Network Data Model and BerlinMOD Benchmark

Simone Jandt

Last Update: July 23, 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.

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 [14,19,31,32] 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 coordinate pairs in the xy plane. 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 [6,25,27,28,35] 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 [34].

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 can be 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 [31] 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 [31] also supports arc interpolation of movement. [31] 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, [31] 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, [31] 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 use 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 positions are given related to 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: [31] is implemented as data cartridge [32, 33] 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 [31] 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 been 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 [26]. 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 [29] and [30]. 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 [37], or support only short-term observations like [4], or 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 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 $\{\perp\}$. $\perp$ represents an undefined value.

The carrier sets for the BASE data types in the discrete data model are defined by the corresponding programming language data types $\underline{int}$, $\underline{real}$, $\underline{bool}$, and $\underline{string}$. The carrier set of a value of the data type $\underline{point}$ is $\underline{real} \times \underline{real}$. The two $\underline{real}$ values represent the coordinate pair of the position of the $\underline{point}$ value in the xy plane.

The data type $\underline{points}$ consists of a disjoint set of $\underline{point}$ values.

The carrier set for the data type $\underline{line}$ consists of a finite set of disjoint line segments representing the linear approximation of the line curve in the two dimensional plane. Semanticaly an $\underline{line}$ 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 $\underline{region}$ 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 $\underline{real}$ in the discrete data model. That means each time $\underline{instant}$ is represented by a corresponding $\underline{real}$ value.

The type constructor $\underline{range}$ 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 network constrained data model NET 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 the advantage that the representation of moving objects that move over several sections of the same route with constant speed becomes much smaller. We only have to store an new unit if the moving object changes the route or the speed, not every time it passes a junction like in the edge based 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

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

where d is the distance from the start of the route and the connectivity code cc encodes which lanes of the routes 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 $RLoc(R)$ 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 of *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 an undefined value.

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 *route interval* 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_i is called region of N_i. The set of all possible regions in a network N_i is denoted as $Reg(N_i)$.

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

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

[1] See [20] for a detailed explanation of the different connectivity code values.

. The set of *route intervals* 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 of a *mgpoint* are 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 use in our experiments later in Section 3 and Section 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 [36] 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 NCDM, 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 *trip* (*mgpoint*), the *licence*, the *type*, and the *model* of the object (all *string*).

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* (int). *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 *point* value
QueryRegions	Object identifier and *region* value
QueryInstants	Object identifier and time instant
QueryPeriods	Object identifier and time interval
QueryLicences	Object identifier and a *string* representing a licence value

Table 1: Query Object Relations of BerlinMOD Benchmark

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

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

- known or unknown object identity,
- standard 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 4.4 together with our NET algorithms for these queries.

3 Implementation of NET

In Section 2.3 we described the implementation of NET provided by [20]. The provided implementation has been changed and extended by us at some points to support faster query execution. In this section we describe the current implementation of the NET data structure and data types in SECONDO DBMS. In

Section 3.1 we present our implementation of the NET data types. In Section 3.2 we introduce so called (temporal) network bounding boxes that can be used in R-Trees to index (temporal) network positions. In Section 3.3 we describe the implementation of NET operations used by the BerlinMOD Benchmark.

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 have been integrated in the data type *network*. An R-Tree has been integrated indexing the *route 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 (rid)
- if two *route intervals* have the same route identifier (rid), the *route interval* with the smaller start position d_1 is stored first if the *route intervals* are disjoint
- for all *route intervals*, *start position* $\leq$ *end position*

We introduced this definition and the 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 *gline* is computed once, but many different algorithms can be executed faster for sorted *gline* values.

The implementation of the *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 <u>*mgpoint*</u> 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 <u>*ugpoint*</u> is the same as in the data model of [20]. The *drivenDist* is the total length of all <u>*ugpoint*</u> in the <u>*mgpoint*</u>. The DBArray of *route intervals* represents all network positions ever traversed by the <u>*mgpoint*</u>. The flag indicates if the *trajectory* is well defined, because this attribute is not maintenanced in every operation changing <u>*mgpoint*</u> 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 <u>*gline*</u> values the *trajectory* value makes it much faster to decide whether an <u>*mgpoint*</u> ever passed a given network position or not. Instead of a linear check of all m units of an <u>*mgpoint*</u> 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 operations like **passes**.

The spatio-temporal minimum bounding box was introduced as an attribute to the <u>*mgpoint*</u> because the computation of this value is very expensive in NET. Although each unit of an <u>*mgpoint*</u> 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 Computation Spatio-Temporal Unit Bounding Box

1: Get *route curve* for *rid* using B-Tree index of *routes* relation
2: Extract subline of unit from *route curve*
3: Compute bounding box of subline
4: Add time 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 <u>*mgpoint*</u> value. Therefore the time complexity to compute a *mbr* is $O(m \log R + \sum_{i=1}^{m} l_i)$.

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 <u>*mgpoint*</u> to compute the spatial bounding box. The third dimension can be added using the start and the end time instant of the <u>*mgpoint*</u> 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 Network Environment Indexes

Spatial and spatio-temporal bounding boxes are used to support 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. In our experimental evaluation we also tried the MON-Tree [8] and more detailed spatio-temporal unit bounding box R-Trees, but they all haven been outperformed 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 *route interval* is a degenerated two dimensional rectangle (rid, rid, d_1, d_2). Analogous the TNBB of a ugpoint value is a degenrated three dimensional rectangle (rid, rid, d_1, d_2, t_1, t_2).

We use these NBB and TNBB to create R-Trees indexing the network positions (NPI) and temporal network positions (TNPI) of the mgpoint values of the BerlinMOD Benchmark. The NPI is created from the NBB of the *route intervals* of the *trajectory* attribute of the mgpoint values. And the TNPI is created from the TNBB of the ugpoint values of the mgpoint values.

3.3 Network Operations used in BerlinMOD Benchmark

The operations used to construct an network object, 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.

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

Table 2 gives an overview of the simple operations for the NET representation of the BerlinMOD Benchmark.

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 gline values, respectively, two mgpoint values as input and mixes them up to an single sorted gline, respectively, mgpoint value.

gline × gline → gline	*gline*1 **union** *gline*2
mgpoint × mgpoint → mgpoint	*mgpoint*1 **union** *mgpoint*2

Table 2: Simple NET Operations

Name	Signature	Explanation	Complexity
routes	*network* → *relation*	Returns the *routes* relation of the *network* object	O(R)
no_components	*mgpoint* → *real*	Returns the number of units of the *mgpoint*	O(1)
length	*mgpoint* → *real*	Returns the driven distance of the *mgpoint*	O(1)
trajectory	*mgpoint* → *gline*	Returns the *trajectory* of the *mgpoint* value as sorted *gline*	O($m \log r + r$) if *trajectory* is defined, O(r) otherwise
units	*mgpoint* → *stream*(*ugpoint*)	Returns the *ugpoint* values of the *mgpoint* value as stream	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 network position 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 NBB for each *route interval* of the *gline* value in a stream	O(r)
gpoint2rect	*gpoint* → *rectangle*	Returns the NBB of the *gpoint* value	O(1)
unitbox	*ugpoint* → *rectangle3D*	Returns the TNBB of the *ugpoint* value	O(1)

Algorithm 2 union(gl_1, gl_2)

```
1: if gl1 is sorted AND gl2 is sorted then
2:   Perform parallel scan of gl1 and gl2
3:   if Current route intervals do not intersect then
4:     Add smaller route interval to result
5:     Continue Scan with next route interval
6:   else
7:     Merge route intervals into one
8:     if Next route intervals intersect the merged one then
9:       merge them too
10:    end if
11:    Add merged route interval to result
12:    Continue Scan
13:  end if
14: else
15:   for Each route interval of gl1 and gl2 do
16:     Insert route interval into RITree
17:   end for
18:   Copy sorted route intervals from RITree to result
19: end if
20: return result
```

The Algorithm 2 distinguishes between two cases: If both *gline* values are sorted (line 2 to line 13) the algorithm has a time complexity of O($r_1 + r_2$). If one or both *gline* are not sorted (line 15 to line 18) the time complexity is 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 mentioned before (see Section 3.1) we think that this time is well invested, because many 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 *ugpoint*s 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 × periods → 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 × periods → bool   *mgpoint* **present** *periods*

The operation **present** returns *TRUE* if the mgpoint value is defined at least at one time instant inside the given Periods value. 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 $O(\log m)$.

gpoint × gline → bool   *gpoint* **inside** *gline*

The operation **inside** returns *TRUE* if a gpoint is inside a gline. 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.

gline × gline → bool   **intersects**(*gline*1, *gline*2)

The operation **intersects** returns *TRUE* if two gline values intersect, *FALSE* otherwise. The Algorithm 3 for **intersects** differentiates three cases:

1. If both gline values are sorted (see line 2 to line 5) the time complexity is $O(r_1 + r_2))$

2. If both gline values are not sorted (see line 8 to line 14) the time complexity is $O(r_1 r_2)$

3. If only one gline value is sorted (see line 16 to line 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.

[2] The first and last time interval might be splitted at the start (resp. end) time instant of the periods value in O(1) time.

