Network Data Model and BerlinMOD Benchmark

Simone Jandt

Last Update: April 13, 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. Two representants of this data models, one for each category, have been implemented in Secondo DBMS. We show in this paper that the network data model outperforms the data model of free movement in the BerlinMOD Benchmark 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 [9,1] for spatio-temporal data objects moving freely in space and [10,16,20] for spatio-temporal data objects that are constrained by given networks to name just a few.

For our experiments we choosed one example data model as representative for each of the both categories to compare the capabilities of the both different data models. Spatio-temporal objects moving freely in two dimensional space are represented by the data model presented in [9]. And network constrained data models are represented by the network data model presented in [10]. Both data models are available in the Secondo DBMS and use the same temporal representation. So we can exclude that the use of different DBMS or temporal representations bias the results of our comparision.

As we use Secondo it seems to be a logical next step to use the BerlinMOD Benchmark [3] 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 also available in the Secondo DBMS. Furthermore the BerlinMOD Benchmark data model is the data model of freely movement in two dimensional space that we use for our comparison. So we only have to transform the BerlinMOD Benchmark spatial and spatio-temporal data types once into our network data model representation. This avoids sources of errors and makes the correctness controll of the query results more easy.

In our experiments the network data model outperforms the data model of free movement in space significantly. The total run time for all benchmark queries for the network data model is less than 50% of the total run time of the data model of free movement in the space. In the same way the storage space used for the trips in the network data model is less than 60% of the storage space used by the data model of free movement in the space. Even the number of units in the network data model is less than 50% of the number of units in the data model of free movement in the

space. Such that we conclude that it is useful to develope specialised databases and operations for network constrained moving objects to save storage space and run time for database operations.

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 choosed as representatives for our comparision. In section (3) we present the setup for our experiments. The translation of the BerlinMOD Benchmark data and query set in the network data model representation is described in section (4). We present the results of our experiments in section (5). We conclude our work in section 6.

2 Related Work

As mentioned before in the past other data models for free movement in the space [9, 1] and for network constrained movement have been presented [10, 16, 20]. As well there are some more benchmarks [13, 19] and database systems for spatial and spatio-temporal datatypes [14, 11]. But they don't provide a combination of different implemented and supported data types together with a existing benchmark like the actual SECONDO DBMS. So it is self-evident for us to use the SECONDO DBMS in combination with the provided data types and the BerlinMOD Benchmark to compare the capabilities of the both different data models.

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 [5, 11] 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 so called SECONDO algebra modules to introduce new data types and operations on this data types. The user also may define additional viewers for the graphical user interface or write optimization rules or cost functions to extend the optimizer. SECONDO is free available in the web [8] 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 space 2.3 and the network data model 2.4. Furthermore the BerlinMOD Benchmark descriped 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 [7] and provides a well defined data-set and queries for the experimental evaluation of different moving object data representations. The BerlinMOD Benchmark emphasises the development of complete systems and simplifies experimental repeatability pointing out the weakness and the potency of the benchmarked systems.

The data-sets of the BerlinMOD Benchmark are created using the street map of the German captial Berlin [15] and statistical data about the regions of Berlin [17, 18] 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. Besides this randomly choosen cars will make additional trips in

the evening and up to six times at the weekend to randomly choosen targets in Berlin. Because for all this the street network of Berlin is used to generate the moving object data sets the BerlinMOD Benchmark can also be used to benchmark network constrained moving object data models.

The number of observed cars and the duration of the observation period can be influenced 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 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 moving unit. This means that if for example a car parks 8 hours in front of the work node there will be only one entry in the cars history of movement with a time interval of 8 hours instead of 14.400 entries one for each GPS 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. One of them (dataMcar) contains the static data for each object like licence, type, and model together with an object identifier. The other relation (dataMtrip) contains for each object identifier several tuples each of them containing a single trip of the moving object (e.g. each time the car drives from home node to work node) or a longer stop (e.g. the time the car parks in front of the office).

Besides the moving objects the BerlinMOD Benchmark provides several data sets each of them containing 100 pseudo randomly generated data objects $QueryPoints$(point), $QueryRegions$(region), $QueryInstants$(time instants), $QueryPeriods$(periods), and $QueryLicences$(licences) which are used as query objects in the benchmark queries.

