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

June 18, 2008

Abstract

This document presents a method to design scalable and representative moving object data (MOD) and two sets 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 DBMSs. 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, a field is mature only when 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 two sets 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 executable commands collected in a *script*. To our knowledge, this is the first time, a DBMS is used in this way. While this is quite unusual, it demonstrates the capabilities of a modern database system, whose modular extensions and algebraic approach allows for synergetic effects and surprising applications. Also, this arrangement provides the user with the possibility to directly compare her own system with another one.

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 [40, 4]. Hence anyone can repeat the experiments reported here. Customized map data can be imported and the generated benchmark data can be exported to various file formats for later conversion to the input formats of other systems.

2 Related Work

In this section, we present results related to our contribution. After starting with some notes on general MOD traits we present some works on data generators and existing benchmarks. Last, we give an overview on interesting query types proposed.

2.1 Moving Object Databases

Moving objects are time dependent geometries, i.e. geometries described as a function of time. In contrast to earlier work on spatio-temporal databases, which in fact described temporal databases for spatial objects containing "snapshots" of the spatial data at certain time stamps, 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 [42, 57, 59, 58].

A second approach represents complete histories of moving objects, for moving point objects also called *trajectory databases*. This approach was pursued, for example, in [13, 19, 15]. 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 [31]. 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 [55, 12, 21], especially with regard to real world applications. As an example, in [8] 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 [19] 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 [21]. This implementation is underway in SECONDO.

2.2 Data Generators

One of the first generators for moving object data, *GSTD* [52], creates unconstrained moving point or rectangle data. A refinement [36] 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 [16]. 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.

G-TERD (generator of time-evolving regional data) [53] is a framework for the generation of large sets of continuously changing 2D region object data. Regions have a location, shape, size, and color as changing attributes. Parameters as for maximum speed, rotation-angle, object interactions, the initial position and movement of the scene-observer, and the statistical distributions used can be defined by the user. Output format are multicolored raster images. This generator produces well-suited data for tasks as tracking objects. However, as the motion of objects is unconstrained and fully stochastically, it is improper for the simulation of vehicles on a road network.

Another generator for spatio-temporal data, called *Oporto* [39], 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.

Many simulators have been proposed to generate traffic data. While *macroscopic simulations* deal with the traffic flow as such, *microscopic simulations* deal with single vehicles. The *SMARTEST* project (Simulation Modelling Applied to Road Transport European Scheme Tests) [44, 43] has provided a broad overview on existing microscopic traffic simulations and developed a best practice manual of modelling micro-simulation tools. The survey covers different tools modelled with regard to different aspects of traffic, like urban, freeway, mixed, or automated highway systems; some are even more specialized, focussing on aspects like car parking or toll collection. The final project report subsumes [43]: "Most of the micro-simulation models studied have been developed to quantify the benefits of Intelligent Transportation Systems, primarily Advanced Traffic Management Systems and Advanced Traveller Information Systems. The scale of application ranges from a small number of vehicles and intersections to a large number, about 200 nodes and many thousands of vehicles. Huge networks (300+ nodes and 1 million+ vehicles) can be considered by models that run on parallel architectures. The models are usually used to estimate traffic efficiency in terms of speed and travel time, sometimes also considering congestion and queue length. They mainly concentrate on simulating traffic signal control, route guidance and traffic condition estimation. Each model uses its own control strategies and algorithms. Motorcycles, bicycles, pedestrians, public transport, weather conditions and on-street parking receive little attention. [...] A few models have a GUI to input the road network topology and other data. Most models have only been partially validated and calibrated."

All these generators primarily aim at traffic planning applications, and though observing individual vehicles, they do so to create data on the general traffic flow. They do not consider directions of travel, like generating starting locations and destinations for trips, or long time observations of single moving objects. Also, many of these systems are non-free, requiring the payment of licence fees, and prohibiting researchers from reading or changing the implementation to their needs.

An open-source generator and visual interface for objects moving in networks is proposed by Brinkhoff in [6]. 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. The generator aims at the generation of single representative trips, not at the generation of longterm movement histories for single objects. Because objects are created at random (following a creation rate function) and terminate as soon as they reach their destination, long-term observations of objects are not realizable without major changes to the generator mechanisms. 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 [6] 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.

Another more recent example for a microscopic traffic simulator implementation is *SUMO* (Simulation of Urban MObility) [28], an open-source project performing a time-discrete, space-continuous simulation of traffic, capturing private and public transport. The simulator performs collision free vehicle movement with a step duration of 1s, multi-lane streets, junction-based right-of-way rules, and lane-to-lane connections. The generated data can be logged in a XML raw-format with each entry providing a vehicle's name, position and speed at a given time step; alternatively, the readings of induction loop type sensors located on arbitrary network positions can be generated by the simulator.

As with other microscopic traffic simulators, using SUMO creates large amounts of accurate moving object data, but this still requires the user to carefully specify trip data (starting and ending locations) or complete route data for a large amount of vehicles. As the vehicles' behaviour relies on their proper interaction, they will only behave representatively, if the whole traffic is shaped in an appropriate (i.e. realistic) way. Though SUMO provides several tools to ease this task (like loop detector based vehicle emission and trips), we consider this task too demanding for the sake of benchmark data generation.

Traffic simulators aim at generating traffic data for sets of vehicles on a network, where the vehicles starting instants/positions, and destinations are given. While these parameter sets are usually generated using some statistic distributions, the framework *ST-ACTS* presented in [17] tries to provide representative *sequences of activities* for a collection of individuals (called *simpersons* there). Commercial datasets on danish demographic data, daily movements, business and consumer survey data are combined into a model to generate home locations, working places/schools, but also a "personality" and "interests" profile of such simpersons. Activities covered are working/learning, staying at home, visiting cultural and leisure facilities (attending concerts, visiting museums), shopping, etc. Activities are performed at according facilities. Several sets of constraints determine which facilities are frequented and the duration of the activity. The route between the locations is not considered, only the estimated time it takes to reach a location is taken into account. Instead, the authors suggest to couple their framework with a traffic simulator to achieve more realistic travel durations or itineraries. The simulator is implemented using the proprietary MATLAB software.

2.3 Benchmarks

For spatial DBMS (SDBMS), the most prominent benchmark is the *Sequoia 2000 storage benchmark* [49]. 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 [56] 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 ("Select 3D matrix data for a given time and 3D region and calculate an arithmetic function for each cell in the matrix.") is only intended to perform a local weighted average operation on raster data. 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.

In [51], Theodoridis proposes a database schema and a set of ten benchmark database queries (plus two operations for loading and updating data) regarding the support of location-based services (LBS). No benchmark data or data generator is proposed, so that the comparability of different benchmark results cannot be ensured. The benchmark includes point and range selection queries on stationary and moving reference objects; distance-, k-NN-, and similarity-based queries; join queries, and unary operations on trajectories of moving objects. The scenario involves people trying to meet their interests by visiting shops, which have certain offers at specific time intervals. The author stresses, that by the vast range of applications for spatio-temporal DBMS, their support by commercial DBMS becomes impractical. Therefore, instead of covering the "full umbrella" of possible query types, he argues to focus on specific applications with similar modeling and functionality. As a consequence, the author does not handle *future queries* (restricting himself to past and current queries), argueing they would not meet the scenario. We subscribe to this point of view and restrict our concerns to historic trajectory databases, leaving current queries and future queries out of the scope of our benchmark.

Also *DynaMark* [34] is a benchmark designed to provide performance metrics for LBS. Performance metrics are defined to quantify the cost of location updates, spatial queries, and spatial index creation and maintenance. The benchmark consists of synthetic datasets and a set of spatial queries. The data is generated using *City Simulator*, which is no longer available from the mentioned IBM developer Web site. City Simulator is said to generate three-dimensional spatio-temporal datasets modelling the movement of people (the authors propose 10,000 to 1 million indiviuals) in a virtual city consisting of multi-lane streets and multi-floor buildings with elevators, also considering traffic jams and other non-stochastic interaction between moving objects. Besides the update benchmark (updating the spatial index to all objects' current positions) and the index benchmark (creating and rebuilding the spatial index from City Simulator's trace data), DynaMark uses four types of queries in its query benchmarks (updating and querying): proximity queries (= range queries) with rectangular and radial ranges, k-NN queries, and sorted distance queries. It does not incorporate predictive queries, spatial data mining, or spatio-temporal queries on historical data, focussing on current queries und updates.

In [54], the authors define a benchmark to perform a comparison of access methods for different time-evolving regional raster image data. The benchmark uses several scenarios of moving region data sets created with G-TERD [53], a generator designed for time-evolving regional data. Several parameters, as storage requirements, index construction costs, and page reads required to perform queries are observed. Only two types of queries are used: snapshot queries retrieving all region data alive at a given timestamp, and time-interval window queries. While this is a valuable framework for the evaluation of indexes, it does not use representative moving point data, and covers only a restricted number of query types.

The same holds for the COST benchmark [25]. It aims at the evaluation and comparison of spatio-temporal indexes. Three types of queries are be distinguished: timeslice queries (objects within a spatial window at a given time instant), window queries (objects within a spatial window during a defined time interval), and moving window queries (objects within a spatial window, that changes linearly during the defined time interval); future queries are included. The benchmark has many parameters regarding data generation (like number of objects, space dimensions, maximum speed, average number of destinations between which objects are moving), queries (maximum spatial/temporal extent, weighting of query types, maximum duration of time that queries may reach into the future), and the workload mix (number of update/query events, update/query frequency, weighting of updates/queries). Queries and updates are intermixed by the workload generator. Object movement can be either random or network-based (i.e.

random walks on a random network). Eight experiments (i.e. parameter settings) are defined with different objectives regarding aspects of index performance. These experiments are applied to TPR-, TPR*-, and B^x-trees.