Algorithm 3 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 gl1 do
      for Each route interval of gl2 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
```

For the Euclidean Distance computation we retranslate our network values into spatial (**gline2line**) respectively spatio-temporal (**mgpoint2mpoint**) values. In [20] all these operations are called **in_space**.

<u>gline</u> → <u>line</u>	**gline2line**($gline$)
<u>mgpoint</u> → <u>mpoint</u>	**mgpoint2mpoint**($mgpoint$)

The operation **gline2line** uses Algorithm 4 to translate an <u>gline</u> value into an <u>line</u> value.

Algorithm 4 gline2line(gl)

```
1: for each route interval ri of gl do
2:   Get route curve rc of ri using B-Tree index of routes relation
3:   Perform binary search on the line segments of rc to find d1 of ri
4:   Copy line segments to result line until d2 of ri is reached.
5: end for
6: return line
```

The loop from line 1 to line 5 is repeated r times. The B-Tree search in line 2 takes $O(\log R)$ time. The binary search in line 3 takes $O(\log l_i)$ time. The copy operation in line 4 takes in the worst case $O(l_i)$ time. The return of the result has a worst case time complexity of $O(\sum_{i=1}^{r} l_i)$. 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 5. The loop from line 3 to line 28 is repeated m times. The statements from line 5 to line 8 has a worst case time complexity from $\mathrm{O}(\log R + l_i)$. But these lines are only executed if the car enters a new *route curve*. We have three different cases for the computing of the *mpoint* units. The two simple cases from line 10 to line 11 and line 13 to line 16 have a time complexity of O(1). The third case (line 18 to line 25) has a worst case time 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 <u>*mpoint*</u> value. In the worst case we get a time complexity for Algorithm 5 of

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

Algorithm 5 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 gp1 of curUGP
8:      upstart = x,y-coordinates of gp1
9:    end if
10:   if gp1 = gp2 then
11:     add unit upstart, upstart to mpoint
12:   else
13:     if gp2 is on l then
14:       upend = x,y-coordinates of gp2
15:       add unit upstart, upend to mpoint
16:       upstart = upend
17:     else
18:       Follow rc in moving direction of curUGP
19:       while not(gp2 is on l) do
20:         add unit from upstart to segment end position to mpoint
21:         upstart = segment end position
22:         l = next line segment of rc in moving direction
23:       end while
24:       upend = x,y-coordinates of gp2
25:       add unit from upstart to upend to mpoint
26:     end if
27:   end if
28: end for
29: return mpoint
```

mgpoint × gpoint → bool   *mgpoint* **passes** *gpoint*
mgpoint × gline → bool   *mgpoint* **passes** *gline*

The operation **passes** returns *TRUE* if a mgpoint value ever 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 $O(r_2 \log r_1)$.

mgpoint × gpoint → mgpoint   *mgpoint* **at** *gpoint*
mgpoint × gline → 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 Algorithm 6 distinguishes two cases. For sorted gline values the execution of the loop from line 2 to 7 needs $O(m \log r)$ time.

Algorithm 6 at (*mgpoint*, *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 the execution of the loops in line 9 to line 14 needs $O(mr)$ time. The result is returned in $O(m_{res})$ time in both cases. In case of sorted $\underline{gline}$ values the total run time is $O(m \log r + m_{res})$ and in case of unsorted $\underline{gline}$ values $O(mr + m_{res})$.

4 Translation of BerlinMOD into NET Representation

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

Executable SECONDO scripts for the network and index creation, object translation, and the executable SECONDO queries for the NET representation of the BerlinMOD Benchmark can be downloaded from our web site [24].

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** operation from the data generated by the BerlinMOD Benchmark.

We use the *streets* object from the BerlinMOD Data Generator to construct our input relation *B_Routes* for the *routes* relation of *net*.

```
B_Routes (rid: int; length: real; curve: line; kind: bool;
start: bo ol)
```

We extract the *geometry* from *streets* as *route curves* and add to each *route curve* a automatic generated integer number as route identifier and compute the length of each *route curve*. The two Boolean values indicating if the route is dual and starts at the smaller end point are set to *TRUE* by default, because the data source lacks information for this facts.

If the *streets* object contains R routes and a route curve of a route r_i has l_i line segments this operation has a time complexity of

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

In the next step we use *B_Routes* to compute the crossings of the street network of Berlin. Therefore we join all *route curves* of *B_Routes* with intersecting spatial bounding boxes and filter the *route curves* that really intersect. For this pairs of *route curves* we compute the positions of the junctions on the *route curves* and fill the resulting data in *B_Junctions* relation.

```
B_Junctions (r1id: int; r1meas: real; r2id: int;
r2meas: real; cc: int)
```

The connectivity code *cc* should tell us 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.

In the worst case this step has a time complexity of $O(R^2)$, but the worst case should never happen in real street network environments

Now we can use *B_Routes* and *B_Junctions* as input relations for the operation **thenetwork** to create our network object *net* representing the street network of Berlin in the NET representation of the BerlinMOD Benchmark.

Algorithm 7 describes how *net* is created from the two input relations and an unifique integer used as network identifier *nid* for *net*. 3.1. Let J be the

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 empty network object *net* with id *nid*
2: Copy *B_Routes* to *routes* relation of *net*
3: Construct B-Tree indexing *rid* in *routes* relation
4: Construct R-Tree indexing *c* in *routes* relation
5: Copy *B_Junctions* to *junctions* relation of *net* and add route tuple identifiers from *routes* relation ($r1rc$ and $r2rc$)
6: Construct two B-Trees indexing the $r1id$ resp. $r2id$ 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 tuple identifiers to the *junctions* relation
12:   **end for**
13: **end for**
14: Construct B-Tree indexing *rid* in the *sections* relation
15: **for** Each junction in *junctions* relation **do**
16:   Find pairs of adjacent sections and fill adjacency lists
17: **end for**

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

For the single steps of Algorithm 7 we get the following time complexities:

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

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

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 *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 *network* object *net* as input. They return the corresponding data type from the network data model NET, respectively an undefined NET object $\perp$ if the input data object is not constrained by *net*.

4.2.1 Translate *point* into *gpoint*

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

The **point2gpoint** operation translates a *point* value p into a corresponding *gpoint* value *gp* if possible. 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. Algorithm 8 describes the operation

Algorithm 8 point2gpoint(p, *net*)

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

The time complexity of the operations in line 1 and line 2 of Algorithm 8 is O(1). *Candidate routes* are routes, which have spatial minimum bounding boxes that intersect with the spatial bounding box of the *point* value. If c is the number of *candidate routes* for a point p the selection of *candidate routes* from the R-tree has a time complexity of $\mathrm{O}(\log R + c)$. 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 line 6 to line 8 takes $\mathrm{O}(l_i)$ 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**.

This should be all to translate the *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 *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 *mgpoint* object passes the junction on route b in all cases, because the definition of **passes** in NET is slightly different from the **passes** operation in SPACE. 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 *gpoint* value gp a stream of *gpoint* values.

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

If gp represents a junction we return all *gpoint* values representing the same junction in *net*, otherwise we return only gp in the stream. Algorithm 9 describes the **point2gpoint** operation in detail.

Algorithm 9 polygpoints(gp)

```
Copy gp to output stream
Use B-Tree on junctions relation to get first junction on with gp.rid
while gp.d ≤ junction position on route do
   if gp.d = junction position on route then
      Copy other junction gpoint into output stream
   end if
   Get next junction with gp.rid
end while
```

The worst case time complexity of the **point2gpoint** operation is $\mathrm{O}(\log J + j_i)$ if j_i is the number of junctions on the route gp belongs to.

In the end we got 221 query *gpoint* values in *QueryPointsNet* for the 100 query *point* values in *QueryPoints* and 22 *gpoint* values in *QueryPoints1Net* for the 10 *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 compared to the SPACE representation.

