Implementation of Network Data Model in SecondoDBMS

Simone Jandt

Last Update: June 23, 2010

Contents

1 Introduction

The network data model was first presented in [6]. The network data model is restricted to represent objects which are constrained by a given network, e.g. cars on a street network. For this network constrained objects the network data model delivers a complete system of data types and operations.

In the next sections we describe our implementation of the network data model in the extensible SECONDO DBMS [1, 4, 7]. Parts of this network implementation have been done by one of our students as final thesis [9].

The central idea of the network data model is that movements are restricted to given networks. Cars use street networks and trains railway networks. It is natural for us to speech about the position of a place relative to the street network, instead of giving its absolute position in coordinates. In the network data model all positions are given relatively to the routes of the network. The temporal element is represented by a time sliced representation of the spatio-temporal elements as described in [6, 8].

The implementation of the network model in SECONDO is splitted into two algebra modules. One contains the spatial data types and operations (`NetworkAlgebra`), and the other one contains the spatio-temporal data types and operations (`TemporalNetAlgebra`).

We describe first the implemented data types of both algebra modules in Section 2 followed by the implemented operations on this data types in Section 3.

2 Implemented Data Types

All data types have a additional Boolean parameter, telling if the object of the data type is well defined or not. We will not mention this flag at every data type description.

2.1 NetworkAlgebra

The four implemented data types of the `NetworkAlgebra`-Module are: *network*, *gpoint*, *gpoints*, and *gline*.

2.1.1 *network*

The data type *network* is the central data type of the network data model. In the data model it consists of two relations routes and junctions describing the spatial structure of the (street) network.

The implementation of the *network* consists of three different relations (see tables 1 - 3); one contains the routes data (streets), one contains the junctions data (crossings), and one contains the sections (street parts between two crossings, or a crossing and the end of the street) of the network. Furthermore, the *network* consists of four B-Trees, indexing the route identifiers of the routes, junctions, and sections relation. A spatial R-Tree, indexing the ROUTE_CURVE attribute of the routes relation. A network identifier (*int*). And two sets connecting section identifiers of sections which are adjacent [1].

[1] Two sections are adjacent, iff they are connected by a junction, and the lanes of the streets are connected by the junction.

Attribute	Data Type	Explanation
JUNCTION_ROUTE1_ID	*int*	Route identifier of the first[2] route of the junction.
JUNCTION_ROUTE1_MEAS	*real*	Length of the route part between the start of the route and the junction.
JUNCTION_ROUTE2_ID	*int*	Route identifier of the second route of the junction.
JUNCTION_ROUTE2_MEAS	*real*	Length of the route part between the start of the route and the junction.
JUNCTION_CC	*int*	The connectivity code[3] tells us for which lanes of the streets an transition exists on the junction.
JUNCTION_POS	*point*	Representing the spatial position of the junction in the 2D space.
JUNCTION_ROUTE1_RC	*TupleIdenitfier*[4]	Identifies the tuple of the first route of the junction in the routes relation.
JUNCTION_ROUTE2_RC	*TupleIdenitfier*	Identifies the tuple of the second route of the junction in the routes relation.
JUNCTION_SECTION_AUP_RC	*TupleIdentifier*	Identifies the tuple of the section upwards of the junction on the first route in the sections relation.
JUNCTION_SECTION_ADOWN_RC	*TupleIdentifier*	Identifies the tuple of the section downwards of the junction on the first route in the sections relation.
JUNCTION_SECTION_BUP_RC	*TupleIdentifier*	Identifies the tuple of the section upwards of the junction on the second route in the sections relation.
JUNCTION_SECTION_BDOWN_RC	*TupleIdentifier*	Identifies the tuple of the section downwards of the junction on the second route in the sections relation.

Table 1: The junctions relation of the data type *network*

Attribute	Data Type	Explanation
ROUTE_ID	*int*	Route Identifier.
ROUTE_LENGTH	*real*	Length of the route.
ROUTE_CURVE	*sline*[5]	Spatial geometry data of the route.
ROUTE_DUAL	*bool*	*TRUE* means that the both lanes of the street are separated.
ROUTE_STARTSSMALLER	*bool*	*TRUE* means the route curve starts at the lexicographical smaller endpoint

Table 2: The routes relation of the data type *network*

Attribute	Data Type	Explanation
SECTION_RID	*int*	Route identifier of the route the section belongs to.
SECTION_MEAS1	*real*	Length of the route part before the section start point from the begin of the route.
SECTION_MEAS2	*real*	Length of the route part before the section end point from the begin of the route.
SECTION_DUAL	*bool*	*TRUE* means that the both lanes of the section are separated.
SECTION_CURVE	*sline*	Spatial geometry of the section in the plane.
SECTION_CURVE_STARTS_SMALLER	*bool*	*TRUE* means the section curve starts at the lexicographical smaller endpoint
SECTION_RRC	*TupleIdentifier*	Identifies the tuple of the route the section belongs to in the routes relation.

Table 3: The sections relation of the data type *network*

2.1.2 *gpoint*

A *gpoint* describes a single position in the network. It consists of the network identifier (*int*), and the route location. The route location is given by the route identifier (*int*), the distance (*real*) from the start of the route, and a parameter side(*side*).

Side has three possible values (Down, Up, None). Up(Down) means a position can only be reached from the up(down) side of a route. None means a position can be reached from both sides of the route. For example motorway service areas on German Highways can always be reached only from one side of the highway.

2.1.3 *gpoints*

gpoints is a set of *gpoint*. Implemented by Jianqiu Xu.

2.1.4 *gline*

A *gline* describes a part of the network. This part of the network might be a path between two *gpoint* or a district of a town. The data type *gline* consists of a network identifier (*int*), and a set of *route intervals*, the length of the gline (*real*), and a Boolean flag telling if the *route intervals* are stored sorted or not.

[2] The first route identifier of a junction will always be the lower route identifier of the two routes which cross in the junction.
[3] See [6] for detailed information about the meaning of the different connectivity code values.
[4] *TupleIdentifier* is a SECONDO data type identifying tuples in other relations.
[5] *sline* is a set of segments representing the curve of the route in the 2D plane.

Every *route interval* consists of a route identifier (*int*), and two position values (*real*), defining the start and the end position of the *route interval* on the route. The positions are given by the distance of the position from the start of the route.[6].

