Network Data Model and BerlinMOD Benchmark

Simone Jandt

Last Update: April 22, 2010

Abstract

In the past several data models for the representation of histories of spatio-temporal data objects have been developed. Among others we can categorize these data models into data models for objects moving freely in the two dimensional space and data models for network constrained moving objects. In this paper we select two representatives, one for each data model category, which are both implemented in the SECONDO DBMS, and compare their capabilities with the BerlinMOD Benchmark. We describe the translation from the BerlinMOD Benchmark into the network constrained data model and show that in our experiments the network constrained data model outperforms the data model of free movement in the two dimensional space by orders of magnitudes.

1 Introduction

In the past several data models for the representation of spatial and spatio-temporal data objects have been developed. Among others we can categorize them into data models for objects moving freely in two dimensional space and data models for network constrained moving objects. For both categories several different data models have been presented like [1, 10] for spatio-temporal data objects moving freely in two dimensional space and [11, 17, 21] for spatio-temporal data objects that are constrained by given networks, to name just a few. Objects which are restricted to use existing networks, like cars are restricted to use road networks, can be represented as moving point object in both data models, while objects, which are not restricted by a given network, like people, can be represented as moving point object only in a data models of free movement in space.

But if everything can be represented by a data model of free movement space why do we spend time on network data models? Now, our experiments presented in this paper show that the network data model outperforms the data model of free movement in two dimensional space significantly. The network data model uses less than 60% of the storage space of the data model of free movement in space, mostly caused by the fact that the number of units for the moving point objects in the network data model is less than 50% of the number of units in the data model of free movement in two dimensional space. The total run time of the BerlinMOD Benchmark for the network data model is less than 50% of the total run time of the data model of free movement in two dimensional space. We think that this results show that it is useful to develop specialised data models for specialised data structures like network data models for network constrained moving objects to save storage space and reduce query run times.

For this paper we choosed two representatives one for each data model category. Both representatives are available in the SECONDO DBMS and use the same temporal representation. So we can exclude that different DBMS or temporal representation issues bias the results of our data model comparison.

The network constrained data model is represented by the network data model presented in [11] and the data model of free movement in two dimensional space is represented by the data model presented in [10].

For us it seems logical to use the BerlinMOD Benchmark [3] which is also available in the Secondo DBMS to compare the capabilities of the both data models. The BerlinMOD Benchmark is best to our knowledge the first benchmark for complete spatio-temporal database systems and it is developed in the Secondo DBMS. Furthermore the data generated by the BerlinMOD Benchmark data generator is restricted to the streets of the German capital Berlin, such that it can be translated into a network constrained environment. And not at last the data model used in the BerlinMOD Benchmark is the data model of free movement in two dimensional space that we use for our comparison. So we only have to translate the spatial and spatio-temporal data types of the BerlinMOD Benchmark once into our network data model representation. This simplifies the control of the query results and avoids errors caused by translation errors.

The translation of the spatial and spatio-temporal data types of the BerlinMOD Benchmark data into the network data model representation described in this paper can be seen as example for the usage of the BerlinMOD Benchmark with other compatible data representations or DBMS.

The rest of the paper is organised as follows: In section 2 we give a short reminder of the underlying Secondo DBMS (2.1), the BerlinMOD Benchmark (2.2) and the two data models (2.3 and 2.4) we choose as representatives for our comparison. The translation of the BerlinMOD Benchmark data and query set in the network data model representation is described in section 3. The resulting experimental benchmark setup is described in section 4 followed by the results of our experiments in section 5. We conclude our work in section 6.

2 Related Work

In the past other data models for free movement in the space [1, 10] and for network constrained movement have been presented [11, 17, 21]. As well there are some more benchmarks [14, 20] and database systems for spatial and spatio-temporal data types [12, 15]. But best to our knowledge they don't actually provide a combination of different implemented and supported data models together with a existing benchmark feasible for moving objects in space and network constrained moving objects like the actual Secondo DBMS. So it is self-evident for us to use the Secondo DBMS in combination with the provided data models and the BerlinMOD Benchmark to compare the capabilities of the network constrained data model and the data model of free movement in two dimensional space provided with the Secondo DBMS.

In the next subsections we give short reminders of the Secondo database system (2.1), the BerlinMOD Benchmark (2.2), and the both data models (2.3, 2.4) we used in our experiments.

2.1 Secondo

The extensible Secondo DBMS presented in [6, 12] provides a platform for implementing various kinds of data models. It provides a clean interface between the data model independent system frame and the content of the single data models. Hence Secondo can be easily extended by the user implementing algebra modules to introduce new data types and operations on this data types. The user may define additional viewers for the graphical user interface or write additional optimization rules or cost functions to extend the the optimizer. Since Secondo version 2.9 the user may publish his extensions as Secondo plugin such that other users can use this plugin to extend their own Secondo system to use the new provided functionalities or repeat the published experiments. Secondo is free available in the web [9] and comes with a number of already implemented spatial and spatio-temporal data types and operations including the spatio-temporal data model of free movement in the two dimensional space 2.3 and the network data model 2.4 used in this paper. Furthermore the BerlinMOD Benchmark described in 2.2 has been developed in the SECONDO DBMS. For our experiments we used the SECONDO version 2.9.StableNewFlob.

2.2 BerlinMOD Benchmark

The BerlinMOD Benchmark was presented in [3] and the provided scripts for the data generator are implemented as SECONDO DBMS operations. The BerlinMOD Benchmark is available in the web [8] and provides a well defined data-set and queries for the experimental evaluation of the capabilities of spatial and spatio-temporal database systems dealing with histories of moving objects. The BerlinMOD Benchmark emphasises the development of complete systems and simplifies experimental repeatability pointing out the weakness and the potency's of the benchmarked systems.

The data-sets of the BerlinMOD Benchmark are created using the street map of the German capital Berlin [16] and statistical data about the regions of Berlin [18, 19] as input relations. The created moving objects represent cars driving in the streets of Berlin, simulating the behaviour of people living and working in Berlin. Every moving object has a home node and a work node. Every weekday each car will do a trip from the home node to work the work node in the morning and vice versa in the late afternoon. Beside this randomly chosen cars will make additional trips in the evening and up to six times at the weekend to randomly chosen targets in Berlin and back home. The BerlinMOD Benchmark uses the data model of free movement in two dimensional space described in section 2.3. Because the BerlinMOD Benchmark generates all data sets restricted to the street map of Berlin the BerlinMOD Benchmark can also be used for network constrained data models, if the spatial and spatio-temporal data types are translated into a corresponding network data model, like we did for our experiments.

The number of observed cars and the duration of the observation period can be influenced by the user setting the $scalefactor$ to different values in the data generation script of the BerlinMOD Benchmark. For example at $scalefactor$ 1.0 the data generator creates 2000 moving point objects observed for 28 days. Each of them sending a GPS-signal every 2 seconds. This simulated signals are simplified such that time intervals when a car doesn't move or moves in the same direction at the same speed are merged into one single time interval. For example: If a car parks in front of the work node for 8 hours there will be only one entry in the history of the cars movement with a time interval of 8 hours instead of 14.400 entries one for each GPS time interval.

