PP.Pos;
```

Just as for the last query, but as a join is required, the impact of indexes on *Trip* will be stronger. In the TBA, we might need to use **SELECT DISTINCT** instead of a simple **SELECT**.

unknown - spatio-temporal - range - single - no aggr

Query 13 Which vehicles travelled within one of the regions from `QueryRegions1` during the periods from `QueryPeriods1`?

```
SELECT RR.Region AS Region, PP.Period AS Period,
       C.Licence AS Licence
FROM dataScar C, QueryRegions1 RR, QueryPeriods1 PP
WHERE NOT(isempty(((C.Trip atperiods PP.Period)
          at RR.Region)));
```

For the TBA we need to use **SELECT DISTINCT**. This is a spatio-temporal range query with 3D volumes (cuboid) as query windows. Again, this tests the performance of spatio-temporal access methods, for a range query in this case.

Query 14 Which vehicles travelled within one of the regions from `QueryRegions1` at one of the instants from `QueryInstants1`?

```
SELECT RR.Region AS Region, II.Instant AS Instant,
       C.Licence AS Licence
FROM dataScar C, QueryRegions1 RR, QueryInstants1 II
WHERE val(C.Trip atinstant II.Instant) inside RR.Region;
```

Once more, we use **SELECT DISTINCT** for the TBA query. This is a spatio-temporal range query, with query windows degenerated into 2D spatial rectangles. Together with Queries 13 and 15, this query will help to assess the performance of spatio-temporal indexes.

Query 15 Which vehicles passed a point from `QueryPoints1` during a period from `QueryPeriods1`?

```
SELECT PO.Pos AS Pos, PR.Period AS Period,
       C.Licence AS Licence
FROM dataScar C, QueryPoints1 PO, QueryPeriods1 PR
WHERE NOT(isempty(((C.Trip atperiods PR.Period) at PO.Pos)));
```

As before, we **SELECT DISTINCT** for the TBA query. Being another spatio-temporal range query, the query windows are degenerated to 1D volumes (temporal intervals) here.

unknown - spatio-temporal - range - relation - no aggr

Query 16 List the pairs of licences for vehicles, the first from `QueryLicences1`, the second from `QueryLicences2`, where the corresponding vehicles are both present within a region from `QueryRegions1` during a period from `QueryPeriod1`, but do not meet each other there and then.

```
SELECT PP.Period AS Period, RR.Region AS Region,
       C1.Licence AS Licence1, C2.Licence AS Licence2
FROM dataScar C1, dataScar C2, QueryRegions1 RR,
     QueryPeriods1 PP, QueryLicences1 LL1, QueryLicences2 LL2
WHERE C1.Licence = LL1.Licence AND C2.Licence = LL2.Licence
      AND LL1.Licence < LL2.Licence
      AND (C1.Trip at PP.Period) passes RR.Region
      AND (C2.Trip at PP.Period) passes RR.Region
      AND isempty((intersection(C1.Trip, C2.Trip)
        atperiods PP.Period) at RR.Region);
```

For the TBA, we formulate:

```
(SELECT RR.Region AS Region, PP.Period AS Period,
        C1.Licence AS Licence1, C2.Licence AS Licence2
 FROM dataMtrip C1, dataMtrip C2, QueryRegions1 RR,
       QueryPeriods1 PP, QueryLicences1 LL1, QueryLicences2 LL2
 WHERE C1.Licence < C2.Licence
       AND C1.Licence = LL1.Licence
       AND (C1.Trip at PP.Period) passes RR.Region
       AND C2.Licence = LL2.Licence
       AND (C2.Trip at PP.Period) passes RR.Region
 GROUP BY RR.Region, PP.Period, C1.Licence, C2.Licence )
EXCEPT
(SELECT RR.Region AS Region, PP.Period AS Period,
        C1.Licence AS Licence1, C2.Licence AS Licence2
 FROM dataMtrip C1, dataMtrip C2, QueryRegions1 RR,
      QueryPeriods1 PP, QueryLicences1 LL1, QueryLicences2 LL2
 WHERE C1.Licence < C2.Licence
       AND C1.Licence = LL1.Licence
       AND (C1.Trip at PP.Period) passes RR.Region
       AND C2.Licence = LL2.Licence
       AND (C2.Trip at PP.Period) passes RR.Region
       AND NOT(isempty(deftime((intersection(C1.Trip, C2.Trip)
                    atperiods PP.Period) at RR.Region)))
 GROUP BY RR.Region, PP.Period, C1.Licence, C2.Licence )
```

Since this query depends on four parameters, we will explore not only 100, but 10,000 combinations.

This query tests the performance of the spatio-temporal operators. In the TBA, spatial, temporal or spatio-temporal indexes might raise the performance when used with additional **present** and/or **passes** conditions for both *Trip* attributes. Otherwise, we don't expect indexes to accelerate this query.

unknown - spatial - unbounded - relation - aggr

Query 17 Which points from `QueryPoints` have been visited by a maximum number of different vehicles?

```
CREATE VIEW PosCount AS
   SELECT PP.Pos AS Pos, COUNT(C.Licence) AS Hits
   FROM QueryPoints PP, dataScar C
   WHERE C.Trip passes PP.Pos
   GROUP BY PP.Pos;

SELECT Pos
FROM   PosCount AS N
WHERE  N.Hits = (SELECT MAX(Hits) FROM PosCount);
```

For the TBA we have:

```
CREATE VIEW PosCount AS
   SELECT PP.Pos AS Pos, COUNT DISTINCT(C.Licence) AS Hits
   FROM QueryPoints PP, dataMtrip C
   WHERE C.Trip passes PP.Pos
   GROUP BY PP.Pos;

SELECT Pos
FROM   PosCount AS N
WHERE  N.Hits = (SELECT MAX(Hits) FROM PosCount);
```

6 Implementation

In this section, we decribe how we create the benchmark data using the extensible database system Secondo [5, 13, 19]. To our knowledge, this is the first time a DBMS is used to create spatio-temporal benchmark data. Secondo is a research prototype DBMS that can be extended by algebra modules, defining new types and operations on them using the second order signature mechanism [11].

For the generation of moving objects, a new algebra module, the "SimulationAlgebra" has been implemented. Other algebra modules providing types and operations for relations, graphs, spatial, and temporal data already exist in the system and are also used.

To create the moving objects, we use a database script containing a sequence of Secondo commands. Each command is formulated at the so-called executable level. At this level, each expression is written as a set of operators, constant objects, and existing database objects.

In the following sub-sections, the principle of the data generation is explained. The complete database script can be found in Appendix A. Numbers within comments of the script refer to the corresponding numbers in the following text.

We have set up a web site where the Secondo system and documentation, all the scripts from the appendices, and scripts with executable benchmark queries for Secondo (the ones we used in Section 7.2) are available for download [1].

Since within the scripts many different operators are employed, we cannot give in-depth explanations for all of them and restrict ourselves to the most important ones here. The interested reader will find more detailed information on data types, operators etc. using the self-inquiry features built into Secondo.

6.1 Map Import and Parameters

As a very first step (6.1.1), we create a new database and import the map of Berlin using the `restore` command. We use converted street data from the bicycle routing program **bbbike** [17]. The result is a relation containing the name of each street, its geometry, and street type:

`streets`: **relation**{*Name*: *string*, *Typ*: *string*, *geoData*: *line*}

Afterwards, the parameters of the script are set (6.1.2). To define these, we create some simple database objects. For example, the command

```
let SCALEFACTOR = 1.0;
let SCALEFCARS = sqrt(SCALEFACTOR);
```

creates a database object of the type *real* with name *SCALEFACTOR* and value 1.0. After this, a second object *SCALEFCARS* of type *real* is computed using the first object in its definition. Further constants are defined in the same way, e.g. the **create_duration**(*p1*,*p2*) command produces an object of the type *duration* with a length of *p1**86,400,000 + *p2* milliseconds.

6.2 Creating the Street Graph

After removing impassable streets, we derive a maximum speed for each street according to its type (6.2.1).

To give an impression how a command is processed, we will describe this command in more detail. The corresponding lines from the script are (here in a reformatted style):