The computation time of many algorithms on *gline* values can be reduced, if the set of *route intervals* is sorted. Sorted means that the set of *route intervals* fullfills the following conditions:

- All *route intervals* are disjoint.
- The *route intervals* are sorted by ascending route identifiers.
- If two *route intervals* have the same route identifier the *route interval* with the smaller start position is stored first.
- All start positions are less or equal to the end positions.

Unfortunately not all *gline* values can be stored sorted. If we describe an district or the parts of the network traversed by an *mgpoint* (See 2.2.1) we can store the *route intervals* sorted, because it is regardless in which sequence we read the *route intervals* describing the network part. But, if the *gline* represents a path between two *gpoint* a and b the *route intervals* must be stored in the sequence they are used in the path. And this will nearly never be a sorted sequence of *route intervals* as defined before. We introduced the Boolean flag *sorted* to solve this problem. Whenever a *gline* value can be stored sorted we do so and set the sorted flag to *TRUE*. Algorithms which can take profit from sorted sets of *route interval*s check the sorted flag and performs a binary scan for sorted and a linear scan for unsorted *gline* values. If r ist the number of *route interval*s of a *gline* the computation time is reduced from $O(r)$ for unsorted *gline* values to $O(\log r)$ for sorted *gline* values.

We pay for the advantage of reduced compuatation time for sorted *gline* values by the higher time complexity of algorithms which produce *gline* values. Sorting and compressing *route interval*s needs time. But we think, that this time is well invested, because sorting a *gline* is done once, whereas the sorted *gline* value can be used many times by other algorithms.

2.2 TemporalNetAlgebra

In the moment only the temporal version of the *gpoint* the so called *mgpoint* (short form from *moving*(*gpoint*)) is implemented in the `TemporalNetAlgebra`. This includes the implementation of the *unit*(*gpoint*) (short *ugpoint*) and the *intime*(*gpoint*) (short *igpoint*). As explained in the following subsections.

Recently the `TemporalNetAlgebra` has been extended by a new data type *mgpsecunit* and operations **mgpsecunits** to create *stream*s of this data type from *mgpoint* values. The new data type is expected to be usefull in the context of traffic estimation.

2.2.1 *mgpoint*

The main parameter of a *mgpoint* is a set of *ugpoint*s (see 2.2.2) with disjoint time intervals. The time intervals of the *ugpoint*s must be disjoint, because nothing can be at two different places at the same time. The *ugpoint*s are stored in the *mgpoint* sorted by ascending time intervals. This allows us to perform a binary scan on the units of the *mgpoint* to find a given time instant within the definition time of the *mgpoint*. An *igpoint* (see 2.2.3), gives the position of the *mgpoint* at a given time instant.

In our experiments we extended the *mgpoint* from [6] with some additional parameters:

- *length* (*real*): The length parameter stores the distance driven by the *mgpoint*.
- *trajectory* (sorted set of *route interval*s)[7]: Represents all the places ever traversed by the *mgpoint*.
- *trajectory_defined* (*bool*): *TRUE*, if the trajectory parameter is well defined.
- *bbox* (*rect3* three dimensional rectangle): Spatio-temporal bounding box of the *mgpoint*.

The trajectory parameter reduces the time to decide if a *mgpoint* ever passed a given place (*gpoint* or *gline*) or not. Instead of an linear check of all m units of an *mgpoint* we can perform a binary scan on the much lower number r of route intervals of the trajectory parameter. Such that the time complexity is reduced from $O(m)$ to $O(\log r)$ with $r \ll m$. The trajectory parameter is not maintenanced by every operation. The Boolean flag *trajectory_defined* tells us, if the trajectory parameter is actually well defined or if it has to be recomputed first.

In the network data model all spatial information is only stored in the central network object. Therefore it is very expensive to get spatial informations especially for *mgpoint*s. Although the *mgpoint* stays on the

[6]As you can see different from [6] the *side* value for *route intervals* is not implemented yet.

[7]This is a work around, caused by the fact that the SECONDO DBMS does not allow us to use a *gline* value as parameter of an *mgpoint*.

same route with the same speed the *mgpoint* might move in different spatial directions within a single unit. For example a car may drive downhill in serpentines. In this case it is not enough to look at the start and end position of the unit to compute the spatial part of the bounding box. The complete route part passed in this unit must be inherited in the computation of the spatial part of the bounding box. That makes the computation of the bounding box of the *mgpoint*, which is the union of all the unit bounding boxes very expensive. We introduced the bbox parameter to save our computational work. The bbox value is not maintenanced at every change of the *mgpoint* and it is only computed on demand using the trajectory of the *mgpoint* or stored if we could get it for free [8].

2.2.2 *ugpoint*

The *ugpoint* consists of a time interval, an start, and an end *gpoint*, whereby both *gpoint* have the same network and route identifiers.

The time interval consists of an starting time instant, an end time instant, and two boolean flags, one for each of the both time instants, indicating if the time instant is part of the interval or not.

With help of this parameters we could compute the exact position of the *ugpoint* at each time instant within the time interval. And, assumed the *ugpoint* reaches the query *gpoint* within the time interval, we can compute the time instant when a *ugpoint* reaches a given *gpoint*.

2.2.3 *igpoint*

The *igpoint* consists of a time instant and a *gpoint* representing the position of the *mgpoint* at the given time instant.

2.2.4 *mgpsecunit*

The data type *mgpsecunit* (see 2.2.4) was introduced in November 2009 to support better traffic estimation. It reduces the complex informations given in a *mgpoint* to the values which are useful for traffic estimation. Possibly this reduced information can be used to build a spatio-temporal index over *mgpoint* values in a later SECONDO version.

secId	*int*	Section identifier for a network section
partNo	*int*	Partition number on this section[9].
direct	*int*	Moving direction of the *mgpoint* within this section part. (0 = Down, 1 = Up)
avgSpeed	*real*	Average speed of the *mgpoint* within this section (part)
time	*Interval¡Instant¿*	Time interval the *mgpoint* moved within this section

Table 4: Description of data type *mgpsecunit*

3 Implemented Operations

In the next subsections we describe the implemented operations of the network data model. For every operator we present its signature, an example call and informations about the used algorithms and if interesting the time complexity of the algorithm.

