Network Data Model and BerlinMOD Benchmark

Simone Jandt

Last Update: July 8, 2010

Abstract

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

Todo list

1 Introduction

In the past, several data models for the representation of spatio-temporal data objects have been developed. We can categorize them into data models for objects moving freely in two dimensional space (DMFS) and data models for network constrained moving objects (NCDM). For both categories several different data models have been presented like [14,19,32,33] for DMFS and [10,20,39,43] for NCDM, to name just a few. Objects which are restricted to use existing networks, like cars are restricted to use road networks, can be represented as moving point objects in both data models, whereas objects, which are not restricted by a given network, like people, can be represented as moving point objects only in DMFS.

Why do we spend time on NCDM, if everything can be represented by DMFS? Now, it is natural to give positions related to the street network instead of x,y-coordinates. NCDM are expected to use less storage space, because geographical information's about street curves are stored only once in the network, whereas in DMFS each street curve is stored in each moving point object using this street. NCDM can support query processing with specialized indexes using their knowledge of the underlying network. It is much easier to formulate queries about the relationships between moving objects and the network in the NCDM. And not at last, the results of our experiments show that our network constrained data model outperforms our data model of free movement in two dimensional space by orders of magnitude. The network constrained data model uses less than 60% of the storage space and less than 50% of the total query run time of the data model of free movement in space, which we used in our experiments. We think that these results show that it is useful to develop specialized data models for specialized data structures like NCDM for network constrained moving objects to save storage space and reduce query run times.

For our benchmark experiments, presented in this paper, we chose two data models one for each data model category. Both data models use the same temporal representation and are available in SECONDO DBMS [9,18]. So we can exclude that different DBMS or temporal representation issues bias the results of our data model comparison with the BerlinMOD Benchmark [5]. The DMFS we use is the data model presented in [14,19] (SPACE). And the NCDM we use is the data model presented in [20] (NET).

We used the BerlinMOD Benchmark [5] to compare the capabilities of the two data models, because the BerlinMOD Benchmark is to the best of our knowledge the first benchmark for complete spatio-temporal database systems. It is developed and available in SECONDO DBMS. And the data generated by the BerlinMOD Benchmark data generator are restricted to the streets of the German capital Berlin, such that they can be translated into a network constrained environment. And not at last, the data model used in the BerlinMOD Benchmark is SPACE that we use for our comparison. So we only have to translate the spatial and spatio-temporal data types of the BerlinMOD Benchmark once into our NET representation. This simplifies the control of the query results and avoids errors caused by translation. The translation of the spatial and spatio-temporal data types of the BerlinMOD Benchmark data into the NET representation described in Section 4 can be seen as an example for the usage of the BerlinMOD Benchmark with other compatible data representations or DBMS.

Besides the comparsion of the both data models, we describe in this paper the first real implementation of NET (see Section 3) providing some further concepts which were only sketched in [20].

The rest of the paper is organised as follows: We present some related work in Section 2, including short reviews of the underlying SECONDO DBMS (Section 2.1), the two data models (SPACE Section 2.2, NET Section 2.3) we chose for our comparison, and the BerlinMOD Benchmark (Section 2.4). In Section 3 we give some information's about our implementation of NET, the used operations and indexes. The translation of the BerlinMOD Benchmark data and query set into the NET representation is described in Section 4. The resulting experimental benchmark setup is described in Section 5 followed by the results of our experiments in Section 6. We conclude our work in Section 7.

2 Related Work

In the past many different spatio-temporal data models have been presented. Many of them support only discrete spatio-temporal changes like [6,25,28,29,36] or deal only with current and future positions of continuously moving objects like [38]. More detailed reviews of these and other spatio-temporal data models beyond the scope of our paper can be found in [35].

In this paper we will focus on spatio-temporal data models for complete histories of continuously moving objects. These can be categorized into data models for objects moving freely in two dimensional space (DMFS) and data models for network constrained moving objects (NCDM).

[40] proposes an incomplete abstract DMFS. Basic idea is that spatio-temporal data types can be modeled by linear constraints and queries are formulated using formulas from differential geometry.

[19] proposes an abstract DMFS including the idea of an time sliced representation for moving objects. This basic idea of time sliced representation is used by the DMFS [32] and [14]. The last one is used in our experiments and therefore reviewed in more detail in Section 2.2. The main difference between the both data models is that [14] supports only linear interpolation of movement, whereas [32] also supports arc interpolation of movement. [32] uses only one spatial object containing all spatial geometries for the representation of spatial objects and one moving object for the representation of the different moving object data types, whereas [14] uses different spatial and moving objects for the representation of the different spatial and moving data types. According to this [32] provides only a single operator that distinguishes between the different topological relationships via a parameter, whereas [14] uses different operations to estimate topological relationships. Overall, [32] offers a more flexible object oriented design than [14].

The spatio-temporal framework of [19] used by [14] has been used for the definition of a NCDM in [20].

The most NCDM use edge based graph representations for the representation of the underlying network data only a small number of NCDM uses route oriented data models or combine route and edge based data models.

[43] proposes an edge based NCDM. The edges and their attributes are stored in a relation representing the network as an undirected graph. Moving objects are assumed to drive always on the path with the lowest cost, in terms of distance or travel-time. They are defined by the source point, the target point, and the starting time instant of the trip. The trajectory is computed by this assumption using the length and speed attributes of the graph edges within the shortest respectively fastest path computation. The advantage of this definition is the reduced storage space for the representation of moving point objects. The drawback is the high computational effort for query evaluation on moving objects.

[39] uses also an edge based network representation. The paper proposes a combination of an two dimensional geometrical edge representation with an directed graph representation of the same network. The both representations are connected by transition policies. The two dimensional geometrical representation handles the spatial information's, whereas the connectivity information is mostly embedded in the directed graph representation. Moving objects are represented by sets of five tuples. Each five tuple contains an edge identifier,

the position of the moving point on the edge in terms of weight and length, the speed and direction of the movement, and the time instant of this information.

Another two-layer network representation is proposed by [10]. The authors of [10] combine the advantages of the dynamic edge-based [12] and the dynamic route-based [11] NCDM approaches. The route-based environment reduces the update intervals and used storage space for the database representation of moving point objects, whereas the edge-based environment supports a more detailed view on the traffic conditions of the different edges belonging to the same route. Moving objects are represented by a set of pairs. The pairs consist of an motion vector and an Boolean flag. The Boolean flag tells if the motion vector contains current or historical information. Each motion vector consists, in parts similar to [39], of a time stamp, a network position, and a speed vector. Similar to [20] the network position bases on routes and junctions and not on edges. Different from all other NCDM in this section [10] uses time depending dynamic attributes in the representation of the network parts. Therefore, changes in the network environment can be handled without loss of information in this NCDM.

To the best of our knowledge only a few of the proposed data models have been implemented into database management systems: [32] is implemented as data cartridge [33, 34] for the commercial Oracle®object-relational DBMS [7]; [10] is implemented as extension of the open source database project PostgreSQL [16]; [14] and [20] are implemented in the freely available extensible SECONDO DBMS [18].

Although [32] for the DMFS and [10] for the NCDM provide greater flexibility we decided to use [14] and [20] in our experiments, because both data models are available in the same DBMS, which is also the DBMS in which the BerlinMOD Benchmark [5] has bee developed.

The BerlinMOD Benchmark [5] is to the best of our knowledge the only benchmark testing the capabilities of complete spatio-temporal database systems. Coming with a well defined data set, and two query sets feasible for DMFS and NCDM. Other benchmarks for spatio-temporal databases systems provide only well defined query sets and a database description without any data set like [41]. Or they come with well defined data generation, workload sets and experiments but evaluate only the capabilties of indexes for current and near future positions like [27]. Or they focus on time-evolving regional data and associated index methods like [42].

The focus on index benchmarking in the most benchmarks is dued by the fact that indexes have a great influence on query run times. Therefore, many spatio-temporal indexes have been developed in the last ten years. An survey about existing spatio-temporal indexes can be found in the two parted work [30] and [31]. The most presented spatio-temporal indexes base on the R-Tree [21] and its variants. The R-Trees are used stand alone or in hierarchical combinations. B-Trees [1] and their variants are also used within spatio-temporal indexes. SECONDO comes with implementations of R-Tree, B-Tree, and MON-Tree [8] in Section 3 we give a detailed description how we used these indexes in our experiments.

The BerlinMOD Benchmark comes with his own data generator. Like mentioned before other benchmarks like [41] define only database descriptions. The users have to generate their own corresponding data.

Therefore, several data generators have been developed. Some of them generate only unconstrained moving point objects like [26], or support only short-term observations like [4], or are unavailable respectively require additional software like [15].

In the sequel we give short reviews of the Secondo DBMS (Section 2.1), the both data models we used in our experiments (Section 2.2 and Section 2.3), and the BerlinMOD Benchmark (Section 2.4).

2.1 Secondo DBMS

The extensible Secondo DBMS presented in [9, 18] provides a platform for implementing various kinds of data models. It provides a clean interface between the data model independent system frame and the content of the single data models. Hence Secondo can be easily extended by the user implementing algebra modules to introduce new data types and operations on these data types. The user may define additional viewers for the graphical user interface or write additional optimization rules or cost functions to extend the the optimizer. Since Secondo version 2.9 the users may publish their extensions as a Secondo plugin such that other users can use these plugins to extend their own Secondo system. They may use the newly provided functionalities or repeat the published experiments. Secondo is freely available on the web [23]. It comes with a number of already implemented spatial and spatio-temporal data types and operations including SPACE (Section 2.2) and NET (Section 2.3). Furthermore, the BerlinMOD Benchmark described in Section 2.4 has been developed in the Secondo DBMS. For our experiments we used the Secondo version 3.0.

2.2 Data Model of BerlinMOD Benchmark (SPACE)

[13] presents the basic idea of the DMFS that is used by the BerlinMOD Benchmark. The abstract data model for SPACE was pubished in [19] and the discrete data model in [14]. Abstract data models are useful as conceptual models, but they cannot be implemented, because computers can only use finite sets. In the sequel description we will mainly focus on the discrete data model.

The type system in [19] is defined using the techniques presented in [17]. The basic idea of [19] is to define type constructors that create new data types if they are applied to an data type of a given set of basic data types.

Basic data types are the standard data types integer, real, string and boolean (BASE); the spatial data types point, points, line, and region (SPATIAL); and the temporal type instant (TIME).

The carrier sets for all data types in the discrete data model contain $\perp$, representing an undefined value. This is not mentioned any more in the sequel.