The BerlinMOD Benchmark provides two sets of queries. One set BerlinMOD/R addresses range queries and the other one BerlinMOD/NN nearest neighbour queries. We will focus on the range queries in this paper, 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:

- object identity (known / unknown),

- dimension (standard / spatial / temporal / spatio-temporal),

- query interval (point / range / unbounded),

- condition type (single object / object relations),

- aggregation (with or without aggregation).

We will present the 17 queries in more detail in section 4.4 together with our network data model translation of this queries.

2.3   BerlinMOD Data Model

The data model used by the BerlinMOD Benchmark is the same data model of freely moving in two dimensional space presented in [6, 9, 12]. All spatial positions are given in x,y-coordinates. 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 two dimensional space. Streets are represented by *line*

values. A *line* value consists of a set of half segments representing the geometry of the line in the two dimensional space. Each half segment consists of two *point* values which are interpreted as start and end point of the half segment. Regions are represented by the data type *region*. A *region* consists of a set of half segments interpreted as outer (and inner) border of the region in the two dimensional space.

All this spatial data types and many standard data types can be lifted to become time dependent *moving* values. For all data types $\underline{\alpha}$ the constructor *moving* creates a new data type *moving*($\underline{\alpha}$) (short form $\underline{m\alpha}$). A car may be represented by a *mpoint*. A *mpoint* is a point object changing its position within time. Therefore 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 represents the position of the *mpoint* at the start of the time interval and the second point represents the position of the *mpoint* at the end of the time interval. It is assumed that the point moves on the straight line between this two points with constant speed. The speed 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 *mpoint* 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 *moving* allows us to compute the position of a *mpoint* at every time instant within its definition time. We can also compute the time instant the point passed a given position assumed the *mpoint* ever passes this position. The position of a *point* at a given time instant is represented by a *intime*(*point*) (short form *ipoint*). A *ipoint* consists of a time instant and a *point* value.

Some other data types of SECONDO which are used in the BerlinMOD Benchmark are shown in table 1.

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

Table 1: Other Data Types of BerlinMOD Benchmark

2.4   Network Data Model

The central idea of the network data model presented in [10] is that every movement is constrained by a network and every position is given related to this network. The data type *network* is the central data type in the network data model. All other data types of the network data model are related to a given *network* object by the unique network identifier that is part of each *network* object.

The *network* object contains all spatial information of the represented network in three main relations. The first relation contains the attributes of the routes (streets) like id, route curve, route length, and two boolean flags indicating if the route starts at the lexicographically smaller end point or not and if the lanes of the route are separated like on German Highways or not. The second relation contains all attributes of the junctions (street crossings) like the identifiers of the first and second route crossing in the junction, the distance of the junction from the start of the first respectively second route, tuple identifiers of the both routes 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 routes are connected by the junction. The third main relation

of the <u>network</u> object is the sections relation containing the attributes of the sections (street parts between two junctions or a junction and the end of the street) like the route identifier of the route the section belongs to, the tuple identifier of this route in the routes relation, start and end position of the section on the route, section curve, and two boolean flags *startssmaller* and *dual* with the same meaning as in the routes relation.

We introduce four B-Tree indexes and one R-Tree index to support faster query execution. The four B-Trees indicate the route identifier attributes in the three main relations. And the R-Tree indicates the curve attribute of the routes relation. Furthermore there are two sets which provide 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.

Single positions in the network are given as <u>gpoint</u> values. Besides the network identifier a <u>gpoint</u> consists of a route identifier, a distance from the start of the route to the position of the <u>gpoint</u> and a *side* value (*up*, *down*, *none*). The *side* value is necessary if the route is dual. It tells us if the position is reachable from the *up* or the *down* side of the route. For simple routes 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 <u>gline</u> values. Besides the network identifier a <u>gline</u> consists of a set of route intervals, and two boolean flags telling us if the <u>gline</u> is defined and if the set of route intervals is sorted.

Each route interval 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 route intervals sorted if the following conditions are fullfilled:

- all route intervals are disjoint

- the route intervals are stored in ascending order of their route identifiers

- if two disjoint route intervals have the same route identifier the route interval with the smaller start position is stored first

- for all route intervals in the set the condition: $startPosition \leq endPosition$ holds

We introduced the sorted <u>gline</u> because many algorithms take profit from sorted <u>gline</u> values. For example the computation if a <u>gpoint</u> is inside a <u>gline</u> needs $O(n)$ time for unsorted and $O(\log n)$ time for sorted <u>gline</u> values, if n is the number of the route intervals in the <u>gline</u>.