2.4 Query Types for Spatio-temporal DBMS

Many different kinds of queries have been proposed for spatio-temporal databases. Beside temporal, spatial and spatio-temporal point and range queries, the following types have some impact:

k-nearest neighbour based (k-NN) queries [38] have been proposed for static high dimensional data [29], and for moving query points [45]. More recently, reversed nearest neighbour (r-NN) queries for (non-moving) spatial [27] and high-dimensional data [41, 50], for moving query and moving data points [3] have been addressed. For two sets of points P, Q, aggregate nearest neighbour based (a-NN) queries ask for those objects in P, that have minimum aggregated distance to all query points from Q. The aggregated distance depends on an aggregation (e.g. *sum*, *max*, *min*) of individual distances. For spatial data (point sets) P and Q, and *sum* as aggregation function, the problem has been investigated in [35]. [33] gives a solution to the continuous a-NN problem for a static point set P, a set of moving query objects Q, and arbitrary aggregation functions. [32] handles different continuous NN query types for static points P and moving points Q over sliding windows for position updates. The use of all these nearest neighbour queries naturally emanates from real-world applications like location based services, but most work on NN-queries deals with static spatial data or operates within the tracking context, dealing with current, predictive or continuous queries.

For tracking systems (dealing with current and predictive queries) continuous (ongoing) variants of all these query types have been proposed, e.g. continuous k-NN queries [23, 5], or continuous r-NN queries [46]. Other authors investigated density-based queries on such systems [22, 24]. Instead of presenting continuous (ongoing) query types, we present analogue queries considering the complete history of moving objects, as we focus on (historic) trajectory databases.

As the notion of similarity for spatio-temporal data is highly application dependent, we leave similarity-based queries [14] out of our scope.

Aggregation queries for spatio-temporal data have been examined in [26], but we will only use simple forms of value-based spatial, temporal and spatio-temporal aggregation in our benchmark queries, leaving out aggregations relying on structure-based strategies.

2.5 Overview

In the remainder of the paper we first describe the data model used throughout this work in Section 3, before describing the scenario simulated during data generation in Section 4. In Section 5 we describe the database model with two different representations for the generated data. After identifying groups of queries we propose two sets of interesting queries (BerlinMOD/R and BerlinMOD/NN) for benchmarking (Section 6). After this, we explain how to use the data generator (Section 7). As a proof of concept, we create the benchmark to evaluate the Secondo DBMS with the BerlinMOD/R query set in Section 8. Finally, we conclude the paper in Section 9.

3 Data Model

In this section, we present the data model for (spatio-) temporal data types used throughout this work. The model has also been implemented within the Secondo DBMS, that will be used to generate the benchmark data. Whereas Secondo provides more complex data types like arrays or graphs (which are also used by the generator), we focus on the (spatio-) temporal sub system, which is used to implement the data generator and formulate the benchmark queries.

An abstract data model for representing moving objects has been proposed in [19]. The model introduces a type constructor *moving*, that creates a type for moving object data when applied to a spatial data type (there are also other type constructors, like *range* to create interval types from a base type). While the representation of moving point data is a major contribution of this article, also other kinds of data are covered, e.g. the types *moving*(*bool*), *moving*(*int*), and *moving*(*real*) are defined, which are capable of storing temporal evolutions of the particular base types. Based on this model, a discrete

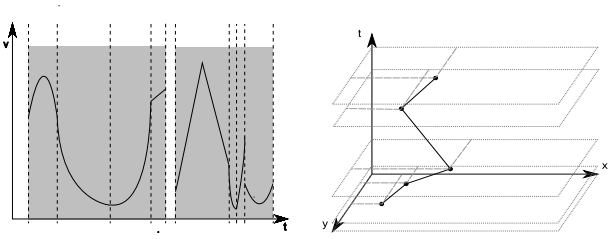


Figure 1: Sliced Representations of a Moving Real and a Moving Point

The *mreal* shown in the left figure consists of 8 *ureal* units (time slices). The total definition time is shaded in gray. For instants outside the shaded area, the *mreal* is undefined. Each unit can be described by a simple function. The right figure depicts an *mpoint* consisting of 4 *upoint* units. Here, the lines between the 5 shown points represent the time slices, for which linear interpolation is used to approximate the object's trajectory.

BASE	=	{*int*, *real*, *string*, *bool*}
SPATIAL	=	{*point*, *points*, *line*, *region*}
TEMPORAL	=	*instant*
RANGE	=	{*rint*, *rreal*, *rbool*, *periods*†}
UNIT	=	{*uint*, *ureal*, *ubool*, *upoint*, *uregion*}
MOVING	=	{*mint*, *mreal*, *mbool*, *mpoint*, *mregion*}
INTIME	=	{*iint*, *ireal*, *ibool*, *ipoint*, *iregion*}

Table 1: Type System

†: *periods* is just an alias for what would be named *rinstant*.

data model has been presented in [15], using a "*sliced representation*" to represent the development of a moving object.

In the *sliced representation*, the development of a moving object is expressed by a set of **units** (as illustrated by Figure 1). Each unit describes the behaviour of a moving object o for a certain (short) time interval (t_i, t_j), $t_i \leq t_j$. A "simple" function is used to approximate the concrete value at each instant of time contained in the unit's definition time interval. For an *int* or *bool* unit for example, a constant function is applied, for a *real* unit a quadratic polynomial function or its square root is used, and for a moving *point*, the function is represented by the static values (i.e. *point* values) taken at the unit's starting instant t_i and ending instant t_j and linearly interpolating between these values for any instant t, $t_i \leq t \leq t_j$. Obviously, for a moving object, the definition time intervals of all units belonging to the history of that object, are required to be disjoint.

SECONDO has implemented a subset of the type system proposed in [15], as shown in Table 1.

We have seven classes of types: **BASE** types, which represent static simple data (like *int*, *real*); **SPATIAL** types, representing 2-dimensional spatial objects (*point*, *points* (=multipoint), *line*, and *region*); **TEMPORAL**, with type *instant* to represent single instants of time; **RANGE** types, providing a collection of disjoint intervals defined on a base type or temporal type — they correspond to the types created using the *range* type constructor from [19]; **UNIT** types, which represent time slices of the temporal development of base types, as described in [15]; **MOVING** types, which describe the complete history of a moving object by a collection of temporally disjoint units and correspond to the types created by the *moving* type constructor in [19]; **INTIME** types, representing a concrete value at a certain temporal instant (an (*instant*, *value*) pair).

All these types can be used for attributes within relations. Large objects (like the history of units of an *mpoint*, or the segments of a *line* object) are not stored completely within the tuples' disk representation, but in a separate file, from which they are only read, when explicitly accessed.

From the type system, we only need the subset given in Table 2. Besides these types, SECONDO provides a large set of operations on the presented data types. Such a large set of operators allows for interesting and complex queries, but using them within the benchmark queries would exclude any DBMS failing to provide them. Hence, we restrict the set of operators to the most important ones. Table 3 shows the operators used to generate the data and formulate the benchmark queries.

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: Data Types Used by the Data Generator

4 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).

4.1 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 can be chosen by one of two alternative methods, as described by Brinkhoff [6]. The first one, called "network-based approach" always chooses nodes from the complete set of nodes, using a uniform distribution. The second one, the "region based approach" partitions the set of all nodes according to predefined spatial regions, and uses a probabilistic function (depending on weights for the different regions) to chose one of the node partitions. From this partition, a node is selected using a uniform distribution.

Also, each object has a *WorkNode* representing the owner's working place. Again, this node is chosen randomly among all nodes of Berlin (network based approach) or using a spatial distribution function (region based approach; with weights different from those for selecting *HomeNode*s).

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.

4.2 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

[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

Name	Signature / Description	Name	Signature / Description
val:	*ipoint* → *point* Extracts the point from the (*instant*, *point*) pair.	**atinstant**:	*mpoint* × *instant* → *ipoint* Extract the current position of *mpoint* at the given *instant*.
create_instant:	*string* → *instant* Creates an *instant* from its textual representation.	**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 within the *mpoint*'s definition time.	**passes**:	*mpoint* × *point* → *bool* *mpoint* × *region* → *bool* Tests, whether the *mpoint* ever passes the second argument.
trajectory:	*mpoint* → *line* Projects the complete *mpoint* into the 2D space.	**distance**:	*line* × *line* → *real* *mpoint* × *mpoint* → *mreal* Computes the minimum distance between the arguments.
inst:	*ipoint* → *instant* Extract the *instant* (timestamp) from the (*instant*, *point*) pair.	**initial**:	*mpoint* → *ipoint* Computes the initial position of the argument.
intersection:	*mpoint* × *mpoint* → *mpoint* Computes the spatio-temporal intersection of its arguments. The result is the *mpoint* with the common history of both arguments.	**at**:	*mbool* × *bool* → *mbool* *mpoint* × *point* → *mpoint* *mpoint* × *region* → *mpoint* Restricts the first argument to the temporal intervals, where its projection is included by/equal to the second argument.
length:	*mpoint* → *real* Computes the driving distance of the moving point.	**atperiods**:	*mpoint* × *periods* → *mpoint* Restricts *mpoint*'s history to the given time intervals.
deftime:	*mpoint* → *periods* Returns the set of all time intervals where the *mpoint* is defined (i.e. the time covered by it's history).	**inside**:	*point* × *region* → *bool* Tests, whether the *point* is inside the *region*.
sometimes:	*mbool* → *bool* Returns 'TRUE', if the *mbool* (a temporal boolean function) ever becomes 'TRUE'.	**isempty**:	X → *bool* Returns 'TRUE', if the argument is undefined or empty (for a moving type or periods value X).
$<=$:	*mreal* × *real* → *mbool* Returns a temporal boolean function with the same definition time as the first argument, yielding 'TRUE' for periods, where the first argument is less than or equal to the second, and 'FALSE' where it is greater.	**concat**:	*mpoint* × *mpoint* → *mpoint* Concatenates the histories of two moving points to a common one. Both arguments must have distinct definition times.

Table 3: Operator Descriptions

The operators listed correspond to those described in [19]. As [19] provides an algebraic approach to MOD, the set of operators was designed to be somewhat "complete", and to enable interesting query types. However, the following extensions make the model more comfortable to the user: Operators **circle**, **create_instant** and **concat** have been added to ease the data generation process. 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)))).