4.2.2 Translate *mpoint* into *mgpoint*

mpoint × *network* → *mgpoint* **mpoint2mgpoint**(*net*, *mpoint*)

The second operation **mpoint2mgpoint** is described in Algorithm 10. The operation translates an *mpoint* value *mp* into an *mgpoint* value *mgp*. The main idea of Algorithm 10 is to use the continuous movement of *mp* to reduce computation time. We need the route search with the B-Tree of the *routes* relation of **point2gpoint** operation only for the first unit of the *mpoint* value $O(\log R + \sum_{i=1}^{c} l_i)$. 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 single steps of Algorithm 10 of the **mpoint2mgpoint** operation have the following time complexities:

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

Initialize empty *mgp*
upoint = first unit of *mp*
Initialize *ugp* = *net* values of *upoint* {Uses variant of **point2gpoint** for two point on same route}
for Each *upoint* of *mp* **do**
  if *p*2 of *upoint* is on same route than *ugpoint* **then**
    if Direction and speed stay the same **then**
      Extend *ugpoint* to include *upoint*
    else
      Add *ugpoint* to *mgp*
      Add *route interval* of *ugpoint* to *RITree*
      ugpoint = *net* values of *upoint*
    end if
  else
    Add *ugpoint* to *mgp*
    Add *route interval* of *ugpoint* to *RITree*
    Search *upoint* on adjacent sections route curves
    Change current route curve to route curve where the *upoint* has been found
    ugpoint = *net* values of *upoint*
  end if
end for
Add *ugpoint* to *mgpoint*
Add *route interval* of *ugpoint* to *RITree*
Build *trajectory* from *RITree*
Copy bounding box of *mp* to *mgp*
return *mgp*

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

line 1 + 2: O(1)

line 3: $O(\log R + \sum_{i=1}^{c} l_i)$

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

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

$$
\begin{aligned}
&O(\log R + \sum_{i=1}^{c} l_i + m_{in}(l_{1i} + \log r + \sum_{i=1}^{s_i} l_{2i} + l_{1i}) + r + \log r + m_{res}) \\
&= O(\log R + m_{in} \log r + m_{in} \sum_{i=1}^{s_i} l_{2i}) + m_{res})
\end{aligned}
$$

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 $\mathrm{O}(\sum_{i=1}^{R} l_i \log \sum_{i=1}^{R} l_i)$. The operation **line2gline** in line 4 uses Algorithm 12 and has therefore a time complexity of

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

Algorithm 12 line2gline(l, net)

for Each line segment l_i of l **do**
  Use variant of **point2gpoint** to find *route curve* including l_i
  Insert corresponding *route interval* into *RITree*
end for
return sorted and compressed *gline* value from *RITree*

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

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

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 more 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.

The B-Tree indexes for the *licence* 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.

Table 3: Indexes on NET Representation of BerlinMOD Benchmark

Name of Index	**Explanation**
dataSNcar_licence_btree	B-Tree on *licence* in *dataSNcar*
dataMcar_licence_btree	B-Tree on *licence* in *dataMcar*
dataMcar_Moid_btree	B-Tree on *moid* in *dataMcar*
dataMNtrip_Moid_btree	B-Tree on *moid* in *dataMNtrip*
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*
dataMNtrip_SpatioTemp	R-Tree on the spatio-temporal bounding box of *trip* in *dataMNtrip*

The R-Tree index of the spatio-temporal bounding boxes of the *trip* attribute in the *dataMNtrip* relation is 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 trip is inserted in the index, whereas in SPACE representation a bounding box for each single unit is inserted in the index. As mentioned before (see Section 3.1) the computation of the unit bounding boxes in NET representation is very expensive. The building of an spatio-temporal unit index for NET analogous to the SPACE index takes up to several days at higher *scalefactors* and the improvement of query run time is nearly not detectable. Such that we decided to ommit the creation and usage of the more detailed index in our experiments.

4.4 Translate Benchmark Queries

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

In Appendix A the resulting executable SECONDO queries for NET are given in detail. As mentioned before, the executable SECONDO scripts with the queries can be downloaded from our web site [24]. In this Section we give a short overview of the query algorithms used for the NET representation of the BerlinMOD Benchmark.

Every time we need a licence in the result or have a query licence number we need an additional step in the TBA queries to connect the *dataMNtrip* and *dataMcar* relation using the *moid* attribute and the corresponding B-Tree indexes *dataMNtrip_Moid_btree* respectively *dataMcar_Moid_btree*. 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 NET representation of the BerlinMOD Benchmark.

Query 3 searches for the positions of the ten cars from *QueryLicences1* at the ten time instants from *QueryInstants1*. 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 *QueryInstants1*, 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 NBB for each gpoint in *QueryPointsNet* and use our specialised NPI 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 and run times comparable.

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

Query 6 asks for the pairs of licences from "trucks" that have been as close as 10m or less to each other. We select the "trucks" from *dataSNcar* relation, respectively *dataMcar* relation, extend their spatio-temporal bounding boxes in all spatial dimensions by 5m, and retranslate the mgpoint values into mpoint values in a first step. In a second step, we 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 and 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 *QueryPointsNet* first of all "passenger" cars during the observation period. We select all "passenger" cars that passed one of the points from *QueryPointsNet* before 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 selected.

Query 8 computes the overall travelled distances of the vehicles with licences from *QueryLicences1* within the periods from *QueryPeriods1*. We select the candidate cars using the licence B-Tree, restrict the trips to the query periods, and return the driven distance of the trips in the OBA. In the TBA we have to sum up the driven distances of the different trips driven by the same car within each query period.

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

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

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

Query 11 asks for the vehicles that passed a point from *QueryPoints1Net* at one of the time instants from *QueryInstants1*. We build TNBB as query box from the *QueryInstants1* and *QueryPoints1Net* relation and use the TNPI 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 the TBA possible duplicate licence pairs are removed and the resulting *moid*s are 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 TNBB from the query objects to select candidate trips using the TNPI. 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*. Almost similar to query 14 we select the candidate trips building TNBB of the query parameters and use TNPI to select candidate trips. We refine the result selecting all candidate trips really fullfilling the query constraints.

Query 16 asks for the licence pairs of vehicles one from *QueryLicences1* and one from *QueryLicences2*, which were both present in one of the regions from *QueryRegions1Net* within a period from *QueryPeriods1*, but did not meet there and then. We select the candidate trips using the licence B-Tree index and restrict the resulting trips to be **present** during the query periods and move **inside** the query regions. 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 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 (points) 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.

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 [24]. We compiled the SECONDO sources with activated optimization flag "O3 -marche=native" in the `makfile.options` file. In the file "SecondoConfig.ini" we set the `MaxMemPerOperator` parameter to 65536.

With this setup 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. For each of this three databases we called the "CreateObjects" scripts for SPACE and NET representation.

Table 4 shows the created amounts of data for the different *scalefactor* values in both data models. As you can see, NET 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 NET compared to SPACE. This is a very good result and we expect this effect to increase if the cars make long distance trips instead of moving in a single town like they do in the benchmark. In towns cars more often change the street or the velocity than cars that do long distance trips and so the compact route representation of NET should become more effective than for cars driving in towns. The long creation time of the NET representation is caused by the expensive mapping of spatial and spatio-temporal positions into network positions. The indexes themselves are built faster in NET representation than in SPACE representation because they have less entries.

We found some isolated mismatches in some query results as we compared the results of the SPACE queries and the NET 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

	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	
	SPACE	NET	SPACE	NET	SPACE	NET
Data Translation and Index Build	208.74s	515.49s	1,047.03s	2,164.45s	6122.23s	11,027.7s
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

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.

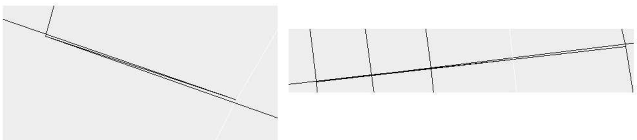

Figure 1: Example Failures in Street Map

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 Figure 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 NET is around 50% less than the total query run time of SPACE at each *scalefactor*.

For the queries 1 and 2, the query run times are almost the same for all data models and approaches at the different *scalefactors*. These results are expected, because both queries deal only with standard attributes and standard indexes, which are not influenced by the different data models.