```
let streets1 =
  streets feed
  filter[ not (.Typ contains "gesperrt") ]
  extend[ Vmax :
    ifthenelse( .Typ contains "Wichtig" , 70.0,
       ifthenelse( .Typ contains "Haupt" , 50.0,
         ifthenelse( .Typ contains "Neben", 30.0 ,-1.0)))]
  consume;
```

The **feed** operator produces a stream of tuples from the relation "streets". Only tuples fulfilling the given condition pass the **filter** operator. Using **extend**, we add a new attribute *Vmax* (with the maximum allowed speed) to each tuple, whose value is derived from the type of the street. Finally, the **consume** operator collects all tuples of the stream into a relation which is the result of this expression. The **let** command stores the result as a database object named *streets1*.

Now, we collect all remaining streets into a single line object and select the largest component *allstreets* of the result to be sure to produce a connected graph (though the original imported street network is connected, some streets are closed for motor vehicles) (6.2.2).

This single component line object is divided into sections using crossings and end points of streets as split points. Crossings are automatically recognized by the **polylines** operator, the end points of the streets are used as an argument for this operator to avoid the loss of street transitions without a crossing (6.2.3-4).

We extract the end points of the sections as nodes of the graph and assign an id to each of them (6.2.5-6).

We join the sections and the nodes with respect to the position of the node and the sections' end points, respectively (6.2.7). Similarly, the maximum speed is assigned to each section. To produce an undirected graph, we add all reverse edges (6.2.8). To build the graph, we compute the cost for each section using its length and maximum allowed speed. For a visualization of the graph, we also append its geographic positions to each node (6.2.9).

6.3 Creating a Relation for the Labour Path

For each car, we select a *HomeNode* and a *WorkNode*. To do that, we can use one of two methods: The simple selection (as implemented by the script from Appendix A) uses a uniform distribution over all nodes of the network. Thus, the density of the nodes in the network corresponds to the densitity of such nodes. Fig. 2 a) shows a distribution of the selection of 1000 nodes within the network using this approach. We use this method for data generation in BerlinMOD.

The second (alternative) possibility implements the region based approach described in [2]. The according script is provided in Appendix B. To realize it, we first define an array of regions shown in Fig. 3.

We assume that the population density drops with increasing array index. If a node occurs in more than one region, it is assigned to the one with the lowest array index. From these regions, we compute a partition of all nodes. We assign the following probabilities to select one of the nodes for the given array index:

a) Network Based b) Region Based

Figure 2: Distributions Generated Using Both Node Selection Algorithms.

Index	0	1	2	3	4	5
Prob	0.4	0.2	0.15	0.1	0.08	0.07

For simplicity, we use the same regions but the probabilities in reversed order for selecting the work nodes.

Between the nodes within a single partition a uniform distribution is used to select one of the nodes. The result of this approach applied to the work nodes is shown in Fig. 2 b).

If you want to use the Region Based Approach, you just need to insert the code from Appendix B after line 143 into the generator script in Appendix A, and change the original line 155 of the generator script. Replace

```
rng_intN((Nodes count)) + 1
```

by

```
selectHomeNode()
```

to select the *HomeNode* region based. In a similar way, you can change line 156 to modify the selection of the *WorkNode*.

Using one of these methods (BerlinMOD standard is the Network Based approach), we create a relation *vehicle1* containing an *Id*, the id of its *HomeNode* and the id of the *WorkNode* (6.3.1). Note, that the *HomeNode* may be equal to the *WorkNode* (home workers).

Starting from this relation, we compute the neighbourhood of the *HomeNode* and add the count of neighbours to the *vehicle* relation for further use (6.3.2-3).

For computing the paths to and from the work, we just replace *WorkNode* and *HomeNode* by the shortest path in the graph (6.3.4). To find the shortest path, we apply Dijkstra's algorithm [6] that has been implemented in the GraphAlgebra.

6.4 Creating the Moving Points

To create the moving point data (which we will call "moving points" furtheron for the sake of simplicity), we use a set of user defined functions. The most important one is called *Path2MPoint* (6.4.2), which produces a single trip along a path starting at a given time. Basically, this function builds a stream of *edges* from the path and connects each edge with the underlying geometry and the allowed maximum speed for this section using the **loopjoin** operator. This produces a stream of tuples with all attributes required for the **sim_create_trip** operator as described in Algorithm 1. For the simulation of a GPS device, we use the **samplempoint** operator. This operator traces the original position and changes the interval length between two consecutive sample points lying on colinear segments of the street network

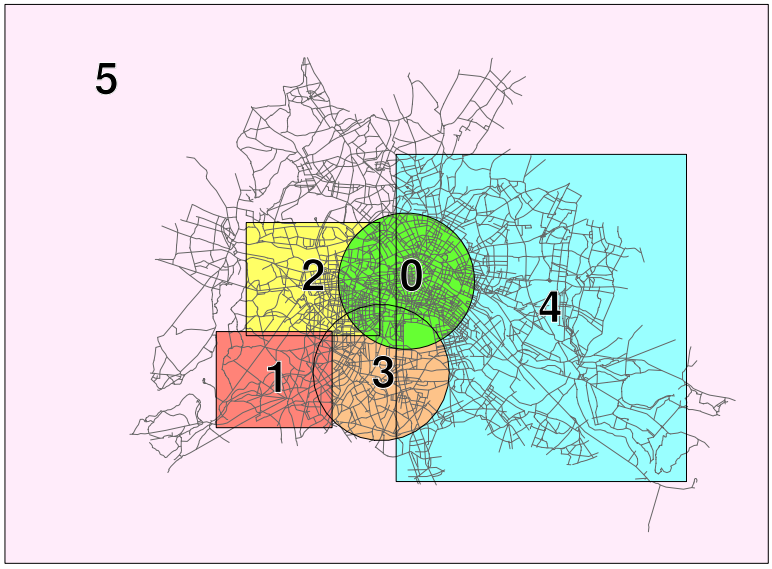

Figure 3: Regions Defined for Selecting Home and Work Nodes

to *GPSINTERVAL* milliseconds. At any change of direction, a sample point is created (even if the *GPSINTERVAL* is not elapsed). This corresponds to the simulation of exact GPS measures combined with a mapping to the street network.

Trips within the spare time are created by the function *CreateAdditionalTrip* (6.4.7). It is based on the functions *SelectDestNode* (6.4.3), *CreatePause* (6.4.4-5), and *Path2MPoint* (6.4.2).
SelectDestNode selects a destination node applying Algorithm 2. Thereby, some instances of the **ifthenelse** operator ensure the correct probabilities for the different spare time trips.

The observation of a vehicle for a complete day is performed by the function **CreateDay** (6.4.8). It distinguishes between working days and weekend days to produce two kinds of different behaviour.

Additional information is stored into further attributes, e.g. the vehicles' model (manufacturer) and type (passenger car, bus or truck). Also, in real applications, a car is not identified by an id but by a licence plate number, e.g. represented as a string value. To append that kind of information, we use the auxiliary functions *LicenceFun* (6.4.11), computing a unique licence plate number from an id, *RandModel* (6.4.9), and *RandType* (6.4.10) returning a model and a type for each car respectively. Both latter functions use predefined probabilities to choose from the sets of possible values.

The "main" function of the script is *CreateVehicles*. For each vehicle, *NUMDAYS* days are created using the function *CreateDay*. Then we aggregate over all movements for each car. Possible gaps in the definition time are closed using the **sim_fill_up** operator. Additionally, for each vehicle the licence plate number, the type and the model are created.

The actual creation of the moving points is started calling the *CreateVehicles* function.

6.5 Deriving the Trip Based Representation

Our simulation yields long time observations of the given number of vehicles. Thus, all created moving points have the same definition time. For this reason, a temporal index will either return all cars or nothing for any given instant or interval. To avoid this, we can use the trip based representation. In this representation, the complete movement of a car is split into its different trips. We define a trip as a connected movement without a break longer than a predefined constant *MINPAUSE*. Because there is a special operator for splitting moving points at such major breaks (**sim_trips**), the creation of the trip based representation is very simple (6.5).

6.6 Creating Query Relations

For the execution of the benchmark queries, some relations containing values used to instantiate query parameters are needed. We create these data by sampling the data used during the creation of the moving point data (6.6).

6.7 Simulating Measuring Errors (Optional)

As all created vehicles never leave the street network, a projection of their according moving points into the plane, will get a part of the network. In a real application, using GPS devices to capture their routes, this is improbable due to measurement errors. We assume that a GPS device has a maximum error depending on some factors (kind of device, weather etc.), but between each two consecutive measurements, its error changes only slightly. To simulate this, a special operator **disturb** can be used. It changes a given moving point randomly depending on the maximum error and the error difference between two measures. To simulate such errors, this operator can be inserted into the generator script (Appendix A) directly after the **sim_fillup_mpoint** operator in line 389 within the *CreateVehicles* function (6.4.12). For example