†: As genuine circular regions are not covered by the used data model, this operator is intended to create polygonal regions approximating circles by using large numbers for n, and therefore was named "circle".

benchmark will always be the same, except possibly for very small numeric differences on different platforms and operating systems.

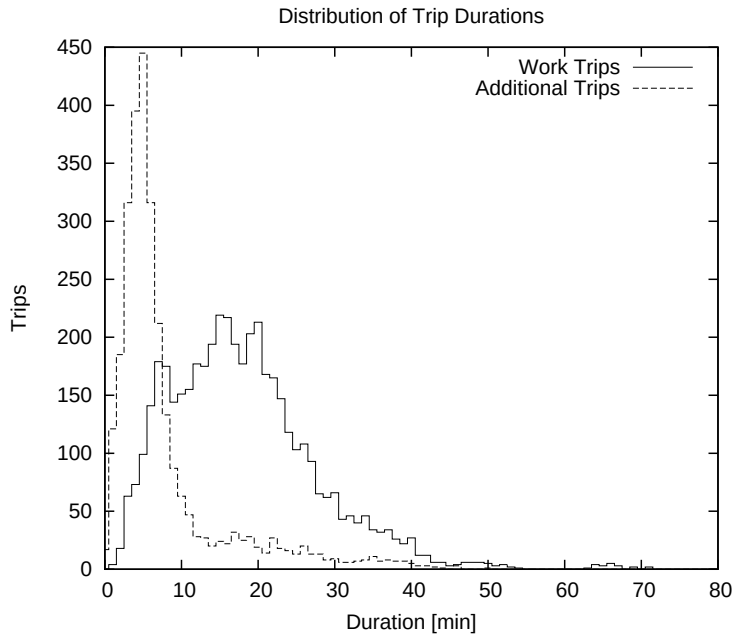


Figure 2: Typical Distribution of Single Trip Durations

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 2 for a typical distribution of trip durations), we can be almost 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.

4.3 Trip Creation Algorithm

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

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.

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 4: 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 4, apply a stop event;
  end for
end for
```

Deceleration event — The vehicle's current velocity v_c is reduced to $v_c \cdot X/20$, where $X \sim B(20, 0.5)$ is 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 4).

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.

[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.

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*

5 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. A vehicle's 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 position, 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 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 P_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, because they are more similar to real geometries used in real applications. 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 P_SAMPLESIZE query points, randomly chosen from `Nodes`.*Pos*. *Id* is a generated key for this relation.

`QueryRegions`: **relation**{*Id*: *int*, *Region*: *region*} — relation of P_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 P_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 P_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 P_SAMPLESIZE query licence plate numbers, where *Id* is a key. Licence plate numbers are uniformly sampled from all vehicles' licence plate numbers.

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`.

6 Benchmark Queries

In this section, we present two sets of queries for the BerlinMOD benchmark, working on the data set described before: BerlinMOD/R and BerlinMOD/NN. While BerlinMOD/R contains range type queries, BerlinMOD/NN provides nearest neighbour (NN) queries for historical moving object databases.

Partitioning the benchmark into different query sets makes sense, as for NN queries there is no accepted way to formulate queries in SQL and most of the needed algorithms still seem to be open problems for the history based MOD approach.

Two subsections handle the two benchmark query sets, both having a common layout: First, we will determine, which kinds of queries may arise. To do so, we define **query types**. Then we examine the types, for whether they contain interesting benchmark queries, and finally formulate concrete queries.

6.1 BerlinMOD/R

This part of the benchmark defines query types and benchmark queries for range and point queries.

6.1.1 Query Types for BerlinMOD/R

To get an overview on possible query types, we distinguish five aspects of 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.

[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.

2. **Dimension** (standard / temporal / spatial / spatio-temporal)
   This criterion refers to the dimension(s) used in the query. "standard" means, that no temporal, spatial, or spatio-temporal attribute or condition is of concern for this query. This type of queries 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.

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 properties, we can identify 96 query types. We have listed all combinations in Table 5. Not all types of queries are realizable. For example, if the object identity is known within a temporal point query on a single object, this will exclude any kind of aggregation.

6.1.2 Selection of Queries for BerlinMOD/R

Though we have 96 query types, not all of them are interesting for a collection of benchmark queries.

Nine of the combinations are unfeasible and marked with "n/a" in the last column of Table 5.

As in the benchmark data all objects are observed for the complete time, the twelve query types with dimensions "temporal" and "relation" (marked with "n/m") provide no meaningful queries and are omitted.

A large bunch of query types (21), which are marked with "s_1" in the table is rather uninteresting for benchmarking a spatio-temporal DBMS, since only standard data is relevant to these queries. However, we select two of these query types as representatives for the complete class of queries on standard data. This is to test, whether the spatio-temporal DBMS can also handle the standard part of the data well.

Fourteen query types (marked with "s_2") are quite "simple", because they only contain temporal or spatio-temporal point queries. These "degenerated" spatio-temporal queries can be processed quite fast and easily and are hence represented by three queries only.

Of the feasible 24 purely "spatial" query types (marked "s_3"), we only cover four with benchmark queries. Though all 24 types may be interesting, similar queries are covered by existing spatial benchmarks, like SEQUOIA.

Fourteen query types with temporal/spatio-temporal dimension and unbounded query interval (marked "slow") must be expected to have very long running times on a large database. Therefore, we only select one of these types to be covered by a benchmark query.

Table 5 indicates, that query type (unknown, spatio-temporal, range, single) is covered by three queries. This is due to the fact, that the notion of "spatio-temporal range" subsumes real 3D-ranges, spatial ranges at fixed temporal instants, and fixed points at temporal ranges, and we want to provide the possibility to compare access methods for these cases.

There are only a few meaningful simple aggregations on moving point data. Obviously, one can calculate statistics on lengths of trajectories. More interesting calculations involving aggregation (e.g. calculating the routes used during a given time as a *line* value, or calculating the traffic density on the network) require additional operators not covered by the benchmark's minimum requirements (see Table 3). Therefore, we decided to cover query types with aggregations by only only two queries, one on standard data, the other on trajectories of *mpoint*s.

At this point, two query types remain uncovered: (known, spatio-temporal, range, single, no aggregation) (1) and (unknown, temporal, range, single, no aggregation) (2). As for (1), the type contains queries testing a single known object (vehicle) on some spatio-temporal range condition. Query type (2) selects single objects by checking a temporal range predicate. For both Types, more complex query types are covered by the benchmark: For (1) we have a similar type with Condition Type = "relation", for (2) we have the corresponding combination with an additional aggregation.

The queries themselves correspond to and are motivated by the scenario used: long-term observation of vehicles on a transportation network. We therefore predominantly query for vehicles whose movement fulfils certain conditions, or compute derived values from these movements, using the operators introduced in Table 3. While some of the proposed queries seem to have no meaning in the given context at first glance, there may be good reasons to pose them in a different application scenario with objects moving similarly. Hence, we keep them within the benchmark.

6.1.3 Queries for BerlinMOD/R

We present a set of 17 interesting queries of different types. First, we formulate the query in common English and then in a more formal, SQL-like notation. Table 2 provides 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 [19].

As the performance of most of the queries strongly depends on the point/range values and the objects' identity, we do not run these queries for just a single combination of parameters (parameters involved are query points, regions, instants, periods and vehicle identities/licences), but usually 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 5. 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 usually 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. For three queries, we only use one instance (the query does not depend on any parameter), for one query 10 instances (regarding to the total runtime), and for another one 1000 (depends on three parameters) instances. As an information, we list the query type of each query together with the number of query instances in a header above each query.

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.

known - standard - point - single - no aggr (100)

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

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

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.

unknown - standard - point - single - aggr (1)

Object Identity	Dimension	Query Interval	Condition Type	Aggregation	Queries	Comment
known	standard	point	single	yes		n/a
known	standard	point	single	no	1	s_1
known	standard	point	relation	yes		s_1
known	standard	point	relation	no		s_1
known	standard	range	single	yes		n/a
known	standard	range	single	no		s_1
known	standard	range	relation	yes		s_1
known	standard	range	relation	no		s_1
known	standard	unbounded	single	yes		n/a
known	standard	unbounded	single	no		s_1
known	standard	unbounded	relation	yes		s_1
known	standard	unbounded	relation	no		s_1
known	temporal	point	single	yes		n/a
known	temporal	point	single	no	3	s_2
known	temporal	point	relation	yes		s_2, n/m
known	temporal	point	relation	no		s_2, n/m
known	temporal	range	single	yes		n/a
known	temporal	range	single	(no)	8	
known	temporal	range	relation	yes		n/m
known	temporal	range	relation	(no)		n/m
known	temporal	unbounded	single	yes		n/a
known	temporal	unbounded	single	(no)		slow
known	temporal	unbounded	relation	yes		slow, n/m
known	temporal	unbounded	relation	(no)		slow, n/m
known	spatial	point	single	yes		s_3
known	spatial	point	single	(no)		s_3
known	spatial	point	relation	yes		s_3
known	spatial	point	relation	(no)		s_3
known	spatial	range	single	yes		s_3
known	spatial	range	single	(no)		s_3
known	spatial	range	relation	yes		s_3
known	spatial	range	relation	(no)		s_3
known	spatial	unbounded	single	yes		s_3
known	spatial	unbounded	single	(no)		s_3
known	spatial	unbounded	relation	yes		s_3
known	spatial	unbounded	relation	(no)	5	s_3
known	spatio-temporal	point	single	yes		n/a
known	spatio-temporal	point	single	no		s_2
known	spatio-temporal	point	relation	yes		s_2
known	spatio-temporal	point	relation	no		s_2
known	spatio-temporal	range	single	yes		n/a
known	spatio-temporal	range	single	(no)		
known	spatio-temporal	range	relation	yes		
known	spatio-temporal	range	relation	(no)	10	
known	spatio-temporal	unbounded	single	yes		n/a
known	spatio-temporal	unbounded	single	(no)		slow
known	spatio-temporal	unbounded	relation	yes		slow
known	spatio-temporal	unbounded	relation	(no)		slow
unknown	standard	point	single	yes	2	s_1
unknown	standard	point	single	no		s_1
unknown	standard	point	relation	yes		s_1
unknown	standard	point	relation	no		s_1
unknown	standard	range	single	yes		s_1
unknown	standard	range	single	no		s_1
unknown	standard	range	relation	yes		s_1
unknown	standard	range	relation	no		s_1
unknown	standard	unbounded	single	yes		s_1
unknown	standard	unbounded	single	no		s_1
unknown	standard	unbounded	relation	yes		s_1
unknown	standard	unbounded	relation	no		s_1
unknown	temporal	point	single	yes		s_2
unknown	temporal	point	single	no		s_2
unknown	temporal	point	relation	yes		s_2, n/m
unknown	temporal	point	relation	no		s_2, n/m
unknown	temporal	range	single	yes	9	
unknown	temporal	range	single	(no)		
unknown	temporal	range	relation	yes		n/m
unknown	temporal	range	relation	(no)		n/m
unknown	temporal	unbounded	single	yes		slow
unknown	temporal	unbounded	single	(no)		slow
unknown	temporal	unbounded	relation	yes		slow, n/m
unknown	temporal	unbounded	relation	(no)		slow, n/m
unknown	spatial	point	single	yes		s_3
unknown	spatial	point	single	(no)	4	s_3
unknown	spatial	point	relation	yes		s_3
unknown	spatial	point	relation	(no)		s_3
unknown	spatial	range	single	yes		s_3
unknown	spatial	range	single	(no)		s_3
unknown	spatial	range	relation	yes		s_3
unknown	spatial	range	relation	(no)		s_3
unknown	spatial	unbounded	single	yes		s_3
unknown	spatial	unbounded	single	(no)		s_3
unknown	spatial	unbounded	relation	yes		s_3
unknown	spatial	unbounded	relation	no	7	s_3
unknown	spatio-temporal	point	single	yes		s_2
unknown	spatio-temporal	point	single	no	11	s_2
unknown	spatio-temporal	point	relation	yes		s_2
unknown	spatio-temporal	point	relation	no	12	s_2
unknown	spatio-temporal	range	single	yes		
unknown	spatio-temporal	range	single	(no)	13, 14, 15	
unknown	spatio-temporal	range	relation	yes		
unknown	spatio-temporal	range	relation	(no)	16	
unknown	spatio-temporal	unbounded	single	yes		slow
unknown	spatio-temporal	unbounded	single	(no)		slow
unknown	spatio-temporal	unbounded	relation	yes		slow
unknown	spatio-temporal	unbounded	relation	(no)	6	slow

Table 5: Query Type Selection for BerlinMOD/R

n/a: Query type not applicable. n/m: Query type contains no meaningful query.

s_1: Simple (only standard data involved). s_2: Simple (temporal or spatio-temporal point query).

s_3: Simple (spatial data only).

slow: Query with temporal/ spatio-temporal dimension and unbounded query interval may require very long run times.

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

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

Also testing standard type queries, but having a more selective predicate and using a final aggregation (count).

known - temporal - point - single - no aggr (100)

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 (100)

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 (100)

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. Though the query may seem to have no meaning in the given context, it starts making sense when the moving objects represent entities, that should be kept apart (e.g. animals placing odor marks on their paths).

unknown - spatio-temporal - unbounded - relation - no aggr (1)

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 [9].

unknown - spatial - unbounded - relation - no aggr (100)

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 (100)

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 [9] is used.

unknown - temporal - range - single - aggr (100)

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 (10)

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**: $X \times X \to X$ 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 (100)

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 (100)

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 (100)

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 (1000)

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 the parameters `QueryLicences1` and `QueryLicences2` are just used to reduce the search space, we only count 100, but not 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 (1)

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);
```