The carrier sets for the BASE data types in the discrete data model are defined by the corresponding programming language data types $\underline{\textit{int}}$, $\underline{\textit{real}}$, $\underline{\textit{bool}}$, and $\underline{\textit{string}}$. The carrier set of a value of the data type $\underline{\textit{point}}$ is $\underline{\textit{real}} \times \underline{\textit{real}}$. The two $\underline{\textit{real}}$ values represent the x,y-coordinates of the $\underline{\textit{point}}$ value in the two dimensional plane. The data type $\underline{\textit{points}}$ consists of a disjoint set of $\underline{\textit{point}}$ values. The carrier set for the data type $\underline{\textit{line}}$ consists of a finite set of disjoint line segments representing the linear approximation of the line curve in the two dimensional plane. Semanticaly an $\underline{\textit{line}}$ value is the union of the points of all its line segments. A region is the union of all points covered by the region. The carrier set $\underline{\textit{region}}$ is defined to be a finite set of line segments building a polygon representing the linear approximation of the outer and, if the region contains wholes, inner borders of the region. The borders are defined to belong to the region.

The carrier set for the TIME data type is given by *real* in the discrete data model. That means each time *instant* is represented by a corresponding *real* value.

The type constructor *range* converts BASE and TIME data types α into a type whoes values are finite sets of intervals over α. Range types are used to represent collections of time intervals, or the values taken by a moving real. Intervals are represented by their start and end point and two flags indicating if the start respectively end point is part of the interval or not.

The other important type constructor of the abstract data model is *moving*. In the abstract data model*moving* maps each data type α from BASE and SPATIAL into an time dependend spatio-temporal moving data type *moving*(α) (*m*α for short) of kind TEMPORAL. The discrete data model introduces some additional type constructors to implement the moving type constructor of the abstract data model. A detailed description of the type constructors is skiped due to place limitations. We explain the realisation of *moving* at the example of a *moving*(*point*) (short *mpoint*) object. An *mpoint* value may represent a car, which changes its position in the plain within time.

An *mpoint* consists of a set of so called *unit*(*point*) values (*upoint* for short). Each *upoint* consists of a time interval and two *point* values. The first *point* value represents the position of the *upoint* at the start of the time interval and the second *point* value represents the position of the *upoint* at the end of the time interval. It is assumed that the object represented by the *upoint* moves on the straight line between these two points with constant speed within the given time interval. The velocity of the object is given by the ratio from the distance of the two points and the length of the time interval of the *upoint*.

All *upoint* values of an *mpoint* must have disjoint time intervals, because a car cannot be at two different positions at the same time. The set of *upoint* values is sorted by ascending time intervals.

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

Other data types of SECONDO which are used in the BerlinMOD Benchmark are *mbool*, *mreal*, and *periods*. A *mbool* value consists of a set of *ubool* values. Each *ubool* value is is constant *TRUE* or *FALSE* for the given time interval. A *mreal* value consists of a set of *ureal* values. Each *ureal* value is defined by a function of time representing the *real* value at each time instant. A *periods* value is a set of disjoint an not connected time intervals.

2.3 Network Data Model(NET)

The central idea of the NDM presented in [20] is that every movement is constrained by a given network and every position can be described relative to this network. Contrary to the most other NCDM NET models the network in terms of routes, corresponding to roads or highways in real life. Positions are given by a route identifier and the distance from the start of the route. This is a more natural representation of network positions as the directed graph representation of networks, where junctions are vertexes and the pieces between junctions are represented by edges, which is used in the most NCDM. We have names for roads not for junctions or pieces between junctions.

The routes based network representation has also the advantage that the representation of moving objects that move over several sections of the same route with constant speed becomes much smaller, because we only have to store new information if the moving object changes the road or the speed. Not every time it passes a junction like in the other NCDM.

In NET the data type <u>*network*</u> is modeled by two main components. One is the set of routes (streets) and the other one the set of junctions (crossings). The domain of routes is defined as

$$\begin{aligned} Route = \{(id,\ l,\ c,\ kind,\ start) \mid\ & id \in \underline{\text{int}},\ l \in \underline{\text{real}},\ c \in \underline{\text{line}},\\ & kind \in \{simple,\ dual\},\\ & start \in \{smaller,\ larger\}\},\end{aligned}$$

where *id* is a distinct route idenitfier, *l* is the length of the route, *c* is the route curve as <u>*line*</u> value (see Section 2.2), *kind* indicates if the lanes of the route are separated, and *start* indicates how the route curve is embedded into space.

If R is a set of distinct routes, the domain of junctions in R is defined as

$$\begin{aligned} Junction(R) = \{(rm_1,\ rm_2,\ cc) \mid\ & rm_1,\ rm_2 \in RMeas(R),\\ & rm_1 = (r_1,\ d_1),\ rm_2 = (r_2, d_2),\\ & r_1 \neq r_2, cc \in \underline{\text{int}}\}.\end{aligned}$$

Where the set of possible positions in R $RMeas(R)$ is defined as

$$\begin{aligned} RMeas(R) = \{(rid, d)\ \mid\ & rid \in \underline{\text{int}},\ d \in \underline{\text{real}},\\ & \exists (rid,\ l,\ c,\ k,\ s) \in R \wedge 0 \leq d \leq l\},\end{aligned}$$

and d is the distance from the start of the route. The connectivity code cc encodes which lanes of the roads are connected by the junction[1].

A network N is a pair $(R,\ J)$, where R is a finite set of distinct routes and J is a finite set of junctions in R. The carrier set for network positions $Loc(N)$ is equal to the set of route locations and defined as

$$\begin{aligned} RLoc(R) = \{(rid,\ d,\ side) \mid\ & (rid,\ d) \in RMeas(R),\\ & side \in \{up,\ down,\ none\}\}.\end{aligned}$$

The *side* value indicates for *dual* routes if a position can be reached from the *up* or the *down* side of the route. For routes with $kind = simple$ the *side* value is always *none*.

Let $N = \{N_1, \ldots, N_k\}$ be a set of networks. A single network position in a network N_i is represented by the data type <u>*gpoint*</u>. The carrier set of <u>*gpoint*</u> is defined as

$$\{(i,\ gp) \mid 1 \leq i \leq k \wedge gp \in RLoc(R) \cup \{\perp\}\},$$

where $\perp$ again represents a undefined value.

[1]See [20] for detailed explanations.

A *route interval* in N_i is a pair of network positions on the same route. The *route interval* is represented by a quadruple $(rid,\ d_1,\ d_2,\ side)$, with $(rid,\ d_1,\ side)$, $(rid,\ d_2,\ side) \in Loc(N)$ and $d_1 \leq d_2$. Semantically a *routeinterval* represents all route locations $(rid,\ d,\ side)$ with $d_1 \leq d \leq d_2$. A finite set of disjoint *route intervals* of a network N is called region of N. The set of all possible regions in a network N is denoted as $Reg(N)$.

A region within an network N_i is represented by the data type gline. The carrier set of gline is defined as

$$\{(i,\ gl) \mid 1 \leq i \leq k \wedge gl \in Reg(N_i)\}$$

. The set of *routeintervals* defining a network region may be empty.

The type systesm of [19] is extended by [20] to contain a new kind GRAPH consisting of the data types gpoint and gline. The type constructors moving is also extended to be feasible for data types of kind GRAPH. Therefore the data type moving(gpoint) (mgpoint for short) is defined similar to the mpoint explained in detail in Section 2.2. The units of a mgpoint consist of ugpoint values and single positions can be given by igpoint values.

The paper provides numerous operations on the network and the network data types. We give reviews of the operations that were used in our experiments later in this paper (see sections 3 - 4).

[20] proposes also implementational issues for NET in SECONDO. The implementation of the data type network consists of three relations called *routes*, *junctions*, and *sections*, and a persistent adjacency list data structure supporting trip and path computations.

The three relations have the following schemas:

```
routes (id:int; length: real; curve: line; kind: bool;
start: bool)

junctions (r1id: int; r1rc: int; pos1: real; r2id: int;
r2rc: int; pos2: real; cc: int; pos: point)

sections (rid: int; rrc: int; pos1: real; pos2: real;
dual: bool; length: real; curve: line)
```

As you can see, the *routes* relation is equivalent to the domain of routes *Route*. The tuple of the *junctions* relation is somewhat different from $Junctions(R)$. The record identifiers $r1rc$ and $r2rc$ support faster access to the corresponding tuples in the *routes* relation and the point value *pos* supports the connection to the two dimensional plane.

The *sections* relation is derived from the other two relations. The meaning of the *rrc* value is similar to the meaning of $r1rc$ in the *junctions* relation. The entries of the *sections* relation correspond to the edges of a network graph. They are used internally to support operations like **shortestpath**. The adjacency list data structure consists of two arrays and provides 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.

For the data types gpoint and gline [20] proposes the following implementations:

```
gpoint: record {nid: int; rid: int; pos: real;
side: {up, down, none};}
```

```
gline: record{nid: int; rints: DBArray of record {
rid: int; pos1: real; pos2: real;
side: {up, down, none};}
```

For the data type *mgpoint* the implementation consists of a set of *ugpoint*. The set is stored in a DBArray in ascending order of the time intervals of the *ugpoint*. Each *ugpoint* is defined as:

```
ugpoint: record {nid: int; rid: int; pos1: real;
pos2: real; side: {up, down, none};
t1: Instant; t2: Instant;}
```

The *ugpoint* is expected to move from pos_1 to pos_2 with constant speed on route *rid* in the given network *nid* within the time interval defined by t_1 and t_2. Every time a *mgpoint* changes the speed or changes the route a new *ugpoint* is written.

We extended this implementation proposed by [20] within our experiments to support faster query execution. Our changes will be described in detail in Section 3.

2.4 BerlinMOD Benchmark

The BerlinMOD Benchmark was presented in [5] and the provided scripts for the data generator are implemented as SECONDO DBMS operations. The BerlinMOD Benchmark is available on the web [22] and provides a well defined data-set and queries for the experimental evaluation of the capabilities of spatial and spatio-temporal database systems dealing with histories of moving objects. The BerlinMOD Benchmark emphasises the development of complete systems and simplifies experimental repeatability pointing out the capabilities and the weaknesses of the benchmarked systems.

The data-sets of the BerlinMOD Benchmark are created using the street map of the German capital Berlin [37] and statistical data about the regions of Berlin [2, 3] 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 the work node in the morning and vice versa in the late afternoon. Beside this, randomly chosen cars will make additional trips in the evening and up to six times at the weekend to randomly chosen targets in Berlin and back home. The BerlinMOD Benchmark uses the data model of free movement in two dimensional space described in Section 2.2. Because the BerlinMOD Benchmark generates all data sets restricted to the street map of Berlin, the BerlinMOD Benchmark can also be used for network constrained data models, if the spatial and spatio-temporal data types are translated into a corresponding NDM, like we did for our experiments.

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

The BerlinMOD Benchmark provides two different approaches to store the histories of moving objects, called the object-based approach (OBA) and the trip based approach (TBA), respectively.

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

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

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

Name of Data Set	**Tuple Content**
QueryPoints	Object identifier and <u>point</u> value
QueryRegions	Object identifier and <u>region</u> value
QueryInstants	Object identifier and time instant
QueryPeriods	Object identifier and time interval
QueryLicences	Object identifier and a <u>string</u> representing a licence value

Table 1: Query Object Relations of BerlinMOD Benchmark

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

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

- known or unknown object identity,
- standard attribute, spatial, temporal, or spatio-temporal dimension,
- point, range, or unbounded query interval,
- single object or object relationships condition type,

- with or without aggregation.