The BerlinMOD Benchmark provides two different approaches to store the histories of moving objects. On the one hand the object-based approach (OBA) and on the other hand the trip based approach (TBA).

In the OBA the complete history for each moving object is kept together into one single entry. There is only one relation *dataScar* containing one tuple for each object consisting of the spatio-temporal data of the object *journey*, the *licence*, the *type*, and the *model* of the object.

In the TBA we have two relations *dataMcar* and *dataMtrip*. *dataMcar* contains the static data for each object like *licence*, *type*, and *model* together with an object identifier *moid*. *dataMtrip* contains for each *moid* several tuple each of them containing all units of a single trip of the moving object or a single unit for a longer stop. For example each time the car drives from home node to work node is a single trip and each time the car parks in front of the office is also a single trip.

Beside the moving point objects the BerlinMOD Benchmark provides several data sets each of them containing 100 pseudo randomly generated data objects which are used in the benchmark queries. Table 1 gives an overview of this query objects. The BerlinMOD Benchmark deals also with subsets from this query object sets consisting of the first or second 10 query objects of a query object set. They are labeled by the name of the query object set followed by a 1 for the first 10 or a 2 for the second 10 query objects of the query object set.

Name of Data Set	**Tuple Content**
QueryPoints	Object identifier and *point* values.
QueryRegions	Object identifier and *region* value.
QueryInstants	Object identifier and time stamps.
QueryPeriods	Object identifier and space of time.
QueryLicences	Object identifier and a *string* representing a licence value.

Table 1: Query Object Relations of BerlinMOD Benchmark

The BerlinMOD Benchmark provides two sets of queries BerlinMOD/R and BerlinMOD/NN. BerlinMOD/R addresses range queries and BerlinMOD/NN nearest neighbour queries. In this paper we will focus on the range queries, which are the main aspect of the BerlinMOD Benchmark up to now.

The query set BerlinMOD/R includes 17 queries selected of the set of possible combinations of the 5 aspects:

- known or unknown object identity,
- standard, spatial, temporal, or spatio-temporal dimension,
- point, range, or unbounded query interval,
- single object or object relations condition type,
- with or without aggregation.

We will present the 17 queries in more detail in section 3.4 together with our network data model algorithms for this queries.

2.3 Data Model of Free Movement in two dimensional space

The data model used by the BerlinMOD Benchmark is the same data model of free movement in two dimensional space presented in [7, 10, 13]. Spatial positions are assumed to be located into a two dimensional space. A single spatial position is represented by the data type *point*. A *point* consists of a pair of *real* values interpreted as x,y-coordinates in the assumed two dimensional plane.

Streets are represented by *line* values. A *line* value consists of a set of *halfsegments* representing the geometry of the line in the two dimensional space. Each *halfsegment* consists of two *point* values and a Boolean flag telling us if the left or the right point is the dominating point of the *halfsegment*.

Regions are represented by the data type *region*. A *region* consists of a set of *halfsegments* defining the outer border of the region in the two dimensional space. If a region contains wholes the inner border is also formed by the *halfsegments*.

In Secondo all this spatial data types and many standard data types can be lifted to become time dependent *moving* values. For all data types α the constructor *moving* creates a new data type *moving*(α) (short form *m*α).

A car may be represented by a *moving*(*point*) short *mpoint*. A *mpoint* consist of a set of units called *unit*(*point*) (short form *upoint*). Each *upoint* consists of a time interval and two *point* values. The first *point* value represents the position of the *mpoint* at the start of the time interval and the second <u>*point*</u> value represents the position of the <u>*mpoint*</u> at the end of the time interval. It is assumed that the object represented by the <u>*mpoint*</u> moves on the straight line between this two points with constant speed within the given time interval. The velocity of the object is given by the ratio from the distance of the two points and the length of the time interval of the unit. All units of a <u>*mpoint*</u> must have disjoint time intervals, because a car cannot be at two different positions at the same time. The units are sorted by ascending time intervals.

This spatio-temporal data model of <u>*moving*</u> allows us to compute the position of a <u>*mpoint*</u> at every time instant within its definition time. We can also compute the time instant the point passed a given position assumed the <u>*mpoint*</u> ever passes this position. The position of a <u>*point*</u> at a given time instant is represented by a <u>*intime*</u>(<u>*point*</u>) (short form <u>*ipoint*</u>). A <u>*ipoint*</u> consists of a time instant and a <u>*point*</u> value.

Some other data types of SECONDO which are used in the BerlinMOD Benchmark are shown in table 2.

Data Type	**Description**
<u>*bool*</u>	Usual boolean data type.
<u>*int*</u>	Usual integer number.
<u>*real*</u>	Usual real number.
<u>*instant*</u>	A point in time.
<u>*periods*</u>	A set of disjoint and not connected spaces of time.
<u>*mbool*</u>	A time dependent boolean value, which will be constant *TRUE* or *FALSE* within each <u>*ubool*</u>
<u>*mreal*</u>	Time dependent real number. Each unit will be defined by a function of time representing the <u>*real*</u> value at each time instant.

Table 2: Other Data Types of BerlinMOD Benchmark

2.4 Network Data Model

The central idea of the network data model presented in [11] is that every movement is constrained by a given network and every position can be described relative to this network. The data type <u>*network*</u> is the central data type in the network data model. All other data types of the network data model are related to a given <u>*network*</u> object by the unique network identifier that is part of each <u>*network*</u> object.

The <u>*network*</u> object contains all spatial information of the represented network. The <u>*network*</u> consists of three main relations (*routes*, *junctions*, and *sections*), two arrays providing fast access to adjacent network sections, and some B-Tree and R-Tree indexes to supporting faster access to the main relations.

The relation *routes* of the <u>*network*</u> contains the attributes of the streets like *id*, *routecurve*, *routelength*, and two Boolean flags. The first flag indicates if the route starts at the lexicographically smaller end point or not. The second flag indicates if the lanes of the street are separated like on German Highways or not.

The *junctions* relation of the <u>*network*</u> contains all attributes of the street crossings like the two route identifiers of the first and second street meeting at the crossing, the distance of the crossing from the start of the first respectively second street, tuple identifiers of the both streets in the routes relation, tuple identifiers of the sections connected by this junction in the sections relation, and a connectivity code telling us which lanes of the two streets are connected by the crossing.

The *sections* relation of the <u>*network*</u> object contains the attributes of the street parts between two crossings or a crossing and the end of the street. This are the route identifier of the street the section belongs to, the tuple identifier of this street in the routes relation, the start position and end position of the section on the street, the section curve, and again two Boolean flags with the same meaning as in the routes relation.

Furthermore there are two arrays in the network object providing a fast access from each section to their adjacent sections with respect to the driving direction. Two sections are adjacent if their lanes are connected by a junction.

We created four B-Tree indexes for the route identifier attributes in the *routes*, *junctions* and *sections* relation, and a R-Tree index over the curve attribute of the *routes* relation to support faster execution of operations dealing with the relations of the network object.