6.2 BerlinMOD/NN

This part of the benchmark defines query types and benchmark queries for nearest neighbour (NN) queries on historical moving object databases. This is motivated by the fact that nearest neighbour queries constitute — besides range queries — the most fundamental query type in spatial and spatio-temporal databases and there exists a lot of research especially on algorithms for various such query types, as discussed in Section 2.4. There are some obstacles to including them in the benchmark, however:

1. Nearest neighbour queries on histories of moving objects to our knowledge have not yet been addressed in the literature. Existing research considers queries for tracking databases dealing with current and predicted linear movement as well as the online maintenance of nearest neighbours (continuous queries). For this reason, established definitions of the semantics of such queries on historical databases are not available.

2. Whereas for the queries in Section 6.1 a precise language framework exists based on embedding operations in a standard way into SQL, this is not the case for nearest neighbour queries. The main problem is that such queries cannot be formulated in terms of conditions in the where-clause or expressions in the select-clause. Instead, global extensions of the SQL language are needed at the level of, say, a *group by* construct. To our knowledge, no satisfactory solutions are available even for static nearest neighbour queries. A proposal [2] formulates nearest neighbour conditions in the where-clause, but this solution suffers from the fact that conditions are not commutative anymore.

Also, only a single constant parameter can be used (i.e. it is not possible to find nearest neighbours for many objects). Oracle Spatial offers some formulations for nearest neighbour queries that are awkward at best and suffer from similar problems.[6]

Much less are adequate formulations available for the dynamic spatio-temporal versions or for other query types such as reverse or aggregate nearest neighbours.

3. No algorithms have been published for nearest neighbour queries on histories of moving objects nor are systems available that can execute them.

Nevertheless, nearest neighbour analyses on histories of moving objects appear interesting and relevant and we include some queries in the benchmark, as a challenge for the research community. We define semantics for such queries in the sequel. However, it is not possible to offer query formulations in SQL. Also, SECONDO is not capable of executing such queries; hence they are not part of the experimental evaluation of Section 8.2.

6.2.1 Semantics of Nearest Neighbour Queries on Historical Moving Objects Databases

Our definition of semantics for nearest neighbour queries on moving object histories is based on the following principle: The result of a time dependent operation must be consistent with the result of the corresponding static operation for each instant of time. This agrees with the definition of *lifting* for operations in [19].

We use the following notations: Let $d(p, q)$ denote the distance between two points p and q. For a point p and a point set Q, let $d(p, Q, aggr)$ denote the aggregate distance using an aggregation operator $aggr$ from the set $\{min, max, sum\}$. Let $mp(i)$ denote the position of moving point mp at instant i.

A static *k-nearest neighbour query* can be formulated as follows: Given a query point q and a relation R with a point attribute loc, return the k tuples of R with the smallest values of $d(q, loc)$.

Definition 6.1 (k-NN Query) *A* spatiotemporal k-nearest neighbour query *is defined as follows: Given a query* mpoint *mq and a relation R with an attribute $mloc$ of type* mpoint*, return a subset R' of R where each tuple has an additional attribute $mloc'$, such that the following conditions hold:*

1. *For each tuple $t \in R'$, there exists an instant of time i such that $d(t.mloc(i), mq(i))$ is among the k smallest distances from the set $\{d(u.mloc(i), mq(i)) \mid u \in R\}$.*
2. *$mloc'$ is defined only at the times condition (1) holds.*
3. *$mloc'(i) = mloc(i)$ whenever it is defined.* □

A static *reverse nearest neighbour query* is: Given a query point q and a relation R with a point attribute loc, return $\{t \in R \mid \forall u \in R, d(t.loc, q) \leq d(t.loc, u.loc)\}$.

Definition 6.2 (r-NN Query) *A* spatiotemporal reverse-nearest-neighbour query *is defined as follows: Given a query* mpoint *mq and a relation R with an attribute $mloc$ of type* mpoint*, return a subset R' of R where each tuple has an additional attribute $mloc'$, such that the following conditions hold:*

1. *For each tuple $t \in R'$, there exists an instant of time i such that*

$$\forall u \in R, d(t.loc(i), mq(i)) \leq d(t.loc(i), u.loc(i)).$$

2. *$mloc'$ is defined only at the times when condition (1) holds.*
3. *$mloc'(i) = mloc(i)$ whenever it is defined.* □

A static *aggregate nearest neighbour query* is: Given a relation Q with point attribute pos and a relation R with point attribute loc, return the tuples in R for which the aggregate distance $d(loc, Q.pos, aggr)$ is minimal.

[6] These remarks refer to Release 2 of Oracle 10g (see e.g. [7]).

Definition 6.3 (a-NN Query) *A* spatiotemporal aggregate nearest neighbour query *is defined as follows: Given a query relation Q with mpoint attribute mpos and a relation R with mpoint attribute mloc, return a subset R' of R where each tuple has an additional attribute mloc′, such that the following conditions hold:*

1. *For each tuple $t \in R'$, there exists an instant of time i such that*

$$\forall u \in R, d(t.loc(i), Q.mpos(i), aggr)) \leq d(u.loc(i), Q.mpos(i), aggr)).$$

2. *mloc′ is defined only at the times when condition (1) holds.*
3. *$mloc'(i) = mloc(i)$ whenever it is defined.* □

All definitions are given for time dependent data and query objects, i.e., for the most general case. Restrictions to static query or data objects are straightforward. Note that for static data points, the result relation will have moving points.

6.2.2 Query Types for BerlinMOD/NN

For NN queries, we distinguish the following properties:

1. **Condition Type** (k-NN/ r-NN / a-NN)
   This property determines which characteristic is used to perform the NN query: k nearest neighbours (k-NN), reversed nearest neighbour (r-NN), or aggregated nearest neighbour (a-NN).
2. **Query Object Type** (static / moving)
   This property determines whether the query object (point/region) is a static or a moving object.
3. **Candidate Type** (static / moving)
   This property determines whether the candidate objects (points/regions) are static or moving objects.

Therefore, we have a total of 12 different NN query types.

Formally, the result of a spatio-temporal NN query is a temporal function, which can be represented as a spatio-temporal object or using a collection of spatio-temporal objects.

6.2.3 Selection of Queries for BerlinMOD/NN

From the 12 NN query types, we select all four k-NN types, as this seems to be be best-researched field (though most works cover current/predictive queries only and do not handle historic MOD). Hence, we also include (k-NN, static, static), which is in fact the spatial k-NN problem and therefore the most simple NN problem to cover. We also select all non-static r-NN types (omitting the most simple, purely spatial variant), and from the a-NN problems, we select the hardest (a-NN, moving, moving) and the most interesting spatio-temporal variant (a-NN, moving, static), which seems to support the more interesting applications.