We will present the 17 queries in more detail in Section 4.4together with our NDM algorithms for these queries.

Check reference when paper finished. Parts of query explanations might be moved to the appendix.

3 Implementation of NET

We started a description of the implementation of NET in SECONDO in Section 2.3 with the description of the implementation proposed in [20]. In the meantime this implementation has been changed and extended in some points to support faster query execution.

In the sequel we describe the current implementation of the NET data types in Section 3.1, new data types for network position indexes are presented in Section 3.2, and the implementation of operations used by the BerlinMOD Benchmark in Section 3.3.

3.1 Data Type Implementation

First of all the *network* object itself has been changed. The *junctions* relation has been extended by four additional record identifiers, one for each section connected within this junction. Four B-Tree indexes for the route identifier attributes in the *routes*, *junctions*, and *sections* relations haven been integrated. An R-Tree has been integrated indexing the curve attribute of the *routes* relation. All this has been done to support faster access to spatial routes data in query evaluation.

The *side* value of the *route intervals* is not yet part of the implementation.

The record of *gline* was extended by an attribute *length* of *real*, storing the length of the *gline*, and an sorted flag, indicating if the *route intervals* in the DBArray are stored sorted or not. We call a set of *route intervals* sorted if it fullfills the following conditions:

- all *route intervals* are disjoint
- the *route intervals* are stored in ascending order of their route identifiers
- if two *route intervals* have the same route identifier, the *route interval* with the smaller start position is stored first if the *route intervals* are disjoint
- for all *route intervals*, *start position* $\leq$ *end position*

We introduced this definition and sorted flag, because many algorithms take profit from sorted *gline* values. Let r be the number of *route intervals* in a *gline*, the decision, if a *gpoint* is inside the *gline* needs $O(r)$ time for unsorted and $O(\log r)$ time for sorted *gline* values.

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

For sorting and compressing *route intervals* we introduced a binary tree data structure called *RITree*. This *RITree* sorts and compresses the inserted *route intervals*. If r_i is the number of inserted *route intervals* and r_{res} the number of resulting *route intervals* sorting and compressing takes $O(r_i \log r_{res})$ time. The sorted *route intervals* are returned in $O(r_{res})$ time. We think that this time is well invested, because the sorted $\underline{gline}$ is computed once, but many different algorithms can be executed faster for sorted $\underline{gline}$ values.

The implementation of the $\underline{ugpoint}$ has been changed to:

```
ugpoint: record {gp1: gpoint; gp2: gpoint;
ti : Interval;}
```

Where t_i is a time interval consisting of two time instants t_1 and t_2 and two Boolean flags, indicating if t_1 respectively t_2 is part of the time interval or not.

At the same time the implementation of $\underline{mgpoint}$ has been extended to:

```
mgpoint: record {units: DBArray of ugpoint;
drivenDist: real; trajDefined: bool; mbr: rectangle3D
trajectory: DBArray of sorted route intervals;}
```

The DBArray of $\underline{ugpoint}$ is consistent with the data model of [20]. The *drivenDist* is the total length of all $\underline{ugpoint}$ in the $\underline{mgpoint}$. The DBArray of *route intervals* represents all network positions ever traversed by the $\underline{mgpoint}$. The flag indicates if the *trajectory* is well defined, because this attribute is not maintenanced in every operation changing $\underline{mgpoint}$ values. And the minimum spatio-temporal bounding box *mbr* can be used for a preselection in spatio-temporal queries.

Why this extensions? Now, analogous to sorted $\underline{gline}$ values the *trajectory* value makes it much faster to decide whether an $\underline{mgpoint}$ ever passed a given network position or not. Instead of a linear check of all m units of an $\underline{mgpoint}$ we can perform a binary scan on the much smaller number r of the passed *route intervals*. This reduces the time complexity from $O(m)$ to $O(\log r)$ for the operations like **passes**.

The spatio-temporal minimum bounding box was introduced as an attribute to the $\underline{mgpoint}$ because the computation of this value is very expensive in the NDM. Although each unit of an $\underline{mgpoint}$ stays on the same route at the same speed it may follow different spatial directions. For example, a route may lead uphill in serpentine. A spatial bounding box only computed from the spatial start and end position may not enclose all spatial positions of the car within the unit. Therefore we always have to examine the spatial dimensions of the *route interval* passed within a unit to compute the units bounding box using algorithm 1.

Algorithm 1 Berechnung Unit Bounding Box

```
1: Get route curve usint B-Tree index of routes relation
2: Extract subline of unit from route curve
3: Compute bounding box of subline
4: Add third dimension from unit time interval
```

If the number of routes in the *routes* relation is R the first step has a time complexity of $O(\log R)$ time. If the route curve r_i consists of l_i line segments the time complexity of step 2 and 3 is $O(l_i)$ in the worst case. Step 4 is done in

O(1) time. Together we get a time complexity of $O(\log R + l_i)$ to compute the bounding box of a single unit.

To compute the *mbr* this computation must be done for each of the m units of the mgpoint value. That takes $O(m \log R + \sum_{i=1}^{m} l_i)$ time.

The time complexity can be reduced if the *trajectory* is defined. We can use the r *route intervals* of the *trajectory* similar to the m units of the mgpoint to compute the spatial bounding box. The third dimension can be added using the start and the end time instant of the mgpoint value. This a algorithm has a time complexity of $O(r \log R + \sum_{i=1}^{r} l_i)$, with $r \ll m$ in nearly all cases.

But the computation is still expensive. So the *mbr* is only computed on demand or if we can get it for free. For example we can copy the bounding box of an mpoint at the translation time into an mgpoint in O(1) time. Analogous to the *trajectory* the *mbr* is not maintained. If the mgpoint value changes, the *mbr* is set to be undefined until recomputing is necessary.

3.2 Indexes in Network Environment

Spatial and spatio-temporal bounding boxes are used to support for spatial and spatio-temporal indexing of spatial respectively spatio-temporal positions. Because of the special problems with spatio-temporal bounding boxes in network environments (see Section 3.1) we use only for the trip based approach a spatio-temporal bounding box tree *dataMNtrip_SpatioTemp* and this tree only indexes spatio-temporal bounding boxes for complete mgpoint values. The advantages of more detailed indexes have been outperfomred within our experimental evaluation of the network implementation for the BerlinMOD Benchmark.

We introduced Network Bounding Boxes (NBB) and Temporal Network Bounding Boxes (TNBB) in our implementation to support indexing in terms of network positions with R-Trees instead of spatial indexing.

Let ugpoint and *route interval* be defined like in 2.3. The NBB of a *routeinterval* is defined as (rid, rid, $pos1$, $pos2$), which can be seen as an degenerated two dimensional rectangle. The TNBB of a ugpoint value is defined as (rid, rid, $pos1$, $pos2$, $t1$, $t2$), which analogous can be seen as an degenerated three dimensional rectangle.

We use these NBB and TNBB to create Network Position Indexes (NPI) and Temporal Network Position Indexes (TNPI) indexing the network positions of the mgpoint values of the BerlinMOD Benchmark.

An NPI is an R-Tree build from the NBB of the *routeintervals* of the *trajectory* attributes of the mgpoint values.

An TNPI is an R-Tree build from the TNBB of the ugpoint values of the mgpoint values.

3.3 Network Operations used in BerlinMOD Benchmark

The operations used to construct the network object (**thenetwork**), and to translate spatial and spatio-temporal values into network values are described in Section 4. In this section we give an overview of the operations on network objects used in the benchmark queries of the BerlinMOD Benchmark.

Table 2 gives an overview of the simple benchmark operations. The more complex operations are described in the sequel.

Table 2: Simple Operations on Network Data Types

Name	**Signature**	**Explanation**	**Complexity**
routes	*network* → *relation*	Returns the *routes* relation fo the *network* object	O(R)
no_components	*mgpoint* → *real*	Returns the number of units of the *mgpoint*	O(1)
length	*mgpoint* → *real*	Returns the *length* of the argument.	O(1)
trajectory	*mgpoint* → *gline*	Returns the *trajectory* of the *mgpoint* value as *gline*	O(r), if trajectory is defined, O($m \log r + r$) otherwise
units	*mgpoint* → *stream*(*ugpoint*)	Returns a stream of the *ugpoint* values of the *mgpoint* value	O(m)
initial	*mgpoint* → *igpoint*	Returns the first position and the start time of the *mgpoint* value	O(1)
atinstant	*mgpoint* × *instant* → *igpoint*	Returns the network position of the *mgpoint* value at the given time instant	O($\log m$)
val	*igpoint* → *gpoint*	Returns the *gpoint* of the *igpoint* value	O(1)
inst	*igpoint* → *instant*	Returns the time instant of the *igpoint* value	O(1)
isempty	*gline* → *bool*	Returns *TRUE* if the *gline* has no *route intervals*	O(1)
routeintervals	*gline* → *stream*(*rectangle*)	Returns the *netbox* for each *route interval* of the *gline* value	O(r)
gpoint2rect	*gpoint* → *rectangle*	Returns a *netbox* for the *gpoint* value	O(1)
unitbox	*ugpoint* → *rectangle3D*	Returns the *netbox* of the *ugpoint*	O(1)

We use m as the number of units of the mgpoint value, r (resp. r_1, r_2) as the number of *route intervals* of the gline value or the *trajectory* attribute of an mgpoint value (resp. of the first, second argument), R as the number of routes in the network object, l_i the number of line segments of a route curve r_i, and p as the number of time intervals in a periods value.

For the Euclidean Distance computation we retranslate our network values into spatial (**gline2line**) respectively spatio-temporal (**mgpoint2mpoint**) values. In the original paper this operations are called **in_space**.

gline → line	**gline2line**(*gline*)
mgpoint → mpoint	**mgpoint2mpoint**(*mgpoint*)

The operation **gline2line** uses the Algorithm 2 to translate a gline value into an line value.

Algorithm 2 gline2line(*gl*)

```
1: for each route interval of gl do
2:     Get route curve of route interval using B-Tree index of routes relation
3:     Perform binary search on line segments of route curve to find the start
       position of the route interval
4:     Copy line segments to result until end pos of route interval is reached.
5: end for
6: return resulting line
```

The loop from line 1 to line 5 is repeated r times. The B-Tree search in line 2 take $\mathrm{O}(\log R)$ time. The binary search in line 3 takes $\mathrm{O}(\log l_i)$ tim. The copy operation in line 4 takes $\mathrm{O}(l_i)$ time in the worst case. The return of the result has a worst case time complexity of $\mathrm{O}(\sum_{i=1}^{r} l_i)$. For the whole algorithm we get

$$O(r \log R + \sum_{i=1}^{r} \log l_i + 2 \sum_{i=1}^{r} l_i) = O(r \log R + \sum_{i=1}^{r} l_i)$$

.