The data type gpoint represents single positions in a given network. Besides the network identifier a gpoint consists of a route identifier, a distance from the start of the route to the position of the gpoint and a *side* value (*up*, *down*, *none*) telling us if the position is reachable from the *up* or the *down* side of the route in case of separated lanes. For simple streets or positions which are reachable from both sides of the route the side value is always *none*.

Parts of the network, regardless if they represent paths or regions, are given as gline values. Besides the network identifier a gline consists of a set of *routeinterval*s, and two Boolean flags. The Boolean flags tell us if the gline is defined and if the set of *routeinterval*s is sorted.

Each *routeinterval* consists of a route identifier identifying the route the route interval belongs to, and the start and the end position from the route interval on this route [1]. We call a set of *routeinterval*s sorted if the following conditions are fullfilled:

- all *routeinterval*s are disjoint
- the *routeinterval*s are stored in ascending order of their route identifiers
- if two disjoint *routeinterval*s have the same route identifier the *routeinterval* with the smaller start position is stored first
- for all *routeinterval*s the $startposition \leq endposition$

Many algorithms take profit from sorted gline values. For example: If n is the number of the route intervals in a gline the decision, if a gpoint is inside the gline needs $O(n)$ time for unsorted and $O(\log n)$ time for sorted gline values,

Unfortunately not all gline values can be stored sorted. If a gline value represents a path between two gpoint in the network, we need the route intervals exactly in the sequence they are used in the path. This will nearly never be a sorted set like defined before. We store gline values sorted whenever this is possible to support faster query execution, and introduced a Boolean sorted flag. Every algorithm which deals with gline values checks this flag and uses the corresponding code.

Mostly similar to the mpoint of the other data model there is a mgpoint in the network data model. A mgpoint consists of a set of ugpoint with disjoint time intervals. Each ugpoint consists of a time interval and two gpoint values. Every time the mgpoint changes the route or the speed a new ugpoint is written. Each ugpoint is assumed to follow the same route from the start to the end position at the same speed. So accordingly to the mpoint we can compute the network position of the mgpoint at every time instant within the definition time of the mgpoint as intime(gpoint).

In deviation from the original network data model we extended the implementation of the mgpoint with four additional attributes to support faster query execution:

1. The total driven distance
2. A sorted set of *routeinterval*s representing the positions ever traversed by the mgpoint
3. A Boolean defined flag for the set of *routeintervals*

[1] In the original paper the *routeinterval* includes a *side* value analogous to the gpoint. But this parameter is not part of the implementation yet.

4. A spatio-temporal minimum bounding box

The sorted set of *routeinterval*s was introduced, because analogous to sorted gline values it makes it much faster to decide if a mgpoint ever passed a given network position or not. Instead of a linear check of all m ugpoints of a mgpoint we can perform a binary scan on the much lower number r of the passed *routeinterval*s. This reduces the time complexity from $O(m)$ to $O(\log r)$ for the **passes** operation. Logically the set of *routeinterval*s should be a sorted gline value but the SECONDO DBMS restricts us to use a sorted set of *routeinterval*s instead.

The spatio-temporal minimum bounding box was introduced as parameter to the mgpoint because the computation of this value is very expensive in the network data model. Although each unit of a mgpoint stays on the same route at same speed it may follow different spatial directions. For example a route may lead uphill in serpentine. A spatial bounding box only computed from the spatial start and end position may not enclose all spatial positions of the car within the unit. Therefore we have always to examine the spatial dimensions of the *routeinterval* passed within a unit to compute the units bounding box. This needs a access to the route curve in the *routes* relation of the corresponding network object. If r is the number of routes of the network and h the number of $halfsegments$ belonging to the *routeinterval* passed in a unit we need $O(h + \log r)$ time to compute the bounding box for a single unit. The bounding box of the mgpoint is the union of the bounding boxes of its m units. So the computation of the spatio-temporal respectively spatial bounding box of a mgpoint needs $O(m(h + \log r))$ time. This very expensive computation is only done on demand or if we can get the bounding box for free. For example we can copy the bounding box of a mpoint at the translation time into a mgpoint without big computational costs. The bounding box attribute is not maintained. If the mgpoint value changes the bounding box attribute is set to be undefined until recomputing is necessary.

3 Translation of BerlinMOD into Network Data Model

In this section we describe the creation of the network object from the *streets* value of the BerlinMOD Benchmark in section 3.1. Followed by the description how to use this new created network value as reference for the translation of all spatial and spatio-temporal data type objects of the BerlinMOD Benchmark into the network data model representation in section 3.2. In section 3.3 we describe the indexes we build on the network data model representation to support faster query execution. We close this part with section 3.4 were we describe the executable SECONDO queries for the network data model representation of the BerlinMOD Benchmark.

Executable SECONDO scripts for the network and index creation, object translation, and the network benchmark queries can be downloaded from our website [8].

3.1 Create Network Object

The network object *net* is created by extracting the *routes* data from the *streets* object that is created by the BerlinMOD Data Generator. The extracted data r is used to compute the crossings of the routes of Berlin j. The data source lacks on information about the connectivity of the street crossings, such that we use the maximum value for the connectivity code of each crossing as default value in this step. Now we can use r and j as input relations for the operator **thenetwork** to create our network object *net* representing the streets of Berlin in the network data model representation of the BerlinMOD Benchmark.

The network creation algorithm first copies all tuple of r to the *routes* relation of *net* and creates the B-Tree index of the route identifiers and the R-Tree index of the route curves of the *routes* relation of *net*. Then all tuple of j are copied to the *junctions* relation of *net* and the tuple identifiers for the both routes connected by this junction are added to the junctions entry. After that we build two B-Trees indexing the route identifiers of the first respectively second route in the *junctions* relation. Next for every route of the *routes* relation all junctions on this route are taken from the *junctions* relation to compute the up and down sections for each of this junctions on the route. The up and down sections are inserted into the *sections* relation of *net* and the tuple identifiers of the sections are added to the entry of the corresponding junction in the *junctions* relation. After that the B-Tree index for the route identifiers in the *sections* relation is created and the adjacency lists of *net* are filled with the adjacent section pairs defined by the *junctions* relation.

If $|r|$ is the number of routes and $|j|$ is the number of junctions. The algorithm needs $O(|r| \log |r|)$ time to copy r to the *routes* relation of *net* and create the indexes of the *routes* relation. The creation of the *junctions* relation and the build of the B-Trees indexes takes $O(|j| \log |j|)$ time. $O(|r||j|)$ time is needed to fill the *sections* relation and $O(|j|)$ time to fill the *adjacencylists* of *net*. Altogether the complete algorithm needs: $O(|r| \log |r| + |j| \log |j| + |r||j|)$ time to create the *net* from the two input relations r and j.

3.2 Translate Spatial and Spatio-Temporal Data Types

In this section we describe the translation of the spatial and spatio-temporal data types of the BerlinMOD Benchmark data set into network respectively network-temporal objects. All translations are done relative to the <u>*network*</u> object *net* that we described in section 3.1. All algorithms in this section get a spatial respectively spatio-temporal BerlinMOD Benchmark data type object and the <u>*network*</u> object *net* as input they return the corresponding network data model object respectively an undefined network data model object if the input data object is not constrained by *net*.