6.2.4 Queries for BerlinMOD/NN

As explained for the BerlinMOD/R queries (Section 6.1.3), we usually use 100 query instances to reduce bias and get more significant results.

k-NN - static - static (100)

Query 18 For each vehicle with a licence plate number from `QueryLicences1` and each instant from `QueryInstants1`: Which are the 10 vehicles that have been closest to that vehicle at the given instant?

This query is a rather simple spatial k nearest neighbours (k-NN) query, as though it uses a moving reference object and moving candidate objects, the temporal dimension is fixed to an instant, therefore reducing all moving objects to points.

k-NN - moving - static (100)

Query 19 For each vehicle with a licence from `QueryLicences1` and each period from `QueryPeriods1`: Which points from `QueryPoints` have when been the 3 closest to that vehicle during that period?

This k-NN query is typical for location based service applications. It involves only one vehicle at a time, whereas the nearest three points (of interest) are asked for during a given temporal range.

k-NN - static - moving (100)

Query 20 For each region from `QueryRegions1` and period from `QueryPeriods1`: What are the licences of the 10 vehicles that are closest to that region during the given observation period?

This k-NN query is not trivial, as a static region object is used as the static reference for moving candidate objects. (Strictly speaking, this type of query is not covered by the formal definition of Section 6.2.1, but a generalization to regions is straightforward.)

k-NN - moving - moving (100)

Query 21 For each vehicle with a licence plate number from `QueryLicences1` and each period from `QueryPeriods1`: Which are the 10 vehicles that are closest to that vehicle at all times during that period?

This query is the most complex k-NN query within this benchmark, using a moving query object and moving candidate objects.

r-NN - moving - static (100)

Query 22 For each vehicle with a licence from `QueryLicences`: Report the points from `QueryPoints`, whose nearest neighbour the vehicle has been, together with the according time intervals.

This is a reversed nearest neighbour (r-NN) query using a moving query object.

r-NN - static - moving (100)

Query 23 For each point from `QueryPoints1` and period from `QueryPeriods1`: Report the licences of the vehicles, having that point as the nearest network node, and the temporal intervals, for that these relations hold during the given period.

This r-NN query has a static query object, but moving candidate objects.

r-NN - moving - moving (100)

Query 24 For each vehicle with a licence plate number from `QueryLicences1` and each period from `QueryPeriods1`: Report the licences of vehicles, having the given vehicle as the nearest vehicle and the according time intervals during the given period.

This is the most complex r-NN query of this benchmark. It provides a moving query object and several moving candidate objects.

a-NN - moving - static (100)

Query 25 For each group of ten vehicles having ten disjoint consecutive licences from `QueryLicences` and each period from `QueryPeriods1`: Report the point(s) from `QueryPoints`, having the minimum aggregated distance from the given group of ten vehicles during the given period.

This is the "simple" case of an a-NN query with static candidate objects and moving query objects. We restrict the search space to the given periods.

a-NN - moving - moving (1000)

Query 26 Create the ten groups of vehicles having ten disjoint consecutive licences from `QueryLicences`. For each pair of such groups and each period from `QueryPeriods1`: Report the licence(s) of the vehicle(s) within the given first group of ten vehicles, having the minimum aggregated distance from the given other group of ten vehicles during the given period.

This is a challenging a-NN query, using ten moving query objects and ten moving candidate objects.

7 Using the Data Generator

In this section, we describe how the data generator for BerlinMOD is used. Also, the principles of the data generation are explained.

7.1 The Data Generator Script

We create the benchmark data using the extensible database system Secondo [10, 20, 40]. 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 [18].

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 object data, 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. The script is a simple text file and can be changed using any text editor.

We have set up a web site where the Secondo system and documentation, the benchmark generator script and data objects, and also the scripts with executable benchmark queries for Secondo (the ones we used in Section 8.2) are available for download [4].

7.2 Data Import

The benchmark data is created using map data for the German capital Berlin. We use converted street data from the bicycle routing program **bbbike** [37], and statistical data on the Berlin districts provided by the Berlin Statistical Office [47, 48]. The data is provided in form of three text files containing data in Secondo's nested list format:

File "streets.data" provides the relation `streets`: **relation**{*Vmax*: *real*, *geoData*: *line*} — relation of all streets with the maximum allowed speed in km/h *Vmax*, and the street geometry *geoData*.

File "homeRegions.data" provides a relation `homeRegions`: **relation**{*Priority*: *int*, *Weight*: *real*, *geoData*: *region*} — relation describing the distribution of HomeNodes, where *geoData* is a region defining an area, that will provide HomeNodes according to its *Weight* in relation to the total of all *Weight* values. *Priority* defines the priority of each region for the case of overlapping areas.

File "workRegions.data" provides a relation `workRegions`: **relation**{*Priority*: *int*, *Weight*: *real*, *geoData*: *region*} — relation describing the distribution of WorkNodes, attribute definitions are as for relation `homeRegions`.

We recommend to use the original data objects provided with the benchmark script in order to achieve comparable benchmark results. However, any other geo data can be used as input to the data generation process without changing the generator parts of the script.

The only requirement is that data must be available in SECONDO nested list format or as Shapefiles and have the stated names and relation schemas, so that they can be imported directly. There also exist several converter tools provided with SECONDO, that will for example convert Shapefiles, or TIGER LINE files to the SECONDO nested list format.

7.3 Setting the Parameters

All parameters are set up in the initial section of the generator script. Any of the parameters can be modified by editing the script within a text editor. Table 6 provides an overview on all parameters and their default settings.

To keep scaling the benchmark data simple, we make use of a global scale factor called `SCALEFACTOR`. Changes to this parameter determine the amount of data generated — the data grows linearly with `SCALEFACTOR`. This is achieved by distributing changes to the secondary parameters `SCALEFCARS` and `SCALEFDAYS`, which internally set the tertiary parameters `P_NUMCARS` and `P_NUMDAYS`. Direct changes to the secondary or tertiary scaling parameters are possible. We decided to use a single scaling factor in order to distribute scale changes to both parameters — amount of observed objects, and observation periods — avoiding to favour one of them. We do not scale the spatial size, since this does neither match the scenario, nor related real world applications.

Parameter `P_SAMPLESIZE` determines the cardinality of the relations providing query objects used within the benchmark queries. `P_TRIP_MODE` allows to choose between "Network Based" and "Region Based" node selection. Two different models for distance computations are available. By setting `P_TRIP_DISTANCE` to the according value, "Fastest Path" or "Shortest Path" can be selected for use within the routing algorithm.

If disturbed (GPS-like) data shall be generated, `P_DISTURB_DATA` needs to be set to `TRUE`. Then, the secondary parameters `P_GPS_TOTALMAXERR` (maximum deviation in meters) and `P_GPS_STEPMAXERR` (maximum deviation per interval in meters) are observed.

`P_STARTDAY` simply determines the date of the first observation day. The default setting starts observation on a monday. `P_NEIGHBOURHOOD_RADIUS` sets the maximum distance around a node defining it's neighbourhood.

While `P_GPSINTERVAL_MS` determines the intervals between consecutive sampling instants (default: 2 seconds), two parameters are used to separate vehicle movements into shorter trips for the TBA: `P_MINPAUSE_MS` determines which period of immobility will be interpreted as a separating pause between consecutive trips; `P_MINVELOCITY` is the maximum velocity (meters per hour), at which vehicles will be interpreted as "immobile" — this is particularly important when the disturbed data generation is used.

Two parameters (`P_HOMERANDSEED`, `P_TRIPRANDSEED`) allow to specify the seeds of the pseudorandom number generators used during data generation.

Though all parameters explained in the context of Algorithm 1 (acceleration rate, several stochastic parameters regarding event generation, maximum subsegment length etc.) can be modified by changing the according definitions in the generator script, we recommend using the default settings.

At last, parameter `P_EXPORT_TYPE` determines the kind of export format for the generated data. While data is always stored in the SECONDO binary format, you can choose to export all generated data to Shapefile ("Shape") format, to a comma-separated text file ("CSV"), or to SECONDO nested list text file ("Secondo Nested List"). More on data export, like data formats, and export file names, will be explained in Section 7.7.

7.4 Running the Script

After having installed and compiled SECONDO in directory `$SECONDO_BUILD_DIR`, change to subdirectory `$SECONDO_BUILD_DIR/bin`. Copy the generator script file `BerlinMOD_DataGenerator.SEC`, and the data files `streets.data`, `homeRegions.data` and `workRegions.data` to this directory. You can now modify the parameter settings at the script's head section. We strongly recommend to deactivate logging during the data creation process by changing the file "`SecondoConfig.ini`": Just remove the leading "#" from line "`#RTFlags += SMI:NoTransactions`". Start SECONDO by typing `SecondoTTYBDB`. At the SECONDO

Parameter Name	Possible Settings	Default Setting	Meaning
SCALEFACTOR	$r \geq 0.025$	1.0	Global scale factor
SCALEFCARS	$r > 0.0$	$\sqrt{\texttt{SCALEFACTOR}}$	Scaling the number of observed vehicles
SCALEFDAYS	$r > 0.0$	$\sqrt{\texttt{SCALEFACTOR}}$	Scaling the number of observation days
P_NUMCARS	$n \geq 1$	SCALEFCARS $\cdot$ 2000	Number of observed vehicles
P_NUMDAYS	$n \geq 1$	SCALEFCARS $\cdot$ 28	Number of observation days
P_SAMPLESIZE	$n \geq 1$	100	Cardinality for relations with query objects
P_TRIP_MODE	"Network Based" "Region Based"	"Network Based"	Apply network based nodes selection Apply region based nodes selection
P_TRIP_DISTANCE	"Fastest Path" "Shortest Path"	"Fastest Path"	Apply fastest way distance metric Apply shortest way distance metric
P_DISTURB_DATA	FALSE TRUE	FALSE	Generate precise positioning data Generate disturbed positioning data
P_GPS_TOTALMAXERR	$0.0 \leq r$	100.0	Maximum total deviation of positioning data
P_GPS_STEPMAXERR	$0.0 \leq r$	1.0	Maximum total deviation per interval
P_STARTDAY	MININT $\leq n \leq$ MAXINT	2702	Date of the first day of observation. Represents a n days setoff to the NULLDAY (01/03/2000)
P_NEIGHBOURHOOD_RADIUS	$r > 0.0$	3000.0	The radius defining a node's neighbourhood.
P_GPSINTERVAL_MS	$n \geq 1$	2000	The maximum interval between consecutive position samplings in milliseconds
P_MINPAUSE_MS	$n \geq 1$	300000	Minimum pause between consecutive trips [ms]
P_MINVELOCITY	$r \geq 0.0$	0.041667	Minimum speed for non-static vehicles [m/h]
P_HOMERANDSEED	MININT $\leq n \leq$ MAXINT	0	Random seed for node selection
P_TRIPRANDSEED	MININT $\leq n \leq$ MAXINT	4277	Random seed for trip creation
P_EXPORT_TYPE	"None" "Secondo Nested List" "Shape" "CSV"	"None"	No data export Export as SECONDO Nested List textfiles Export as Shapefiles/dBase files Export as CSV text files

Table 6: Data Generator Parameters

n stands for an integer, r for a real (the decimal point is essential)

prompt, type `@BerlinMOD_DataGenerator.SEC` to run the generator script. The script will produce lots of output on your screen. When the script has finished, type `quit;` to leave SECONDO and quit to the shell.