Although the query algorithms for both data models and approaches are almost the same, SPACE outperforms NET for query 3 by tenths of a second. On the one hand, the number of units in NET is less than the number of units in SPACE, such that the unit which contains the query time instant should be found faster. But 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 each unit in NET has more elements than in SPACE, therefore access and copying

	Scalefactor 0.05			
	SPACE		NET	
Query	**OBA**	**TBA**	**OBA**	**TBA**
1	0.160	0.109	0.197	0.097
2	0.003	0.002	0.003	0.002
3	0.418	0.279	0.462	0.595
4	9.461	11.709	0.338	1.508
5	1.087	1.632	1.760	1.238
6	13.391	6.461	5.933	4.016
7	3.568	3.213	5.670	7.024
8	0.324	0.351	0.218	0.208
9	88.124	188.896	21.618	19.758
10	101.294	31.166	63.217	79.528
11	0.152	0.104	1.044	1.018
12	0.285	0.109	0.203	0.177
13	9.952	6.606	1.135	1.141
14	0.505	0.625	1.513	3.747
15	1.036	0.783	0.599	0.501
16	5.768	16.744	0.580	1.501
17	1.144	1.180	0.208	0.292
Total	236.670	269.970	104.698	122.352

	Scalefactor 0.2			
	SPACE		NET	
Query	**OBA**	**TBA**	**OBA**	**TBA**
1	0.122	0.101	0.166	0.113
2	0.003	0.003	0.004	0.003
3	0.468	0.318	0.584	0.817
4	32.742	39.531	0.604	8.663
5	1.649	3.033	2.990	2.434
6	66.301	48.417	17.046	14.982
7	18.444	11.435	25.204	31.675
8	0.451	0.358	0.229	0.252
9	246.767	373.958	33.143	44.976
10	487.731	136.743	286.764	294.473
11	0.244	0.153	2.956	3.136
12	4.480	0.140	0.260	0.261
13	30.072	15.216	5.314	4.646
14	1.124	1.168	3.325	9.655
15	8.617	4.281	2.399	1.985
16	6.967	28.592	0.365	0.824
17	8.703	6.718	0.333	0.734
Total	914.886	670.165	381,685	419.631

	Scalefactor 1.0			
	SPACE		NET	
Query	**OBA**	**TBA**	**OBA**	**TBA**
1	0.185	0.196	0.302	0.205
2	0.005	0.004	0.006	0.004
3	0.948	0.540	1.207	1.418
4	199.990	159.760	1.207	33.035
5	3.326	6.347	5.851	5.675
6	1295.508	2099.649	300.661	235.301
7	108.222	44.631	110.542	123.349
8	0.840	0.502	0.240	0.280
9	795.955	1887.512	108.343	138.995
10	3564.303	2099.858	2358.162	1661.170
11	0.737	0.650	7.804	8.702
12	39.648	0.216	0.713	0.289
13	118.512	78.304	27.203	34.018
14	11.494	4.990	9.423	39.700
15	29.831	17.227	10.140	6.304
16	9.802	60.795	0.517	1.919
17	84.209	162.730	0.554	5.829
Total	6263.514	6623.912	2942.663	2295.765

Figure 2: Compare Query Run Times in Seconds

needs a little more time in NET than in SPACE.

In query 4 NET outperforms SPACE significantly at all *scalefactors* (> 3 min OBA, > 2 min TBA at *scalefactor* 1.0). The NET index used in query 4 is much smaller (OBA 24 MB, TBA 160 MB, at *scalefactor* 1.0) than the spatial unit index of SPACE (OBA 3.7 GB, TBA 3.7 GB at *scalefactor* 1.0) and more precise, such that we do not need an additional expensive refinement step after the index usage in NET, like we do in SPACE.

We expected NET to be slower than SPACE in the queries 5, 6, and 10, because we retranslate intermediate results from NET representation into SPACE representation. For query 5 this holds in the OBA. We need a little more time in NET than in SPACE. But in TBA NET outperforms SPACE. This is due to the fact that a <u>*gline*</u> value has less *route intervals* than a <u>*line*</u> value has line segments, such that the union of <u>*gline*</u> values in the aggregate step of query 5 in TBA can be computed much faster in NET than the union of <u>*line*</u> values in SPACE.

NET outperforms SPACE again significantly at query 6 for all amounts of data and approaches (> 15 minutes for OBA, > 30 minutes for TBA, at *scalefactor* 1.0). In NET we reduce the number of candidate pairs for the distance computation by some filter steps, while in SPACE in the OBA no filtering is used. NET uses the operation **everNearerThan**, which stops computation immediately if the distance between two units is less than the query value. Whereas the operation **distance** used by SPACE in OBA always checks all units of the query objects. For the TBA SPACE uses **everNearerThan** in-

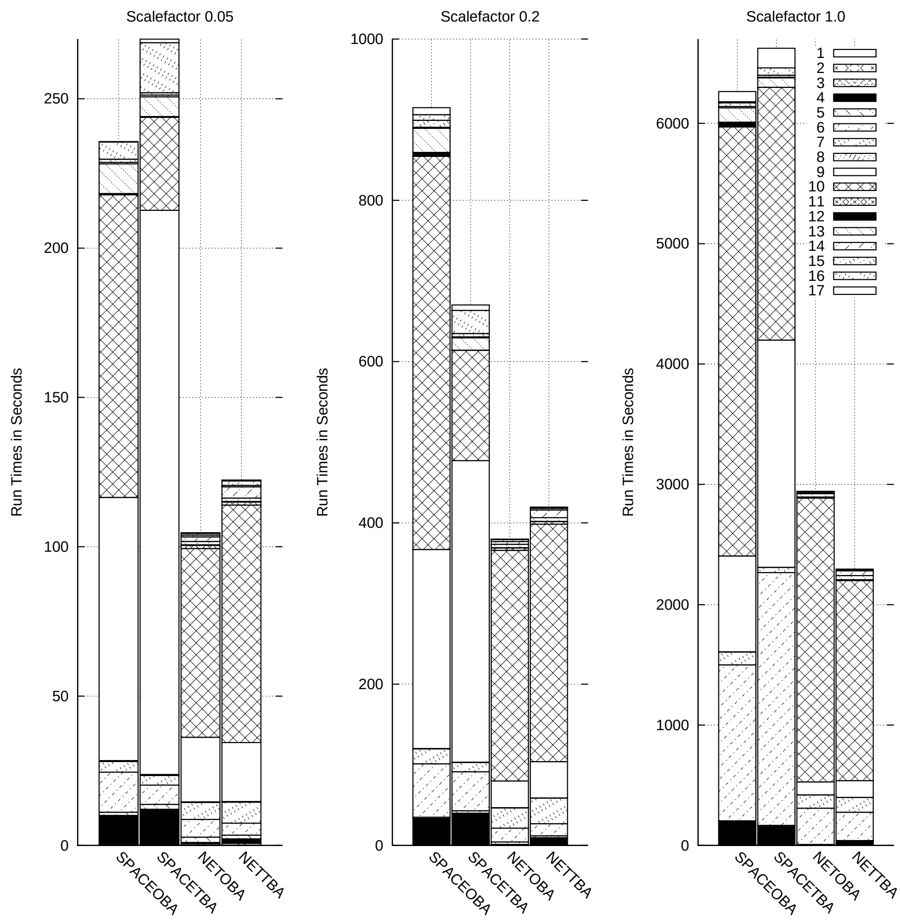

Figure 3: Compare Total Run Times

stead of **distance** and a filter step before. But, whereas NET uses in TBA a spatio-temporal filtering, SPACE uses only a spatial filtering, such that many false candidates are included in the expensive distance computation part of the algorithm.

After the very good results from query 4 we did not expect SPACE to outperform NET at query 7. The weakness of NET is that we have to do the expensive **at** operation for the double number of query $\underline{gpoint}$ compared with SPACE (see Section 4.2). But in the end, NET looses at *scalefactor* 1.0 around 2 seconds in OBA and 80 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 NET is more than 30% less than the query run time of SPACE. This is caused by the *length* attribute of the $\underline{mgpoint}$ and the smaller number of units of a $\underline{mgpoint}$ compared with the corresponding $\underline{mpoint}$.

For query 9, NET outperforms SPACE by orders of magnitude. The advantages named in the analysis of query 8 have a much more impact when the number of examined trips becomes higher. At *scalefactor* 1.0 this saves more than 10 minutes time in the OBA and more than 25 minutes 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 NET outperforms SPACE at all *scalefactors* and at *scalefactor* 1.0 in TBA, whereas SPACE outperforms NET for the lower amounts of data in TBA significantly. Before our experiments we expected that SPACE would outperform NET in all cases, because of the expensive retranslation of intermediate results. So why is NET 20 minutes faster at *scalefactor* 1.0 in OBA than SPACE? In NET we use bounding boxes to preselect candidate trips that step is not performed in SPACE. The disadvantage of NET in TBA becomes smaller at higher *scalefactors* we think this is due to the fact that the number of units in $\underline{mgpoint}$ values is always smaller than in $\underline{mpoint}$ values such that the final aggregation of the different trips of the same cars can be done faster in NET than SPACE.