```
[...]
sim_fillup_mpoint[ create_instant(StartDayP - 1,0),
      create_instant((StartDayP + NumDaysP) + 1,0),
    TRUE, FALSE, FALSE] ]
disturb[100.0,1.0]
extend[ Licence: LicenceFun(.Id),
[...]
```

will generate errors of up to 1.0 metre between each pair of successive measurements, but the total absolute error is limited to 100.0 metres.

Dealing with inexact data must be reflected by adapting the spatial and spatio-temporal queries. As an example, please recall Query 4:

Query 4: Which licence plate numbers belong to vehicles that have passed the points from `QueryPoints`?

```
SELECT PP.Pos AS Pos, C.Licence AS Licence
FROM dataScar C, QueryPoints PP
WHERE C.Trip passes PP.Pos;
```

When handling inexact data, we cannot apply the **passes** operator directly to the *point* values *PP.Pos* anymore. We are required to modify the query: Instead of using each point value itself, we rather construct a circular *region* around it and apply the **passes** operator to that region. To do so, we need to replace PP.Pos by **circle**(PP.Pos, 10, 50) within the above query. This will also hold for all other spatial/ spatio-temporal queries and some other kinds of **WHERE** conditions (e.g. regarding distances).

As a default, we use the exact, undisturbed data in the BerlinMOD benchmark.

6.8 Benchmark Data Export

The produced data are stored into a database using the internal representation of SECONDO. It can be exported into a simple text file using the **save** command. The output is a nested list whose format is decribed on the SECONDO website [19]. A moving point is an atomic data type which can be used as an attribute in a relation. Converting the output for database systems without this property, the representation can be changed using the **gps** operator. It produces a stream of tuples consisting of an instant and the position of the moving point at this instant. Thus, we can change the complete relation containing the movement of all cars using the following command:

```
let dataScarExport = dataScar feed
      loopjoin[ gps(.Trip, [const duration value (0 2000)]) ]
      project[Id, Time, Position]  consume;
```

This relation can also be exported into a nested list text file, which can be used as the input for arbitrary converters to other data formats:

```
save dataScarExport to BerlinMODdata;
```

7 Creating and Using BerlinMOD

We have created BerlinMOD on a standard PC with the configuration shown in Table 4. We generated the benchmark data for three different scale factors, and tested Secondo using each of the data sets.

7.1 Data Generator: Performance and Statistics

The creation of the full-scaled database ($SCALEFACTOR = 1.0$, 2,000 cars, 28 days) takes about 9:33:17 hours on the mentioned system. In total, the creation of the full-scaled database together with several non-spatial, spatial, temporal and spatio-temporal indexes for both representations needs 14:50:35 hours. For $SCALEFACTOR = 0.2$, this requires only 2:35:04 h, and only 25:27 min for $SCALEFACTOR = 0.05$.

The creation time of about 10, respectively 15, hours for the full-scaled database is long, but we feel it is acceptable for a 20 GB database based on a fairly sophisticated simulation scenario. The resulting database has about 53 million temporal units. Note that in fact many more units are created temporarily based on "observations" but disappear again due to data reduction. GPS receivers would produce 1,209,600 timestamps per car within 28 days, leading to an overall figure of about 2.4 billion observations, but our representation removes subsequent sampling points having identical motion vectors, thus compressing the data, e.g. for an immobile car.

If only one of the representations is needed (object/trip based), the creation time can be reduced considerably.

Some more statistical information is given in Table 5.

7.2 Applying BerlinMOD to Secondo

We have used the benchmark to test the Secondo DBMS with its spatio-temporal extension algebras. The benchmark was carried out with hand-translated executable queries on the same PC that was used to generate the benchmark data before (see Table 4 for specification). Due to limitations of space, we cannot show and explain the translation of the queries here, but all queries are available as annotated scripts at the Secondo project website.

The results are presented in two tables. Table 6 shows the total response times for all queries. Table 7 shows the average response time per single query instance (calculated by dividing the total response times by the number of query parameter combinations applied). A query instance is a single combination of applicable query parameters. For Query 16, the benchmark applies 10,000 instances, for Query 10 10, for Queries 2, 6 and 17 only 1, and for all other queries 100 query instances.

As expected, the non spatio-temporal Queries 1, and 2 are executed very fast. There are significant differences between both representations. The OBA is significantly faster for queries 6, and 9, while the TBA makes the race in queries 3, 4, 5, 7, 10, 11, 12, 13, 14, 15, and 16. The OBA seems to have an advantage, if the objects' complete histories — or larger parts of them — are accessed (6, 9 and 17). The TBA can take advantage of (spatio-) temporal indices in bounded spatio-temporal queries (10, 11, 12, 13, 14, 15, 16).

Looking at the individual response times shown in Table 7, we observe that all but eight queries have acceptable runtimes (that is, seconds or at most a minute at scale factor 1.0). The outliers are Queries 6, 10, and 17 in both representations; Query 7 in the OBA, and Query 9 in the TBA only. Let us investigate the reason for this behaviour.

The runtime for Query 6 strongly depends on the number of observed vehicles N and the observation time. We need to select all "trucks" and perform a selfjoin on these, selecting $N_{truck} \cdot (N_{truck}/2 - 1)$ combinations. For these, we need a parallel scan of the joined vehicles' MOD histories to check whether

CPU:	Intel Core 2 DUO 2.4 Ghz
Memory:	2 GB
HDD:	2 x 500 GB, RAID 0
OS:	Open SuSe 10.2 (Kernel 2.6.18.2-34-default)

Table 4: Configuration of the used Standard PC

Scalefactor	0.05	0.2	1.0
Time to Build (excl. indexes)	12:16 min	74:34 min	573:17 min
Time to Build (incl. indexes)	25:27 min	155:04 min	890:35 min
On-Disk Size (excl. indexes)	1.91 GB	4.97 GB	19.45 GB
Observed Vehicles	447	894	2,000
Observation Period	6 days	13 days	28 days
Average Number of Units per Vehicle	6,138.857	13,040.283	26,963.197
Minimum Number of Units per Vehicle	1,038	2,498	3,247
Maximum Number of Units per Vehicle	22,016	44,517	94,021
Total Number of Trips	15,049	62,893	292,693
Average Number of Trips per Vehicle	33.667	70.353	146.347
Minimum Number of Trips per Vehicle	21	45	103
Maximum Number of Trips per Vehicle	53	101	199
Average Driven Distance per Vehicle	170.645 km	361.661 km	748.496 km
Average Trip Length	5,068 m	5,141 m	5,114 m
Nodes within the Network Graph	4,468	4,468	4,468
Edges within the Network Graph	13,384	13,384	13,384

Table 5: Statistics on Created Data for Different Scalefactors

Query	SCALEFACTOR = 0.05			SCALEFACTOR = 0.2			SCALEFACTOR = 1.0		
	Response Time		Result	Response Time		Result	Response Time		Result
	OBA	TBA	Size	OBA	TBA	Size	OBA	TBA	Size
Q1	0.406	0.335	100	0.476	0.451	100	0.460	0.407	100
Q2	0.099	0.055	1	0.050	0.140	1	0.113	0.099	1
Q3	2.290	0.616	100	6.303	0.993	100	12.080	1.092	100
Q4	76.664	49.516	794	625.431	273.426	2734	6232.560	966.393	9240
Q5	16.737	20.059	100	45.535	34.710	100	121.885	61.015	100
Q6	71.396	189.435	31	333.341	1942.071	86	7032.000	53910.502	1254
Q7	92.666	35.654	79	2325.340	241.182	99	23324.700	135.724	107
Q8	1.209	1.214	100	5.854	3.521	100	13.989	4.308	100
Q9	392.336	784.329	100	1102.580	3241.180	100	4791.730	21730.800	100
Q10	681.937	221.276	376	3170.480	1189.130	1003	23951.800	16410.200	4130
Q11	0.956	0.580	15	1.862	0.849	20	11.602	1.411	33
Q12	2.188	0.553	0	144.466	0.510	1	964.456	0.625	17
Q13	50.376	45.189	2015	426.079	128.682	1963	2015.680	261.572	2612
Q14	2.129	2.020	267	6.444	3.083	318	138.305	13.075	381
Q15	3.662	4.530	204	121.011	30.562	555	322.635	36.343	625
Q16	144.565	58.967	43	102.139	49.206	2	132.165	74.842	2
Q17	5.129	27.393	1	467.125	242.219	1	5374.080	1097.145	1
Total $[s]$	1544.745	1441.721	—	8884.516	7381.915	—	74440.240	94705.553	—
$[h:m:s]$	0:25:45	0:24:02	—	2:28:05	2:01:01	—	20:40:40	26:18:26	—

Table 6: Benchmarking the Secondo DBMS: Total Response Time.

Listed are the total response times over all query instances (all combinations of query parameters) in seconds. Result sizes are given by tuples.

their distance ever is 10 metres or below. The cost for this increases linearly with the observation time. All in all, this is very expensive. For the TBA, we additionally have to join the `dataMcar` and `dataMtrip` relations and to remove duplicates from the result.

In Query 10, for each car with a appropriate licence plate number, we select it from the relation. Then we join them with all cars that ever have a distance of less then 3 metres. In the TBA, we apply a range query on the spatio-temporal index to find all appropriate candidate trips and filter them; in the OBA we just scan the relation `dataScar` completely and filter directly, because neither the spatial, nor the spatio-temporal index showed to perform better in preceding experiments. In both representations, the number of combinations (candidates) to filter increases quadratically with the number of observed vehicles. Each computation of the spatio-temporal distance between two vehicles needs a parallel linear scan through both MOD histories.

Query	SCALEFACTOR = 0.05 Response Time		SCALEFACTOR = 0.2 Response Time		SCALEFACTOR = 1.0 Response Time	
	OBA	TBA	OBA	TBA	OBA	TBA
Q1	0.004	0.003	0.005	0.005	0.005	0.004
Q2	0.099	0.055	0.050	0.140	0.113	0.099
Q3	0.023	0.006	0.063	0.010	0.121	0.011
Q4	0.767	0.495	6.254	2.734	62.326	9.664
Q5	0.167	0.201	0.455	0.347	1.219	0.610
Q6	71.396	189.435	333.341	1942.071	7032.000	53910.502
Q7	0.927	0.357	23.253	2.412	233.247	1.357
Q8	0.012	0.012	0.059	0.035	0.140	0.043
Q9	3.923	7.843	11.026	32.412	47.917	217.308
Q10	68.194	22.128	317.048	118.913	2395.180	1641.020
Q11	0.010	0.006	0.019	0.008	0.116	0.014
Q12	0.022	0.006	1.445	0.005	9.645	0.006
Q13	0.504	0.452	4.261	1.287	20.157	2.616
Q14	0.021	0.020	0.064	0.031	1.383	0.131
Q15	0.037	0.045	1.210	0.306	3.226	0.363
Q16	0.014	0.006	0.010	0.005	0.013	0.007
Q17	5.129	27.393	467.125	242.219	5374.080	1097.145
Total $[s]$	151.248	248.462	1165.688	2342.939	15180.887	56880.901
$[h:m:s]$	0:02:31	0:04:08	0:19:26	0:39:03	4:13:01	15:48:01

Table 7: Benchmarking the SECONDO DBMS: Response Time per Query Instance

Listed are the response times for single query instances (single combination of query parameters) in seconds.

For Query 17, each QueryPoint P is looked up in the spatial index, to find all candidate tuples with trips passing it. The candidates are filtered for those tuples whose trip really passes P. The remaining tuples are projected to P and the according vehicle's licence plate number L. The results are grouped by $\{P\}$. In each group, duplicate licences are removed and the count of the remainder is added. The result is sorted by descending counters. Last, we select all tuples having the same count as the first tuple. The required time increases with the number of observed vehicles and the observation period. In SECONDO, a linear scan through the MOD histories is needed to check whether an *mpoint* passes a *point*.

In the OBA version of Query 7, the spatial index is used to find candidates for a subsequent filtering step. The filter selects only tuples, whose trip passes QueryPoint P. As mentioned before, this requires SECONDO to perform a linear search in each MOD history. Whereas a longer observation time increases the MOD histories in the OBA, this is not the case for the TBA (basically, there will be more trips of the same length in the TBA). This is the reason for the long response time in the OBA at scale factor 1.0.

For Query 9 we observe, that the OBA has an advantage over the TBA version. Why does the TBA's relative disadvantage become even larger with increasing scale factor? In the OBA, we can just restrict all vehicles to the temporal ranges from the parameter PP indicating the temporal range and calculate the travelled distance on the restricted *mpoint* values. Then we just need to group by the PP to find the maximum travelled distance. In the TBA, we use the temporal index to retrieve all trips present at PP, restrict them to the according temporal range and compute the travelled distance for each trip found. Then, we are first required to group by *Licence* to aggregate the distances of all trips belonging to each vehicle, before we can finally group by PP to select the maximum travelled distance. So, in the TBA, we have to group twice, which with increasing scale factor becomes more expensive.

8 Summary and Future Work

We have presented a method to generate a data set representing cars driving in Berlin. Because the only source is a simple map of Berlin, the scenario can also be applied to any place in the world where a map is available, for example as a shape file. The number of observed cars as well as the number of observation days can be changed easily. But also more complex changes can be made by simple changes to the used script, as demonstrated for the region based approach.

First results have been established for the SECONDO DBMS, which can be used as a reference for benchmarking other spatio-temporal database systems.

For the future, we would like to use the proposed BerlinMOD benchmark with other spatio-temporal DBMSs and compare their performances. Also, we plan to compare the performance of different representations of moving object data within the SECONDO system using the benchmark.

9 Acknowledgements

This work was partially supported by grant Gu 293/8-2 from the Deutsche Forschungsgemeinschaft (DFG), project "Datenbanken für bewegte Objekte" (Databases for Moving Objects).

References

[1] BerlinMOD Benchmark Web Site. `http://www.informatik.fernuni-hagen.de/import/pi4/secondo/BerlinMOD/BerlinMOD.html`, 2007.

[2] T. Brinkhoff. A framework for generating network-based moving objects. *GeoInformatica*, 6(2):153–180, 2002.

[3] V. T. de Almeida and R. H. Güting. Indexing the trajectories of moving objects in networks. *Geoinformatica*, 9(1):33–60, 2005.

[4] V. T. de Almeida, R. H. Güting, and C. Düntgen. Multiple entry indexing and double indexing. In *IDEAS 2007*, pages 181–189. IEEE Computer Society, 2007.

[5] S. Dieker and R. H. Güting. Plug and Play with Query Algebras: SECONDO - A Generic DBMS Development Environment. In *Proc. of the International Symposium on Database Engineering & Applications*, pages 380–392, 2000.

[6] E. W. Dijkstra. A note on two problems in connexion with graphs. In *Numerische Mathematik*, volume 1, pages 269–271. Mathematisch Centrum, Amsterdam, The Netherlands, 1959.

[7] Z. Ding and R. H. Güting. Managing moving objects on dynamic transportation networks. In *SSDBM*, pages 287–296. IEEE Computer Society, 2004.

[8] M. Erwig, R. H. Güting, M. Schneider, and M. Vazirgiannis. Spatio-temporal data types: An approach to modeling and querying moving objects in databases. *GeoInformatica*, 3(3):269–296, 1999.

[9] L. Forlizzi, R. H. Güting, E. Nardelli, and M. Schneider. A data model and data structures for moving objects databases. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 319–330. ACM Press, 2000.

[10] F. Giannotti, A. Mazzoni, S. Puntoni, and C. Renso. Synthetic generation of cellular network positioning data. In *GIS '05: Proceedings of the 13th annual ACM international workshop on Geographic information systems*, pages 12–20, New York, NY, USA, 2005. ACM Press.

[11] R. H. Güting. Second-Order Signature: A Tool for Specifying Data Models, Query Processing, and Optimization. In *Proc. of the ACM SIGMOD International Conf. on Management of Data*, pages 277–286, 1993.

[12] R. H. Güting, M. H. Böhlen, M. Erwig, C. S. Jensen, N. A. Lorentzos, M. Schneider, and M. Vazirgiannis. A foundation for representing and quering moving objects. *ACM Trans. Database Syst.*, 25(1):1–42, 2000.

[13] R. H. Güting, V. T. de Almeida, D. Ansorge, T. Behr, Z. Ding, T. Höse, F. Hoffmann, M. Spiekermann, and U. Telle. Secondo: An extensible DBMS platform for research prototyping and teaching. In *ICDE*, pages 1115–1116, 2005.

[14] R. H. Güting, V. T. de Almeida, and Z. Ding. Modeling and querying moving objects in networks. *The VLDB Journal*, 15(2):165–190, 2006.

[15] M. F. Mokbel, T. M. Ghanem, and W. G. Aref. Spatio-temporal access methods. *IEEE Data Eng. Bull.*, 26(2):40–49, 2003.

[16] D. Pfoser and Y. Theodoridis. Generating semantics-based trajectories of moving objects. *Computers, Environment and Urban Systems*, 27(3):243–263, 2003.

[17] S. Rezic. `http://bbbike.de`, 2007.

[18] J.-M. Saglio and J. Moreira. Oporto: A Realistic Scenario Generator for Moving Objects. *Geoinformatica*, 5(1):71–93, 2001.

[19] Secondo Web Site. `http://www.informatik.fernuni-hagen.de/secondo`, 2007.

[20] A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Querying the uncertain position of moving objects. *Lecture Notes in Computer Science*, 1399:310–337, 1998.

[21] M. Stonebraker, J. Frew, K. Gardels, and J. Meredith. The Sequoia 2000 Benchmark. In P. Buneman and S. Jajodia, editors, *SIGMOD Conference*, pages 2–11. ACM Press, 1993.

[22] Y. Theodoridis, J. R. O. Silva, and M. A. Nascimento. On the generation of spatiotemporal datasets. In R. H. Güting, D. Papadias, and F. H. Lochovsky, editors, *SSD*, volume 1651 of *Lecture Notes in Computer Science*, pages 147–164. Springer, 1999.

[23] M. Vazirgiannis and O. Wolfson. A spatiotemporal model and language for moving objects on road networks. In *SSTD '01: Proceedings of the 7th International Symposium on Advances in Spatial and Temporal Databases*, pages 20–35, London, UK, 2001. Springer-Verlag.

[24] P. Werstein. A performance benchmark for spatiotemporal databases. In *Tenth Annual Colloquium of the Spatial Information Research Centre, 16 - 19 Dec, Dunedin, New Zealand*, pages 1365–1374. University of Otago, 1998.

[25] O. Wolfson, S. Chamberlain, S. Dao, L. Jiang, and G. Mendez. Cost and imprecision in modeling the position of moving objects. In *ICDE*, pages 588–596, 1998.

[26] O. Wolfson, A. P. Sistla, S. Chamberlain, and Y. Yesha. Updating and querying databases that track mobile units. *Distributed and Parallel Databases*, 7(3):257–387, 1999.

[27] O. Wolfson, B. Xu, S. Chamberlain, and L. Jiang. Moving objects databases: Issues and solutions. In *Statistical and Scientific Database Management*, pages 111–122, 1998.

A The BerlinMOD Data Generator Script

This script is intended to be used with the Secondo DBMS. It will create the benchmark data used by BerlinMOD.