7.5 What happens during the execution of the script?

The following paragraphs explain how the generator script works. We have inserted reference numbers enclosed by parentheses into the generator script and the following explanations, so that you can investigate it easily.

7.5.1 Creating the Street Graph

First, map data is imported from the three map data files (Section 1). Then, the parameters are set (Section 2). The main part of the data generator (Section 3) starts initializing SECONDO's Simulation-Algebra according to the parameter settings (3.0). After this, the mere street graph generation process (3.1) is started by removing impassable streets from the map (3.1.1). A street is impassable, if its speed limit *Vmax* is below P_MINVELOCITY. Then, all crossings and end points are extracted from the streets' *geoData* attributes, creating the nodes for the street graph (3.1.2.1, 3.1.2.3-4). Each node comprises a *NodeId* and a position *Pos*. The streets are decomposed into sections, i.e. line fragments between crossings and/or end points (3.1.2.2). The edges of the street graph are created by annotating each street section with the according speed limit *Vmax*, and the incident nodes' *NodeId*s (3.1.2.5–8). From the

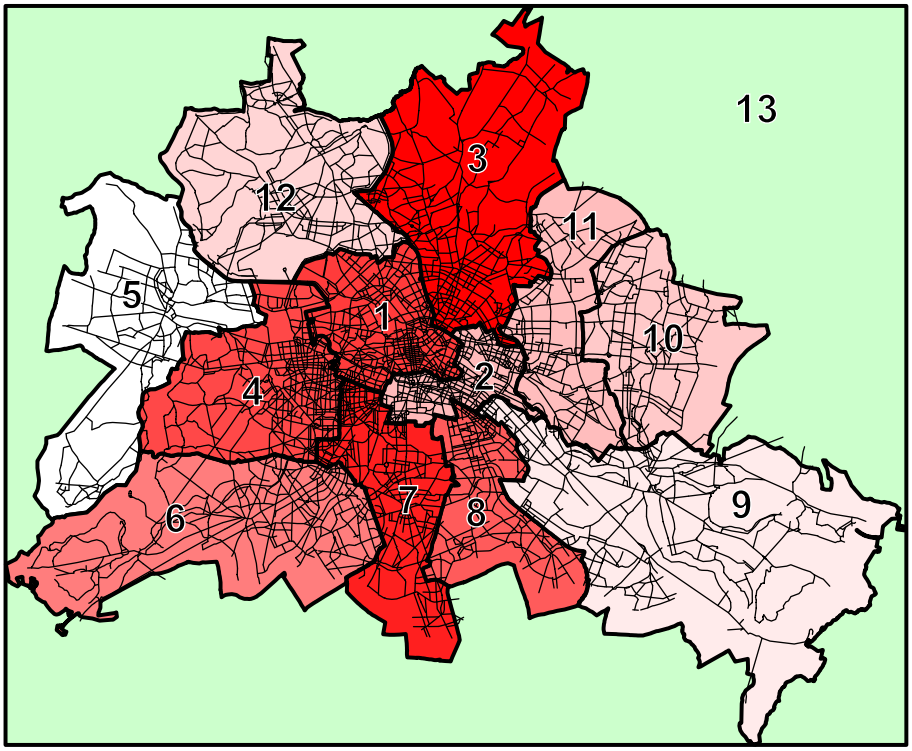


Figure 3: Regions Defined for Selecting Home and Work Nodes

nodes and edges, two graph data structures are generated. The first graph uses edges labeled with the lengths of the corresponding sections' *geoData* (3.1.3.1), the second graph's edges are labeled with the minimum time required to travel the according edges at the maximum allowed velocity *Vmax* (3.1.3.2). Thus, we can use two different distance measures (`"Shortest Path"`, and `"Fastest Path"`). As the last step during graph creation, we restrict each graph to its largest connected component (3.1.3.3). For the provided standard geo data, the created network graph has 4,468 nodes and 13,384 edges.

7.5.2 Creating Labour Paths

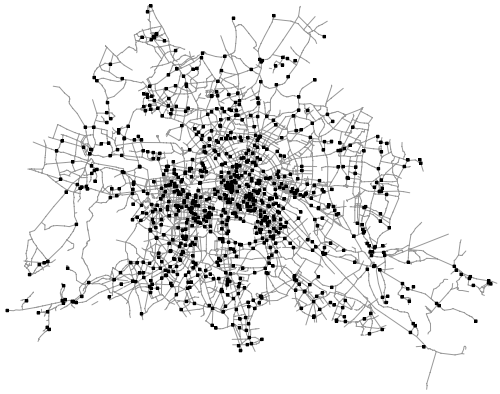
For each vehicle, a *HomeNode* and a *WorkNode* are selected using the policy selected by `P_TRIP_MODE` (3.2). When `"Network Based"` was selected, both nodes are chosen randomly using a uniform distribution over all nodes of the network. Thus, the density of the nodes in the network corresponds to the density of such nodes. Figure 4 a) shows a distribution of the selection of 1000 nodes within the network using this approach.

If the parameter is set to `"Region Based"`, the region based approach as described in [6] is used (3.2.2-7). Then, the script uses the data provided by the files `homeRegions` and `workRegions`, which define location dependent distributions across the graph nodes.

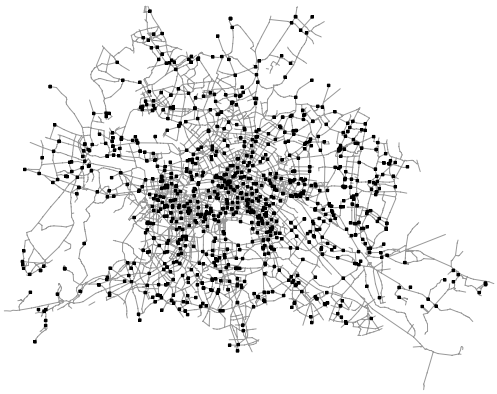
Our standard data uses the regions shown in Figure 3.

The graph nodes are partitioned according to the specifications provided by the data files. If a node occurs in more than one region, it is assigned to the one with the lowest priority.

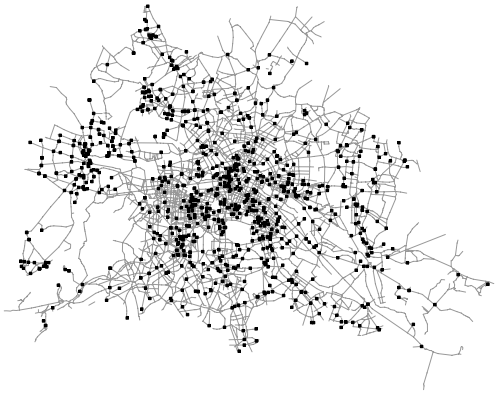
Within the default data, we use 12 regions according to the districts of Berlin [48] (with priority numbers equal to the Berlin Statistical Offices' standard reference system), plus one "fall back" region "Brandenburg" with priority 13, as shown by Figure 3. For the Home Node weights, we use the population statistics on the districts [47]; for the Work Node weights, we use aggregated statistics on employed persons per district:



a) Network Based



b) Region Based (WorkNodes)



c) Region Based (HomeNodes)

Figure 4: Node Selection Strategies

The maps show the default network with 1000 nodes selected using both selection strategies. For the Figure b) (Figure c)) we used the default regional WorkNode (HomeNode) distribution with the regions as illustrated in Figure 3.

Priority	Population	Jobs	District
1	320800	20287	Mitte
2	258500	13082	Friedrichshain-Kreuzberg
3	350500	8868	Pankow
4	314700	8325	Charlottenburg-Wilmersdorf
5	225700	21580	Spandau
6	288500	8882	Steglitz-Zehlendorf
7	334400	20546	Tempelhof-Schöneberg
8	305700	15966	Neukölln
9	234700	11239	Treptow-Köpenick
10	251400	7438	Marzahn-Hellersdorf
11	257500	7042	Lichtenberg
12	245500	17050	Reinickendorf
13	10000	1000	Brandenburg

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 Figure 4 b).

Using one of these node selection methods, for each vehicle a *HomeNode* and a *WorkNode* is selected. Note, that the *HomeNode* may be equal to the *WorkNode* (home workers). For each *HomeNode*, its neighbourhood is computed, and Dijkstra's algorithm [11] is applied to find the fastest (shortest) way between *HomeNode* and *WorkNode*.

7.5.3 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 (3.3). The most important one (*Path2MPoint*) (3.3.8) produces a single trip along a path through the graph starting at a given time instant. Basically, the 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 to `P_GPSINTERVAL_MS` milliseconds. At any change of direction, a sample point is created (even if the `P_GPSINTERVAL_MS` 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*, which is based on the function *SelectDestNode* (3.3.9), selecting a destination node applying Algorithm 2.