Unfortunately not all <u>gline</u> values can be stored sorted. If a <u>gline</u> value represents a path between two <u>gpoint</u> 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 solved this dilemma by introducing the boolean sorted flag. Every algorithm which can take profit from a sorted <u>gline</u> checks this flag and uses the corresponding code. We store <u>gline</u> values sorted whenever this is possible to support faster query execution.

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

In deviation from the original network data model we extended the implementation of the <u>mgpoint</u> with four additional attributes:

---

[1]In the original paper the route interval includes a additional parameter *side* analogous to the <u>gpoint</u>. But this parameter is not part of the implementation yet.

1. The total driven distance

2. A sorted set of route intervals representing the positions ever traversed by the *mgpoint*

3. A boolean defined flag for the set of route intervals

4. A spatio-temporal minimum bounding box

The sorted set of route intervals was introduced, because 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 *ugpoint*s of a *mgpoint* we can perform a binary scan on the much lower number r of route intervals. This reduces the time complexity from $O(m)$ to $O(\log r)$ for all **passes** operations. Logically this should be done by a sorted *gline* value but the SECONDO DBMS restricts us to use a sorted set of route intervals 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 the motion may follow different spatial directions, e.g. a route may lead uphill in serpentine. Not all this positions must be enclosed by a bounding box computed just from the spatial position of the *start* and the *endposition* of the unit. Therefore we have always to examine the spatial dimensions of the complete part of the route passed within a unit to compute the units bounding box. All spatial information of the route curve is hidden in the *network* object. We have to call the route curve from the *network* object to compute the spatial dimensions of the unit bounding box. If r is the number of routes of the network and h the number of half segments of the traversed part of the route curve 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 a *mgpoint* bounding box takes $O(m(h + \log r))$ time. This is a very expensive computation and the bounding box of a *mgpoint* is only computed on demand or if we can get it for free. For example we can copy the bounding box of a *mpoint* if we translate it into a *mgpoint* without computational effort. But we don't maintain this attribute at every change of the *mgpoint*. If the *mgpoint* changes we set the bounding box attribute to be undefined and compute it again on demand if necessary.

3 Experimental Setup

For our experiments we used a Standard PC with a AMD Phenom II X4 Quad Core 2.95 GHz CPU, 8 GB main memory, and 2 TB HDD. We used the Linux openSUSE 11.2 operating system and installed SECONDO DBMS version 2.9.StableNewFlob and the BerlinMOD Benchmark from the web.

We used the data generating script of the BerlinMOD Benchmark with $SCALEFACTOR$ values 0.05, 0.2, and 1.0 to generate three data sets with different amounts of data in three different database directories. After that we build the index structures used by the BerlinMOD Benchmark with the script "BerlinMOD_CreateObjects.SEC" delivered with the BerlinMOD Benchmark data for each database and started the benchmark queries for the object oriented and the trip based approach of the BerlinMOD Benchmark on this databases. We saved the results for each database and measured the run times of the queries several times to be sure that the run times measured are free from other influences.

Next we translated the three databases into network data model representation. Therefore we build a *network* object *net* from the data of *streetsrelation* and translated all spatial and spatio-temporal data types of the BerlinMOD Benchmark data sets relative to *net*. We also define some indexes to support faster query execution on the network data model representation and formulate
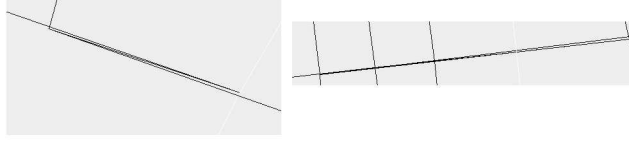
Figure 1: Example Failures in Street Map

executable SECONDO queries for each BerlinMOD/R query on this network data set. We give a detailed description of the translation steps, indexes and queries in section 4. After that we started a first run of our network queries and compared the results of our network queries with the results of the BerlinMOD Benchmark queries to ensure that all results are correct. We found some isolated missmatches in some query results, which were caused by the fact that singular route curves in the source data of the street map were not well defined (see figure 1) at two places. We corrected the source file "streets.data" delivered by BerlinMOD Benchmark at this two routes and restarted the building of the databases and our experiments from the scratch. After all results from the network queries corresponded to the results of the BerlinMOD Benchmark results we figured out the fastest versions of the executable SECONDO network queries and started them several times to measure the run times analogues to the run times of the BerlinMOD Benchmark benchmark queries. The results of the run time measurement are shown in 5.