3.2.1 Translate <u>*point*</u> into <u>*gpoint*</u>

The **point2gpoint** operation translates a <u>*point*</u> value p into a corresponding <u>*gpoint*</u> value gp if possible. This operation is also included in the other translation operations. The algorithm uses the R-Tree index of the *routes* relation of *net* to select the route curve closest to p and computes the position of p on this route curve. In case of the BerlinMOD Benchmark the *side* value of gp is always set to *none*, because the BerlinMOD Benchmark does not differentiate between the different sides of a street.

If r is the number of routes in the *routes* relation and k is the number of possible candidate routes the worst case complexity of the algorithm is $O(\log r + k)$.

This should be all to translate the <u>*point*</u> values of the $QueryPoints$ relation of the BerlinMOD Benchmark into network query positions. But there is a problem with the network data model representation of junctions. In the network data model contrary to the data model of free movement in the two dimensional space each junction has more than one <u>*gpoint*</u> representation, because each junction is related to two or more routes. Hence if a junction position is given related to route a we won't detect the junction as passed if a <u>*mgpoint*</u> object passes the junction on route b in all cases, because the definition of **passes** in the network data model is slightly different from the **passes** operation in the BerlinMOD Benchmark data model. Unfortunately all query points of the BerlinMOD Benchmark are junctions. To make the results comparable we added a operator **polygpoints**, which returns for every input <u>*gpoint*</u> value gp a stream of <u>*gpoint*</u> values. If gp represents a junction we return all <u>*gpoint*</u> values representing the same junction in *net*, otherwise we return only gp in the stream. So we got 221 query <u>*gpoint*</u> values in $QueryPointsNet$ for the 100 query <u>*point*</u> values in $QueryPoints$ and 22 <u>*gpoint*</u> values in $QueryPoints1Net$ for the 10 <u>*point*</u>

values of $QueryPoints1$ of the BerlinMOD Benchmark. This means we have always to compute the results for the doubled number of query points in our network data model than in the data model of free movement in space.

3.2.2 Translate *mpoint* into *mgpoint*

The second operation **mpoint2mgpoint** translates a *mpoint* value s into a *mgpoint* value t. The main idea of the algorithm is to use the continuous movement of s to reduce computation time. We initialize the algorithm by reading the first unit of s and use the **point2gpoint** algorithm to find a route in the network containing the *start* and the *endpoint* of this unit. We initialize the first unit of t with the computed network values. Then we read the next unit of s and try to find the *endpoint* of the new unit on the same route the last unit of s was found. If the *endpoint* is found on the same route we check the moving direction on the route and speed of the point in the unit. If they are equal to the actual unit we extend the actual unit of t to enclose the value of the actual unit of s. If the speed or the moving direction on the route changes we write the actual unit to t and initialize a new unit for t with the network values of the actual unit from s. If the *endpoint* can't be found on the same route than the last unit from s we write the actual unit of t and start a search on the route curves of the adjacent sections to find the route curve that contains the *start* and the *endpoint* of the actual unit of s. We initialize a new unit for t with the estimated network values for the actual unit of s and continue with the next unit of s. At last we add the actual network unit to t.

The time complexity to find the start values for the first unit is O(**point2gpoint**). For the next m units of s the time complexity is O(1) if s don't change the route. And O(a) if the end point is on another route and a is the maximum number of adjacent sections. So we get a worst case time complexity of O(O(**point2gpoint**) + ma) for the translation of a *mpoint* s into a *mgpoint* t.

3.2.3 Translate *region* into *gline*

The translation of the *region* values in the $QueryRegions$ relation of the BerlinMOD Benchmark into *gline* values of our network data model is done in several steps. First of all we build a single big *line* object containing all network streets. Then we compute for each *region* of the $QueryRegions$ the intersection with this big *line* object. At last we translate the resulting *line* objects of the intersection, each representing one *region* of the $QueryRegions$ relation, into sorted *gline* values using the **line2gline** operation.

The algorithm of the **line2gline** operation takes each $halfsegment$ of a *line* value and computes a corresponding network *routeinterval* by searching a common *routecurve* for the *start* and the *endpoint* of the $halfsegment$ using the **point2gpoint** operation. The computed *routeinterval*s are sorted, merged and compressed before the resulting *gline* value is returned. If the number of $halfsegment$s of a *line* value is h and the number of resulting compressed *routeinterval*s is r we get a time complexity of O(hO(**point2gpoint**)+$h \log r + r$) for the whole algorithm. Whereby the summand $h \log r + r$ is caused by the compressing and sorting of the resulting *gline* but as mentioned before in 2.4 we think this time is well invested, because it is needed once and the sorted *gline* value is used many times.

3.3 Create Indexes on Network Data Model

For the use with the BerlinMOD Benchmark we created the following indexes on the network data model representation of the BerlinMOD Benchmark data sets:

- B-Tree indexes for the *licences* and *moid* attributes of the relations $dataSNcar$, $dataMcar$, and $dataMNtrip$. This indexes are similar to the indexes created in the BerlinMOD Benchmark for $dataSCcar$, $dataMCcar$, and $dataMCtrip$, because the relations $dataSNcar$, $dataMcar$ and $dataMNtrip$ contain the network data model representation of the $dataScar$, $dataMcar$ and $dataMtrip$ relation of the BerlinMOD Benchmark. We don't explain them in more detail.

- An R-Tree index of the spatio-temporal bounding boxes of the $\underline{mgpoint}$ attributes in the $dataMNtrip$ and the $dataSNcar$ relation. Different from the data model that uses the spatio-temporal units for the spatio-temporal indexes we used only the big bounding boxes of the whole trips instead of much more small bounding boxes for each single unit like it is done in the data model of free movement in two dimensional space.

- For every unit of each $\underline{mgpoint}$ we build a three dimensional *netbox* and for every *routeinterval* of every $\underline{mgpoint}$s trajectory a two dimensional *netbox*. This *netboxes* are used to create R-Trees indexing the network and network-temporal positions of the $\underline{mgpoint}$s. A temporal-network bounding box is a degenerated three dimensional rectangle. The coordinates are defined to be $x_1 = x_2 = routeidentifier$ as $\underline{real}$ value (The equality of x_1 and x_2 makes the degeneration.), $y_1 = \min(starposition, endposition)$, $y_2 = \max(startposition, endposition)$, and, $z_1 = starttime$ as $\underline{real}$ value and $z_2 = endtime$ as $\underline{real}$ value. The network bounding boxes are defined to be degenerated two dimensional rectangles with x,y-coordinates analogous to the temporal-network boxes.

3.4 Translate Benchmark Queries

We developed executable SECONDO queries for each of the 17 BerlinMOD/R queries for the object based approach (OBA) and the trip based approach (TBA) using our network indexes to support faster query execution. The SECONDO optimizer is not able to optimize SQL-queries on network data model objects yet, so we tested in our experiments many different query formulations for each query to get optimal queries delivering the correct result in a minimum of time.