The operation **mgpoint2mpoint** is described by Algorithm 3. The loop from line 3 to line 28 is repeated m times. The statements from line 5 to line 8 take $\mathrm{O}(\log R + l_i)$ every time they are executed. But they are only executed at the first run of the loop and if the car changes the route. We have three different cases for the computing of the *mpoint* units. The simple cases from line 10-11 and line 13-16 have a time complexity of O(1). The third case has a worst case complexity from $\mathrm{O}(l_i)$. The last line 29 has a time complexity of $\mathrm{O}(m_{res})$ if m_{res} is the number of units of the resulting mpoint value. In the worst case we get a time complexity of

$$O(m \log R + \sum_{i=1}^{m} l_i + m_{res}).$$

Algorithm 3 mgpoint2mpoint(mgp)

```
1: $actRID$ = -1
2: Initialize resulting $mpoint$
3: for each unit $curUGP$ of $mgp$ do
4:   if not (rid from $curUGP$ == $actRID$) then
5:     $actRID$ = rid from $curUGP$
6:     $rc$ = route curve of $actRID$ {$rc$ determined using B-Tree Index of
       $routes$ relation}
7:     Perform binary search on the line segments of $rc$ to find line segment $l$
       with start position of $curUGP$
8:     $upstart$ = x,y-coordinates of start position
9:   end if
10:  if end position == start position then
11:    add unit $upstart$, $upstart$ to $mpoint$
12:  else
13:    if end position is on $l$ then
14:      $upend$ = x,y-coordinates of end position
15:      add unit $upstart$, $upend$ to $mpoint$
16:      $upstart = upend$
17:    else
18:      Follow $rc$ in moving direction of $curUGP$
19:      while not(end position is on $l$) do
20:        add unit from $upstart$ to $segment\ end\ position$ to $mpoint$
21:        $upstart = segment\ end\ position$
22:        $l$ = next segment in moving direction
23:      end while
24:      $upend$ = x,y-coordinates of end position
25:      add unit from $upstart$ to $upend$ to $mpoint$
26:    end if
27:  end if
28: end for
29: return $mpoint$
```

In the trip based approach (TBA) the result values of the different trips must be aggregated to an single result value. Therefore the operation **union** was introduced. **union** gets two $\underline{gline}$ values or two $\underline{mgpoint}$ values as input and mixes them up to a single sorted $\underline{gline}$ or a single $\underline{mgpoint}$ value.

$\underline{gline} \times \underline{gline} \rightarrow \underline{gline}$	$gline1$ **union** $gline2$
$\underline{mgpoint} \times \underline{mgpoint} \rightarrow \underline{mgpoint}$	$mgpoint1$ **union** $mgpoint2$

Algorithm 4 union(gl_1, gl_2)

```
if $gl_1$ is sorted AND $gl_2$ is sorted then
  Perform parallel scan of $gl_1$ and $gl_2$
  if Current route intervals do not intersect then
    Add smaller route interval to result
    Continue Scan with next route interval
  else
    Merge route intervals into one
    if Next route intervals intersect the merged one then
      merge them too
    end if
    Add merged route interval to result
    Continue Scan
  end if
else
  for Each route interval of $gl_1$ and $gl_2$ do
    Insert route interval into RITree
  end for
  Copy sorted route intervals from RITree to resulting gline
end if
return resulting gline
```

The Algorithm 4 distinguishes between two cases. If both gline values are sorted the algorithm has a time complexity of $O(r_1+r_2)$ and if one or both gline are not sorted it has a time complexity of $O((r_1+r_2)\log r_{res}+r_{res})$, if r_{res} is the number of *route intervals* of the resulting gline. The additional time results from sorting and compressing the resulting *route intervals*. As explained before (see Section 3.1) we think that this time is well invested, because several algorithms take profit from sorted gline values.

The computation of the union of two mgpoint values works almost similar to the **union** operation for two sorted gline. We perform a parallel scan through the mgpoint values and add the units in the sequel of their time intervals to the resulting mgpoint value. If two units have overlapping time intervals and the positions of the ugpoints within the overlapping time interval are not the same the resulting mgpoint is not defined, otherwise one unit is added to the result for the overlapping time interval. The time complexity of the **union** operation is $O(m_1 + m_2)$.

mgpoint → rect3 **mgpbbox**(*mgpoint*)

The operation **mgpbbox** returns the spatio-temporal bounding box of an mgpoint value. As explained before (see Section 3.1) the *mbr* value is not maintained such that the operation **mgpbbox** knows three different cases:

1. If the *mbr* is defined it can be returned in $O(1)$ time.

2. The *mbr* is not defined but the *trajectory* can be used to compute the *mbr* the time complexity is $O(r \log R + \sum_{i=1}^{r} l_i)$

3. If the *mbr* and *trajectory* are not defined the time complexity will be $O(m \log R + \sum_{i=1}^{m} l_i)$

mgpoint $\times$ periods $\rightarrow$ mgpoint *mgpoint* **atperiods** *periods*

The operation **atperiods** restricts a mgpoint value to the given periods. The operation performs a binary scan of the units of the mgpoint to find the unit including the start time instant of the periods value, respectively the last unit with an time interval smaller than the start time instant of the periods value in $O(\log m)$ time. From that position the k_i units, which have time intervals that intersect with the periods value are copied to the result in $O(k_i)$ time[2]. The total time complexity of the operation is $O(\log m + k_i)$

mgpoint $\times$ periods $\rightarrow$ bool *mgpoint* **present** *periods*

The operation **present** returns *TRUE* if the mgpoint value is defined at least at one time inside the given Periods value, *FALSE* otherwise. The algorithm works almost similar to the **atperiods** operation, but the scan of the mgpoint units is stopped immediately if a intersecting unit is found. The worst case time complexity for the operation is $O(\log m + k_i)$.

gpoint $\times$ gline $\rightarrow$ bool *gpoint* **inside** *gline*
mgpoint $\times$ gline $\rightarrow$ mbool *mgpoint* **inside** *gline*

The operation **inside** returns *TRUE* if a gpoint is inside a gline, *FALSE* otherwise. For sorted gline values the algorithm performs a binary scan of the *route intervals* to find a *route interval* including the gpoint in $O(\log r)$ time. For unsorted gline values a linear scan of the *route intervals* is performed in $O(r)$ time to find an *route interval* including the gpoint.

mgpoint $\times$ gpoint $\rightarrow$ bool *mgpoint* **passes** *gpoint*
mgpoint $\times$ gline $\rightarrow$ bool *mgpoint* **passes** *gline*

The operation **passes** returns *TRUE* if a mgpoint value passes a given gpoint or gline value. The algorithm uses the *trajectory* of the mgpoint value.

For an gpoint value a binary scan of the *trajectory* is performed to find a *route interval* that includes the gpoint. The time complexity of this operation is $O(\log r)$.

For an gline value two cases are distinguished. If the gline value is sorted a parallel scan of the set of *route intervals* and the *trajectory* is performed and immediately aborted if two intersecting *route intervals* have been found. In this case the worst case time complexity is $O(r_1 + r_2)$. If the gline value is not sorted a linear scan of the set of *route intervals* of the gline is performed and for every *route interval* a binary scan of the *trajectory* is performed to find a intersecting *route interval*. In this case the worst case time complexity is $O(r_2 \log r_1)$.

[2]The first and last time interval might be splitted at the start (resp. end) time instant of the periods value.

$\underline{mgpoint} \times \underline{gpoint} \rightarrow \underline{mgpoint}$	*mgpoint* **at** *gpoint*
$\underline{mgpoint} \times \underline{gline} \rightarrow \underline{mgpoint}$	*mgpoint* **at** *gline*

The operation **at** restricts a mgpoint to the times and places it was at a given gpoint or moved inside a given gline.

For gpoint values the operation **at** performs a linear scan of all units of the mgponit value. Every time a ugpoint passes the gpoint the time instant of passing is computed and the resulting unit is added to the resulting mgpoint. The computation of the result has a time complexity of $O(m)$.

For gline values the algorithm (see Algorithm 5) distinguishes two cases. For sorted gline values the loop from line 2 to 7 is executed in $O(m \log r)$ time.

Algorithm 5 *mgpoint* **at** *gline*

```
1: if gline is sorted then
2:   for Each unit of mgpoint do
3:     Perform binary scan after unit in set of route intervals of gline
4:     if unit intersects route interval then
5:       Add resulting unit to result
6:     end if
7:   end for
8: else
9:   for Each unit of mgpoint do
10:    for Each route interval of gline do
11:      if unit intersects route interval then
12:        Compute intersection and add it to result
13:      end if
14:    end for
15:  end for
16: end if
17: return result
```

For unsorted gline values line 9 to 14 are executed, which needs $O(mr)$ time. The result is returned in $O(m_{res})$ time in both cases. In case of sorted gline values the total run time is $O(m \log r + m_{res})$ and in case of unsorted gline values $O(mr + m_{res})$.

$\underline{gline} \times \underline{gline} \rightarrow \underline{bool}$	**intersects**(*gline*1, *gline*2)

The operation **intersects** returns *TRUE* if two gline values intersect, *FALSE* otherwise.

Algorithm 6 intersects(gl_1,gl_2)

```
if Both gline values are sorted then
   Perform a parallel scan of the route intervals of the both gline values
   if route intervals intersect then
      return true
   end if
else
   if Both gline not sorted then
      for Each route interval of $gl_1$ do
         for Each route interval of $gl_2$ do
            if route intervals intersect then
               return true
            end if
         end for
      end for
   else
      for Each route interval of the unsorted gline value do
         Perform a binary search on the sorted gline value
         if Intersecting route interval is found then
            return true
         end if
      end for
   end if
end if
return FALSE
```

The Algorithm 6 differentiates three cases:

1. If both gline values are sorted (see lines 2 to 5) the time complexity is $O(r_1 + r_2))$
2. If both gline values are not sorted (see lines 8 to 14) the time complexity is $O(r_1 r_2)$
3. If only one gline value is sorted (see lines 16 - 21) the time complexity is $O(r_1 \log r_2)$ respectively $O(r_2 \log r_1)$, depending on which of the both gline values is sorted.

4 Translation of BerlinMOD into Network Data Model

In this section we describe the creation of the network object from the *streets* value of the BerlinMOD Benchmark in Section 4.1. In Section 4.2 we describe how to use this new created network object in the translation of all spatial and spatio-temporal data objects of the BerlinMOD Benchmark into the NET representation. In Section 4.3 we describe the indexes we build on the database use in the NET representation of the BerlinMOD Benchmark to support faster query execution. We close this section with Section 4.4 a description of our executable SECONDO queries for the NET representation of the BerlinMOD Benchmark.

Executable SECONDO scripts for the network and index creation, object translation, and the network benchmark queries can be downloaded from our web site [24].

Create corresponding website!

4.1 Create Network Object

Before we can use the operator **thenetwork** to construct the network object *net* for the NET version of the BerlinMOD Benchmark we have to build the input relations for **thenetwork** from the data generated by the BerlinMOD Benchmark.