3.1 Network Constructor

int × *relation* × *relation* → *network* **thenetwork**(*n*, *routes*, *junctions*)

The operator **thenetwork** constructs the *network* object with the given identifier n[10] from the two given relations by algorithm 1. Therefore the two input relations should contain the following attributes:

[8] If we translate a *mpoint* into an *mgpoint* we can copy the bounding box of the *mpoint* without compuatitional effort.

[9] For traffic estimation it might be useful to divide long sections into smaller parts. The operation **mgpsecunits** (see 3.10.6) which constructs the *mgpsecunits* from a set of *mgpoints* has a parameter which gives a maximum section length. Sections which are longer than this value will be divided up into several parts of this length. The partitioning starts at the smaller point of the section and the first part has the number 1. The length of the last part might be shorter than the given length value.

[10] If n is already used as network identifier in the database the next free integer value $i \geq n$ is used as network identifier instead of n.

- *routes*: route identifier (<u>*int*</u>), length of the route (<u>*real*</u>), geometry of the route curve (<u>*sline*</u>), and the two Boolean flags dual and startssmaller
- *junctions*: first route identifier (<u>*int*</u>), position on first route (<u>*real*</u>), second route identifier (<u>*int*</u>), position on the second route (<u>*real*</u>), and the connectivity code (<u>*int*</u>)

Algorithm 1 thenetwork $(n, route, junctions)$

Require: An integer $n \geq 0$, *routes* and *junctions* relation as described.

```
1: Create Network empty network object net with id n
2: Copy routes to routes relation of net
3: Construct B-Tree indexing route identifiers in routes relation
4: Construct R-Tree indexing route curves in routes relation
5: Copy junction to junctions relation of net and add route tuple identifiers from routes relation
6: Construct B-Trees indexing the first / second route identifiers in the junctions relation
7: for Each tuple in routes relation do
8:   for Each junction on this route do
9:     Compute the Up and Down sections
10:    Add the sections to the sections relation
11:    Add the section identifiers to the junctions relation
12:  end for
13: end for
14: Construct B-Tree indexing route identifiers in the sections relation
15: for Each junction of the junctions relation do
16:   Find pairs of adjacent sections and fill adjacency list
17: end for
```

Let r be the number of entries in *routes*, j the number of entries in *junctions*. and j_i the number of junctions on route r_i from the routes relation. The number of entries in the sections relation of *net* is $\sum_{i=1}^{r} j_i + 1 = r\sum_{i=1}^{r} j_i$, The time complexities of the single steps of algorithm 1 are:

1 O(1)

2 O(r)

3 + 4 O($r \log r$)

5 O(j)

6 O($j \log j$)

7 - 13 ($\sum_{i=1}^{r} j_i$)

14 O($r\sum_{i=1}^{r} j_i \log r \sum_{i=1}^{r} j_i$)

15 - 17 O(j)

For all steps together we get a time complexity of

$$O(1 + r + r\log r + j + j\log j + \sum_{i=1}^{r} j_i + r\sum_{i=1}^{r} j_i \log(r\sum_{i=1}^{r} j_i) + j) = O(r\sum_{i=1}^{r} j_i \log(r\sum_{i=1}^{r} j_i))$$

, because $r, j \leq r\sum_{i=1}^{r} j_i$.

3.2 Translation from 2D Space into Network Data Model

The next operations are used to translate spatial and spatio-temporal data types from the two dimensional plane data model [2, 5, 8] of the SECONDO DBMS into the network data model representation. In [6] this operations are all called **in_network** with different signatures. All translations will only be successful if the values of the two dimensional data types are aligned to the given network otherwise the network representation of the object is not defined.

<u>*network*</u> × <u>*point*</u> → <u>*gpoint*</u>	**point2gpoint**(*network*, *point*)
<u>*network*</u> × <u>*line*</u> → <u>*gline*</u>	**line2gline**(*network*,*line*)
<u>*network*</u> × <u>*mpoint*</u> → <u>*mgpoint*</u>	**mpoint2mgpoint**(*network*, *mpoint*)

3.2.1 point2gpoint

The operation **point2gpoint** translates a *point* value into a *gpoint* value of the given network if possible. If r_r is the number of routes in the routes relation, and c_r the number of candidate routes the algorithm 2 has a worst case complexity from $O(\log r_r + c_r)$.

Algorithm 2 point2gpoint(p)

```
Use R-Tree of routes relation to get the candidate routes close to the point
found = false
while not found and not isEmpty(candidateRoutes) do
  if Distance of point from route = 0 then
    found = true
    Compute position of point on route
  end if
end while
return gpoint for p
```

3.2.2 line2gline

The operation *line2gline* translates an *line* value into an sorted *gline* value. The algorithm takes every segment of the *line* value and tries to find the start and end of the segment on the same route using a variant of **point2gpoint**. The computed route intervals are sorted, merged and compressed with help of an *RITree*[11] before the resulting *gline* is returned.

If h is the number of segments of the *line* value, r_r and c_r are defined as in **point2gpoint**, and r is the number of resulting *route interval*s the time complexity is $O(h(\log r_r + c_r + \log r))$. The summand $h \log r$ is caused by merging and sorting the *route interval*s with the RITree. As mentioned before (see 2.1.4) we think that the many times reduced runtimes are bigger than the additional computation time invested at this point.

3.2.3 mpoint2mgpoint

The operation **mpoint2mgpoint** translates an *mpoint* value which is constrained by the network into an *mgpoint* value. The single steps of algorithm 3 have the following time complexities. The initialization in 1-2 is

Algorithm 3 mpoint2mgpoint(*mpoint*,*net*)

```
1: Initialize bulkload with empty mgpoint
2: upoint = first unit mpoint
3: Initialize ugpoint = net values of upoint)
4: for Each upoint in mpoint do
5:   if Endpoint of upoint is on same route than ugpoint then
6:     if Direction and speed stay the same then
7:       Extend ugpoint
8:     else
9:       Add ugpoint to mgpoint
10:      Add route interval of ugpoint to trajectory
11:      Ugpoint = net values of upoint
12:    end if
13:  else
14:    Add ugpoint to mgpoint
15:    Add route interval of ugpoint to trajectory
16:    Search upoint on adjacent sections
17:    Ugpoint = net values of upoint
18:  end if
19: end for
20: Add ugpoint to mgpoint
21: Finish bulkload mgpoint
22: Copy bounding box of mpoint to mgpoint
```

[11]The RITree is a binary search tree for *route intervals*. It is implemented in the NetworkAlgebra of SECONDO. It sorts and compress *route interval*s in $O(r_{in} \log r_{out})$ time, if r_{in} is the number of inserted *route interval*s and r_{out} is the number of resulting *route interval*s.

done in O(1). The computation of the ugpoint in 3 needs a variant of the **point2gpoint**, such that O($\log r_r + c_r$) is needed. The for-loop in 4 is executed m times if m is the number of units of the *mpoint* value. The for-loop knows 3 cases, which have different time complexity:

1. extend ugpoint is done in O(1) .
2. write ugpoint and initialize new one on the same route is done in O(1) time
3. write ugpoint and search adjacent route to initialize new one depends on the number of routes x_i connected by the crossing i. In the worst case the time complexity is O($x_i(\log r_r + c_r)$).