The limited space does not allow us to show all our executable SECONDO queries for the network data model in detail, but they can be downloaded as SECONDO scripts from our web page. At this place we give only a short overview over the BerlinMOD Benchmark queries and their network data model algorithms.

Every time we need a licence in the result or have a query licence number we need a additional step in the TBA. Because we have to join the $dataMNtrip$ and $dataMcar$ relation using the *moid* attribute and the corresponding B-Tree indexes. We will not repeat this step at every single TBA query description.

Query 1 asks for the models of the cars with licence plate numbers from $QueryLicences$, and query 2 for the number of vehicles that are "passenger" cars. Both queries deal only with standard attributes so we only changed the relation names and the B-Tree indexes to match the network data model representation.

Query 3 searches for the positions of the ten cars from $QueryLicence1$ at the ten time instants from $QueryInstants1$. We use the licence B-Tree to select the ten cars and computes the positions of these ten cars for each of the ten time instants from $QueryInstants1$ if the time instant is inside the definition time of the trip.

In Query 4 asks for the licence numbers of the cars that passed the points from $QueryPointsNet$. We create a *netbox* for each $\underline{gpoint}$ in $QueryPointNet$ and use our specialised *netbox* R-Tree of the $\underline{mgpoint}$ *routeintervals*s to select the vehicles passing the given query points.

The queries 5, 6, and 10 deal with Euclidean Distance values, which are not very useful in network environments. In networks everything is constrained by the network and regularly the network distances are computed instead of Euclidean Distances. We decided to retranslate intermediate results into spatial respectively spatio-temporal objects and use the existing Euclidean Distance operation to compute the distances between this objects to make the results comparable.

Query 5 asks for the minimum distance between places where vehicles with licences from $QueryLicence1$ and $QueryLicence2$ have been. We select the cars with licence plate numbers from $QueryLicence1$ respectively $QueryLicences2$ using the B-Tree over the *Licence* attribute of $dataSNcar$ relation. In the TBA the resulting trajectories for each car are aggregated into one single trajectory for each car. In both approaches we create a *line* value for each resulting (aggregated) trajectory value of the *mgpoint*s and compute the Euclidean Distance between this *line* values for each pair of licences one from $QueryLicences1$ and one from $QueryLicences2$.

Query 6 asks for the pairs of licences from "trucks" that have been as close as 10m or less to each other. We filter $dataSNcar$ relation, respectively $dataMcar$ relation to select the "trucks" and compute the spatio-temporal bounding box of each trip of a "truck". We extend the spatial dimensions of the bounding boxes by 5m in each spatial direction and retranslate the *mgpoint* values into *mpoint* values in a first step. In a second step we compute the **symmjoin** of the results from step one with itself using the intersection of the bounding boxes as join criteria. We filter the result to include all licence pairs of "trucks" that had sometimes a distance lower than 10m. In the TBA we additionally remove the duplicate licence pairs from the result.

Query 7 asks for the licence plate numbers of the "passenger" cars that reached the points from $QueryPoints$ first of all "passenger" cars during the observation period. The first step to solve query 7 in the network data model is equal to query 4. In a filter step we remove all "not passenger" cars from the first intermediate result. We compute for each remaining candidate trip the times the trip reaches first the query positions. We group the resulting time instants by the identifiers of the query positions and compute the minimum time stamp of each group, which is in fact the first time the query position was reached by a car. In a last step the licences of the "passenger" cars reaching the query positions at this first time instant are computed using the specialised network-temporal index of the network data model.

Query 8 computes the overall travelled distances of the vehicles from $QueryLicence1$ within the periods from $QueryPeriods1$. We select the candidate cars using the licence B-Tree, restrict the trips to the query periods and return the lengths of the trips in the OBA. In the TBA we have to sum up the length of the different trips driven by a single car within each query period.

Query 9 asks for the longest distance travelled by a single vehicle during each of the periods from $QueryPeriods1$. We restrict all trips to the periods, compute the driven distances and select the maximum length for each query periods value. Again we have to do a additional aggregation of the distances driven from the same car in the same period in the TBA.

Query 10 asks when and where vehicles with licences from $QueryLicence1$ meet which other vehicles (distance less than 3m). In the OBA we first retranslate every *mgpoint* value of $dataSNCar$ into a *mpoint* value and extend the spatial bounding box of each of this trips by 1.5 m in every spatial direction. After that we select the ten candidate trips given by $QueryLicences1$, retranslate them and extend their spatial bounding boxes in the same way. We join all trips from the first two steps where the extended bounding boxes intersect and filter the candidate pairs that have different licences and their distance is sometimes less than 3m to each other. We compute the position of the *mgpoint* at the times the distance between the remaining candidate pairs of *mpoint* was less than 3 m and return the licence pairs and the network positions of the first car when it has been closer than 3 m to the other one.

In the TBA we select the trips given by $QueryLicences1$ from $dataMNtrip$, retranslate them into *mpoint* values, and extend their spatio-temporal bounding boxes by 3m in each spatial direction. After that we use the spatio-temporal index of $dataMNtrip$ to select for each trip of the ten cars, the cars of $dataMNtrip$ which spatio-temporal bounding boxes intersect the extended spatio-temporal bounding boxes built before. For every pair of candidate trips we retranslate the second trip and use the Euclidean Distance function for *mpoint* values to determine the times when the both *mgpoint* had a distance less than 3m. At last we restrict the trip of the query *mgpoint* to this times and aggregate the resulting trips into one single trip for each licence pair.

In our experiments we tried out several indexes to support a faster query execution of query 10 including the MON-Tree presented in [4]. The MON-Tree showed very good CPU times but never the less the total run time was very high. In the end the simple form described above showed the best complete run time performance of all indexes.

Query 11 asks for the vehicles that passed a point from $QueryPoints1Net$ at one of the time instants from $QueryInstants1$. We build a network-temporal query box from the $QueryInstant1$ and $QueryPoints1Net$ relation and use the network-temporal index on $dataSNcar$, respectively $dataMNtrip$, to select the resulting trips.

Query 12 asks for the vehicles that met at a point from $QueryPoints1Net$ at an time instant from $QueryInstants1$. The first step of query 12 is identical with query 11. In a second step the Cartesian Product of the result of the first step with itself is computed and filtered for vehicles which have been at the same query point at the same query time instant.

Query 13 asks for the vehicles which travelled within one of the regions from $QueryRegions1Net$ during the periods from $QueryPeriods1$. We restrict the trips to the query regions and check if the restricted trips are defined within the query periods. In TBA possible duplicate licence pairs have to be removed and the resulting *moid*s must be mapped to the licences of the cars to generate the result using the B-Tree *moid* index of $dataMcar$.

Query 14 asks for the vehicles that have been in one of the regions from $QueryRegions1Net$ at a time instant from $QueryInstants1$. We build temporal-netboxes from the query objects to select candidate trips using the temporal-network position index. We refine the result filtering the candidate trips really full filling the query predicates.