We use the *streets* object created by the BerlinMOD Data Generator to construct our input relation B_ROUTES for the *routes* relation of *net*. We extract the routes geometries from the streets relation and add to each route curve a automatic generated integer number as route identifier. We add fields for the length of each route curve and and two Boolean values indicating if the route is dual and start at the smaller end point. If the *streets* object contains R routes and a route curve of a route r_i has l_i line segments this will take

$$O(R + \sum_{i=1}^{R} l_i) = O(\sum_{i=1}^{R} l_i)$$

time.

In a next step we use *B_Routes* to compute the crossings of the street network of Berlin. Therefore we join all route of B_Routes with intersecting spatial bounding boxes and filter the routes that really intersect. For this pairs of routes we compute the positions of the junctions on the route curves and fill the resulting data in *B_Junctions* relation. The relation *B_Junctions* has the attributes: $r1id$, $r1meas$, $r2id$, $r2meas$, and cc. The last one is the connectivity code that should tell which lanes of the two routes are connected by the junction. But the data source lacks information about the connectivity of the street crossings, such that we use the maximum value for the connectivity code of each crossing as default value in this step. This step has a time complexity of $\mathrm{O}(R^2)$ in the worst case.

Now we can use *B_Routes* and *B_Junctions* as input relations for our network constructor **thenetwork** to create our <u>*network*</u> object *net* representing the streets of Berlin in the NET representation of the BerlinMOD Benchmark. The attributes of the *net* object have been explained in Section 3.1.

Algorithm 7 describes how *net* is created from the two input relations.

Algorithm 7 thenetwork(*nid*, *B_Routes*, *B_Junctions*)

```
Require: An unifique integer $nid \geq 0$, $B\_Routes$ and $B\_Junctions$ relation as described.
 1: Create Network empty network object net with id nid
 2: Copy B_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 B_Junctions 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 in junctions relation do
16:   Find pairs of adjacent sections and fill adjacency list
17: end for
```

Let J be the number of entries in *B_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* will be $\sum_{i=1}^{R}(j_i+1) = R + \sum_{i=1}^{R} j_i$,

The time complexities of the single steps of Algorithm 7 are:

line 1: O(1)

line 2: O(R)

line 3 and 4: O($R \log R$)

line 5: O(J)

line 6: O($J \log J$)

line 7 - 13: O($\sum_{i=1}^{R} j_i$)

line 14: O($(R + \sum_{i=1}^{R} j_i) \log(R + \sum_{i=1}^{R} j_i)$)

line 15 - 17: O(J)

For all steps together we get a time complexity of

$$O(1 + R + R\log R + J + J\log J + R + \sum_{i=1}^{R} j_i + (R + \sum_{i=1}^{R} j_i)\log(R + \sum_{i=1}^{R} j_i) + J)$$

$$= O((R + \sum_{i=1}^{R} j_i)\log(R + \sum_{i=1}^{R} j_i)),$$

because $R,\ J \leq R + \sum_{i=1}^{R} j_i$.

4.2 Translate Spatial and Spatio-Temporal Data Types

In this section we describe the translation of the spatial and spatio-temporal data types of the BerlinMOD Benchmark data set into network constrained objects. In the original paper this operations are all called **in_network**. All translations are done relative to the $\underline{network}$ object *net* of the previous section.

All algorithms in this section get a spatial respectively spatio-temporal BerlinMOD Benchmark data type object and the corresponding $\underline{network}$ object *net* as input. They return the corresponding data type from the network data model NET, respectively an undefined NET object if the input data object is not constrained by *net*.

4.2.1 Translate $\underline{point}$ into $\underline{gpoint}$

$\underline{network} \times \underline{point} \rightarrow \underline{gpoint}$ **point2gpoint**(*net*, *point*)

The **point2gpoint** operation translates a $\underline{point}$ value p into a corresponding $\underline{gpoint}$ value gp if possible. Variants of this operation are also included in the other translation operations. The operations in line 1 to 2 of Algorithm 8 need O(1) time.

Algorithm 8 point2gpoint(p)

```
1: bbox = spatial bounding box of p
2: bbox = extend bbox by 1.0 in every direction
3: Select set of candidate routes using bbox and R-Tree of routes relation
4: found = FALSE
5: while not found AND not isEmpty(candidateRoutes) do
6:   if Distance of point from route = 0 then
7:     found = true
8:     Compute position of point on route
9:   end if
10: end while
11: return corresponding gpoint value
```

Candidate routes are routes, which spatial minimum bounding boxes intersect with the spatial bounding box of the $\underline{point}$ value. The selection of *candidate routes* from the R-tree needs O($\log R + c$) time if c is the number of *candidate routes* for p. The assignment in line 4 takes O(1) time. In the worst case the loop from line 5 to line 10 is called c times. The computation in lines 6 to 8 takes in the worst case O(l_i) time, because we have to find the line segment to which p is connected. The result is returned in O(1) time. We get a worst time complexity of

$$O(\log R + c + \sum_{i=1}^{c} l_i + 1) = O(\log R + \sum_{i=1}^{c} l_i)$$

for the operation **point2gpoint**. Let this time complexity be called o_{P2G} in operations using **point2gpoint** operation.

In case of the BerlinMOD Benchmark the *side* value of *gp* is always set to *none*, because the BerlinMOD Benchmark does not differentiate between the different sides of a street.

This should be all to translate the $\underline{\text{point}}$ values of the *QueryPoints* relation of the BerlinMOD Benchmark into network query positions. But there is a problem with the NET representation of junctions. In the NET, contrary to SPACE, each junction has more than one $\underline{\text{gpoint}}$ representation, because each junction is related to two or more routes. Hence if a junction position is given related to route *a* we won't detect the junction as passed if an $\underline{\text{mgpoint}}$ object passes the junction on route *b* in all cases, because the definition of **passes** in the network data model is slightly different from the **passes** operation in the BerlinMOD Benchmark data model. Unfortunately all query points of the BerlinMOD Benchmark are junctions. To make the results comparable, we added an operator **polygpoints**, which returns for every input $\underline{\text{gpoint}}$ value *gp* a stream of $\underline{\text{gpoint}}$ values.

$\underline{\text{gpoint}} \rightarrow \underline{\text{stream}}(\underline{\text{gpoint}})$ **polygpoints**(*gpoint*)

If *gp* represents a junction we return all $\underline{\text{gpoint}}$ values representing the same junction in *net*, otherwise we return only *gp* in the stream. The Algorithm 9 describes the **point2gpoint** in detail. The worst case time complexity of the

Algorithm 9 polygpoints(*gp*)

```
1: Copy gp to output stream
2: Use B-Tree on junctions relation to Get junctions on route gp.rid
3: if gp is junction then
4:    for Each gpoint representing the same junction do
5:       Copy gpoint into output stream
6:    end for
7: end if
```

point2gpoint operation is O($\log J + j_i$) if j_i is the number of junctions on route i.

In the end we got 221 query $\underline{\text{gpoint}}$ values in *QueryPointsNet* for the 100 query $\underline{\text{point}}$ values in *QueryPoints* and 22 $\underline{\text{gpoint}}$ values in *QueryPoints1Net* for the 10 $\underline{\text{point}}$ values of *QueryPoints1* of the BerlinMOD Benchmark. This means we always have to compute the results for the double number of query points in our NET representation of the BerlinMOD Benchmark relativ to SPACE.

4.2.2 Translate $\underline{\text{mpoint}}$ into $\underline{\text{mgpoint}}$

The second operation **mpoint2mgpoint** translates an $\underline{\text{mpoint}}$ value *mp* into an $\underline{\text{mgpoint}}$ value *mgp*.

Algorithm 10 mpoint2mgpoint(*mp*,*net*)

```
1: Initialize empty $mgp$
2: $upoint$ = first unit of $mp$
3: Initialize $ugp = net$ values of $upoint$ {Uses variant of point2gpoint for two
   point on same route}
4: for Each $upoint$ of $mp$ do
5:   if Endpoint of $upoint$ is on same route than $ugpoint$ then
6:     if Direction and speed stay the same then
7:       Extend $ugpoint$ to include $upoint$
8:     else
9:       Add $ugpoint$ to $mgp$
10:      Add $route\ interval$ of $ugpoint$ to $RITree$ for $trajectory$
11:      $ugpoint = net$ values of $upoint$
12:    end if
13:  else
14:    Add $ugpoint$ to $mgp$
15:    Add $route\ interval$ of $ugpoint$ to $RITree$ for $trajectory$
16:    Search $upoint$ on adjacent sections route curves
17:    Change current route curve to route curve where the $upoint$ has been
       found
18:    $ugpoint = net$ values of $upoint$
19:  end if
20: end for
21: Add $ugpoint$ to $mgpoint$
22: Add $route\ interval$ of $ugpoint$ to $RITree$ for $trajectory$
23: Build $trajectory$ from $RITree$
24: Copy bounding box of $mp$ to $mgp$
25: return $mgp$
```

The main idea of the algorithm (see 10) is to use the continuous movement of *mp* to reduce computation time. We need the complete **point2gpoint** operation only for the first unit of the $\underline{mpoint}$ value. After that we can use the found route curve for computing the network position until the car changes the route. And if the car changes the route it can only drive on routes which are adjacent to the last used section, such that we only have to check the route curves of the adjacent sections instead of searching in the R-Tree for a route curve containing the current position.

Let $l_{xi},\ 1 \leq x \leq 2$ the number of line segments of route curve r_i, and s_i the number of route curves of adjacent sections of a section i [3]. The steps of the Algorithm 10 of the **mpoint2mgpoint** operation have the following time complexities:

line 1 + 2: O(1)

line 3: O(o_{P2G})

line 4 - 20: Loop will be called m_{in} times and knows three cases

[3]Each section has one route curve but two adjacent sections of a section may have the same route curve.

line 5: O(l_{1i})

1. : 6 + 7: O(1)
2. line 9 - 11: O($\log r + l_{1i}$)
   - line 9: O(1)
   - line 10: O($\log r$)
   - line 11: O(l_{1i})
3. line 14 - 18: O($\log r + \sum_{i=1}^{s_i} l_{2i} + l_{1i}$)
   - line 14: O(1)
   - line 15: O($\log r$)
   - line 16: O($\sum_{i=1}^{s_i} l_{2i}$)
   - line 17: O(1)
   - line 18: O(l_{1i})

line 21: O(1)

line 22: O($\log r$)

line 23: O(r)

line 24: O(m_{res})

The worst case time complexity for the whole algorithm is

$$O(o_{P2G} + m_{in}(l_{1i} + \log r + \sum_{i=1}^{s_i} l_{2i} + l_{1i}) + r + \log r + m_{res})$$
$$= O(o_{P2G} + m_{in}\log r + m_{in}\sum_{i=1}^{s_i} max(l_{1i},\ l_{2i}) + m_{res})$$

4.2.3 Translate region into gline

The translation of the region values in the *QueryRegions* relation of the BerlinMOD Benchmark into sorted gline values of NET is described in Algorithm 11.