The observation of a vehicle for a complete day is performed by the function **CreateDay** (3.3.14). It distinguishes between working days and weekend days to produce two kinds of different behaviour.

The "main" function of the script is *CreateVehicles* (3.3.18). For each vehicle, `P_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 its licence plate number, type and model are created.

The actual creation of the moving points (3.4.1) is started calling the *CreateVehicles* function. If disturbed data shall be created, *CreateVehiclesDisturbed* (3.3.19) is called instead. Now, the additional vehicle data (licence plate number, type, model) are joined (3.4.2).

7.5.4 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 specified by parameter `P_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 (3.4.3-4).

7.5.5 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. The cardinality of these relations is determined by parameter `P_SAMPLESIZE`.

7.6 Simulating Measuring Errors (Optional)

As by now 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. This kind of data is generated, when parameter `P_DISTURB_DATA` is set to `FALSE`.

In a real application, using GPS devices to capture the vehicles' routes, this exact graph matching is improbable due to measurement errors. Besides the impresice nature of collected real application data, also from the query processing side of view, such "imprecise" position data is an interesting task, as dealing with inexact data must be reflected by adapting 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).

To allow for experiments with imprecise data, the BerlinMOD data generator includes an option to model measurement errors. When changing parameter `P_DISTURB_DATA` to `TRUE`, the following simple procedure is used to disturb the correct position data:

We assume that a GPS device has a maximum total error depending on some factors (kind of device, weather etc.), defined by parameter `P_GPS_TOTALMAXERR`. Between each two consecutive measurements of the GPS device, its error changes only slightly, up to a maximum determined by parameter `P_GPS_STEPMAXERR`. Both parameters are used with a special operator **disturb**, which changes a given moving point randomly depending on the maximum error and the error difference between two measures. The default settings 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.

Though this approach is very simplistic, and does not model GPS measuring errors in a convenient way, it does provide deviated, imprecise position data helpful for testing and benchmarking spatio-temporal DBMS.

7.7 Data Export

The generator allows for exporting the benchmark data to a variety of simple database exchange formats. To choose between export formats, parameter `P_EXPORT_TYPE` should be set to the according value (see Section 7.3).

7.7.1 No Export

If the parameter `P_EXPORT_TYPE` is set to `"None"`, data are only available within the SECONDO-system. No export files are created.

7.7.2 Secondo Nested List Export

SECONDO's proprietary data exchange format are nested lists. Such a list may be atomic (bool, int, real, symbol, string, text) or a list with elements which can be either atomic or a list for itself. Nested lists are written to files in plain ASCII text format. If the parameter `P_EXPORT_TYPE` has the value `"Secondo`

Nested List", eight files containing such lists are created within the directory $SECONDO_BUILD_DIR/bin. The files are datamcar.obj, datamtrip.obj, queryinstants.obj, querylicences.obj, queryperiods.obj, querypoints.obj, queryregions.obj, and streets.obj. Their content corresponds to the same-named relations created by the data generator script. A description of the format for spatial and spatio-temporal data types can be found in [30].

7.7.3 Shapefile/DBase-Export

Shapefiles [1] are a well known format for geometries. Basically, a relation consisting of a spatial attribute and other atomic types is represented by a file containing the spatial attribute (the actual shapefile) and a DBASE-III file containing the relation's standard attributes. If P_EXPORT_TYPE="Shape" holds, the relations QueryPoints, QueryRegions, and streets are exported in this format. Other relations are written to DBase-III-files. The relation dataMcar and the relation QueryLicences are exported without any changes to a file named cars.dbf and querylicences.dbf respectively. Unfortunately, within a dbf-file, we cannot store an *instant* directly because of the absence of an appropriate type in DBase-III. So, we export an *instant* as a string value when exporting the QueryInstants relation. For each *QueryPeriod*, we store the starting and the ending instant as strings within the queryperiods.dbf file. For the moving point data, we create a file trips.dbf, containing a relation with scheme:

trips: **relation**{*Moid*: *int*, *Tripid*: *int*, *TStart*: *instant*, *Tend*: *instant*,
Xstart: *real*, *Ystart*: *real*, *Xend*: *real*, *Yend*: *real*}

Each row describes a linear movement of the car with id *Moid* during the trip *Tripid* within the temporal interval described by instants *Tstart* and *Tend*, starting at position $(X, Y) = (Xstart, Ystart)$, and ending at position $(Xend, Yend)$.

7.7.4 CSV-Export

Setting the parameter P_EXPORT_TYPE to "CSV" forces the export of the created data as comma separated values. The export of the relations dataMcar, QueryInstants, QueryLicenes, QueryPeriods, and QueryPoints is staightforward and not described in detail here.

For the relation QueryRegions, we create a file containing an *id* and a *point*. Collecting all points related to the same id and computing the convex hull of these points will create the corresponding region.

Relation streets is exported by separating the segments of a line. The csv file has attributes *id*, *Vmax*, *X1*, *Y1*, *X2*, and *Y2*. The union of all segments $((X1, Y1), (X2, Y2))$ related to the same *id*-value is the street section corresponding to this id.

The moving point data are exported very similar to the export in a way used for DBase files. For this reason, we omit a detailed description.

8 Creating and Using BerlinMOD

We have created BerlinMOD on a standard PC with the configuration shown in Table 7. We generated the benchmark data for three different scale factors using the network based approach, and for one scale factor using the region based approach, and tested Secondo with each of the data sets and the BerlinMOD/R queries. Since Secondo does not provide the functionality to efficiently support NN queries, we omit the BerlinMOD/NN queries.

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 7: Configuration of the used Standard PC

	Network Based			Region Based
Scalefactor	0.05	0.2	1.0	0.2
Observed Vehicles	447	894	2,000	894
Observation Period	6 days	13 days	28 days	13 days
Time to Build (excl. indexes)	12:16 min	74:34 min	573:17 min	110:23 min
Time to Build (incl. indexes)	25:27 min	155:04 min	890:35 min	226:38 min
On-Disk Size (excl. indexes)	1.91 GB	4.97 GB	19.45 GB	5.18 GB
Average Number of Units per Vehicle	6,138.857	13,040.283	26,963.197	13,535.700
Minimum Number of Units per Vehicle	1,038	2,498	3,247	1,053
Maximum Number of Units per Vehicle	22,016	44,517	94,021	37,620
Total Number of Trips	15,049	62,893	292,693	62,956
Average Number of Trips per Vehicle	33.667	70.353	146.347	70.4205
Minimum Number of Trips per Vehicle	21	45	103	45
Maximum Number of Trips per Vehicle	53	101	199	101
Average Driven Distance per Vehicle	170.645 km	361.661 km	748.496 km	391.095 km
Average Trip Length	5,068 m	5,141 m	5,114 m	5,554 m

Table 8: Statistics on Created Data for Different Scalefactors

8.1 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 (2:46:38 h with the region based approach), 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 8.

8.2 Applying BerlinMOD to Secondo

We have used the benchmark data and the BerlinMOD/R query set 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 7 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 9 shows the total response times for all queries. Table 10 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.

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, 8, 10, 11, 12, 13, 14, 15, 16, and 17. The OBA seems to have an advantage, if the objects' complete histories — or larger parts of them — are accessed (6, 9). The TBA can take advantage of (spatio-) temporal indices in bounded spatio-temporal queries (10, 11, 12, 13, 14, 15, 16).

Query	Network Based @0.05 Response Time		Network Based @0.2 Response Time		Network Based @1.0 Response Time		Region Based @0.2 Response Time	
	OBA	TBA	OBA	TBA	OBA	TBA	OBA	TBA
Q1	0.406	0.335	0.476	0.451	0.460	0.407	0.362	0.341
Q2	0.099	0.055	0.050	0.140	0.113	0.099	0.028	0.021
Q3	2.290	0.616	6.303	0.993	12.080	1.092	7.407	0.963
Q4	76.664	49.516	625.431	273.426	6232.560	966.393	640.642	284.292
Q5	16.737	20.059	45.535	34.710	121.885	61.015	38.160	26.666
Q6	71.396	189.435	333.341	1942.071	7032.000	53910.502	193.308	430.185
Q7	92.666	35.654	2325.340	241.182	23324.700	135.724	564.038	194.097
Q8	1.209	1.214	5.854	3.521	13.989	4.308	2.110	1.403
Q9	392.336	784.329	1102.580	3241.180	4791.730	21730.800	1078.710	2020.960
Q10	681.937	221.276	3170.480	1189.130	23951.800	16410.200	3011.840	627.194
Q11	0.956	0.580	1.862	0.849	11.602	1.411	0.820	0.669
Q12	2.188	0.553	144.466	0.510	964.456	0.625	97.807	0.564
Q13	50.376	45.189	426.079	128.682	2015.680	261.572	139.375	90.711
Q14	2.129	2.020	6.444	3.083	138.305	13.075	4.909	2.873
Q15	3.662	4.530	121.011	30.562	322.635	36.343	51.973	40.674
Q16	144.565	58.967	102.139	49.206	132.165	74.842	118.154	51.944
Q17	5.129	27.393	467.125	242.219	5374.080	1097.145	69.109	194.322
Total $[s]$	1544.745	1441.721	8884.516	7381.915	74440.240	94705.553	6018.751	3967.521
$[h:m:s]$	0:25:45	0:24:02	2:28:05	2:01:01	20:40:40	26:18:26	1:40:19	1:06:07

Table 9: Benchmarking the Secondo DBMS using BerlinMOD/R: Total Response Time.

Listed are the total response times over all query instances (all combinations of query parameters) in seconds.