Query 15 asks for the vehicles passing a point from $QueryPoints1Net$ during a period from $QueryPeriods1$. Analogous to query 14 we build temporal-netboxes of the of the query parameters to select the candidate trips using the temporal-network position index and refine the result filtering the candidates really full filling the query constraints.

Query 16 asks for the licence pairs one from $QueryLicence1$ and one from $QueryLicence2$ of vehicles, which were both present in a region from $QueryRegions1Net$ within a period from $QueryPeriods1$, but did not meet there and then. We select the candidate trips using the licence B-Tree of $dataSNcar$ relation and restrict the resulting trips to be **present** during the query periods and **inside** the query region. This is done one time for the licences from $QueryLicences1$ and one time for the licences from $QueryLicences2$. The both intermediate results are joined and filtered to get the trips of different cars which where at the same period in the same region without meeting each other there and then. In the TBA we have to do a additional selection from trips with the *moids* belonging to the cars selected before by the licences and remove duplicates of licence pairs from the same period and region.

Query 17 asks for the points from $QueryPointsNet$ that have been visited by a maximum number of different vehicles. In a first step we use almost the query algorithm from query 4 to select the trips passing a given query point. After that we group the cars passing query points by the ids of the query points and count the number of cars passing this query point. In a last step the point(s) with the maximum number of passing cars is(are) selected. In the TBA we have to remove duplicate vehicles from the result list before we count the number of passing cars.

4 Experimental Setup

For our experiments we used a standard personal computer with an AMD Phenom II X4 Quad Core 2.95 GHz CPU, 8 GB main memory, and 2 TB hard disk. We installed the Linux openSUSE 11.2 as operating system, SECONDO DBMS version 2.9.StableNewFlob, and the BerlinMOD Benchmark version provided in the web.

We generated three databases with different amounts of data using the data generation script of the BerlinMOD Benchmark with the $SCALEFACTOR$ 0.05, 0.2, and 1.0. The following steps are done with all three databases. We first created the BerlinMOD Benchmark data and indexes using the script "BerlinMOD_CreateObjects.SEC" for the data model of free movement in two dimensional space. The network data model representation of the databases was generated by the the script "Network_CreateObjects.SEC" that uses the algorithms and builds the indexes described in section 3.

Table 3 shows the created data amounts for the different $SCALEFACTOR$ values in the both data models. As you can see the network data model needs less than 40% of the storage space of the BerlinMOD Benchmark data model. The main cause is that the same trip is represented by less than 50% of the units in the network data model compared to the data model of free movement in two dimensional space. This is a very good result and we expect this effect to increase if the cars make long distance trips instead of moving in a single town like they do in the benchmark. In towns cars more often change the street or the velocity than cars that do long distance trips and so the compact route representation in the network data model should become more effect than in the town.

	Scalefactor 0.05		**Scalefactor 0.2**		**Scalefactor 1.0**	
Number of Cars	447		894		2000	
Number of Days	6		13		28	
Data Generation	164.761s		587.299s		3177.46s	
	BMODB	Network	BMODB	Network	BMODB	Network
Data Translation and Index Build	301.72s	535.65s	1,362.72s	2,190.45s	7,419.13s	11,144.13s
Number of Units	2,646,026	1,260,888	11,296,682	5,346,971	52,140,685	24,697,709
Total Storage Space	2.26 GB	0.86 GB	9.51 GB	3.69 GB	45,76 GB	17.28 GB
Data	0.79 GB	0.44 GB	3.35 GB	1.83 GB	15.47 GB	8.40 GB
Indexes	1.48 GB	0.42 GB	6.16 GB	1.86 GB	30.30 GB	8.89 GB

Table 3: Database Statistics

The long creation time of the network data model representation is caused by the expensive mapping of spatial and spatio-temporal positions into network positions. The indexes them self are build faster in the network representation than in the BerlinMOD Benchmark representation because they have less entries and are smaller.

We found some isolated mismatches in some query results as we compared the results of the BerlinMOD Benchmark queries and the network data model queries for the object based (OBA) and the trip based approach (TBA). We detected that the source data of the street map of the BerlinMOD Benchmark is not well defined in all places. Figure 1 shows two examples for the street map failures. Using a very high zoom factor you can see that single streets consist of more than one line. We corrected the source file "streets.data" of the BerlinMOD Benchmark at the places where we detected the errors and restarted the building of the databases and our experiments from the scratch. With the corrected street map all results match each other in the different data models and approaches.

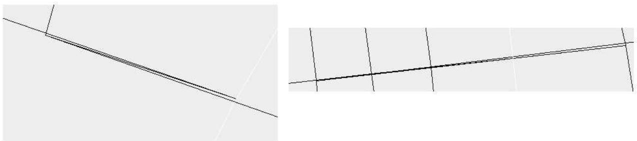

Figure 1: Example Failures in Street Map

5 Experimental Results

We repeated the BerlinMOD Benchmark query execution several times for both data models and approaches. The tables in figure 2 compare the average run times query run times in seconds for the different scale factors, data models, and approaches. As you can see the total run time of all queries in the network data model is around 50% less than the total query run time of the BerlinMOD Benchmark data model at each scalefactor. For the queries 1 and 2 the query run

	Scalefactor 0.05			
	BMODB		Network	
Query	**OBA**	**TBA**	**OBA**	**TBA**
1	0.086	0.100	0.090	0.113
2	0.003	0.003	0.003	0.002
3	0.346	0.345	0.112	0.568
4	9.105	15.403	0.142	1.303
5	1.076	1.623	0.927	1.377
6	16.781	14.934	5.004	4.483
7	2.996	3.007	1.221	6.790
8	0.346	0.424	0.225	0.213
9	99.375	193.929	22.935	24.349
10	139.795	36.636	81.770	84.298
11	0.143	0.111	0.158	0.898
12	0.297	0.133	0.228	0.202
13	11.284	7.341	1.159	1.336
14	0.525	0.727	0.793	3.734
15	1.201	0.802	0.625	0.543
16	43.567	5.346	0.680	1.579
17	1.084	0.935	0.234	0.337
Total	328.009	281.797	116.305	132.127

	Scalefactor 1.0			
	BMODB		Network	
Query	**OBA**	**TBA**	**OBA**	**TBA**
1	0.226	0.166	0.185	0.215
2	0.005	0.004	0.019	0.004
3	0.745	0.845	0.845	1.323
4	149.167	427.583	1.145	31.879
5	3.029	5.821	4.781	5.277
6	1463.214	5110.613	381.622	263.371
7	84.660	50.575	124.602	169.848
8	0.890	0.557	0.258	0.303
9	830.253	3153.720	120.470	156.693
10	4414.632	2015.096	2786.578	1791.941
11	0.656	0.914	6.030	7.738
12	36.210	0.213	0.274	0.267
13	119.685	94.366	29.126	36.086
14	10.912	3.915	36.041	38.552
15	30.874	18.711	10.317	7.989
16	35.490	8.655	0.584	1.969
17	82.242	343.826	0.550	8.026
Total	7262.888	11235.581	3503.429	2521.481