4 Translation of BerlinMOD into Network Data Model

The construction of the central <u>network</u> object from the *streets* value of the BerlinMOD Benchmark is described in section 4.1. We used this <u>network</u> object to transfer the spatial and spatio-temporal data objects into the network data model representation (see 4.2). We build several indexes on this spatial-temporal network constrained data (see 4.3) to support faster query execution. We conclude the subsection with a description of our executable SECONDO network queries for the BerlinMOD Benchmark in section 4.4.

On our website [7] we provide different SECONDO scripts. One for the network data creation, the translation of the BerlinMOD Benchmark data sets into network data representation, and the index generation. And two scripts containing the network query sets for the object and the trip based approach of the BerlinMOD Benchmark.

4.1 Create Network Object

For the creation of the <u>network</u> object *net* representing the street map of Berlin we extract the routes data from the *streets* relation of the BerlinMOD Benchmark data set. The extracted routes data r is used to compute the crossings of the routes of Berlin j. In this step the connectivity code for each crossing is set to the maximum value because the data source lacks on information about the connectivity of the crossings. We use r and j as input relations for the creation of our <u>network</u> object *net* representing the streets of Berlin in the network data model.

The network creation algorithm first copies all tuples of r to the *routesrelation* of *net* and creates the B-Tree index of the route identifiers and the R-Tree index of the route curves of the *routesrelation* of *net*. Then all tuples of j are copied to the *junctionsrelation* of *net* and the *tupleidentifiers* for the both routes connected by this junction are added to the junctions entry in the *junctionsrelation*. After that we build two B-Trees indexes over the route identi-

fiers of the first respectively second route in the *junctionsrelation*. Next for every route of the *routesrelation* all junctions on this route are taken to compute the up and down sections for each of this junctions on the route. The up and down sections are inserted into the *sectionsrelation* of *net* and the *tupleidentifiers* of the sections are added to the entry of the according junction in the *junctionsrelation*. After that the B-Tree index for the route identifiers in the *sectionsrelation* is created and the adjacency lists of *net* are filled with the adjacent section pairs defined by the *junctionsrelation*.

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

4.2   Translate Spatial and Spatio-Temporal Data

In this section we describe the translation of the spatial and spatio-temporal data types of the BerlinMOD Benchmark data set into spatio-temporal network objects. The translation is done relative to the <u>network</u> object *net* (see 4.1). The input for all algorithms is a spatial respectively spatio-temporal BerlinMOD Benchmark data object and the <u>network</u> object *net*. If a input data object is not constrained by *net* the translation result is undefined for all network translation algorithms.

We start the explanation of our translation algorithms with the **point2gpoint** operation, because this operation is used by the other translation algorithms. The algorithm translates a <u>point</u> value p into a corresponding <u>gpoint</u> value *gp*. It uses the R-Tree index of the *net routesrelation* to select the route closest to p and computes the position of p on this route. The *side* value of *gp* is always set to *none*. Because the BerlinMOD Benchmark does not differntiate between the sides of a road. If r is the number of routes in the *net routesrelation* and k is the number of possible candidate routes the worst case complexity of the algorithm is $\mathrm{O}(k + \log r)$.

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 freely moving in space junctions have more than one <u>gpoint</u> representation, because they are 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. As work around 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 *QueryPoints*1*Net* for the 10 <u>point</u> values of *QueryPoints*1 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.

The second operation **mpoint2mgpoint** translates a <u>mpoint</u> value s into a <u>mgpoint</u> 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** operation 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 this route we check the direction and speed of the unit. If they are equal to the last 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 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 least 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*.

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 from all our network streets. Then we compute for each region of the *QueryRegions* the intersection with this big line object. At least 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 half segment of a line value and computes a corresponding network route interval by searching a common route curve for the *start* and the *endpoint* of the half segment using the **point2gpoint** operation. The computed route intervals are sorted, merged and compressed before the resulting gline value is returned. If the number of half segments of a line value is *h* and the number of resulting compressed route intervals 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.