Algorithm 11 Translate region values into sorted gline values

```
1: Build single line object $rl$ from the route curves of $routes$ relation
2: for Each $region$ of $QueryRegions$ do
3:    $lreg$ = intersection of $region$ and $rl$
4:    $netRegion$ = line2gline($lreg$)
5: end for
```

The step in line 1 has a time complexity of O($\sum_{i=1}^{R} l_i$). The loop is called for each entry in *QueryRegions*. In case of BerlinMOD Benchmark this will be 100 times. The intersection of an region value with a line value is computed by a planesweep algorithm using an AVL-Tree for the segments. Therefore line 3 has a time complexity of O($\sum_{i=1}^{R} l_i \log \sum_{i=1}^{R} l_i$). The operation **line2gline** in line 4 uses Algorithm 12 and has therefore a time complexity of

$$O(l_{1i}o_{P2G} + l_{1i}\log r + r)$$

Algorithm 12 line2gline(l, net)

```
1: for Each line segment $l_i$ of $l$ do
2:     Use variant of point2gpoint to find route curve including $l_i$
3:     Insert corresponding route interval into $RITree$
4: end for
5: return  sorted and compressed gline value from $RITree$
```

For the whole translation operation we get a worst case time complexity of

$$O(\sum_{i=1}^{R} l_i + (\sum_{i=1}^{R} l_i)\log(\sum_{i=1}^{R} l_i) + l_{1i}o_{P2G} + l_{1i}\log r + r)$$
$$= O((\sum_{i=1}^{R} l_1 i)\log(\sum_{i=1}^{R} l_1 i) + l_{1i}o_{P2G}).$$

The algorithm is very expensive, because it depends on existing SECONDO operations. This is acceptable for the current use with the BerlinMOD Benchmark, because the 100 regions are fixed and only translated once in the data generation step. It is planned to implement a own efficient translation operation for region values for latter use cases.

4.3 Created Indexes for NET Representation

For the use with the BerlinMOD Benchmark we created the indexes of Table 3 on the NET representation of the BerlinMOD Benchmark data sets.

Table 3: Indexes on NET Representation of BerlinMOD Benchmark

Name of Index	Explanation
dataSNcar_licence_btree	B-Tree on *licences* in *dataSNcar* relation.
dataMcar_licence_btree	B-Tree on *licences* in *dataMcar* relation.
dataMcar_Moid_btree	B-Tree on *moid* in *dataMcar* relation.
dataMNtrip_Moid_btree	B-Tree on *moid* in *dataMNtrip* relation.
dataSNcar_BoxNet_timespace	TNPI on *trip* in *dataSNcar*
dataMNtrip_BoxNet_timespace	TNPI on *trip* in *dataMNtrip*
dataSNcar_TrajBoxNet	NPI on *trip* in *dataSNcar*
dataMNtrip_TrajBoxNet	NPI on *trip* in *dataMNtrip*
dataSNcar_SpatioTemp	R-Tree on the spatio-temporal bounding box of *trip* in *dataSNcar*
dataMNtrip_SpatioTemp	R-Tree on the spatio-temporal bounding box of *trip* in *dataMNtrip*

The B-Tree indexes for the *licences* and *moid* attributes of the relations *dataSNcar*, *dataMcar*, and *dataMNtrip* are similar to the indexes created in the BerlinMOD Benchmark for *dataSCcar*, *dataMCcar*, and *dataMCtrip*, respectively. We don't explain them in more detail.

The Network Position Index (NPI) and the Temporal Network Position Index (TNPI) are new constructions. They are used in query processing to support a faster selection of *mgpoint* values that passed given network positions or network regions at / within a given time (TNPI) or without temporal restrictions (NPI). Detailed explanations of the trees and their construction can be found in Section 3.2.

The R-Tree indexes of the spatio-temporal bounding boxes of the *trip* attributes in *dataMNtrip* and t*dataSNcar* relation are different from the R-Trees of spatio-temporal bounding boxes used in the BerlinMOD Benchmark. In the NET representation only the big bounding boxes of the whole trips inserted in the indexes, whereas in SPACE representation the much smaller bounding boxes for each single unit are inserted. This is done because the computation of the small bounding boxes in NET representation is very expensive (see Section 3.1). It takes several hours to build the spatio-temporal unit bounding box indexes for the *mgpoint* values at the smallest scalefactor. For bigger scalefactors we stopped the attempt of computation after several days, because we think that the results of the current used indexes are not so bad that the additional build time is well invested. We don't expect a great improvement of query run times.

Really senseful this way? Move formal queries to Appendix.

4.4 Translate Benchmark Queries

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

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

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

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

Query 3 searches for the positions of the ten cars from *QueryLicence*1 at the ten time instants from *QueryInstants*1. We use the licence B-Tree to select the ten cars and compute the positions of these ten cars for each of the ten time instants from *QueryInstants*1 if the time instant is inside the definition time of the trip.

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

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

Query 5 asks for the minimum distance between places where vehicles with licences from *QueryLicence*1 and *QueryLicence*2 have been. We select the cars with licence plate numbers from *QueryLicence*1 respectively *QueryLicences*2 using the B-Tree over the *Licence* attribute of the *dataSNcar* relation. In the TBA, the resulting trajectories for each car are aggregated into one single trajectory for each car. In both approaches we create a $\underline{line}$ value for each resulting (aggregated) trajectory value of the $\underline{mgpoint}$s and compute the Euclidean distance between these $\underline{line}$ values for each pair of licences one from *QueryLicences*1 and one from *QueryLicences*2.

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

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

Query 8 computes the overall travelled distances of the vehicles from *QueryLicence*1 within the periods from *QueryPeriods*1. We select the candidate cars using the licence B-Tree, restrict the trips to the query periods and return the lengths of the trips in the OBA. In the TBA we have to sum up the length of the different trips driven by a single car within each query period.

Query 9 asks for the longest distance travelled by a single vehicle during each of the periods from *QueryPeriods*1. We restrict all trips to the periods, compute the driven distances and select the maximum length for each query periods value. Again we have to do an additional aggregation of the distances driven from the same car in the same period in the TBA.

Query 10 asks when and where vehicles with licences from *QueryLicence*1 meet which other vehicles (distance less than 3m). In the OBA we first retranslate every $\underline{mgpoint}$ value of *dataSNCar* into a $\underline{mpoint}$ value and extend the spatial bounding box of each of this trips by 1.5 m in every spatial direction. After that we select the ten candidate trips given by *QueryLicences*1, retranslate them and extend their spatial bounding boxes in the same way. We join all trips from the first two steps where the extended bounding boxes intersect and filter the candidate pairs that have different licences and their distance is sometimes less than 3m to each other. We compute the position of the $\underline{\textit{mgpoint}}$ at the times the distance between the remaining candidate pairs of $\underline{\textit{mpoint}}$ was less than 3 m and return the licence pairs and the network positions of the first car when it has been closer than 3 m to the other one.

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

In our experiments we tried out several indexes to support a faster query execution of query 10 including the MON-Tree presented in [8]. The MON-Tree showed very good CPU times but never the less the total run time was very high. In the end the simple form described above showed the best complete run time performance of all indexes.

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

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

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

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

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

Query 16 asks for the licence pairs one from $QueryLicence1$ and one from $QueryLicence2$ of vehicles, which were both present in a region from $QueryRegions1Net$

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

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

Controll this and next section for right use of NET and SPACE instaed of old abbreviations.

5 Experimental Setup

For our experiments we used a standard personal computer with an AMD Phenom II X4 Quad Core 2.95 GHz CPU, 8 GB main memory, and 2 TB hard disk. We installed the Linux openSUSE 11.2 as operating system, SECONDO DBMS version 3.0, and the BerlinMOD Benchmark version provided in the web.

Provide scripts in web and add link to scripts!

We generated three databases with different amounts of data using the data generation script of the BerlinMOD Benchmark with the $scalefactor$ 0.05, 0.2, and 1.0. The following steps are done with all three databases. We first created the BerlinMOD Benchmark data and indexes using the script "BerlinMOD_CreateObjects.SEC" for the DMFS. The NDM representation of the databases was generated by the the script "Network_CreateObjects.SEC" that uses the algorithms and builds the indexes described in Section 4.

Table 4 shows the created amounts of data for the different $scalefactor$ values in both data models. As you can see, the NDM needs less than 40% of the storage space of the BerlinMOD Benchmark data model. The main cause is that the same trip is represented by less than 50% of the units in the NDM compared to the DMFS. This is a very good result and we expect this effect to increase if the cars make long distance trips instead of moving in a single town like they do in the benchmark. In towns cars more often change the street or the velocity than cars that do long distance trips and so the compact route representation in the NDM should become more effective than in the town.

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

Table 4: Database Statistics

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

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

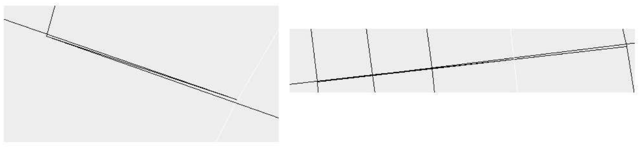

Figure 1: Example Failures in Street Map

6 Experimental Results

We repeated the BerlinMOD Benchmark query execution several times for both data models and approaches. The tables in Figure 2 and the graphic in 3 compare the average query run times in seconds for the different scale factors, data models, and approaches. As you can see, the total run time of all queries in the NDM is around 50% less than the total query run time of the DMFS at each scale factor.

	Scalefactor 0.05			
	DMFS		NDM	
Query	**OBA**	**TBA**	**OBA**	**TBA**
1	0.125	0.109	0.128	0.088
2	0.003	0.002	0.003	0.002
3	0.245	0.205	0.227	0.268
4	6.594	7.514	0.238	0.846
5	1.072	1.585	1.098	1.031
6	14.332	6.280	3.995	3.675
7	3.458	3.191	3.893	3.192
8	0.353	0.379	0.201	0.205
9	96.724	166.434	19.840	21.783
10	104.239	31.555	62.972	76.826
11	0.150	0.096	0.224	0.443
12	0.296	0.120	0.202	0.226
13	9.959	6.551	1.094	1.113
14	0.516	0.659	1.566	1.709
15	1.144	0.857	0.579	0.488
16	6.214	14.354	0.612	1.483
17	1.126	0.719	0.228	0.262
Total	246.551	240.609	97.061	113.640

	Scalefactor 0.2			
	DMFS		NDM	
Query	**OBA**	**TBA**	**OBA**	**TBA**
1	0.123	0.096	0.157	0.099
2	0.003	0.003	0.008	0.003
3	0.501	0.327	0.424	0.349
4	32.323	39.113	0.500	6.522
5	1.657	2.985	2.394	2.198
6	57.453	45.931	15.508	13.451
7	17.184	11.150	27.084	22.814
8	0.390	0.379	0.205	0.220
9	250.626	386.452	36.323	46.187
10	447.679	126.870	286.779	287.101
11	0.247	0.165	2.119	2.958
12	4.171	0.182	0.262	0.316
13	27.028	12.106	5.865	4.669
14	1.125	1.143	8.783	9.101
15	7.800	3.874	2.543	1.876
16	6.441	25.298	0.362	0.776
17	8.722	4.042	0.289	0.641
Total	863.472	660.217	389,605	399.281

	Scalefactor 1.0			
	DMFS		NDM	
Query	**OBA**	**TBA**	**OBA**	**TBA**
1	0.196	0.186	0.387	0.185
2	0.005	0.004	0.006	0.004
3	0.731	0.483	1.020	1.349
4	150.172	157.629	2.089	31.769
5	3.274	6.079	5.230	5.494
6	826.483	2002.002	270.468	235.594
7	99.086	53.099	118.840	125.206
8	0.794	0.524	0.254	0.398
9	775.458	2263.531	106.910	143.150
10	3314.518	1942.155	2150.250	1645.812
11	0.685	0.474	6.080	7.889
12	37.445	0.200	0.272	0.290
13	111.587	72.907	26.880	32.540
14	11.397	4.238	36.728	37.700
15	28.512	16.862	9.696	8.602
16	9.726	53.011	0.571	1.880
17	86.357	152.542	0.530	6.884
Total	5456.427	6725.924	2736.210	2284.747

Figure 2: Compare Query Run Times in Seconds

For the queries 1 and 2, the query run times are almost the same for all data models and approaches at the different scale factors. This is what we expected because both queries deal only with standard attributes and standard indexes, which are not influenced by the different data models.