In the worst case we get a total time complexity for the for-loop from $O(m(\log r_r + c_r)\sum_{i=1}^{m}(x_i))$. The last steps of the algorithm needs only O(1) time again, such that the time complexity of the for-loop dominates the algorithms run time and the worst case time complexity of the algorithm is $O(m(\log r_r + c_r)\sum_{i=1}^{m}(x_i))$. But this worst case takes only place if the car changes the route in each unit, all other cases need only O(1) time such that we will have a much smaller computation time than in the most cases.

3.3 Translation from Network Data Model into 2D Space

gpoint → *point*	**gpoint2point**(*gpoint*)
gline → *line*	**gline2line**(*gline*)
mgpoint → *mpoint*	**mgpoint2mpoint**(*mgpoint*)

In [6] this operations are called **in_space**. The translation from network constrained data types into data types of the two dimensional plane is always possible.

3.3.1 gpoint2point

The operation *gpoint2point* translates a *gpoint* value into a *point* value. The algorithm uses the B-Tree of the routes relation to get the route curve of the route the *gpoint* is connected to in O($\log r_r$) time. Then the spatial position of the *gpoint* on this route is computed in O(h) time if h is the number of line segmenst defining the route curve. Together we get a worst case time complexity of O($h + \log r_r$).

3.3.2 gline2line

The operation **gline2line** translates a *gline* value into an spatial *line* value. The algorithm uses the B-Tree index on the routes relation to get the corresponding route curve for every route interval of the *gline*. And computes the corresponding half segments which are put in the resulting *line*.

Let m be the number of route intervals of the *gline*, and r the number of routes in the network, and h the maximum number of half segments of a *line* value for a route interval. For each route interval we need O($\log r$) time to get the route curve and O(h) time to get the half segments of the route interval. The time complexity of the whole operation is O($m(h + \log r)$)

3.3.3 mgpoint2mpoint

The operation **mgpoint2mpoint** translates a *mgpoint* value into a corresponding *mpoint* value. It is not enough to compute only the *point* values for the start and end *gpoint* of every unit because if a *ugpoint* passes different half segments of the route curve it must be divided up into different *upoint*. Because at every new halfsegment it changes the moving direction and this must be saved in the *mpoint*.

The algorithm gets the first *ugpoint* of *mgpoint* and uses the BTree index of the routes relation to get the tuple with the *ugpoint*s route curve. Find the first *gpoint* of the *ugpoint* on the route curve. For every *ugpoint* of *mgpoint* check if the route identifier of actual route is equal to the route identifier of *ugpoint*. If this is not the case use the BTree index of the routes relation of the *network* to get the tuple with the *ugpoint*s route curve. If the end position of the *ugpoint* is on the same half segment of the route curve than the start position compute *point* for end *gpoint* and write unit to *mpoint* and continue with the next unit of the *mgpoint*. If the end position of the *ugpoint* is not on the same half segment of the route curve and the *ugpoint* is moving up(down) compute the time instant the *ugpoint* reaches the end(start) position of the actual half segment. Write the *upoint* to the resulting *mpoint*. Get next half segment of the route curve in up(down) direction repeat the last computation until the half segment containing the end point of the *ugpoint* is reached.

Let r be the number of routes in the routes relation, and m the number of units of the *mgpoint*, and h the maximum number of half segments of a route curve. The worst case time complexity of the algorithm is O($m(h + \log r)$).

3.4 Extract Attributes

The operators of table 5 return the attributes from the different data types in O(1) time.

Operator	Signature	Explanation
routes	*network*→ *routes relation*	Returns the routes relation of the *network*
junctions	*network* → *junctions relation*	Returns the junctions relation of the *network*
sections	*network* → *sections relation*	Returns the sections relation of the *network*
no_components	*gline* → *int* *mgpoint* → *int*	Returns the number of *route intervals* respectively units of the first argument.
isempty	*gline* → *bool* *mgpoint* → *bool*	Returns *TRUE* if the first argument X is not defined or **no_components**$(X) = 0$.
length	*gline* → *real* *mgpoint* → *real*	Returns the length of the *gline* or the driven distance of the *mgpoint*
initial	*mgpoint*→*igpoint*	Returns the first position and start time of the *mgpoint*.
final	*mgpoint*→*igpoint*	Returns the last position and end time of the *mgpoint*.
unitrid	*ugpoint* → *real*	Returns the route identifier of the *ugpoint*.
unitstartpos	*ugpoint* → *real*	Returns the start position of the *ugpoint*.
unitendpos	*ugpoint* → *real*	Returns the end position of the *ugpoint*.
unitstarttime	*ugpoint* → *real*	Returns the start time instant of the *ugpoint* as *real* value.
unitendtime	*ugpoint* → *real*	Returns the end time instant on the *ugpoint* as *real* value.
startunitinst	*ugpoint* → *instant*	Returns the start time instant of the *ugpoint*.
endunitinst	*ugpoint* → *real*	Returns the end time instant on the *ugpoint*.
val	*igpoint* → *gpoint*	Returns the *gpoint* of the *igpoint*
inst	*igpoint* → *instant*	Returns the time instant of the *igpoint*

Table 5: Operators returning simple attributes

The following operators return the more complex attributes of the network data types.

mgpoint → *gline*	**trajectory**(*mgpoint*)
mgpoint → *periods*	**deftime**(*mgpoint*)
ugpoint → *periods*	**deftime**(*ugpoint*)
mgpoint → *stream*(*ugpoint*)	**units**(*mgpoint*)

3.4.1 trajectory

The operation **trajectory** returns a sorted *gline* value representing all the places traversed by the *mgpoint*. If the trajectory parameter is defined the route intervals are returned as a *gline* value immediately. Otherwise the trajectory parameter is computed by a linear scan of the units of the *mgpoint*. In the last case the route intervals are sorted, merged and compressed with help of a RITree (see 3.2.2).