	Scalefactor 0.2			
	BMODB		Network	
Query	**OBA**	**TBA**	**OBA**	**TBA**
1	0.146	0.123	0.082	0.092
2	0.003	0.003	0.004	0.003
3	0.456	0.523	0.134	0.834
4	32.832	80.881	0.217	7.504
5	1.539	2.824	1.768	2.251
6	70.266	120.294	20.605	14.884
7	14.479	10.423	10.107	35.036
8	0.435	0.446	0.202	0.225
9	237.581	485.998	41.579	50.853
10	605.718	139.565	378.248	309.345
11	0.233	0.149	0.178	3.017
12	4.332	0.160	0.269	0.260
13	30.173	13.791	5.737	5.248
14	1.115	1.166	1.469	9.286
15	8.824	4.286	2.644	2.084
16	29.389	5.500	0.376	0.847
17	8.453	4.169	0.306	0.923
Total	1045.974	870.300	463,926	442.691

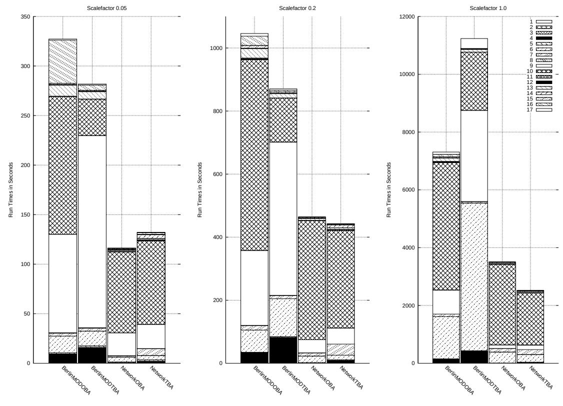

Figure 2: Compare Query Run Times in Seconds

times are almost the same for all data models and approaches at the different scale factors. This is what we expected because the both queries deal only with standard attributes and standard indexes, which are not influenced by the different data models.

For query 3 the run times for all data models and approaches are very small. But we can see a development of the ratio of the run times between the different data amounts, data models and approaches. Although the query algorithms for both data models and approaches is almost the same, the run time ratio for the different data amounts is different. For the small databases (scalefactor 0.05 and 0.2) the network database model outperforms in the OBA the data model of free movement in two dimensional space, while for scalefactor 1.0 and all TBA queries the data model of free movement in space outperforms the network data model. We think that two different effects take place. On the one hand the number of units in the network data model is less than the number of units in the data model of free movement in space, such that the unit which contains the query time instant can be found faster. On the other hand a *gpoint* value has more internal elements (3 *int*, 1 *real*, and 1 *bool*) than a *point* value (2 *real*, and 1 *bool*), such that result computing and copying is a little more expensive in the network data model. The advantage of the smaller number of units in a binary search is smaller if the number of units becomes bigger and the disadvantage of bigger results becomes greater, that explains the different run time ratios for query 3.

In query 4 the network data model outperforms the data model of free movement in space significantly at all scale factors (¿2 min OBA, ¿6 min TBA at scalefactor 1.0). The network data model index used in query 4 is much smaller (OBA 24 MB, TBA 160 MB, at scalefactor 1.0) than the spatial unit index of the BerlinMOD Benchmark (OBA 3.7 GB, TBA 3.7 GB at scalefactor 1.0) and more precise, such that we don't need a additional refinement step after the index usage in the network data model, like we do in the data model of free movement in two dimensional space.

We expected the network data model to be slower than the data model of free movement in two dimensional space, because we retranslate intermediate results from the network data model representation into the BerlinMOD Benchmark data model representation. For the OBA this is correct. We need a little more time in the network data model than in the data model of free movement in two dimensional space. But in TBA the network data model outperforms the data model of free movement in two dimensional space. This is due by the fact that a *gline* value has less *routeintervals* than a *line* value representing the same part of a curve has $halfsegments$, such that the union of two or more *gline* values in the aggregate step of query 5 in TBA can be computed much faster than the union of two or more *line* values.

The network data model outperforms the data model of free movement in space again significantly at query 6 for all data amounts and approaches, although we retranslate intermediate results and use the distance function of the data model of free movement in two dimensional space. In the network data model we reduce the number of candidate pairs for the distance computation by pre selecting intersecting extended bounding boxes and use the operation **notEverNearerThan** in OBA and TBA, while in the BerlinMOD Benchmark in the OBA no filtering is used and the computation is done by **minimum**(**distance**($mp1$,$mp2$)$\leq$10.0), and in the TBA the operation **spatialjoin** is used instead of bounding box intersection. While the operation **distance** has always a run time from O(n) the operation **everNearerThan** stops computation immediately if the distance between two units is less than the query value to reduce computation time. And the operation **spatialjoin** of the SECONDO DBMS seems to have a big weakness in implementation. Otherwise the difference in the TBA query run times could not be so big (¿ 17 min OBA, ¿ 80 min TBA, at scalefactor 1.0).

After the very good results from query 4 we did not expect query 7 to have such results in the run times comparison. In fact at scalefactor 1.0 the data model of free movement in two dimensional space outperforms the network data model significantly in both approaches and at all scale factors in TBA, while at the smaller scale factors in OBA the network data model outperforms the the data model of free movement in two dimensional space. We think there are two main causes on the one hand we have to do the expensive operation *at* for *mgpoint* for the double number of query *gpoint* compared with the data model of free movement in space and on the other hand the test points out a weakness of the network data model implementation of the operation **at** for *mgpoint*. But in the end network data model looses at scalefactor 1.0 less than 45 seconds in the OBA respectively less than 120 seconds in TBA, what is not much compared with the advantages in the other benchmark queries.

Query 8 is a very fast query in both data models, although the query run time of the network data model is more than 50% less than the query run time of the data model of free movement in two dimensional space. Which is caused by the *length* attribute of the *mgpoint* and the smaller number of units of a *mgpoint* compared with the corresponding *mpoint*.

For query 9 the network data model outperforms the data model of free movement in space by orders of magnitudes the advantages named in the analysis of query 8's run time results become a much higher impact when the number of examined trips becomes bigger. At scalefactor 1.0 this saves more than 10 min time in the OBA and more than 50 min time in the TBA.

The ratio of the run times of query 10 changes between the data amounts and the both data models. In the OBA and at scalefactor 1.0 in the TBA the network data model outperforms the data model of free movement in two dimensional space at all scale factors, while in the TBA at the two small databases the data model of free movement in two dimensional space outperforms the network data model significantly. Before our experiments we expected that the BerlinMOD Benchmark would outperform the network data model in all cases, because of the expensive retranslation of intermediate results. So why is the network data model faster (¿20 min in OBA and ¿3 min in TBA at scalefactor 1.0) than the data model of free movement in two dimensional space? In the OBA we use bounding boxes for a preselect of candidate trips that step is not performed in the BerlinMOD Benchmark. In the TBA the results are only better for the big data amounts we think this is due to the fact that the number of units in *mgpoint* values is always smaller than in *mpoint* values such that the final aggregation of the different trips of the same cars can be done faster in the network data model than in the data model of free movement in two dimensional space.