```
1  #==========================================================
2  #===      Section 6.1: Map Import and Parameters        ===
3  #==========================================================
4  # (6.1.1) Create database and restore street data
5  create database berlingraph;
6  open database berlingraph;
7  restore streets from STREETS;
8
9  # (6.1.2) Setting up Parameters
10 #========== Scaling Parameter ============================
11 let SCALEFACTOR = 1.0;
12
13 #========= Secondary Parameters ===========================
14 # distribute the scalefactor
15 let SCALEFCARS = sqrt(SCALEFACTOR);
16 let SCALEFDAYS = sqrt(SCALEFACTOR);
```

```
17
18 # The day the observation starts
19 let STARTDAY = 2702;
20
21 # The number of vehicles to observe
22 let NUMCARS = real2int(round((2000 * SCALEFCARS),0));
23
24 # The number of observation days
25 let NUMDAYS = real2int(round((SCALEFDAYS*28),0));
26
27 # The minimum length of a pause in milliseconds
28 # to distinguish subsequent trips
29 let MINPAUSE = create_duration(0,300000);
30
31 #The velocity below which a vehicle is considered to be static
32 let MINVELOCITY = 1/24;
33
34 # The duration between two subsequent GPS-observations
35 let GPSINTERVAL = create_duration(0, 2000);
36
37 # The random seeds used
38 let HOMERANDSEED = 0;
39 let TRIPRANDSEED = 4277;
40
41 # The size for sample relations
42 let SAMPLESIZE = 100;
43
44 #=================================================
45 #===     Section 6.2: Creating the Street Graph     ===
46 #=================================================
47
48 # (6.2.1) Removing Impassable streets:
49 let streets1 =
50   streets feed filter[ not (.Typ contains "gesperrt")]
51   extend[ Vmax :  ifthenelse( .Typ contains "Wichtig" , 70.0,
52        ifthenelse( .Typ contains "Haupt" , 50.0,
53          ifthenelse( .Typ contains "Neben",
54            30.0 ,-1.0)))]
55   consume;
56
57 # (6.2.2) Aggregate all street lines into a single line
58 # object and extract the largest component:
59 let allstreets = components(streets1 feed
60   aggregateB[geoData; fun(L1: line, L2: line)
61     union_new(L1,L2); [const line value ()]])
62   transformstream extend[NoSeg: no_segments(.elem)]
63   sortby[NoSeg desc] extract[elem];
64
65 # (6.2.3) collect all endpoints  into a single points
66 # object:
67 let Endpoints = streets1 feed
68   projectextend[; B : boundary(.geoData)]
69     aggregateB[B; fun(P3 : points, P4 : points)
70       P3 union P4; [const points value ()]];
71
72 # (6.2.4) Split the latter line object into polylines:
73 let sections2 = allstreets
74   polylines[FALSE, Endpoints]
75   namedtransformstream[Part] consume;
76
77 # (6.2.5) Calculate the positions of the nodes of the
```