If the trajectory is defined and r is the number of route intervals of the trajectory the time complexity is $O(r)$. Otherwise the time complexity is $O(m + m \log r)$, if m is the number of units of the *mgpoint*. The last time complexity value could be reduced to $O(m)$ if we store the computed route intervals immediately to the resulting *gline* value without sorting and compressing. But as mentioned in 2.1.4 we think that the overhead in computation time for sorting and compressing is well invested.

3.4.2 deftime

The operation **deftime** is defined for *mgpoint* and *ugpoint*. It returns the *periods* representing the definition times of the the *mgpoint* respectively the *ugpoint*.

This takes O(1) time for *ugpoint* value and $O(m)$ time for a *mgpoint* value with m units, because every unit of the *mgpoint* must be read to merge the definition times.

3.4.3 units

The operation **units** returns the m units of a *mgpoint* value as *stream* of *ugpoint* in $O(m)$ time.

3.5 Bounding Boxes

We know two different types of bounding boxes in the network data model. On the one hand spatio-temporal bounding boxes analogous to the BerlinMOD Benchmark data model. And on the other hand network bounding boxes where route identifiers and positions become coordinates so that R-Trees can be abused to index non spatial network data.

3.5.1 Spatio-Temporal Bounding Boxes

ugpoint → *rect3*	**unitboundingbox**(*ugpoint*)
mgpoint → *rect3*	**mgpbbox**(*mgpoint*)

The operations return the spatio-temporal bounding boxes of *ugpoint* respectively *mgpoint* as three dimensional rectangle with coordinates x_1, x_2, y_1, y_2, z_1 and z_2 of data type *real*.

unitboundingbox The spatial part of the unitbounding box (x-,y-coordinates) is defined as the spatial bounding box of the *route interval* covered by the *ugpoint*. And the temporal part (z-coordinates) of the unitbounding box is given by the *real* values representing the start and the end time instant of the time interval of the *ugpoint*. In detail the coordinates of the resulting rectangle are defined as follows:

- $x_1 = \min(x-$coordinate of the bounding box of the *route interval*)
- $x_2 = \max(x-$coordinate of the bounding box of the *route interval*)
- $y_1 = \min(y-$coordinate of the bounding box of the *route interval*)
- $y_2 = \max(y-$coordinate of the bounding box of the *route interval*)
- z_1 =start time instant as *real*
- z_2 =end time instant as *real*

The computation takes O(h) time, if h is the number of halfsegments passed within the *ugpoint*.

mgpbbox The spatial-temporal bounding box of the *mgpoint* is defined as the union of the bounding boxes of the units of the *mgpoint*. The computation would take a very long time if we use this definition for computation, because a *mgpoint* has many units. Therefore we introduced the parameter bbox to store a once computed spatio-temporal bounding box. Otherwise we use the trajectory parameter of the *mgpoint* to get the same result in much less time.

The algorithm first checks if the bounding box parameter is defined. If this is the case the bounding box is returned immediately in O(1) time. If the bounding box is not defined we distinguish between two cases:

1. If the trajectory is not defined we first compute the trajectory. And then use the trajectory as it has been defined before.
2. If the trajectory is defined we compute the union of the bounding boxes of the route intervals of the trajectory and extend the resulting two dimensional rectangle to a three dimensional rectangle computing the *real* values of the start and the end time instant of the *mgpoint*.

Let r be the number of *route intervals* of the *mgpoint*, m the number of units of the *mgpoint*, and h the maximum number of half segments covered by a route interval. The algorithm needs O(rh) time in case 2 and O($rh + m \log r$) time in case 1.

3.5.2 Network Bounding Boxes

The following operators return network bounding boxes respectively *streams* of network bounding boxes.

gpoint → *rect*	**gpoint2rect**(*gpoint*)
gline → *stream*(*rect*)	**routeintervals**(*gline*)
ugpoint → *rect3*	**unitbox**(*ugpoint*)
ugpoint → *rect*	**unitbox2d**(*ugpoint*)

All network bounding boxes are two (network) respectively three (network-temporal) dimensional rectangles with coordinates (x_1, x_2, y_1, y_2) respectively $(x_1, x_2, y_1, y_2, z_1, z_2)$. For network bounding boxes the both x-coordinates are always identically and defined by the route identifier of the object. The z-coordinates are defined as *real* value representing the start (z_1) respectively end time (z_2) of the time interval of the *ugpoint*.

gpoint2rect The operator **gpoint2rect** computes the network box of a *gpoint* value in O(1) time. The y-coordinates are defined as $y_1 = position - 0.000001$ respectively $y_2 = position + 0.000001$. The small *real* value is used to avoid problems with the computational inaccuracy of *real* values.

routeintervals The operation **routeintervals** returns a stream of two network boxes, one for each route interval of the *gline*. The y-coordinates are defined: $y_1 = \min$(start position, end position) and $y_2 = \max$(start position, end position).

The operation needs $O(r)$ time if r is the number of route intervals of the *gline*.

unitbox2d Returns a two dimensional rectangle for the *ugpoint* in $O(1)$ time. The y-coordinates are given by $y_1 = \min$(start position, end position) and $y_2 = \max$(start position, end position).

unitbox Returns a three dimensional rectangle for the *ugpoint* in $O(1)$ time. It extends the two dimensional rectangle of unitbox2d with z-coordinates defined by the *real* values of the time instants of the *ugpoint*.

3.6 Boolean Operations

Boolean operations check if the arguments hold special characteristics or conditions and return *TRUE* if this is the case, *FALSE* elsewhere. A special case is the operation **inside** for *mgpoint* because the argument and the returned value are *moving* objects.

gpoint × *gpoint* → *bool*	*gpoint1* = *gpoint2*
gline × *gline* → *bool*	*gline1* = *gline2*
gline × *gline* → *bool*	**intersects**(*gline1*, *gline2*)
mgpoint × *gpoint* → *bool*	*mgpoint* **passes** *gpoint*
mgpoint × *gline* → *bool*	*mgpoint* **passes***gline*
gpoint × *gline* → *bool*	*gpoint* **inside** *gline*
mgpoint × *gline* → *mbool*	*mgpoint* **inside** *gline*
mgpoint × *instant* → *bool*	*mgpoint* **present** *instant*
mgpoint × *periods* → *bool*	*mgpoint* **present** *periods*

3.6.1 =

The operator = compares the parameters of the arguments and returns *TRUE* if they are equal, *FALSE* elsewhere. For two *gpoint*s this can be done in $O(1)$ time. For two *gline* we have to compare all r *route intervals* of the both *gline* this will take $O(r)$ time. But the computation will stop immediately if a difference between the two *gline* is detected and *FALSE* returned.