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

In query 4 the NDM outperforms the DMFS significantly at all scale factors ($>$ 2 min OBA, $>$ 6 min TBA at scale factor 1.0). The NDM index used in query 4 is much smaller (OBA 24 MB, TBA 160 MB, at scalefactor 1.0) than the spatial unit index of the BerlinMOD Benchmark (OBA 3.7 GB, TBA 3.7

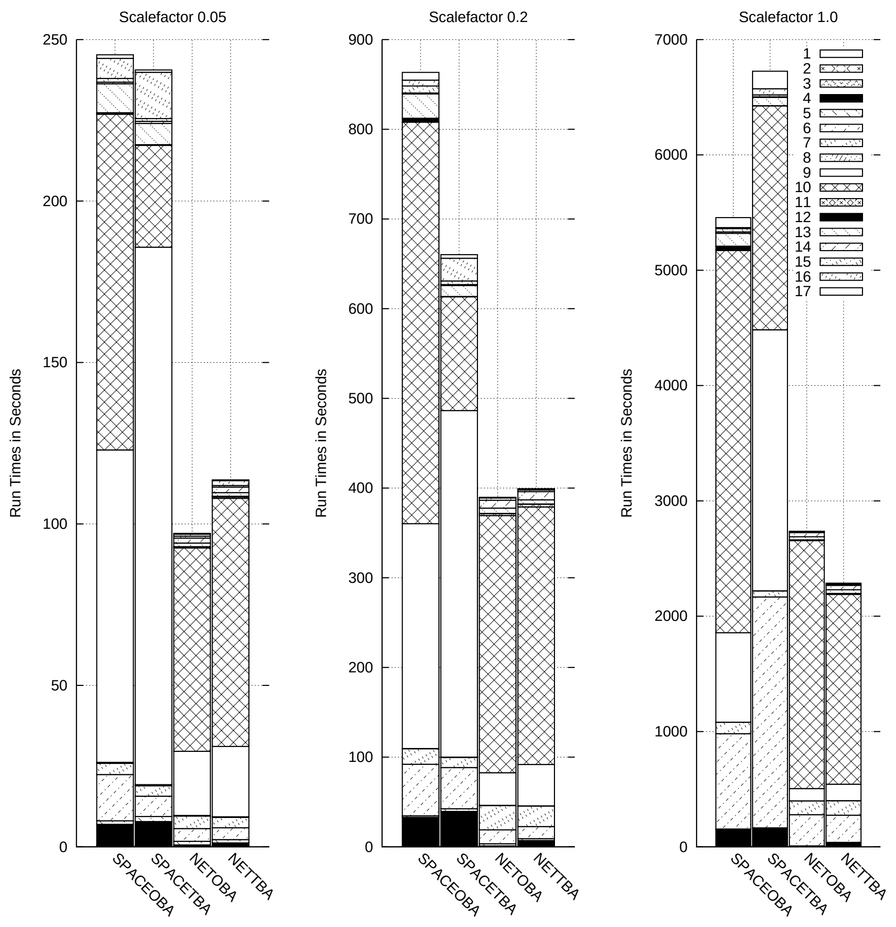

Figure 3: Compare Total Run Times

GB at scalefactor 1.0) and more precise, such that we do not need an additional refinement step after the index usage in the NDM, like we do in the DMFS.

We expected the NDM to be slower than the DMFS in the queries 5,6, and 10, because we retranslate intermediate results from the NDM representation into the DMFS representation. For query 5 this holds in the OBA. We need a little more time in the NDM than in the DMFS. But in TBA the NDM outperforms the DMFS. This is due to the fact that a *gline* value has less *RouteInterval*s than a *line* value representing the same part of a curve has *HalfSegment*s, such that the union of two or more *gline* values in the aggregate step of query 5 in TBA can be computed much faster than the union of two or more *line* values.

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

check analysis for correctness.

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

Query 8 is a very fast query in both data models, although the query run time of the NDM is more than 50% less than the query run time of the DMFS. This is caused by the *length* attribute of the *mgpoint* and the smaller number of units of a *mgpoint* compared with the corresponding *mpoint*.

For query 9, the NDM outperforms the DMFS by orders of magnitude. The advantages named in the analysis of query 8's run time results have a much higher impact when the number of examined trips becomes bigger. At scale factor 1.0 this saves more than 10 min time in the OBA and more than 50 min time in the TBA.

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

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

In our experiments we also tested the MON-Tree [8] as network-temporal index but the elapsed run time performance was not good, although the CPU run times was were small.

The bad performance of the network-temporal position index is also shown by query 13. The NDM outperforms the DMFS significantly, but we do not use any index in the executable NDM queries, while the DMFS uses its spatio-temporal index to preselect candidate trips. The same holds for query 17.

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

Although we detected in our experiments some points of weakness in the network-temporal position indexing, the NDM outperforms the DMFS by orders of magnitude. The weakness of the NDM mostly occurs in queries with short run times, whereas the advantages of the NDM become apparent in the queries with long run times, such that the weakness of the network-temporal position index is covered by the advantages of the network data model.

7 Summary and Future Work

We presented our translation of the BerlinMOD Benchmark into the NDM and compared the capabilities of both data models, with very good results for the NDM. Our experiments show that the NDM outperforms the DMFS by orders of magnitude with respect to storage space and query run times. This is mainly caused by the much lower number of units for an *mgpoint* value compared with the number of units of the corresponding *mpoint*, which also results in smaller indexes for the NDM objects. The BerlinMOD Benchmark of the NDM pointed out that we should spend time in the improvement of the network-temporal position index and the **at** operation for *mgpoint* and *gpoint* values.

The good results of the NDM encourage us to work on an extension of the BerlinMOD Benchmark, which should enable us to compare the capabilities of different spatio-temporal NDMs with respect to the special challenges of NDM, like shortest path and fastest path computation.

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

Another interesting topic for future work on the NDMs is the efficient computation of dynamic network distances between moving network objects.

References

[1] R. Bayer and E.M. McCreight. Organization and maintenance of large ordererd indexes. *Acta Informatica*, 1:173–179, September 1972.

[2] Statistisches Landesamt Berlin. Bevölkerungsstand in Berlin Ende September 2006 nach Bezirken, 2008. `http://www.statistik-berlin.de/framesets/berl.htm`.

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

[4] T. Brinkhoff. A Framework for Generating Network-Based Moving Objects. *GeoInformatica*, 6(2):153–180, June 2002.

[5] T. Behr C. Düntgen and R.H. Güting. BerlinMOD: A Benchmark for Moving Object Databases. *The VLDB Journal*, 18(6):1335–1368, December 2009.

[6] C. X. Chen and C. Zaniolo. SQLST A Spatiotemporal Model and Query Language. In *Conceptual Modeling - ER 2000*, volume 1920/2000, pages 96–111. Springer Berlin, Heidelberg, 2000.

[7] Oracle Corporation. Oracle Web Site, June 2010. `http://www.oracle.com`.

[8] V.T. De Almeida and R.H. Güting. Indexing the Trajectories of Moving Objects in Networks. *Geoinformatica*, 9(1):33–60, 2005.

[9] S. Dieker and R.H. 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.

[10] Z. Ding. Data Model, Query Language, and Real-Time Traffic Flow Analysis in Dynamic Transportation Network Based Moving Objects Databases. *Journal of Software*, 20(7):1866–1884, July 2009.

[11] Z. Ding and R.H. Güting. Managing moving objects on dynamic transportation networks. In *Proc. of the 16th Intern. Conf. on Science and Statistical Database Management.*

[12] Z. Ding and R.H. Güting. Modeling temporally variable transportation networks. In *Proceedings of the 9th Intern. Conf. on Database Systems for Advanced Applications.*

[13] M. Erwig, R. H. Güting, M. Schneider, and M. Vazirgiannis. Spatio-Temporal Data Types: An Approach to Modeling and Querying Moving Objects in Databases. *Geoinformatica*, 3(3):269–296, 1999.

[14] L. Forlizzi, R.H. Güting, E. Nardelli, and M. 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.

[15] G. Gidofalvi and T.B. Pedersen. St–acts: a spatio-temporal activity simulator. In *GIS '06: Proceedings of the 14th annual ACM international symposium on Advances in geographic information systems*, pages 155–162, New York, NY, USA, 2006. ACM.

[16] PostgreSQL Global Development Group. PostgreSQL Web Site, June 2010. `http://www.postgresql.org/`.

[17] R.H. Güting. Second Order Signature: A Tool for Specifying Data Models, Query Processing, and Optimization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data in Washington*, pages 277–286.

[18] R.H. Güting, V. Almeida, D. Ansorge, T. Behr, Z. Ding, T. Hose, F. Hoffmann, M. Spiekermann, and U. 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.

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

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

[21] A. Guttmann. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proc. of the ACM Intl. Conf. on Management of Data SIGMOD*, pages 47–57, June 1984.

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

[23] Fernuniversität Hagen. Secondo Web Site, April 2009. `http://dna.fernuni-hagen.de/secondo/index.html`.

[24] Fernuniversität Hagen. Website Network Data Model and BerlinMOD Benchmark, June 2010. `http://dna.fernuni-hagen.de/secondo/BerlinMOD/Network.html`.

[25] G.J. Hunter and I.P. Williamson. The developement of a historical digital cadastral database. *International Journal of Geograhpich Information Systems*, 4(2), 1990.

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

[27] C.S. Jensen, D. Tiesyte, and N. Tradisauskas. The COST Benchmark - Comparison and Evaluation of Spatio-Temporal Indexes. In *Database Systems for Advanced Applications*, volume 3882/2006, pages 125–140. Springer Berlin, Heidelberg, 2006.

[28] G. Langran. *Time in Geographical Information Systems*. Taylor and Francis, 1992.

[29] G. Langran and N.R. Chrisman. A Framework For Temporal Geographic Information. *Cartographica: The International Journal for Geographic Information and Geovisualization*, 25(3):1–14, October 1988. `http://utpjournals.metapress.com/content/K877727322385Q6V`.