4.3   Create Indexes on Network Data Model

After translating all the BerlinMOD Benchmark data sources we are able to create indexes on our network data representation of the BerlinMOD Benchmark data. First we create B-Trees for the *licences* and *moid* attributes of the relations *dataSNcar*, and *dataMNtrip*. This indexes are similar to the indexes created in the BerlinMOD Benchmark for *dataSCcar*, and *dataMCtrip*, because the relations *dataSNcar* and *dataMNtrip* contain the network data model representation of the *dataScar* and *dataMtrip* relation of the BerlinMOD Benchmark. Then we create R-Tree indexes over the spatio-temporal bounding boxes of the mgpoint attributes in the *dataMNtrip* and the *dataSNcar* relation. At least we create some specialized network indexes indicating network positions and network-temporal positions of moving objects. Therefore we introduce two and three dimensional *netboxes*. A *netbox* is a degenerated two or three dimensional rectangle. The coordinates of the rectangle are defined to be $x_1 = x_2 = routeidentifier$ as real value (The equality of x_1 and x_2 makes the degeneration.), $y_1 = \min(starposition, endposition)$, $y_2 = \max(startposition, endposition)$, and, in the three dimensional case, $z_1 = starttime$ as real value and $z_2 = endtime$ as real value. For every unit of each mgpoint we build a three dimensional *netbox* and for every *routeinterval* of every mgpoints trajectory a two dimensional *netbox*. This *netboxes* are used to create R-Trees indexing the network and network-temporal positions of the mgpoints in the network data model representation of the BerlinMOD Benchmark.

4.4   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. We had to do this manually because the SECONDO optimizer is not able to optimize SQL-queries on network data model objects yet. In our experiments we tried 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 network queries in detail. As mentioned before the complete SECONDO network scripts can be taken from our website. In the following we describe only the algorithms of a few queries in detail.

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 tables *dataMNtrip* and *dataMNcar* using the *moid* attribute and the corresponding B-Tree indexes. We will not repeat this step at every single query description.

Query 1 and 2 work only on standard attributes. They are formulated analogous to the original queries of the BerlinMOD Benchmark only the relation names and the used B-Trees are changed to match the network data model.

Query 3 uses the licence B-Tree to select the ten cars with licences from *QueryLicences*1 from *dataSNcar* then the positions of this cars are computed for each of the ten time instants from *QueryInstants*1.

In Query 4 we produce a *netbox* for each of the *QueryPointsNet* and use our specialized netbox R-Tree of the mgpoint routeintervals to select the passing vehicles.

In the queries 5, 6, and 10 a retransmission of network objects into spatial respectively spatio-temporal objects of the BerlinMOD Benchmark data model is done. This is caused by the fact that the BerlinMOD Benchmark deals with Euclidean Distances. Euclidean Distances are not very useful in network environments because all objects are restricted to use network paths. Therefore in networks normally the Network Distance is computed. To make the results comparable we retransmit the intermediate results of our network data model into spatial and spatio-temporal data types and use the existing spatial and spatio-temporal Euclidean Distance Functions of the BerlinMOD Benchmark data model for the distance computation in the queries 5, 6, and 10.

Query 5 selects the cars with *licences* from *QueryLicence*1 respectively *QueryLicences*2 using the B-Tree over the *Licence* attribute of *dataSNcar* and creates a line value from the list of *routeintervals* passed by every car. Then the Euclidean Distance between this line values is computed for each pair of licences one from *QueryLicences*1 and one from *QueryLicences*2. In the TBA we need a aggregation step building the union of the several mgpoint belonging to each candidate car. This is done with the *routeintervals* returned as trajectory of the mgpoint values because it takes much less time to build the union of the *routeintervals* than of the half segments in the line values representing the same network part.

Query 6 uses the **filter** operation to select the "trucks" from *dataSNcar* (respectively *dataMcar* in TBA) relation. Then the spatio-temporal bounding box of each trip is computed and the spatial dimensions of this box are extended by 5m in every spatial direction. After that the mgpoint values are retranslated into mpoint values. In a second step each result of the first step is joined with all other results of the first step if the extended bounding boxes intersect, the licences are different and the mpoint values have sometimes a distance lower than 10m. The licence pairs of trucks fullfilling this predicate are returned. In the TBA there might be duplicate licence pairs which we have to remove before we return the result.