3.6.2 intersects

The algorithm checks if there is a pair of *route intervals* (one from *gline1* and one from *gline2*) that intersects. Because sorted *gline* can reduce computation time the algorithm knows three cases:

1. If Both *gline*s are sorted, a parallel scan through the route intervals of both *gline* is performed.
2. If only one *gline* is sorted, a linear scan of the unsorted *gline* is performed. For each route interval of the unsorted *gline* a binary search for a overlapping route interval is performed on the sorted *gline*.
3. If both *gline* are not sorted, a linear scan of the first *gline* is performed. And for every route interval a linear scan for overlapping route intervals is performed on the second *gline*.

In all three cases *TRUE* is returned and computation stops immediately if a intersecting pair of *route intervals* has been detected.

If r is the number of *route intervals* of *gline1* and s for *gline2*. We get the following time complexities for the three cases:

1. $O(r + s)$
2. $O(r \log s)$ respectively $O(s \log r)$, depending on which of the both *gline* is sorted.
3. $O(rs)$

3.6.3 passes

The operation **passes** checks if the *mgpoint* ever passes the given *gpoint* respectively *gline*. The algorithm uses the trajectory parameter of the *mgpoint*. If the trajectory is not defined the trajectory is computed first with help of **trajectory**(*mgpoint*). In this case we must add the time complexity of the operator **trajectory** to the time complexity of **passes**. In the following we assume that the trajectory is defined.

gpoint A binary search of a route interval that contains the *gpoint* is performed on the trajectory parameter. This will take $O(\log r)$ time, if r is the number of route intervals in the trajectory parameter.

gline The algorithm is divided up into two cases:

1. If the *gline* is sorted a parallel scan of the route intervals of the *gline* and the route intervals of the trajectory parameter is performed to find a intersecting pair of route intervals.

2. If the *gline* is not sorted a linear scan of the route intervals of the *gline* is performed. And for every route interval a binary search of a intersecting route interval is performed on the trajectory parameter.

In both cases the computation is stopped immediately and *TRUE* returned if a intersecting *route interval* has been found.

If r is the number of route intervals of the *mgpoint*, and s is the number of route intervals of the *gline* we get for case 1 a time complexity of $O(s + r)$ and for case 2 a time complexity of $O(s \log r)$

3.6.4 Inside

The operation checks if the *gpoint* respectively *mgpoint* is inside the *gline*.

gpoint In case of the *gpoint* the algorithm knows two different cases:

1. If the *gline* is sorted a binary search for a route interval containing the *gpoint* is performed on the route intervals of the *gline*.

2. If the *gline* is not sorted a linear scan of the route intervals of the *gline* is performed to find a route interval containing the *gpoint*.

If r is the number of *route intervals* of the *gline* the time complexity will be $O(\log r)$ for a sorted *gline* and $O(r)$ for a unsorted *gline*.

mgpoint For a *mgpoint* a *mbool*[12] which is *TRUE* every time interval the *mgpoint* moves inside *gline* and *FALSE* elsewhere is returned. The algorithm checks for every unit of the *mgpoint* if there is any intersection with the route intervals of the *gline*. Based on this values the resulting *mbool* is computed. The search for intersecting route intervals is different for sorted and unsorted *gline*.

- If the *gline* is sorted a binary search on the route intervals is performed.

- If the *gline* is not sorted a linear scan on the route intervals is performed.

Let m be the number of units of *mgpoint* and r the number of route intervals of the *gline*. The operation takes $O(m \log r)$ time if the *gline* is sorted. And $O(mr)$ time if the *gline* is not sorted.

3.6.5 present

The operation checks the temporal attribute of the *mgpoint*.

instant The algorithm performs a binary search on the units of the *mgpoint* if a corresponding *ugpoint* is found *TRUE* is returned, *FALSE* elsewhere. This takes $O(\log m)$ time if m is the number of units of the mgpoint.

periods The algorithm performs a parallel scan through the units of the *mgpoint* and the *periods* if a intersecting time interval is found *TRUE* is returned, *FALSE* elsewhere. The worst case time complexity is $O(m+n)$ if m is the number of units of the *mgpoint* and n the number of temporal units of the *periods*.

3.7 Merging Data Objects

gline × *gline* → *gline*	*gline1* **union** *gline2*
mgpoint × *mgpoint* → *mgpoint*	*mgpoint1* **union** *mgpoint2*

If possible **union** merges the two argument objects into one result object of the same data type.

[12] Short form of *moving*(*bool*). A *mbool* changes its *bool* value within time. See [?] for more details.

3.7.1 *gline*

The operation returns a sorted *gline* which contains the union of the route intervals of the both *gline*. The algorithm knows two cases:

- both *gline* are sorted.
- one or both *gline* are not sorted.

If both *gline* are sorted we perform a parallel scan through the route intervals and compare the actual route intervals of the both *gline*. If the route intervals don't intersect the smaller route interval is added to the resulting *gline* and the next route interval from the *gline* the added route interval was from is taken to continue the scan. If the both route intervals intersect a single route interval containing both route intervals is created and new route intervals from both *gline* are taken. If one or both new route intervals intersect with the merged route interval the merged route interval is extended to contain the intersecting route interval and the next route interval from the *gline* the last merged route interval was from is taken. We continue merging route intervals until there is no more route interval intersecting with the merged route intervals. The the merged route interval is stored to the result and we continue the scan until all route intervals of both *gline* have been added to the resulting sorted *gline*.

If one or both *gline* are not sorted. All route intervals of both *gline* are filled into a *RItree* to sort and compress them and the resulting sorted *gline* is returned.

Let r respectively s be the number of route intervals of the both *gline* and k the number of resulting route intervals. If both *gline* are sorted the time complexity is $\mathrm{O}(r + s)$ in all other cases we get a time complexity of $\mathrm{O}((r + s) \log k))$.