Query	Network Based @0.05 Response Time		Network Based @0.2 Response Time		Network Based @1.0 Response Time		Region Based @0.2 Response Time	
	OBA	TBA	OBA	TBA	OBA	TBA	OBA	TBA
Q1	0.004	0.003	0.005	0.005	0.005	0.004	0.004	0.003
Q2	0.099	0.055	0.050	0.140	0.113	0.099	0.028	0.021
Q3	0.023	0.006	0.063	0.010	0.121	0.011	0.074	0.010
Q4	0.767	0.495	6.254	2.734	62.326	9.664	6.406	2.843
Q5	0.167	0.201	0.455	0.347	1.219	0.610	0.382	0.267
Q6	71.396	189.435	333.341	1942.071	7032.000	53910.502	193.308	430.185
Q7	0.927	0.357	23.253	2.412	233.247	1.357	5.640	1.941
Q8	0.012	0.012	0.059	0.035	0.140	0.043	0.021	0.014
Q9	3.923	7.843	11.026	32.412	47.917	217.308	10.787	20.210
Q10	68.194	22.128	317.048	118.913	2395.180	1641.020	301.184	62.719
Q11	0.010	0.006	0.019	0.008	0.116	0.014	0.008	0.007
Q12	0.022	0.006	1.445	0.005	9.645	0.006	0.978	0.006
Q13	0.504	0.452	4.261	1.287	20.157	2.616	1.394	0.907
Q14	0.021	0.020	0.064	0.031	1.383	0.131	0.049	0.029
Q15	0.037	0.045	1.210	0.306	3.226	0.363	0.520	0.407
Q16	0.014	0.006	0.010	0.005	0.013	0.007	0.118	0.052
Q17	5.129	27.393	467.125	242.219	5374.080	1097.145	69.109	194.322
Total $[s]$	151.248	248.462	1165.688	2342.939	15180.887	56880.901	590.010	713.940
$[h:m:s]$	0:02:31	0:04:08	0:19:26	0:39:03	4:13:01	15:48:01	0:09:50	0:11:54

Table 10: Benchmarking the Secondo DBMS using BerlinMOD/R: Response Time per Query Instance

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

Looking at the individual response times shown in Table 10, 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 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.

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 similar 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.

Comparing the results for the network based (NB) and the region based (RB) datasets, we can observe the aforementioned differences between the OBA and TBA not only for the NB, but also for the RB data. The only significant difference is Query 17. The reason may be the different distribution of points, that is essential for that particular query.

A pairwise comparison of the NB and RB results shows, that several queries seem to be more sensitive to changes of the data used than others. Significant differences can be observed for Queries 6, 7, 13, 15 and 17 (OBA), and 6, 9 and 10 (TBA). As response times are generally smaller for the region based data, the reason seems to be within the movements themselves. While the number of units per vehicles is greater for the RB (see Table 8), there are major differences regarding the distribution of home and work nodes (see Figure 4). While for the NB data, the spatial distribution of home and work nodes looks like a 2-variate gaussian distribution centered on the centre of the network, the spatial distribution of home and work nodes for the RB data clearly shows different clusters and smaller variance. This may have resulted in significantly changed work trips and therefore different query response times.

Since we expect similar results for scale factors 0.05 and 1.0, we did not repeat the benchmark with the region based dataset at these scale factors.

9 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 BerlinMOD/R queries and 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 extend SECONDO's capabilities to perform a subset of the BerlinMOD/NN queries, and compare the performance of different representations of moving object data within the SECONDO system using the benchmark.

10 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).

We wish to thank Slaven Rezić for the permission to use the Berlin map data. We also thank the anonymous referees for their careful reviews and detailed suggestions, that have helped us to significantly improve the paper.

References

[1] ESRI Shapefile Technical Description, 1998.

[2] M. C. N. Barioni, H. Razente, A. Traina, and J. Caetano Traina. Siren: a similarity retrieval engine for complex data. In *VLDB '06: Proceedings of the 32nd International Conference on Very large Data Bases*, pages 1155–1158. VLDB Endowment, 2006.

[3] R. Benetis, C. S. Jensen, G. Karciauskas, and S. Saltenis. Nearest neighbor and reverse nearest neighbor queries for moving objects. In *IDEAS '02: Proceedings of the 2002 International Symposium on Database Engineering & Applications*, pages 44–53, Washington, DC, USA, 2002. IEEE Computer Society.

[4] BerlinMOD Benchmark Web Site. `http://dna.fernuni-hagen.de/secondo/BerlinMOD/BerlinMOD.html`, 2008.

[5] C. Böhm, B. C. Ooi, C. Plant, and Y. Yan. Efficiently processing continuous k-NN queries on data streams. In *ICDE*, pages 156–165, 2007.

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

[7] T. Brinkhoff. *Geodatenbanksysteme in Theorie und Praxis*. Wichmann Verlag, 2005.

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

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

[14] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In *Proceedings 1994 ACM SIGMOD Conference, Mineapolis, MN*, pages 419–429, 1994.

[15] 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 Iinternational Conference on Management of Data*, pages 319–330. ACM Press, 2000.

[16] 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.

[17] G. Gidófalvi and T. B. Pedersen. ST-ACTS: A spatio-temporal activity simulator. In R. A. de By and S. Nittel, editors, *GIS*, pages 155–162. ACM, 2006.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] M. Hadjieleftheriou, G. Kollios, D. Gunopulos, and V. Tsotras. Line discovery of dense areas in spatio-temporal databases, 2003.

[23] H. Hu, J. Xu, and D. L. Lee. A generic framework for monitoring continuous spatial queries over moving objects. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 479–490, New York, NY, USA, 2005. ACM.

[24] C. S. Jensen, D. Lin, B. C. Ooi, and R. Zhang. Effective density queries on continuously moving objects. In *International Conference on Data Engineering (ICDE)*, page 71, 2006.

[25] C. S. Jensen, D. Tiesyte, and N. Tradisauskas. The COST benchmark - Comparison and evaluation of spatio-temporal indexes. In *DASFAA*, pages 125–140, 2006.

[26] S.-H. Jeong, A. A. A. Fernandes, N. W. Paton, and T. Griffiths. A generic algorithmic framework for aggregation of spatio-temporal data. In *SSDBM '04: Proceedings of the 16th International Conference on Scientific and Statistical Database Management*, page 245, Washington, DC, USA, 2004. IEEE Computer Society.

[27] F. Korn and S. Muthukrishnan. Influence sets based on reverse nearest neighbor queries. *SIGMOD Rec.*, 29(2):201–212, 2000.

[28] D. Krajzewicz, G. Hertkorn, C. Rössel, and P. Wagner. SUMO (Simulation of Urban MObility); An open-source traffic simulation. In *Proceedings of the 4th Middle East Symposium on Simulation and Modelling (MESM2002)*, pages 183–187. SCS European Publishing House, 2002.

[29] C. Lang and A. Singh. A framework for accelerating high-dimensional NN-queries, 2001.

[30] J. A. C. Lema and T. Behr. External representation of spatial and spatio-temporal values. `http://dna.fernuni-hagen.de/Secondo.html/files/SpatialListFormat.pdf`, 2004.

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

[32] K. Mouratidis and D. Papadias. Continuous nearest neighbor queries over sliding windows. *IEEE Transactions on Knowledge and Data Engineering*, 19(6):789–803, 2007.

[33] K. Mouratidis, D. Papadias, and M. Hadjieleftheriou. Conceptual partitioning: an efficient method for continuous nearest neighbor monitoring. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 634–645, New York, NY, USA, 2005. ACM.

[34] J. Myllymaki and J. Kaufman. Dynamark: A benchmark for dynamic spatial indexing. In *MDM '03: Proceedings of the 4th International Conference on Mobile Data Management*, pages 92–105, London, UK, 2003. Springer-Verlag.

[35] D. Papadias, Y. Tao, K. Mouratidis, and C. K. Hui. Aggregate nearest neighbor queries in spatial databases. *ACM Trans. Database Syst.*, 30(2):529–576, 2005.

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

[37] S. Rezić. `http://bbbike.de`, 2008.

[38] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 71–79, New York, NY, USA, 1995. ACM.

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

[40] Secondo Web Site. `http://dna.fernuni-hagen.de/Secondo.html/index.html`, 2008.

[41] A. Singh, H. Ferhatosmanoglu, and A. Tosun. High dimensional reverse nearest neighbor queries, 2003.

[42] 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.

[43] SMARTEST. Final Report for Publication. Technical report, European Commission Transport RTD Programme of the 4th Framework Programme, 1999. Project Reference: RO-97-SC.1059. `http://www.its.leeds.ac.uk/projects/smartest/finrep.PDF`.

[44] SMARTEST Project Web Site. `http://www.its.leeds.ac.uk/projects/smartest/`, 1999.

[45] Z. Song and N. Roussopoulos. K-nearest neighbor search for moving query point. In *SSTD '01: Proceedings of the 7th International Symposium on Advances in Spatial and Temporal Databases*, pages 79–96, London, UK, 2001. Springer-Verlag.

[46] I. Stanoi, D. Agrawal, and A. E. Abbadi. Reverse nearest neighbor queries for dynamic databases. In *ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, pages 44–53, 2000.

[47] Statistisches Landesamt Berlin. Bevölkerungsstand in Berlin Ende September 2006 nach Bezirken, 2008.

[48] Statistisches Landesamt Berlin. Interaktiver Stadtatlas: `http://www.statistik-berlin.de/framesets/berl.htm`, 2008.

[49] M. Stonebraker, J. Frew, K. Gardels, and J. Meredith. The Sequoia 2000 Benchmark. In *SIGMOD Conference*, pages 2–11, 1993.

[50] Y. Tao, D. Papadias, and X. Lian. Reverse knn search in arbitrary dimensionality. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 744–755. VLDB Endowment, 2004.

[51] Y. Theodoridis. Ten benchmark database queries for location-based services. *Comput. J.*, 46(6):713–725, 2003.

[52] Y. Theodoridis, J. R. O. Silva, and M. A. Nascimento. On the generation of spatiotemporal datasets. In *SSD*, pages 147–164, 1999.

[53] T. Tzouramanis, M. Vassilakopoulos, and Y. Manolopoulos. the generation of time-evolving regional data, 2002.

[54] T. Tzouramanis, M. Vassilakopoulos, and Y. Manolopoulos. Benchmarking access methods for time-evolving regional data. *Data Knowl. Eng.*, 49(3):243–286, 2004.

[55] 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.

[56] 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.

[57] 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.

[58] 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.

[59] 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.