In our experiments we also tested the MON-Tree [8] as spatio-temporal index. But, although the CPU run time was very small, the elapsed run time performance was very bad. Such that the primitive index outperformed the MON-Tree in all cases.

Query 11 is identical with the first part of query 12. So it is surprising that the run time of query 11 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 of cache effects resulting from query 11 running immediately before query 12. Another weakness of NET pointed out by the run times of query 11 and some other queries is that our TNPI has bad run times for query TNBB created from a single $\underline{gpoint}$ 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 TNBB constructed from a single $\underline{gpoint}$ and a time interval. We have to spend some more work to figure out the problem and develop a better TNPI to improve our NET implementation.

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

The NET version of query 16 takes profit from the smaller number of units in NET and outperforms SPACE significantly.

Although we detected in our experiments some points of weakness in the temporal network position index (TNPI), NET outperforms SPACE by orders of magnitude. The weakness of NET almost occurs in queries with short run times, whereas the advantages of NET become apparent in the queries with long run times, such that the weakness of the TNPI in some cases is covered by the advantages of NET in the other cases.

7 Summary and Future Work

We presented our translation of the BerlinMOD Benchmark into the network constrained data model NET and compared the capabilities of both data models, with very good results for NET. Our experiments show that NET outperforms SPACE 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 $\underline{mgpoint}$ value compared with the number of units of the corresponding $\underline{mpoint}$, which also results in smaller indexes for NET objects. The BerlinMOD Benchmark of NET pointed out that we should spend time in the improvement of the TNPI.

The good results of NET encourages to spend further work in the network representation of network constrained objects.

We want to extend the BerlinMOD Benchmark, with an additional query set, covering the special challenges of spatio-temporal network constrained databases like shortest path computing. The new query set should enable us to compare the capabilites of database systems dealing with spatio-temporal NCDM.

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

An interesting topic for future work on NCDM is the efficient computation of dynamic network distances between moving objects in the network.

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*, pages 287–296. IEEE Computer Society Press, Washington, 2004.

[12] Z. Ding and R.H. Güting. Modeling temporally variable transportation networks. In Lee D. Lee YJ, Li J. Whang KY, editor, *Proceedings of the 9th Intern. Conf. on Database Systems for Advanced Applications*, pages 154–168. Springer Berlin / Heidelberg, 2004.

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

[14] 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, 1993.

[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 Geographical Information Systems*, 4(2), 1990.

[26] 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.

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

[28] 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`.

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

[30] 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.

[31] N. Pelekis. *STAU: A Spatio-Temporal Extension to ORACLE DBMS*. PhD Thesis, University of Manchester Institute of Science and Technology, 2002.

[32] N. Pelekis and Y. Theodoridis. An Oracle Cartridge for Moving Objects. Technical report, December 2007. http://infolab.cs.unipi.gr/pubs/TR/UNIPI-ISL-TR-2007-04.pdf.

[33] 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.

[34] N. Pelekis, B. Theodoulidis, I. Kopanakis, and Y. Theodoridis. Literature review of spatio-temporal database models. *The Knowledge Engineering Review*, (4):235–274, 2004.

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

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

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

[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 NET 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 if 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 Q5h1OBA =
    QueryLicences1 feed {l1}
    loopsel [dataSNcar_Licence_btree dataSNcar exactmatch [.Licence_l1]
        projectextend [Licence; TrajLine: gline2line(trajectory(.Trip))]]
consume;
let Q5h2OBA =
    QueryLicences2 feed {l2}
    loopsel [dataSNcar_Licence_btree dataSNcar exactmatch [.Licence_l2]
        projectextend [Licence; TrajLine: gline2line(trajectory(.Trip))]]
consume;
let Q5OBA =
    Q5h1OBA feed {c1}
    Q5h2OBA feed {c2}
    product
    projectextend [Licence_c1, Licence_c2; Distance: distance(.TrajLine_c1, .TrajLine_c2)]
consume;
delete Q5h1OBA;
delete Q5h2OBA;

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;


oplet Q7hOBA =
  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 [Licence; Id: attr(t, Id), Instant: inst(initial(.Trip at attr(t, Pos)))]]
  filter [not(isempty(.Instant))]
  sortby [Id asc, Instant asc]
consume;
let Q7cOBA =
  Q7hOBA feed groupby [Id; FirstTime: group feed min [Instant]] {b}
  Q7hOBA feed {a}
  symmjoin [..Id_a = .Id_b]
  filter [.Instant_a ≤ .FirstTime_b]
  project [Id_a, Licence_a]
consume;
delete Q7hOBA;
```

```
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 Q7aTBA =
    Q7hTBA feed groupby [Id; FirstTime: group feed min [TimeAtPos]] {b}
    Q7hTBA feed {a}
    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;

let Q8OBA =
    QueryLicences1 feed {l}
        loopsel [dataSNcar_Licence_btree dataSNcar exactmatch [.Licence_l]]
    QueryPeriods1 feed filter[not(isempty(.Period))] {p}
    product
    projectextend [Licence, Period_p; Distance: length(.Trip atperiods .Period_p)]
consume;

let Q8TBA =
    QueryLicences1 feed {l}
        loopjoin [dataMcar_Licence_btree dataMcar exactmatch [.Licence_l]]
        project [Licence, Moid]
        loopsel [fun (t:TUPLE) dataMNtrip_Moid_btree dataMNtrip exactmatch [attr(t, Moid)]
            projectextend[Trip; Licence: attr(t, Licence)]]
    QueryPeriods1 feed
    symmjoin [.Trip present ..Period]
    projectextend [Licence, Period, Id; Distance: length(.Trip atperiods .Period)]
    sortby [Id asc, Licence asc, Distance desc]
    groupby [Id, Period, Licence; Dist: group feed sum [Distance]]
    project [Licence, Period, Dist]
consume;

let Q9OBA =
    dataSNcar feed {c}
    QueryPeriods feed filter [not(isempty(.Period))] {p}
    product
    projectextend [Id_p, Period_p, Licence_c; Dist: length(.Trip_c atperiods .Period_p)]
    sortby [Id_p asc, Period_p asc, Dist desc]
    groupby [Id_p, Period_p; Distance: group feed max [Dist]]
    project [Id_p, Period_p, Distance]
    sortby [Id_p asc]
    project [Period_p, Distance]
consume;

let Q9TBA =
    dataMNtrip feed {c}
    QueryPeriods feed filter [not(isempty(.Period))] {p}
    symmjoin[.Trip_c present ..Period_p]
    projectextend [Moid_c, Period_p, Id_p; Distance: length(.Trip_c atperiods .Period_p)]
    sortby [Id_p asc, Moid_c asc, Distance desc]
    groupby [Id_p, Period_p, Moid_c; Dist: group feed sum [Distance]]
    groupby [Id_p, Period_p; Dista: group feed max [Dist]]
    project [Period_p, Dista]
consume;

let Q10OBA =
    dataSNcar feed
        projectextend[Licence; TripA: mgpoint2mpoint(.Trip), BBox: mgpbbox(.Trip)]
        projectextend[Licence, TripA; Box: rectangle2((minD(.BBox, 1) - 1.5), (maxD(.BBox, 1) + 1.5),
                (minD(.BBox, 2) - 1.5), (maxD(.BBox, 2) + 1.5))] {c1}
    QueryLicences1 feed
        loopsel [dataSNcar_Licence_btree dataSNcar exactmatch [.Licence]]
        projectextend [Licence, Trip; BBox: mgpbbox(.Trip)]
```