```
78 # graph as a single points object
79 let nodes2 = sections2 feed
80   projectextend[; EndPoints: boundary(.Part)]
81   aggregateB[EndPoints;
82     fun(P1: points, P2: points) P1 union P2
83     ; [const points value ()]];
84
85 # (6.2.6) Create a relation with all nodes
86 let Nodes = components(nodes2)
87   namedtransformstream[Pos]
88   addcounter[NodeId,1] consume;
89
90 # (6.2.7) Join sections, nodes, and Vmax
91 let Sections =
92   sections2 feed addcounter[SecId, 1]
93   extendstream[EP: components(boundary(.Part))]
94   Nodes feed
95   hashjoin[EP, Pos, 99997]
96   sortby[SecId]
97   groupby[ SecId; Part: group feed
98     extract[Part],
99     NodeId_s1: group feed extract[NodeId],
100    NodeId_s2: group feed addcounter[Cnt,0]
101      filter[.Cnt = 1] extract[NodeId]]
102  remove[SecId]
103  streets1 feed spatialjoin[Part,geoData]
104  filter[ not(isempty(
105    intersection_new(.Part,.geoData)))]
106  project[Part, NodeId_s1, NodeId_s2, Vmax]
107  consume;
108
109
110 # (6.2.8) We encode sections by
111 # (SourceNodeId * 10000 + TargetNodeId)
112 let SectionsUndir =
113   Sections feed extendstream[B :intstream (0,1)]
114   projectextend[ Vmax , Part
115     ; NodeId_s1 : ifthenelse(.B=0, .NodeId_s1, .NodeId_s2),
116       NodeId_s2 : ifthenelse(.B=0, .NodeId_s2, .NodeId_s1)]
117   sortby[NodeId_s1, NodeId_s2] krdup[NodeId_s1, NodeId_s2]
118   extend[Key: (.NodeId_s1 * 10000) + .NodeId_s2]
119   consume;
120
121 derive SectionsUndir_Key =
122   SectionsUndir createbtree[Key];
123
124 # (6.2.9) Create the Graph with drivetime distances
125 let berlingraphtime =
126   Nodes feed {n2}
127   ( Nodes feed {n1}
128     Sections feed
129     projectextend[NodeId_s1, NodeId_s2
130       ; Costs : (3.6 * size(.Part)) / .Vmax]
131     hashjoin[NodeId_n1, NodeId_s1, 9999]
132     project[NodeId_s1, NodeId_s2, Costs, Pos_n1] )
133   hashjoin[NodeId_n2, NodeId_s2, 9999]
134   project[NodeId_s1, NodeId_s2, Pos_n1, Pos_n2, Costs]
135   extendstream[D : intstream(0,1) ]
136   projectextend[Costs
137     ; NodeId1 : ifthenelse(.D=0 , .NodeId_s1, .NodeId_s2),
138       NodeId2 : ifthenelse(.D=0 , .NodeId_s2, .NodeId_s1),
```

```
139         Pos1 : ifthenelse(.D=0 , .Pos_n1, .Pos_n2),
140         Pos2 : ifthenelse(.D=0 , .Pos_n2, .Pos_n1)]
141     constgraphpoints[NodeId1, NodeId2, .Costs, Pos1, Pos2];
142
143
144 #=============================================================
145 #===      Section 6.3: Creating the LabourPaths            ===
146 #=============================================================
147
148 # (6.3.1) A relation with all vehicles, their HomeNode,
149 # WorkNode and Number of Neighbourhood nodes
150 # The second relation contains all neighours
151 # for a vehicle:
152 query rng_init(14, HOMERANDSEED);
153 let vehicle1 =
154     intstream(1,NUMCARS) namedtransformstream[Id]
155     extend[HomeNode: rng_intN(Nodes count) + 1,
156       WorkNode: rng_intN((Nodes count)) + 1]
157     consume;
158
159 # (6.3.2) encoding for index:
160 # Key is (VehicleId * 100000) + NeighbourId
161 let neighbourhood =
162     vehicle1 feed filter[seqinit(1)]
163     projectextend[ ; Vehicle: .Id,
164      HomePos: pos(thevertex(berlingraphtime, .HomeNode))]
165     extendstream[
166       Vertex: (vertices(berlingraphtime) transformstream) ]
167     filter[distance(pos(.Vertex), .HomePos) <= 3000.0]
168     projectextend[Vehicle ; Node: key(.Vertex), Id: seqnext()]
169     extend[Key: (.Vehicle * 100000) + .Id]
170     consume;
171
172 derive neighbourhood_Key =
173     neighbourhood createbtree[Key];
174
175 # (6.3.3)
176 let vehicle =
177     neighbourhood feed groupby[Vehicle
178       ; NoNeighbours: group count] {nbr}
179     vehicle1 feed
180     mergejoin[Vehicle_nbr, Id]
181     projectextend[Id, HomeNode, WorkNode
182       ; NoNeighbours: .NoNeighbours_nbr]
183     consume;
184
185 # (6.3.4) A relation containing the paths for the labour trips
186 let labourPath =
187     vehicle feed projectextend[ Id
188       ; ToWork: shortestpath(berlingraphtime,
189           .HomeNode, .WorkNode),
190         ToHome: shortestpath(berlingraphtime,
191           .WorkNode, .HomeNode)]
192     consume;
193
194 #=============================================================
195 #===      Section 6.4: Creating the Moving Points          ===
196 #=============================================================
197
198 # (6.4.1) Function BoundedGaussian
199 # Computes a gaussian distributed value
```

```
# within [Low, High]
let BoundedGaussian = fun(Sigma: real,
  Low: real, High: real)
  rng_gaussian(Sigma) within[
    ifthenelse(. < Low,Low,ifthenelse(. > High, High, .))];

# (6.4.2) Function Path2Mpoint:
# Creates a trip as a mpoint following path P
# and starting at instant Tstart.
let Path2Mpoint = fun(P: path, Tstart: instant)
  (samplempoint(
   (P feed namedtransformstream[Path]
    filter[seqinit(1)] extend[Dummy: 1]
    projectextendstream[ Dummy
     ; Edge : edges(.Path) transformstream ]
    projectextend[ ; Source : source(.Edge) ,
      Target : target(.Edge), SeqNo: seqnext()]
    loopjoin[SectionsUndir_Key SectionsUndir
      exactmatch[(.Source * 10000) + .Target] ]
    projectextend[ SeqNo, Vmax ; Line: .Part ]
    sortby[SeqNo asc]
    sim_create_trip[ Line, Vmax, Tstart,
      pos(vertices(P) extract[Vertex]), 0.0 ]),
   GPSINTERVAL, TRUE, TRUE ));

# (6.4.3) Function SelectDestNode
# Selects a destination node for an additional
# trip. 80% of the destinations are from the
# neighbourhood 20% are from the complete graph
let SelectDestNode = fun(VehicleId: int,
  NoNeighbours: int)
    ifthenelse( (rng_intN(100) < 80 ),
     ( neighbourhood_Key neighbourhood
       exactmatch[(VehicleId*100000) +
         (rng_intN( NoNeighbours ) + 1)]
       extract[Node] ),
     ( rng_intN(Nodes count) + 1 ) );

# (6.4.4) Function CreatePause
# Creates a random duration of length [0ms, 2h]
let CreatePause = fun( )
  create_duration(0,real2int(((BoundedGaussian(1.4, -6.0, 6.0)
      * 100.0) + 600.0) * 6000.0));