Query 11 is identically with the first part of query 12 so it is surprising that the run time of query 11 at scalefactor 1.0 is longer than the run time of query 12, which does additional computations. In our experiments with the different queries we have seen that there exist numerous cache effects depending on the sequence of the queries. So we think that query 12 takes profit cache effects resulting from query 11 running immediately before query 12. Another weakness of the network data model pointed out by the run times of query 11 and query 14 is that our network-temporal position index has bad run times for query *netbox* objects constrained from a single *gpoint* and a single time instant. Which becomes more worse with a higher number of indexed units. As you can see at query 15 this does not hold for query *netboxes* constrained from a single *gpoint* and a time interval. We have to spend some more work to figure out the problem and develop a better network-temporal position index to improve our network data model system.

In our experiments we also tested the MON-Tree [5] as network-temporal index but the elapsed run time performance was not good, although the CPU run times was very well.

The bad performance of the network-temporal position index is also shown by query 13. The network data model outperforms the data model of free movement in two dimensional space significantly, but we don't use any index in the executable network data model queries, while the BerlinMOD Benchmark uses its spatio-temporal index to preselect candidate trips. The same holds for query 17.

The network data model version of query 16 takes profit from the smaller number of units in the network data model and outperforms the data model of free movement in two dimensional space.

Although we detected in our experiments some points of weakness in the network-temporal position indexing, the network data model outperforms the data model of free movement in two dimensional space by orders of magnitudes. The weakness of the network data model almost occurs in queries with short run times, while the advantages of the network data model take place in the queries with long run times, such that the weakness of the network-temporal position index is covered by the advantages of the network data model.

6 Summary and Future Work

We presented our translation of the BerlinMOD Benchmark into the network data model and compared the capabilities of the both data models, with very good results for the network data model. Our experiments show that the network data model outperforms the data model of free movement in the two dimensional space by orders of magnitudes with respect to storage space and query run times. This is manly caused by the much lower number of units for a $\underline{mgpoint}$ value compared with the number of units of the corresponding $\underline{mpoint}$, which also results in smaller indexes for the network data model objects. The BerlinMOD Benchmark of the network data model pointed out that we should spend time in the improvement of the network-temporal position index and the **at** operation for $\underline{mgpoint}$ and $\underline{gpoint}$ values.

The good results of the network data model encourages us to work on a extension of the BerlinMOD Benchmark, which should enable us to compare the capabilities of different spatio-temporal network data models with respect to the special challenges of network data models, like shortest path and fastest path computation.

Another direction of our actual work is traffic flow estimation and traffic jam representation in the network data model.

Other interesting themes for future work on the network data models is the efficient computation of dynamic network distances between moving network objects.

References

[1] Cindy Xinmin Chen and Carlo Zaniolo. Sql^{s}t a spatiotemporal model and query language. In *Conceptual Modeling - ER 2000*, volume 1920/2000, pages 111–182. Springer Berlin, Heidelberg, 2000.

[2] Thomas Behr Christian Düntgen and Ralf Hartmut Güting. Berlinmod: A benchmark for moving object databases. Informatik Berichte 340, FernUniverstität in Hagen, 2007.

[3] Thomas Behr Christian Düntgen and Ralf Hartmut Güting. Berlinmod: A benchmark for moving object databases. *The VLDB Journal*, 2009.

[4] Victor Teixeira de Almeida and Ralf Hartmut Güting. Indexing the trajectories of moving objects in networks. Informatik Berichte 309, FernUniverstität in Hagen, 2004.

[5] Victor Teixeira De Almeida and Ralf Hartmut Güting. Indexing the trajectories of moving objects in networks*. *Geoinformatica*, 9(1):33–60, 2005.

[6] Stefan Dieker and Ralf Hartmut Güting. Plug and play with query algebras: Secondo-a generic dbms development environment. In *IDEAS '00: Proceedings of the 2000 International Symposium on Database Engineering & Applications*, pages 380–392, Washington, DC, USA, 2000. IEEE Computer Society.

[7] Martin Erwig, Ralf Hartmut Güting, Markus Schneider, and Michalis Vazirgiannis. Spatio-temporal data types: An approach to modeling and querying moving objects in databases. *Geoinformatica*, 3(3):269–296, 1999.

[8] Fernuniversität Hagen. *BerlinMOD Benchmark Web Site*, June 2008.

[9] Fernuniversität Hagen. *Secondo Web Site*, April 2009.

[10] Luca Forlizzi, Ralf Hartmut Güting, Enrico Nardelli, and Markus Schneider. A data model and data structures for moving objects databases. In *SIGMOD '00: Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 319–330, New York, NY, USA, 2000. ACM.

[11] Hartmut Güting, Victor Teixeira de Almeida, and Zhiming Ding. Modeling and querying moving objects in networks. *The VLDB Journal*, 15(2):165–190, 2006.

[12] Ralf Hartmut Güting, Victor Almeida, Dirk Ansorge, Thomas Behr, Zhiming Ding, Thomas Hose, Frank Hoffmann, Markus Spiekermann, and Ulrich Telle. Secondo: An extensible dbms platform for research prototyping and teaching. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering*, pages 1115–1116, Washington, DC, USA, 2005. IEEE Computer Society.

[13] Ralf Hartmut Güting, Michael H. Böhlen, Martin Erwig, Christian S. Jensen, Nikos A. Lorentzos, Markus Schneider, and Michalis Vazirgiannis. A foundation for representing and querying moving objects. *ACM Trans. Database Syst.*, 25(1):1–42, 2000.

[14] Christian S. Jensen, Dalia Tiesyte, and Nerius Tradisauskas. The cost benchmark -comprasion and evaluation of spatio-temporal indexes. In *Database Systems for Advanced Applications*, volume 3882/2006, pages 125–140. Springer Berlin, Heidelberg, 2006.

[15] Nikos Pelekis, Yannis Theodoridis, Spyros Vosinakis, and Themis Panayiotopoulos. Hermes - a framework for location based data management. In *Advances in Database Technology - EDBT 2006*, volume 3896/2006, pages 1130–1134. Springer Berlin, Heidelberg, 2006.

[16] S. Rezić. *Berlin Road Map*, 2008.

[17] Laurynas Speičvcys, Christian S. Jensen, and Augustas Kligys. Computational data modeling for network-constrained moving objects. In *GIS '03: Proceedings of the 11th ACM international symposium on Advances in geographic information systems*, pages 118–125, New York, NY, USA, 2003. ACM.

[18] Statistisches Landesamt Berlin. *Bevoelkerungsstand in Berlin Ende September 2006 nach Bezirken*, 2008.

[19] Statistisches Landesamt Berlin. *Interaktiver Stadtatlas Berlin*, 2008.

[20] Yannis Theodoridis. Ten benchmark database queries for location-base services. *The Computer Journal*, 46(6):713–725, 2003.

[21] Michalis Vazirgiannis and Ouri Wolfson. A spatiotemporal model and language for moving objects on road networks. In *Advances in Spatial and Temporal Databases*, volume 2121/2001, pages 20–35. Springer Berlin, Heidelberg, 2001.