The first part of the first step of query 7 is almost equal to the selection of cars passing a query point in query 4. The intermediate result is filtered to remove all "not passenger" cars and for every remaining trip the time the trip reaches first the query position is computed for every query

position and every candidate trip. In a second step the resulting time instants are grouped by the *id* of the query positions and the minimum time stamp of each group is computed. This minimum time stamp is for every query position the first time it was reached by a car. In the third and last step the licences of the cars reaching the query positions at this first time instant are computed by a join of the results of the first two steps by query position id and the equality of the time stamps.

In query 8 we just select the candidate cars with the licence B-Tree and compute for every car the length of the trip at the query periods in the OBA. In the TBA we have to aggregate over all the distances driven in the single trips by a car within a query period.

For query 9 we compute the length of every trip in every query period, and select the maximum driven distance for every period. In the TBA again we have to do a aggregation of the distances driven from the same car in the same period.

For query 10 in 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 direction. Second we select the ten candidate trips given by *QueryLicences*1, retranslate them and extend their spatial bounding boxes. Than we join all trips from the first and the second step where the extended bounding boxes intersect. We filter the pairs that have different licences and are sometimes nearer than 3m. For this pairs we compute the position of the *mgpoint* at the times the distance between the both *mpoint* has been smaller than 3 m. We return the licence pairs and the positions when they have been closer than 3 m to each other. In TBA we select the trips given by *QueryLicences*1 from *dataMNtrip* retranslate them into *mpoint* values. Then we use the spatio-temporal index of *dataMNtrip* to select for each of the ten cars the cars of *dataMNtrip* which bounding boxes intersect the extended spatio-temporal bounding boxes of the first selected candidate trips. For every pair of candidate trips we retranslate the second trip into free movement and use the Euclidean Distance function for *mpoint* values to determine the deftimes when the both *mgpoint* had a distance lower than 3m. At least we restrict the trips on 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 faster query execution of especially of query 10 including the MON-Tree [4]. But at the end this simple form shows the best complete run time performance of all indexes.

In query 11 we build a network-temporal query box from the product of *QueryInstant* and *QueryPoints*1*Net* relation. And use the network-temporal index on *dataSNcar* (respectively *dataMNtrip*) to select the resulting trips.

The first step of query 12 is identical with query 11. In a second step a product of the result of the first step with itself is computed and checked for vehicles which have been at the same query point at the same query time instant.

Query 13 first computes uses the trajectory value of the *mgpoint* to select the trips passing a given region. Restricts the trips to the times they pass the region and tests the resulting trips for their existenz in a given time interval. In TBA possible duplicate licence pairs have to be removed and the resulting *moids* must be mapped to the licences of the cars to generate the result.

Query 14 and 15 build sets of three dimensional network boxes of the query parameters and use the three dimensional network box tree to select the candidate trips. The resulting candidate trips are filtered to be sure they realy fullfill the query constraints.

Query 16 selects the candidate trips using the licence B-Tree, filters them to be **present** within the query periods and restricts them to the times of the query periods. The restricted trips are filtered with the **passes** operation and restricted to the times they were inside the query region. This is done one time for *QueryLicences*1 and one time for *QueryLicences*2. The both results are joined 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 a third step. Again 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 to

remove duplicates of licence pairs in the same period.

Query 17 again uses the methods from query 4 to find the trips passing a given query point. The passing cars are grouped by the passed query points and the number of cars per query point is computed. In a second step the point with the maximum number of hits is selected and his id and the number of passing cars is returned. In the TBA we have to remove the duplicate cars from the result before computing the hits.

5   Benchmark Results

We made several runs for each data amount and each query to get correct average execution times for each query in both data models. Table 2 compares the creation times, the number of units and the storage space of the both data models at the different *scale factors*.

	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	7419.130	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 2: Database Statistics

As you can see the network data model needs less than 40% of the storage space from 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 the two dimensional space. And that although in towns cars often change the speed or the used street. We expect this effect to be much higher if the cars go for long trips between different towns especially if they use highways.

The long creation time for the network data model representation is caused by the expensive mapping of spatial and spatio-temporal positions into network positions. The Indexes are build faster in the network representation because they are much smaller than in the BerlinMOD Benchmark data representation.

Figure 2 compares the run times for the different scalefactors and data models in seconds. As you can see the total run time of all queries in the network data model is almost less than a half of the total query run time of the BerlinMOD Benchmark data model at each scalefactor. The network data model outperforms the data model of free movement in the two dimensional space significantly in the queries that have a the longest run times.