# (6.4.5) Function CreatePauseN
# Creates a random non-zero duration of length
# [2ms, N min - 4ms] using flat distribution
let CreatePauseN = fun(Minutes: int)
  create_duration(0, ( 2 + rng_intN(((Minutes
    + 1) * 60000) - 4) ) );

# (6.4.6) Function CreateDurationRhoursNormal
# Creates a normally distributed duration
# within [-Rhours h, +Rhours h]
let CreateDurationRhoursNormal =
  fun(Rhours: real)
    create_duration((rng_gaussian(1.0) * Rhours
     * 1800000)/86400000)
      within[ ifthenelse( duration2real(.)
        between[(Rhours / -24.0), (Rhours / 24.0)],
        . , ifthenelse( duration2real(.) > (Rhours / 24.0),
```

```
261           create_duration(Rhours / 24.0),
262           create_duration(Rhours / -24.0)) )];
263
264 # (6.4.7) Function CreateAdditionalTrip
265 # Creates an 'additional trip' for a vehicle,
266 # starting at a given time
267 let CreateAdditionalTrip =
268   fun(Veh: int, Home: int, NoNeigh: int,
269     Ttotal: duration, Tbegin: instant)
270     ( ( ifthenelse( rng_intN(100) < 80, 1,
271         ifthenelse( rng_intN(100) < 50, 2, 3) )
272       feed namedtransformstream[NoDests]
273       extend[ S0: Home,
274         S1: ifthenelse( .NoDests >=1 ,
275           SelectDestNode(Veh, NoNeigh), Home ),
276         S2: ifthenelse( .NoDests >=2 ,
277           SelectDestNode(Veh, NoNeigh), Home ),
278         S3: ifthenelse( .NoDests >=3 ,
279           SelectDestNode(Veh, NoNeigh), Home )]
280       extend[ D0: .S1, D1: .S2, D2: .S3, D3: Home ]
281       projectextend[ NoDests
282         ; Trip0: ifthenelse( .S0 # .D0,
283             Path2Mpoint(shortestpath(
284               berlingraphtime, .S0, .D0), Tbegin),
285             [const mpoint value ()] ),
286             Trip1: ifthenelse( .S1 # .D1,
287               Path2Mpoint(shortestpath(
288                 berlingraphtime, .S1, .D1), Tbegin),
289               [const mpoint value ()]),
290             Trip2: ifthenelse( .S2 # .D2,
291               Path2Mpoint(shortestpath(
292                 berlingraphtime, .S2, .D2), Tbegin),
293              [const mpoint value ()]),
294             Trip3: ifthenelse( .S3 # .D3,
295               Path2Mpoint(shortestpath(
296                 berlingraphtime, .S3, .D3), Tbegin),
297               [const mpoint value () ]) ]
298       extend[ Pause0: CreatePause(),
299         Pause1: CreatePause(), Pause2: CreatePause()]
300       extend[ Result: ifthenelse( .NoDests < 2,
301        .Trip0 translateappend[.Trip1, .Pause0],
302        ifthenelse( .NoDests < 3,
303          (.Trip0 translateappend[.Trip1, .Pause0])
304          translateappend[.Trip2, .Pause1],
305          ((.Trip0 translateappend[.Trip1, .Pause0])
306          translateappend[.Trip2, .Pause1])
307          translateappend[.Trip3, .Pause2]) ) ]
308       extract[Result] ));
309
310 # (6.4.8) Function CreateDay
311 # Creates a certain vehicle's movement
312 # for a specified day
313 let CreateDay = fun(VehicleId: int, DayNo: int,
314   PathWork: path, PathHome: path,
315   HomeIdent: int, NoNeighb: int)
316   ifthenelse(
317     ( (weekday_of(create_instant(DayNo, 0)) = "Sunday") or
318       (weekday_of(create_instant(DayNo, 0)) = "Saturday")),
319     ( ifthenelse(rng_intN(100) < 40,
320         CreateAdditionalTrip(VehicleId, HomeIdent, NoNeighb,
321           [const duration value (0 18000000)],
```

```
322           create_instant(DayNo,32400000) + CreatePauseN(120)),
323           [const mpoint value () ] )
324       ifthenelse(rng_intN(100) < 40,
325         CreateAdditionalTrip(VehicleId, HomeIdent, NoNeighb,
326          [const duration value (0 18000000)],
327         create_instant(DayNo, 68400000) + CreatePauseN(120)),
328          [const mpoint value () ] ) concat  ),
329     ( ( Path2Mpoint(PathWork,
330           create_instant(DayNo, 28800000)
331             + CreateDurationRhoursNormal(2.0))
332         Path2Mpoint(PathHome,
333           create_instant(DayNo, 57600000)
334             + CreateDurationRhoursNormal(2.0))
335         concat )
336       ifthenelse(rng_intN(100) < 40,
337         CreateAdditionalTrip(VehicleId, HomeIdent, NoNeighb,
338           [const duration value (0 14400000)],
339           create_instant(DayNo, 72000000) + CreatePauseN(90)),
340         [const mpoint value () ]  ) concat  ) )
341   within[ifthenelse(hour_of(inst(final(.))) between [6,8],
342     [const mpoint value ()], . )];
343
344 # (6.4.9) Function RandModel(): Return a random
345 # vehicle model string:
346 let ModelArray = makearray("Mercedes-Benz", "Volkswagen",
347   "Maybach", "Porsche", "Opel", "BMW", "Audi", "Acabion",
348   "Borgward", "Wartburg", "Sachsenring", "Multicar");
349 let RandModel = fun()
350   get(ModelArray, rng_intN(size(ModelArray)));
351
352 # (6.4.10) Function RandType(): Return a random vehicle
353 # type (0 = passenger, 1 = bus, 2 = truck):
354 let TypeArray = makearray("passenger", "bus", "truck");
355 let RandType = fun()
356   get(TypeArray, ifthenelse( ( rng_intN(100) < 90 ), 0,
357     ( ( ifthenelse( ( rng_intN(100) < 50 ) , 1 , 2) ))));
358
359 # (6.4.11) Function Licence(): Return the unique licence
360 #  string for a given vehicle-Id 'No':
361 #  for 'No' in [0,26999]
362 let LicenceFun = fun(No: int)
363   ifthenelse( (No > 0) and (No < 1000),
364     "B-" + char(rng_intN(26) + 65) + char(rng_intN(25) + 65)
365       + " " + num2string(No),
366     ( ifthenelse( (No mod 1000) = 0,
367         "B-" + char((No div 1000) + 65) + " "
368           + num2string(rng_intN(998) + 1),
369         "B-" + char((No div 1000) + 64) + "Z "
370           + num2string(No mod 1000))));
371
372 # (6.4.12) Function CreateVehicles():
373 #   Create data for 'NumVehicle' vehicles and
374 #    'NumDays' days starting at 'StartDay'
375 let CreateVehicles = fun(NumVehicleP: int,
376   NumDaysP: int, StartDayP: int)
377   vehicle feed head[NumVehicleP] {v}
378   labourPath feed {p}
379   symmjoin[.Id_v = ..Id_p]
380   intstream(StartDayP, (StartDayP + NumDaysP) - 1)
381   transformstream
382   product
```

```
  projectextend[; Id: .Id_v, Day: .elem,
    TripOfDay: CreateDay(.Id_v, .elem, .ToWork_p, .ToHome_p,
      .HomeNode_v, .NoNeighbours_v)]
  sortby[Id, Day]
  groupby[Id; Trip:
    (group feed projecttransformstream[TripOfDay] concatS)
  sim_fillup_mpoint[ create_instant(StartDayP - 1,0),
      create_instant((StartDayP + NumDaysP) + 1,0),
    TRUE, FALSE, FALSE] ]
  extend[ Licence: LicenceFun(.Id),
    Type: RandType(), Model: RandModel() ]
  consume;