```
    projectextend [Licence, Trip; TripA: mgpoint2mpoint(.Trip), Box: rectangle2(minD(.BBox,1) - 1.5),
          (maxD(.BBox, 1) + 1.5), (minD(.BBox, 2) - 1.5), (maxD(.BBox, 2) + 1.5))] {c2}
  symmjoin [.Box_c1 intersects ..Box_c2]
  filter [.Licence_c1 ≠ .Licence_c2]
  filter [everNearerThan(.TripA_c1, .TripA_c2, 3.0)]
  projectextend [Licence_c1, Licence_c2;
          Pos: .Trip_c2 atperiods deftime((distance(.TripA_c1, .TripA_c2) < 3.0) at TRUE]
  filter [not(isempty(.Pos))]
  project [Licence_c2, Licence_c1, Pos]
  sortby [Licence_c2 asc, Licence_c1 asc]
consume;

let Q10TBA =
  QueryLicences1 feed project [Licence] {V1}
  loopsel [fun (t:TUPLE) dataMcar_Licence_btree dataMcar exactmatch [attr(t, Licence_V1)]
    project [Moid]
    loopjoin [dataMNtrip_Moid_btree dataMNtrip exactmatch [.Moid] remove[Moid]] {V3}
    extend [t3bbx: mgpbbox(.Trip_V3)]
    extend [ptripA: mgpoint2mpoint(.Trip_V3)]
    loopjoin [fun (u:TUPLE) dataMNtrip_SpatioTemp windowintersectsS[rectangle3(
          minD(attr(u, t3bbx), 1) - 3.0, maxD(attr(u, t3bbx), 1) + 3.0,
          minD(attr(u, t3bbx), 2) - 3.0, maxD(attr(u, t3bbx), 2) + 3.0,
          minD(attr(u, t3bbx), 3), maxD(attr(u, t3bbx), 3))]
      sort rdup dataMNtrip gettuples
      filter [.Moid ≠ attr(u, Moid_V3)]
      projectextend [Moid; ptripB: mgpoint2mpoint(.Trip)]
      filter [everNearerThan(attr(u, ptripA), .ptripB, 3.0)]
      projectextend [Moid; Times: deftime((distance(attr(u, ptripA), .ptripB) < 3.0) at TRUE)]
      filter [not(isempty(.Times))]
      loopjoin [dataMcar_Moid_btree dataMcar exactmatch [.Moid] project [Licence]]]
    projectextend [Times, Trip_V3; QueryLicence: attr(t, Licence_V1), OtherLicence: .Licence]
    projectextend [QueryLicence, OtherLicence; Pos: .Trip_V3 atperiods .Times]
    filter [not(isempty(.Pos))]]
  sortby [QueryLicence asc, OtherLicence asc]
  groupby [QueryLicence, OtherLicence; AllPos: group feed aggregateB[Pos;
        fun (M1:mgpoint, M2:mgpoint) M1 union M2; [const mgpoint value()]]]
  project [QueryLicence, OtherLicence, AllPos]
consume;

let Q11bOBA =
  QueryInstant1 feed {i}
  QueryPoints1Net feed projectextend [Id, Pos; Prect: gpoint2rect(.Pos)] {p}
  product
  projectextend [Id_p, Instant_i; Box: box3d(.Prect_p, .Instant_i)]
  loopsel [fun(t:TUPLE) dataSNcar_BoxNet_timespace windowintersectsS [attr(t, Box)]
    sort rdup dataSNcar gettuples
    projectextend [Licence; Id: attr(t, Id_p), Instant_i: attr(t, Instant_i)]]
consume;

let Q11TBA =
  QueryInstant1 feed {i}
  QueryPoints1Net feed projectextend [Id, Pos; Prect: gpoint2rect(.Pos)] {p}
  product
  loopsel [fun (t:TUPLE) dataMNtrip_BoxNet_timespace windowintersectsS[
        box3d(attr(t, Prect_p), attr(t, Instant_i))]
    sort rdup dataMNtrip gettuples
    projectextend [Moid; Id: attr(t, Id_p), Instant: attr(t, Instant_i)]] {a}
  loopjoin [dataMcar_Moid_btree dataMcar exactmatch [.Moid_a]]
  project [Id_a, Instant_a, Licence]
  sortby [Id_a asc, Instant_a asc, Licence asc]
  krdup [Id_a, Instant_a, Licence]
consume;

let Q12hOBA =
  QueryInstant1 feed {i}
  QueryPoints1Net feed projectextend [Id, Pos; Prect: gpoint2rect(.Pos)] {p}
  product
  loopsel [fun (t:TUPLE) dataSNcar_BoxNet_timespace windowintersectsS[
        box3d(attr(t, Prect_p), attr(t, Instant_i))]
    sort rdup dataSNcar gettuples
    projectextend [Licence; Id_p: attr(t, Id_p), Pos_p: attr(t, Pos_p), Instant_i: attr(t, Instant_i)]]
  sortby [Id_p asc, Instant_i asc, Licence asc]
consume;
let Q12OBA =
```

```
    Q12hOBA feed {c1}
    Q12hOBA feed {c2}
    symmjoin [(.Licence_c1 < ..Licence_c2) and (.Id_p_c1 = ..Id_p_c2) and (.Instant_i_c1 = ..Instant_i_c2)]
    project [Id_p_c1, Pos_p_c1, Instant_i_c1, Licence_c1, Licence_c2]
    sortby [Id_p_c1 asc, Instant_i_c1 asc, Licence_c2 asc]
consume;
delete Q12hOBA;

let Q12hTBA =
    QueryPoints1Net feed projectextend [Id, Pos; Prect: gpoint2rect(.Pos)] {p}
    QueryInstant1 feed {i}
    product
    projectextend [Id_p, Pos_p, Instant_i; Box: box3d(.Prect_p, .Instant_i)]
    loopsel [fun (t:TUPLE) dataMNtrip_BoxNet_timespace windowintersectsS [attr(t, Box)]
        sort rdup dataMNtrip gettuples
        projectextend [Moid; Id: attr(t, Id_p), Instant: attr(t, Instant_i)]] {a}
    loopjoin [dataMcar_Moid_btree dataMcar exactmatch [.Moid_a]]
    projectextend [Moid, Licence; Id: .Id_a, Instant: .Instant_a]
consume;
let Q12TBA =
    Q12hTBA feed {A}
    Q12hTBA feed {B}
    symmjoin [(.Id_A = ..Id_B) and (.Instant_A = ..Instant_B) and (.Moid_A < ..Moid_B)]
    project [Id_A, Instant_A, Licence_A, Licence_B]
    sortby [Id_A asc, Instant_A asc, Licence_B asc]
consume;
delete Q12hTBA;

let Q13OBA =
    dataSNcar feed {c}
    QueryRegions1Net feed filter [not(isempty(.Region))] {r}
    symmjoin [.Trip_c passes ..Region_r]
        projectextend [Licence_c, Id_r, Region_r; Trip: .Trip_c at .Region_r]
    QueryPeriods1 feed filter [not(isempty(.Period))] {p}
    symmjoin [.Trip present ..Period_p]
    projectextend [Id_r, Period_p; Licence: .Licence_c, Trip: .Trip atperiods .Period_p]
    filter [no_components(.Trip) > 0]
    project [Id_r, Period_p, Licence]
    sortby [Id_r asc, Period_p asc, Licence asc]
consume;

let Q13TBA =
    dataMNtrip feed {c}
    QueryRegions1Net feed filter [not(isempty(.Region))] {r}
    symmjoin [.Trip_c passes ..Region_r]
        projectextend [Moid_c, Id_r; Trip: .Trip_c at .Region_r]
    QueryPeriods1 feed filter [not(isempty(.Period))] {p}
    symmjoin [.Trip present ..Period_p]
    loopjoin [dataMcar_Moid_btree dataMcar exactmatch [.Moid_c]]
    project [Licence, Id_r, Period_p]
    sortby [Id_r asc, Period_p asc, Licence asc]
    krdup [Id_r, Period_p, Licence]
consume;

let Q14aOBA =
    dataSNcar feed
    QueryInstant1 feed
    product
        projectextend [Licence, Instant; PosX: val(.Trip atinstant .Instant)]
        projectextendstream [Licence, Instant; Pos: polygpoints(.PosX, B_NETWORK)]
    QueryRegions1Net feed filter [not(isempty(.Region))]
    symmjoin [.Pos inside ..Region]
    project [Id, Instant, Licence]
    sortby [Id asc, Instant asc, Licence asc]
    krdup[Id, Instant, Licence]
consume;

let Q14TBA =
    QueryRegions1Net feed filter [not(isempty(.Region))]
        projectextendstream [Id, Region; Brect: routeintervals(.Region)] {r}
    QueryInstant1 feed {i}
    product
    projectextend [Id_r, Region_r, Instant_i; Box: box3d(.Brect_r, .Instant_i)]
    loopsel [fun (t:TUPLE) dataMNtrip_BoxNet_timespace windowintersectsS [attr(t, Box)]
```