This can be explained by the effects of the new attributes trajectory and length of the mgpoint and by the much lower number of units for a mgpoint value compared with the corresponding mpoint values. In the BerlinMOD Benchmark data model every unit must be checked if we want to know if a mpoint passed a given position at an arbitrary time. In the network data model we can do a binary check of the much lower number of sorted *routeintervals* in the trajectory of the mgpoint, which are additionally indexed by a specialised network position index, that is much smaller than the corresponding spatial index of the mpoint values. This effect becomes bigger if cars and units grows, especially if the observed cars drive several times the same route, like they do in the BerlinMOD Benchmark for the work trips. This effect can be seen if you compare the run times of the queries 4, 6, 13, 15, and 16 at the different scalefactors.

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

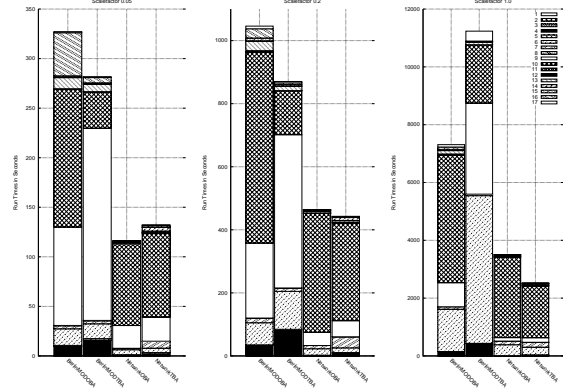| Scalefactor 0.2				
	BMODB		Network	
Query	OBA	TBA	OBA	TBA
1	0.146	0.123	0.082	0.092
2	0.003	0.003	0.004	0.003
3	0.456	0.523	0.134	0.834
4	32.832	80.881	0.217	7.504
5	1.539	2.824	1.768	2.251
6	70.266	120.294	20.605	14.884
7	14.479	10.423	10.107	35.036
8	0.435	0.446	0.202	0.225
9	237.581	485.998	41.579	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 Run Times

The queries 8 and 9 take profit from the new length attribute of the mgpoint the attribute takes very small storage space compared with the trip data of a mgpoint but reduces the computation time of driven distances from O(n) to O(1) if n is the number of units of the mgpoint. The advantage can be seen very good if you compare the different run times of query 9. While the data model of the BerlinMOD Benchmark has to add the length of each unit of the n mpoint, which takes O(n) time, the network data model returns the driven distance in O(1) time. The advantage is bigger for bigger scalefactors.

Interesting is also the comparision of the run times of query 10. In the object based approach (OBA) the network data model outperforms the data model of free movement in two dimensional space at all three scalefactors, although we do a retranslation of the mgpoint values into mpoint values to use the existing Euclidean Distance operation for mpoint values. In the trip based approach (TBA) the data model of free movement outperforms the network data model at the two smaller scalfactors, while at scalefactor 1.0 the network data model outperforms the BerlinMOD Benchmark data model.

Only the queries 7, 11, and 14 show a bad performance in the network data model compared to the BerlinMOD Benchmark data model. The queries 7 and 14 use a common central functionality to decide if a position given by a gpoint is inside a given set of *routeintervals* or not. This function seems to be the problem of our current network data model implementation in SECONDO. All other functions used by the queries 7 and 14 are also used in other queries which have a significantly better performance.

The bad performance of query 11 is not so simple to explain. Query 12 extends query 11, such that we expected the runtime of query 12 to be longer than the the runtime of query 11. But the

experimental results are contrary. In our experiments we detected that the run time of a query are influenced by cache effects depending on the environment of the single query. Such that depending of the query in a sequence of queries the run time for the single query changes. While query 11 is not supported by cache effects in combinatino with query 10, query 12 is supported by cache effects from query 11. The run time differences are not so big compared to the total run time of all queries that the very good total result would be significantly changed by a little longer run time of query 12.

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 showed that the network data model outperforms the data model of free movement in the two dimesional space by orders of magnitudes with respect to storage space and query run times.

The good results of the network data model encourages us to work on a extension of the Berlin-MOD Benchmark to 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. Which will be presented in near future.

Other directions of our actual work is data mining and aggregation in our network data model to examine traffic flow, and detect and represent traffic jams.

Other interesting themes for future work on the network data models is the efficient computation of network distances especially 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] 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.

[6] 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.

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

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

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

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

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

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

[20] 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.