# (6.4.13) Create Movement data for NUMCARS vehicles and
#  NUMDAYS days
query sim_set_rng( 14, TRIPRANDSEED );

# long time observation of objects
let dataScar = CreateVehicles(NUMCARS, NUMDAYS, STARTDAY);

#=============================================================
#===  Section 6.5: Deriving the Trip Based Representation ===
#=============================================================

# observation of single trips
let dataMcar =
  dataScar feed
  projectextendstream[Id, Licence, Type,
    Model; Subtrip: .Trip sim_trips[MINPAUSE, MINVELOCITY]]
  addcounter[TripId, 1] remove[Subtrip] consume;

let dataMtrip =
  dataScar feed
  projectextendstream[Licence
    ; SubTrip: .Trips[MINPAUSE, MINVELOCITY]]
  addcounter[TripId, 1] remove[Licence] consume;

#=============================================================
#===          Section 6.6: Creating Query Relations        ===
#=============================================================

# SAMPLESIZE random node positions
let QueryPoints =
  intstream(1, SAMPLESIZE) namedtransformstream[Id]
  extend[Pos: get(nodes2, rng_intN(no_components(nodes2)) )]
  consume;
let QueryPoints1 = QueryPoints feed head[10] consume;
let QueryPoints2 = QueryPoints feed filter[.Id > 10]
  head[10] consume;

# SAMPLESIZE random regions
let QueryRegions =
  intstream(1, SAMPLESIZE) namedtransformstream[Id]
  extend[Region: circle(get(nodes2, rng_intN(no_components(
    nodes2))), rng_intN(997) + 3.0, rng_intN(99) + 1)]
  consume;
let QueryRegions1 = QueryRegions feed head[10] consume;
let QueryRegions2 = QueryRegions feed filter[.Id > 10]
  head[10] consume;

# SAMPLESIZE random instants
```

```
444 let QueryInstants =
445   intstream(1, SAMPLESIZE) namedtransformstream[Id]
446   extend[Instant: create_instant(STARTDAY, 0)
447     + create_duration(rng_real() * NUMDAYS )]
448   consume;
449 let QueryInstants1 = QueryInstants feed head[10] consume;
450 let QueryInstants2 = QueryInstants feed filter[.Id > 10]
451   head[10] consume;
452
453 # SAMPLESIZE random periods
454 let QueryPeriods =
455   intstream(1, SAMPLESIZE) namedtransformstream[Id]
456   extend[
457     StartInstant: create_instant(STARTDAY, 0)
458      + create_duration(rng_real() * NUMDAYS ),
459     Duration: create_duration(abs(rng_gaussian(1.0)))
460   ]
461   projectextend[Id; Period: theRange( .StartInstant,
462     .StartInstant + .Duration, TRUE, TRUE)]
463   consume;
464 let QueryPeriods1 = QueryPeriods feed head[10] consume;
465 let QueryPeriods2 = QueryPeriods feed filter[.Id > 10]
466   head[10] consume;
467
468 # SAMPLESIZE random Licences
469 let LicenceList =
470   dataScar feed project[Licence] addcounter[Id,1] consume;
471 let LicenceList_Id = LicenceList createbtree[Id];
472 let QueryLicences =
473   intstream(1, SAMPLESIZE) namedtransformstream[Id1]
474   loopjoin[ LicenceList_Id LicenceList
475     exactmatch[rng_intN(NUMCARS) + 1] ]
476   project[Id, Licence] consume;
477 let QueryLicences1 = QueryLicences feed head[10] consume;
478 let QueryLicences2 = QueryLicences feed filter[.Id > 10]
479   head[10] consume;
```

B Alternative Selection of Nodes

```
1
2 let regions1 = makearray(
3   circle(makepoint(9450.0,11700.0),5000.0,50),
4   rectangle2(-4500,4000,950,8000) rect2region,
5   rectangle2(-2300,7500,7700,16000)rect2region,
6   circle(makepoint(7600, 5000), 5000.0, 30),
7   rectangle2(8700,30000,-3000,21000)rect2region,
8   rectangle2(-20000,36000,
9              -9000,32000) rect2region);
10
11 let partition1 =
12   intstream(0, size(regions1) - 1)
13   namedtransformstream[i] extend [
14     reg : (intersection(get(regions1,.i),
15       nodes2)) minus ( intstream(0,.i - 1)
16         namedtransformstream[m]
17         extend[tmpp : intersection(get(
18           regions1,.m),nodes2)]
19         aggregate[tmpp
20           ;fun(p1: points, p2 : points )
```

```
21            p1 union p2
22           ; [const points value ()] ] ) ]
23   distribute [i] loop [ . feed extract[reg]]
24
25 let Nodes_pos = Nodes feed addid sortby[Pos]
26                 bulkloadrtree[Pos]
27
28 let selectHomePos = fun ()
29   get( partition1 ,
30        rng_intN(100) feed transformstream
31        projectextend[ ; ind:
32          ifthenelse( .elem < 40, 0,
33          ifthenelse( .elem < 60, 1,
34          ifthenelse( .elem < 75, 2,
35          ifthenelse( .elem < 85, 3,
36          ifthenelse( .elem < 93, 4,5)))))]
37        extract[ind] )
38   feed namedtransformstream[AllPos]
39   extend[ Pos : get(.AllPos,
40     rng_intN(no_components(.AllPos)- 1))]
41   extract[Pos]
42
43 let selectHomeNode = fun ()
44   selectHomePos() feed
45   namedtransformstream[P1]
46   extendstream[ Id: Nodes_pos Nodes
47      windowintersects[.P1]
48      projecttransformstream[NodeId] ]
49   extract[Id]
```

Verzeichnis der zuletzt erschienenen Informatik-Berichte

[324] Roth, J.: 2. GI/ITG KuVS Fachgespräch Ortsbezogene Anwendungen und Dienste

[325] Fernandez, A.: Groupware for Collaborative Tailoring

[326] Grubba, T., Hertling, P., Tsuiki, H., Weihrauch, K.: CCA 2005 - Second International Conference on Computability and Complexity in Analysis

[327] Heutelbeck, D**.:** Distributed Space Partitioning Trees and their Application in Mobile Computing

[328] Widera, M., Messing, B., Kern-Isberner, G., Isberner, M., Beierle, C.: Ein erweiterbares System für die Spezifikation und Generierung interaktiver Selbsttestaufgaben

[329] Fechner, B.: A Fault-Tolerant Dynamic Multithreaded Microprocessor

[330] Keller, J., Schneeweiss, W.: Computing Closed Solutions of Linear Recursions with Applications in Reliability Modelling

[331] Keller, J.: Efficient Sampling of the Structure of Cryptographic Generators' State Transition Graphs

[332] Fisseler, J., Kern-Isberner, G., Koch, A., Müller, Chr., Beierle, Chr..: CondorCKD – Implementing an Algebraic Knowledge Discovery System in a Functional Programming Language

[333] Cenzer, D., Dillhage, R., Grubba, T., Weihrauch, K..: CCA 2006 - Third International Conference on Computability and Complexity in Analysis

[334] Fechner, B., Keller, J.: Enhancement and Analysis of a Simple and Efficient VLSI Model

[335] Wilkes, W., Ondracek, N., Oancea, M., Seiceanu, M.: Web services to resolve concept identifiers supporting effective product data exchange

[336] Kunze, C., Lemnitzer,L., Osswald, R. (eds.): GLDV-2007 Workshop - Lexical-Semantic and Ontological Resources

[337] Scheben, U.: Simplifying and Unifying Composition for Industrial Models

[338] Dillhage, R., Grubba, T., Sorbi, A., Weihrauch, K., Zhong, N.: CCA 2007 – Fourth International Conference on Computability and Complexity in Analysis

[339] Beierle, Chr., Kern-Isberner, G. (Eds.): Dynamics of Knowledge and Belief - Workshop at the 30th Annual German Conference on Artificial Intelligence, KI-2007