```
        sort rdup dataMNtrip gettuples
        filter [(val(.Trip atinstant (attr(t, Instant_i)))) inside (attr(t, Region_r))]
        projectextend [Moid;Instant: attr(t, Instant_i), Id: attr(t, Id_r)]] {a}
    loopjoin [dataMcar_Moid_btree dataMcar exactmatch [.Moid_a]]
    projectextend[Licence; Id: .Id_a, Instant: .Instant_a]
    sortby [Id asc, Instant asc, Licence asc]
    krdup[Id, Instant, Licence]
consume;

let Q15OBA =
    QueryPoints1Net feed projectextend[Id, Pos; Prect: gpoint2rect(.Pos)] {p}
    QueryPeriods1 feed filter[not(isempty(.Period))] {t}
    product
    projectextend[Id_p, Pos_p, Period_t; Box: box3d(.Prect_p, .Period_t)]
    loopsel [fun(t:TUPLE) dataSNcar_BoxNet_timespace windowintersectsS[attr(t, Box)]
        sort rdup dataSNcar gettuples
        filter [(.Trip atperiods (attr(t, Period_t))) passes (attr(t ,Pos_p))]
        projectextend [; Id: attr(t, Id_p), Period: attr(t, Period_t), Licence: .Licence]]
    sortby [Id asc, Period asc, Licence asc]
    krdup [Id, Period, Licence]
consume;

let Q15TBA =
    QueryPoints1Net feed projectextend [Id, Pos; Prect: gpoint2rect(.Pos)] {p}
    QueryPeriods1 feed filter[not(isempty(.Period))] {t}
    product
    loopsel[fun (t:TUPLE) dataMNtrip_BoxNet_timespace windowintersectsS[
            box3d(attr(t, Prect_p), attr(t, Period_t))]
        sort rdup dataMNtrip gettuples
        filter [(.Trip atperiods (attr(t, Period_t))) passes (attr(t, Pos_p))]
        projectextend [Moid;Period: attr(t, Period_t), Id: attr(t, Id_p)]] {a}
    loopjoin [dataMcar_Moid_btree dataMcar exactmatch [.Moid_a]]
    projectextend [Licence; Id: .Id_a, Period: .Period_a]
    sortby [Id asc, Period asc, Licence asc]
    krdup [Id, Period, Licence]
    project [Licence, Id, Period]
consume;

let Q16OBA =
    QueryLicences1 feed {l}
            loopjoin [dataSNcar_Licence_btree dataSNcar exactmatch [.Licence_l]] {c}
        QueryPeriods1 feed filter[not(isempty(.Period))] {p}
        symmjoin [.Trip_c present ..Period_p]
            projectextend [Id_p, Period_p; Licence: .Licence_c, Trip: .Trip_c atperiods .Period_p]
            filter [no_components(.Trip) > 0]
        QueryRegions1Net feed filter [not(isempty(.Region))] {r}
        symmjoin [.Trip passes ..Region_r]
        projectextend [Licence, Id_r, Region_r, Id_p, Period_p; Trip: .Trip at .Region_r]
        filter [no_components(.Trip) > 0] {a}
    QueryLicences2 feed {l}
            loopjoin [dataSNcar_Licence_btree dataSNcar exactmatch [.Licence_l]] {c}
        QueryPeriods1 feed filter [not(isempty(.Period))]{p}
        symmjoin [.Trip_c present ..Period_p]
            projectextend [Id_p, Period_p; Licence: .Licence_c, Trip: .Trip_c atperiods .Period_p]
            filter [no_components(.Trip) > 0]
        QueryRegions1Net feed filter [not(isempty(.Region))]{r}
        symmjoin[.Trip passes ..Region_r]
        projectextend [Licence, Id_r, Region_r, Id_p, Period_p; Trip: .Trip at .Region_r]
        filter [no_components(.Trip) > 0] {b}
    symmjoin [(.Id_r_a = ..Id_r_b) and (.Id_p_a = ..Id_p_b)]
    filter [.Licence_a ≠ .Licence_b]
    filter [not(.Trip_a intersects .Trip_b)]
    project [Id_r_a, Period_p_a, Licence_a, Licence_b]
consume;

let Q16TBA =
    QueryLicences1 feed {l}
            loopjoin [dataMcar_Licence_btree dataMcar exactmatch[.Licence_l]] {a}
            loopjoin [dataMNtrip_Moid_btree dataMNtrip exactmatch[.Moid_a]]
        QueryPeriods1 feed filter [not(isempty(.Period))] {p}
        symmjoin [.Trip present ..Period_p]
            projectextend [Id_p, Period_p; Licence: .Licence_a, Trip: .Trip atperiods .Period_p]
            filter [no_components(.Trip) > 0]
        QueryRegions1Net feed filter [not(isempty(.Region))] {r}
```

```
    symmjoin [.Trip passes ..Region_r]
      projectextend [Licence, Id_p, Period_p, Id_r; Trip: .Trip at .Region_r]
      filter [no_components (.Trip) > 0] {a}
  QueryLicences2 feed {l}
      loopjoin [dataMcar_Licence_btree dataMcar exactmatch [.Licence_l]] {a}
      loopjoin [dataMNtrip_Moid_btree dataMNtrip exactmatch[.Moid_a]]
    QueryPeriods1 feed filter [not(isempty(.Period))] {p}
    symmjoin [.Trip present ..Period_p]
      projectextend [Id_p, Period_p; Licence: .Licence_a, Trip: .Trip atperiods .Period_p]
      filter [no_components(.Trip) > 0]
    QueryRegions1Net feed filter [not(isempty(.Region))] {r}
    symmjoin [.Trip passes ..Region_r]
      projectextend [Licence, Id_p, Id_r; Trip: .Trip at .Region_r]
      filter [no_components (.Trip) > 0]{b}
  symmjoin [(.Id_r_a = ..Id_r_b) and (.Id_p_a = ..Id_p_b)]
  filter [.Licence_a ≠ .Licence_b]
  filter [not(.Trip_a intersects .Trip_b)]
  project [Id_r_a, Id_p_a, Period_p_a, Licence_a, Licence_b]
  sortby [Id_r_a asc, Id_p_a asc, Licence_a asc, Licence_b asc]
  krdup [Id_r_a, Id_p_a, Licence_a, Licence_b]
consume;

let Q17hOBA =
  dataSNcar feed {c}
  QueryPointsNet feed {p}
  symmjoin [.Trip_c passes ..Pos_p]
  project [Id_p, Licence_c]
  sortby [Id_p, Licence_c]
  krdup [Id_p, Licence_c]
  groupby [Id_p; Hits: group feed count]
consume;
let Q17OBA =
  Q17hOBA feed filter [.Hits = (Q17hOBA feed max[Hits])]
  project [Id_p, Hits]
consume;
delete Q17hOBA;

let Q17hTBA =
  QueryPointsNet feed projectextend [Id; Elem: gpoint2rect(.Pos)]
  loopsel [fun (t:TUPLE) dataMNtrip_TrajBoxNet windowintersectsS [attr(t, Elem)]
    sort rdup dataMNtrip gettuples
    projectextend [Moid; Id_p: attr(t, Id)]]
  sortby [Id_p asc, Moid asc]
  krdup[Id_p, Moid]
  groupby[Id_p; Hits: group feed count]
consume;
let Q17TBA =
  Q17hTBA feed filter [.Hits = (Q17hTBA feed max[Hits])]
  project [Id_p, Hits]
consume;
delete Q17hTBA;
```

B Executable Secondo SPACE Queries

In the sequel we present the executable Secondo queries delivered with the BerlinMOD Benchmark for the SPACE representation in our experiments. The name of the query result object indicates the number of the query, and if it is a query for the object based approach (OBA), or for the trip based approach (TBA).

```
let OBACRres001 =
  QueryLicences feed {O}
  loopjoin [ dataSCcar_Licence_btree dataSCcar exactmatch [.Licence_O]]
  project[Licence, Model]
consume;

let TBACRres001 =
  QueryLicences feed {O}
  loopjoin [dataMCcar_Licence_btree dataMCcar exactmatch [.Licence_O]]
  project [Licence, Model]
```