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

[31] L.-V. Nguyen-Dinh, W.G. Aref, and M. Mokbel. Spatio-temporal Access Methods: Part 2(2003 - 2010). *IEEE Data Eng. Bull.*, 33(2):46–55, June 2010.

[32] N. Pelekis. *STAU: A Spatio-Temporal Extension to ORACLE DBMS, school = University of Manchester Institute of Science and Technology, year = 2002, type = PhD Thesis,*. PhD thesis.

[33] N. Pelekis and Y. Theodoridis. An Oracle Cartridge for Moving Objects. Technical report, December 2007.

[34] N. Pelekis, Y. Theodoridis, S. Vosinakis, and T. 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.

[35] N. Pelekis, B. Theodoulidis, I. Kopanakis, and Y. Theodoridis. *The Knowledge Engineering Review*, (4):235–274, 2004.

[36] A. Ramachandran, F. MacLoad, and S. Dowers. Modeling temporal changes in a GIS using an object oriented approach. In *Advances in GIS Research, 6th International Symposium on Spatial Data Handling*, 1994.

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

[38] P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modelling and Querying Moving Objects. In *Proceedings of the 13th International Conference of Data Engineering (ICDE13)*, pages 422–432. IEEE Computer Society, 1997.

[39] L. Speičvcys, C.S. Jensen, and A. 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.

[40] J. Su, H. Xu, and O.H. Ibarra. Moving objects: Logical relationships and queries. In *Advances in Spatial and Temporal Databases*, volume 2121 of *Lecture Notes in Computer Science*, pages 3–19. Springer Berlin / Heidelberg, 2001.

[41] Y. Theodoridis. Ten Benchmark Database Queries for Location-Based Services. *The Computer Journal*, 46(6):713–725, 2003.

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

[43] M. Vazirgiannis and O. 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.

A Executable Secondo Queries

In the sequel we present our executable SECONDO queries for the NET representation of the BerlinMOD Benchmark. The name of the query result object indicates the number of the query, and it is a query for the object based approach (OBA), or for the trip based approach (TBA).

```
let Q1OBA =
    QueryLicences feed {l}
        loopjoin [dataSNcar_Licence_btree dataSNcar exactmatch [.Licence_l]]
            project [Licence, Model]
consume;

let Q1TBA =
    QueryLicences feed {l}
        loopjoin [dataMcar_Licence_btree dataMcar exactmatch[.Licence_l]]
        project [Licence, Model]
consume;

let Q2OBA = dataSNcar feed filter [.Type = 'passenger'] count;

let Q2TBA = dataMcar feed filter [.Type = 'passenger'] count;

let Q3OBA =
    QueryLicences1 feed {l}
        loopjoin [dataSNcar_Licence_btree dataSNcar exactmatch [.Licence_l]]
        project [Licence, Trip]
    QueryInstant1 feed {i}
    product
    projectextend[Licence, Instant_i; Pos: val(.Trip atinstant .Instant_i)]
consume;

let Q3TBA =
    QueryLicences1 feed {l}
        loopsel[dataMcar_Licence_btree dataMcar exactmatch[.Licence_l] {ll}]
            loopjoin[dataMNtrip_Moid_btree dataMNtrip exactmatch[.Moid_ll]]
    QueryInstant1 feed {i}
    symmjoin [.Trip present ..Instant_i]
        projectextend[Instant_i, Licence_ll; Pos: val(.Trip atinstant .Instant_i)]
consume;

let Q4OBA =
    QueryPointsNet feed projectextend[Id, Pos; Prect: gpoint2rect(.Pos)]
    loopjoin[dataSNcar_TrajBoxNet windowintersectsS[.Prect]
            sort rdup dataSNcar gettuples]
    project [Id, Licence]
    sortby [Id asc, Licence asc]
    krdup [Id, Licence]
consume;

letQ4TBA =
    QueryPointsNet feed projectextend[Id; Elem: gpoint2rect(.Pos)]
    loopjoin[dataMNtrip_TrajBoxNet windowintersectsS[.Elem]
            sort rdup dataMNtrip gettuples]
    project [Moid, Id]
    loopsel[fun(t:TUPLE) dataMcar_Moid_btree dataMcar exactmatch[attr(t, Moid)]
        projectextend[Licence; Id: attr(t,Id)]]
    sortby [Id asc, Licence asc]
    krdup[Id, Licence]
consume;

let Q5OBA =
    QueryLicences1 feed {l1}
        loopsel[dataSNcar_Licence_btree dataSNcar exactmatch [.Licence_l1]
            projectextend[Licence; TrajLine: gline2line(trajectory(.Trip))]]{c1}
    QueryLicences2 feed {l2}
        loopsel[dataSNcar_Licence_btree dataSNcar exactmatch [.Licence_l2]
            projectextend[Licence; TrajLine: gline2line(trajectory(.Trip))]]{c2}
    product
    projectextend [Licence_c1, Licence_c2; Distance: distance(.TrajLine_c1, .TrajLine_c2)]
consume;

let Q5TBA =
    QueryLicences1 feed project[Licence] {LL1}
        loopsel[fun (t:TUPLE) dataMcar_Licence_btree dataMcar exactmatch[attr(t,Licence_LL1)] {CAR}
            loopsel[dataMNtrip_Moid_btree dataMNtrip exactmatch[.Moid_CAR]]
            projectextend[;Traj: trajectory(.Trip)]
            aggregateB[Traj; fun (L1: gline, L2: gline) L1 union L2; [const gline value ()]]
            feed namedtransformstream[Traxj]
            extend[Licence: attr(t,Licence_LL1)]]
        projectextend[Licence; Trax: gline2line(.Traxj)]{c1}
    QueryLicences2 feed project[Licence] {LL2}
        loopsel[fun (s:TUPLE) dataMcar_Licence_btree dataMcar exactmatch[attr(s,Licence_LL2)] {CAR}
            loopsel[dataMNtrip_Moid_btree dataMNtrip exactmatch[.Moid_CAR]]
            projectextend[;Traj: trajectory(.Trip)]
            aggregateB[Traj; fun (L3: gline, L4: gline) L3 union L4; [const gline value ()]]
            feed namedtransformstream[Traxj]
            extend[Licence: attr(s,Licence_LL2)]]
        projectextend[Licence; Trax: gline2line(.Traxj)]{c2}
    product
    projectextend[Licence_c1, Licence_c2; Distance: distance(.Trax_c1, .Trax_c2)]
consume;

let Q6hOBA =
    dataSNcar feed filter [.Type = 'truck']
    projectextend [Licence; ptrip: mgpoint2mpoint(.Trip), BBox: mgpbbox(.Trip)]
    projectextend [Licence, ptrip; Box: rectangle3(minD(.BBox,1) - 5.0, maxD(.BBox,1) + 5.0,
            minD(.BBox,2) - 5.0, maxD(.BBox,2) + 5.0,minD(.BBox,3), maxD(.BBox,3))]
consume;
let Q6OBA =
    Q6hOBA feed {a}
    Q6hOBA feed {b}
    symmjoin[(.Box_a intersects ..Box_b) and (.Licence_a ¡ ..Licence_b) and
            (everNearerThan(.ptrip_a, ..ptrip_b, 10.0))]
    project [Licence_a, Licence_b]
    sortby [Licence_a asc, Licence_b asc]
    krdup [Licence_a, Licence_b]
consume;
delete Q6hOBA;

let Q6hTBA =
    dataMcar feed filter [.Type = "truck"] project [Licence, Moid] {c}
    loopjoin[dataMNtrip_Moid_btree dataMNtrip exactmatch[.Moid_c]]
    projectextend [;Licence: .Licence_c, BBox: mgpbbox(.Trip), ptrip: mgpoint2mpoint(.Trip)]
    projectextend [Licence, ptrip; Box: rectangle3((minD(.BBox,1) - 5.0), (maxD(.BBox,1) + 5.0),
            (minD(.BBox,2) - 5.0), (maxD(.BBox,2) + 5.0), minD(.BBox,3), maxD(.BBox,3))]
consume;
let Q6TBA =
    Q6hTBA feed {c1}
    Q6hTBA feed {c2}
    symmjoin[(.Box_c1 intersects ..Box_c2) and (.Licence_c1 ¡ ..Licence_c2)]
    filter [everNearerThan(.ptrip_c1, .ptrip_c2, 10.0)]
    project [Licence_c1, Licence_c2]
    sortby [Licence_c1 asc, Licence_c2 asc]
    krdup [Licence_c1, Licence_c2]
consume;
delete Q6hTBA;

let Q7OBA =
  QueryPointsNet feed projectextend[Id, Pos; Prect: gpoint2rect(.Pos)] {a}
  QueryPointsNet feed projectextend[Id, Pos; Prect: gpoint2rect(.Pos)]
    loopsel[fun (t:TUPLE) dataSNcar_TrajBoxNet windowintersectsS[attr(t,Prect)]
        sort rdup dataSNcar gettuples
      filter[.Type = "passenger"]
      projectextend[; Id: attr(t,Id) , Instant: inst(initial(.Trip at attr(t,Pos)))]]
      filter[not(isempty(.Instant))]
      sortby[Id asc, Instant asc]
      groupby[Id; FirstTime: group feed min[Instant]]{b}
  symmjoin[.Id_a = ..Id_b]
  projectextend[Id_a, FirstTime_b, Pos_a; MBR: box3d(.Prect_a, .FirstTime_b)]
  loopjoin[dataSNcar_BoxNet_timespace windowintersectsS[.MBR]
        sort rdup dataSNcar gettuples]
  filter[.Type = "passenger"]
  projectextend[Licence, FirstTime_b, Id_a; Instant: inst(initial(.Trip at .Pos_a))]
  filter[not(isempty(.Instant))]
  filter[.Instant ¡= .FirstTime_b]
  project[Id_a, Licence]
consume;
let Q7hTBA =
  QueryPointsNet feed projectextend[Id, Pos; Prect: gpoint2rect(.Pos)]
  loopsel[fun (t:TUPLE) dataMNtrip_TrajBoxNet windowintersectsS[attr(t,Prect)]
        sort rdup dataMNtrip gettuples
    loopjoin[dataMcar_Moid_btree dataMcar exactmatch [.Moid]
      filter[.Type = "passenger"]
      project[Licence] {X}]
    projectextend[ Licence_X;TimeAtPos: inst(initial(.Trip at attr(t,Pos))), Id: attr(t, Id)]]
  sortby [Id asc, TimeAtPos asc]
consume;
let Q7TBA =
  Q7hTBA feed {a}
  Q7hTBA feed groupby [Id; FirstTime: group feed min[TimeAtPos]]{b}
  symmjoin[(.Id_a = ..Id_b)]
  filter[.TimeAtPos_a ¡= .FirstTime_b]
  project [Id_a, Licence_X_a]
  sortby [Id_a asc, Licence_X_a asc]
  krdup [Id_a, Licence_X_a]
consume;
delete Q7hTBA;
```