If we don't want to store the resulting *gline* sorted we could simply add every route interval of the both *gline* into the new *gline* in $\mathrm{O}(r + s)$ time. But as mentioned before in 2.1.4 many algorithms take profit from sorted *gline* values. We think that the additional time is well invested.

3.7.2 *mgpoint*

The operation merges two *mgpoint* if the time intervals of all units of the both *mgpoint* are disjoint or the units with the same time intervals have identical values. The algorithm performs a parallel scan through the units of the both *mgpoint* and writes the units of the *mgpoints* in ascending order of their time intervals to the resulting *mgpoint*. If there are overlapping time intervals the algorithm checks if the both *ugpoints* are identically. If the *ugpoints* are identically one of them is written to the result and the other one ignored. If the *ugpoints* are not identically the computation is stopped and the result is undefined. This takes $\mathrm{O}(m + n)$ if m respectively n is the number of units of the both *mgpoint*.

3.8 Path Computing

gpoint × *gpoint* → *gline* **shortest_path**(*gpoint1*, *gpoint2*)

The operation computes the shortest path in the network between *gpoint1* and *gpoint2* using Dijkstras Algorithm of shortest paths [3]. In the worst case this takes $\mathrm{O}(s + j \log j)$ time if j be the number of junctions and s be the number of sections in the network.

3.9 Distance Computing

There is a big difference between the Euclidean Distance and the Network Distance between between two places a and b. The Euclidean Distance is given by the length of the beeline between the two places regardless from existing paths in the network between the two locations. Contrary to this the Network Distance is given by the length of the shortest path between a and b in the network. According to this, and contrary to the Euclidean Distance, the Network Distance from a to b might be another than the Network Distance from b to a. Because there might be one way routes in the shortest path from a to b, which cannot be used in the shortest path from b to a.

3.9.1 Euclidean Distances

gpoint × *gpoint* → *real* **distance**(*gpoint1*, *gpoint2*)
gline × *gline* → *real* **distance**(*gline1*, *gline2*)
mgpoint × *mgpoint* → *mreal*[13] **distance**(*mgpoint1*, *mgpoint2*)

Although Euclidean Distances don't make much sense in a network environment we implemented the **distance** operation which computes the Euclidean Distance of two *gpoint*, *gline*, or *mgpoint* for network objects for convenience. All following algorithms for Euclidean Distance computing do first a translation of the network data types into equivalent 2D data types using the operators of 3.3 before they use the existing distance operation of this equivalent 2D data types to compute the Euclidean distance between the network objects. The time complexity is therefore always given by the sum of the translation time and the time for the distance computation. We get the following time complexities:

- *gpoint*: O(O(**gpoint2point**) + O(**distance** (*point*, *point*))).
- *gline*: O(O(**gline2line**) + O(**distance**(*line*, *line*)))
- *mgpoint*: O(O(**mgpoint2mpoint**) + O(**distance** (*mpoint*, *mpoint*)).

3.9.2 Network Distances

gpoint × *gpoint* → *real*	**netdistance**(*gpoint1*, *gpoint2*)
gline × *gline* → *real*	**netdistance**(*gline1*, *gline2*)

As mentioned before the Network Distance is given by the length of the shortest path between the arguments. In the simple case of two *gpoint* we use **length** (**shortest_path** (*gpoint1*, *gpoint2*)) and get a time complexity of O(O(**shortest_path** + O(**length** (*gline*))[14].

If the arguments are *gline*s we have to return the length of the minimum shortest path between a *gpoint* from *gline1* and a *gpoint* from *gline2*. The algorithm first computes the bounding gpoints[15] for each of the both *gline*. Then we check for each possible pair of bounding gpoints one of *gline1* and one of *gline2*:

- If one of the both *gpoint* is inside the other *gline*. If this is the case, the both *gline*s intersect. The Network Distance is 0.0 and computation is stopped immediately.
- If the *gline*s do not intersect the distance of the both *gpoint* is computed using **netdistance**(*gpoint*, *gpoint*)

If the distances of all pairs of bounding gpoints has been computed the minimal computed distance value is returned.

Let s be the number of sections covered by the route intervals of *gline1* and t the number of sections covered by route intervals of *gline2*. The computation of the bounding gpoints of both *gline*s will take $O(s + t)$ time. Let i respectively j be the number of bounding gpoints of the both *gline*. The Network Distance computation between this points will take O(ij O(**netdistance**(*gpoint*, *gpoint*))) time. We get a time complexity of O($n + m + ij$ O(**netdistance**(*gpoint*, *gpoint*))) for the whole operation.

3.10 Restricting and Reducing

The following operations restrict a *mgpoint* to given times or places or reduce the number of units of the *mgpoint*.

mgpoint × *instant* → *igpoint*	*mgpoint* **atinstant** *periods*
mgpoint × *periods* → *mgpoint*	*mgpoint* **atperiods** *periods*
mgpoint × *gpoint* → *mgpoint*	*mgpoint* **at**(*gpoint*)
mgpoint × *gline* → *mgpoint*	*mgpoint* **at**(*gline*)
mgpoint × *mgpoint* → *mgpoint*	**intersection**(*mgpoint1*, *mgpoint2*)
mgpoint × *real* → *mgpoint*	**simplify**(*mgpoint*,*real*)

3.10.1 atinstant

Restricts the *mgpoint* to the given time instant. Therefore a binary scan of the units of the *mgpoint* is performed to find the unit containing the given time instant. If a corresponding unit is found the resulting *igpoint* is computed. The time complexity depends on the number m of units of the *mgpoint* and is $O(\log m)$.

[13] We have *moving* data types so the result is also a moving data type. See [?] for detailed explanation of *mreal*.

[14] See 3.8 for information about **shortest_path** respectively 3.4 for information about **length**(*gline*)

[15] Bounding gpoints means at least a set of *gpoint*s. Where each *gpoint* of the set must be passed by everyone who wants to reach the inside of the *gline* from the outside of the *gline* and vice versa. This bounding gpoints are the interesting *gpoint*s for Network Distance computing between two *gline*, because every other place inside a *gline* can only be reached by passing one of the bounding gpoints of the *gline*.

3.10.2 atperiods

Restricts the *mgpoint* to the given *periods*. Therefore a parallel scan of *periods* and the units of the *mgpoint* is performed. And the (parts) of units which are inside the *periods* value are written to the resulting *mgpoint*. The time complexity is $O(m + p)$ if m is the number of units of the *mgpoint* and p is the number of time intervals of the *periods*.

3.10.3 at

Restricts the *mgpoint* to the times and places it passed a given *gpoint* respectively *gline*.

gpoint The algorithm performs a linear scan on the units of the *mgpoint* and checks for every unit if the *mgpoint* passes the *gpoint*. If this is the case a *ugpoint* for the time the *mgpoint* was at the *gpoint* is computed and added to the resulting *mgpoint*. The computation takes $O(m)$ time if m is the number of units of the *mgpoint*.

gline The algorithm performs a linear scan on the units of the *mgpoint*. If the *gline* is sorted a binary search on the route intervals of the *gline* is performed for each unit of the *mgpoint*. If the *gline* is not sorted a linear scan of the route intervals is performed for each unit of the *mgpoint*. In both cases it is checked if the actual *ugpoint* passes any route interval of the *gline*. If this is the case the times and places of passing are computed as *ugpoints* and added to the resulting *mgpoint*.

The time complexity for the operation is $O(m \log r)$ for sorted and $O(mr)$ for unsorted *gline*, if m is the number of units of the *mgpoint*, and r the number of *route intervals* of the *gline*.

3.10.4 intersection

Returns a *mgpoint* value representing the times and places where both *mgpoints* have been at the same time. The algorithm first computes the refinement partitions[16] of the both *mgpoint*. Then it performs a parallel scan through the refinement partitions of the both *mgpoint* and checks for every pair of units if the positions intersect. If this is the case a *ugpoint* with the intersection value is computed and written to the resulting *mgpoint*.

Let m respectively n be the number of units of the both *mgpoint* and r the number of units of the refinement partitions. The time complexity of the algorithm is $O(m + n + r)$.

3.10.5 simplify

The operation reduces the number of units of the *mgpoint*, by merging the units, where the *mpoint* moves on the same route, in the same direction, and the speed difference is smaller as the given *real*. To do this a linear scan on the units of the *mpoint* is performed and the condition is checked for every unit. This will take $O(m)$ time if m is the number of units of the *mgpoint*. Detailed information about the simplification can be taken from [9].

3.10.6 mpgsecunits, mgpsecunits2, and mgpsecunits3

mgpsecunits:	*rel*(*tuple*($(a_1\ x_1)(a_2\ x_2)...(a_2\ x_2)$)) $\times a_i \times$ *network* $\times$ *real* $\rightarrow$ *stream*(*mgpsecunit*)
mgpsecunits2:	*mgpoint* $\times$ *real* $\rightarrow$ *stream*(*mgpsecunit*)
mgpsecunits3:	*stream*(*mgpoint*) $\times$ *real* $\rightarrow$ *stream*(*mgpsecunit*)

We implemented three different versions of the **mgpsecunits**. They all get different input values. While the second operation **mgpsecunits2** is provided to support later on a new spatio-temporal network index for *mgpoint* values. The first (**mgpsecunits**) and the last (**mgpsecunits3**) should support traffic estimation operations. This traffic estimation operations will be part of another new algebra module called `TrafficAlgebra`.

The algorithm is for all three versions of the **mgpsecunits** operation almost the same. The input is a set of *mgpoint* or in case of **mgpsecunits2** a single *mgpoint* and the output a stream of *mgpsecunits* containing all the information's given by the input *mgpoints*. The algorithm computes for all units of every *mgpoint* a corresponding set of *mgpsecunits*. The resulting *mgpsecunits* for a single *mgpoint* are merged as far as possible. Merging means here that as long as the *mgpoint* moves in the same section part in the same direction the different *mgpsecunits* are merged into one *mgpsecunit*. At least the result is returned as *stream* of *mgpsecunits*.

[16] Refinement partition means that the units of both *mgpoint* are parted, so that in the end the units of both *mgpoint* have the same time intervals for the times they both exist.

3.11 ugpoint2mgpoint

ugpoint → *mgpoint* **ugpoint2mgpoint**(*ugpoint*)

The operation constructs a *mgpoint* from a single *ugpoint*.

3.12 polygpoints

gpoint → *stream*(*gpoint*) **polygpoints**(*gpoint*)

A problem of the network data model is that junctions belong to more than one route. Therefore they are represented by more than one *gpoint*. Operators like **passes** or **inside** doesn't check if the query *gpoint* is a junction and probably has more than one representation, because the interpretation of passing a network junction in [**?**] is slightly different from passing a *point* in the 2D space. So if, for example, a *mgpoint* passes a junction on the one route and the *gpoint* representing the junction is given related to another route we get *FALSE* as result. This is correct in the network data model but doesn't correspond to the **passes** interpretation of the BerlinMOD Benchmark.

We introduced the operation **polygpoints** to bypass this problem in the BerlinMOD Benchmark. This operation returns for every given *gpoint* a *stream* of *gpoint*. This stream contains only the *gpoint* itself if the *gpoint* is not a junction, and the *gpoint* itself and all the alias *gpoint*s representing the same place if the *gpoint* is a junction.

The algorithm **polygpoints** first copies the argument *gpoint* to the output stream. Then it checks if the *gpoint* represents a junction by selecting all junctions from the junctions relation which are on the route with the route identifier of the *gpoint* with help of the junctions relation B-Tree. This junctions are checked if they are identified by the *gpoint*. If this is the case all other *gpoint* values identifying the same junction on other routes are returned in the output stream.

The check if the *gpoint* is a junction takes $O(k + \log j)$ time, if the number of junctions in the network is j and the number of junctions on the route is k. If i is the number of related junctions we can find and return them in $O(i + \log k)$ time. Altogether we get a time complexity of $O(k + i + \log j + \log k) = O(k + \log j)$, because it holds $i \leq k \leq j$.

References

[1] 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.

[2] 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.

[3] E.W.Dijkstra. *A Note on Two Problems in Connexion with Graphs*, pages 269–271. Numerische Mathematik 1, 1959.

[4] Fernuniversität Hagen. *Secondo Web Site*, April 2009. = http://dna.fernuni-hagen.de/Secondo.html/index.html.

[5] 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.

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

[7] 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.

[8] 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.

[9] Martin Scheppokat. Datenbanken für bewegliche Objekte in Netzen Prototypische Implementierung und experimentelle Auswertung. Diplomarbeit, Fernuniversität in Hagen, 2007.