```
consume;
let OBACRres002 = dataSCcar feed filter [.Type = 'passenger'] count;

let TBACRres002 = dataMCcar feed filter [.Type = 'passenger'] count;

let OBACRres003 =
    QueryLicences feed head [10] {LL}
        loopjoin [dataSCcar_Licence_btree dataSCcar exactmatch [.Licence_LL]]
    QueryInstants feed head [10] {II}
    product
    projectextend [; Licence: .Licence_LL, Instant: .Instant_II,
                Pos: val(.Journey atinstant .Instant_II)]
consume;

let TBACRres003 =
    QueryLicences feed head [10] {LL}
        loopsel [dataMCcar_Licence_btree dataMCcar exactmatch [.Licence_LL]
            project [Licence, Moid] {LL}]
        loopjoin [dataMCtrip_Moid_btree dataMCtrip exactmatch [.Moid_LL]]
    QueryInstants feed head [10] {II}
    symmjoin [.Trip present ..Instant_II]
    projectextend [; Licence: .Licence_LL, Instant: .Instant_II,
                Pos: val(.Trip atinstant .Instant_II)]
    sortby [Instant asc, Licence asc]
consume;

let OBACRres004 =
    QueryPoints feed
    loopjoin [dataSCcar_Journey_sptuni windowintersectsS [bbox(.Pos)]
        sort rdup dataSCcar gettuples]
    filter [.Journey passes .Pos]
    project [Id, Licence]
    sortby [Id asc, Licence asc]
    krdup[Id, Licence]
consume;

let TBACRres004 =
    QueryPoints feed
    loopjoin [dataMCtrip_Trip_sptuni windowintersectsS [bbox(.Pos)]
        sort rdup dataMCtrip gettuples]
    filter [.Trip passes .Pos]
    project [Id, Moid]
    loopsel [fun (t1: TUPLE) dataMCcar_Moid_btree dataMCcar exactmatch [attr(t1, Moid)]
        projectextend [Moid; Id: attr(t1, Id), Licence: .Licence]]
    sortby [Id asc, Licence asc]
    krdup [Id, Licence]
    project [Id, Licence]
consume;

let OBACRres005tmp1 =
    QueryLicences feed head [10]
    loopsel [dataSCcar_Licence_btree dataSCcar exactmatch [.Licence]]
    projectextend [Licence; Traj: simplify(trajectory(.Journey), 0.000001)]
consume;
let OBACRres005tmp2 =
    QueryLicences feed head [20] filter [.Id > 10]
    loopsel [dataSCcar_Licence_btree dataSCcar exactmatch [.Licence]]
    projectextend [Licence; Traj: simplify(trajectory(.Journey), 0.000001)]
consume;
let OBACRres005 =
    OBACRres005tmp1 feed {V1}
    OBACRres005tmp2 feed {V2}
    product
    projectextend [; Licence1: .Licence_V1, Licence2: .Licence_V2, Dist: distance(.Traj_V1, .Traj_V2)]
    sort rdup
consume;
delete OBACRres005tmp1;
delete OBACRres005tmp2;

let TBACRres005Traj1 =
    QueryLicences feed head [10] project [Licence] {LL1}
    loopsel [fun (t:TUPLE) dataMCcar_Licence_btree dataMCcar exactmatch [attr(t, Licence_LL1)] {CAR}
        loopsel[dataMCtrip_Moid_btree dataMCtrip exactmatch[.Moid_CAR]]
        projectextend [; Traj: simplify(trajectory(.Trip), 0.000001)]
```

```
        aggregateB [Traj; fun (L1: line, L2: line) union_new(L1, L2); [const line value ()]]
        feed namedtransformstream [Traj]
        extend [Licence: attr(t, Licence_LL1)]]
consume;
let TBACRres005Traj2 =
    QueryLicences feed head [20] filter [.Id > 10] {LL1}
    loopsel [fun (t:TUPLE) dataMCcar_Licence_btree dataMCcar exactmatch [attr(t, Licence_LL1)] {CAR}
        loopsel [dataMCtrip_Moid_btree dataMCtrip exactmatch [.Moid_CAR]]
        projectextend [; Traj: simplify(trajectory(.Trip), 0.000001)]
        aggregateB [Traj; fun (L1: line, L2: line) union_new(L1, L2); [const line value ()]]
        feed namedtransformstream [Traj]
        extend [Licence: attr(t, Licence_LL1)]]
consume;
let TBACRres005 =
    TBACRres005Traj1 feed {LL1}
    TBACRres005Traj2 feed {LL2}
    product
    projectextend [; Licence1: .Licence_LL1, Licence2: .Licence_LL2, Dist: distance(.Traj_LL1, .Traj_LL2)]
consume;
delete TBACRres005Traj1;
delete TBACRres005Traj2;

let OBACRres006 =
    dataSCcar feed {V1} filter [.Type_V1 = 'truck']
    dataSCcar feed {V2} filter [.Type_V2 = 'truck']
    symmjoin [.Licence_V1 < ..Licence_V2]
    filter [minimum(distance(.Journey_V1, .Journey_V2)) ≤ 10.0 ]
    projectextend [; Licence1: .Licence_V1, Licence2: .Licence_V2 ]
consume;

let TBACRres006BBoxMtrip =
    dataMCcar feed filter [.Type = 'truck']
    project [Licence, Moid]
    loopsel [fun (t: TUPLE) dataMCtrip_Moid_btree dataMCtrip exactmatch [attr(t, Moid)]
        projectextend [Trip, Moid; BBox: bbox(.Trip), Licence: attr(t, Licence)]
        projectextend [Moid, Licence, Trip, BBox; Box: rectangle2((minD(.BBox, 1) - 5.0),
                (maxD(.BBox, 1) + 5.0), (minD(.BBox, 2) - 5.0), (maxD(.BBox, 2) + 5.0))]]
consume;
let TBACRres006 =
    TBACRres006BBoxMtrip feed {C1}
    TBACRres006BBoxMtrip feed {C2}
    spatialjoin [Box_C1, Box_C2]
    filter [.Moid_C1 < .Moid_C2]
    filter [everNearerThan(.Trip_C1, .Trip_C2, 10.0)]
    projectextend [; Licence1: .Licence_C1, Licence2: .Licence_C2]
    sort rdup
consume;
delete TBACRres006BBoxMtrip;

let OBACRres007PointMinInst =
    QueryPoints feed
    loopjoin [dataSCcar_Journey_sptuni windowintersectsS [bbox(.Pos)]
        sort rdup dataSCcar gettuples]
    filter [.Type = 'passenger']
    projectextend [Id, Pos; Instant: inst(initial(.Journey at .Pos))]
    filter [not(isempty(.Instant))]
    sortby [Id asc, Instant asc]
    groupby [Id, Pos; FirstTime: group feed min [Instant]]
consume;
let OBACRres007 =
    OBACRres007PointMinInst feed extend [MBR: box3d(bbox(.Pos), .FirstTime)]
    loopjoin [dataSCcar_Journey_sptmpuni windowintersectsS [.MBR]
        sort rdup dataSCcar gettuples]
    filter [.Type = 'passenger']
    filter [.Journey passes .Pos]
    projectextend [Licence, FirstTime, Id ; Instant: inst(initial(.Journey at .Pos))]
    filter [.Instant ≤ .FirstTime]
    project [Id, Licence]
    sortby [Id asc, Licence asc]
consume;
delete OBACRres007PointMinInst;

let TBACRres007PointMinInst2 =
    QueryPoints feed project [Pos]
```

```
    loopjoin [fun (t:TUPLE) dataMCtrip_Trip_sptuni windowintersectsS [bbox(attr(t, Pos))]
        sort rdup dataMCtrip gettuples
        filter [.Trip passes attr(t, Pos)]
        loopjoin [dataMCcar_Moid_btree dataMCcar exactmatch [.Moid]
            filter [.Type = 'passenger']
            project [Licence] {X}]
        projectextend [; TimeAtPos: inst(initial(.Trip at attr(t, Pos)))]
        min [TimeAtPos]
        feed namedtransformstream [FirstTime]
        filter [not(isempty(.FirstTime))]]
consume;
let TBACRres007 =
    TBACRres007PointMinInst2 feed
    loopjoin [fun (t:TUPLE) dataMCtrip_Trip_sptmpuni windowintersectsS[
                box3d(bbox(attr(t, Pos)), attr(t, FirstTime))]
        sort rdup dataMCtrip gettuples
        filter [val(.Trip atinstant attr(t, FirstTime)) = attr(t, Pos)]
        loopsel [fun (t2: TUPLE) dataMCcar_Moid_btree dataMCcar exactmatch[attr(t2, Moid)]
            filter [.Type = 'passenger']
            project [Licence, Moid]]]
    project [Pos, Licence, Moid ]
    sort rdup
    project [Pos, Licence]
consume;
delete TBACRres007PointMinInst2;

let OBACRres008 =
    QueryLicences feed head [10] {LL}
        loopsel [dataSCcar_Licence_btree dataSCcar exactmatch [.Licence_LL]]
    QueryPeriods feed head [10] {PP}
    product
    projectextend [Licence; Period: .Period_PP, Dist: round(length(.Journey atperiods .Period_PP), 3)]
    project [Licence, Period, Dist]
    sortby [Licence asc, Period asc]
consume;

let TBACRres008 =
    QueryPeriods feed head [10]
    QueryLicences feed head [10] project [Licence]
    product
    loopsel [fun (t:TUPLE) dataMCcar_Licence_btree dataMCcar exactmatch [attr(t, Licence)] {CAR}
        extend[Dist: round(
                dataMCtrip_Moid_btree dataMCtrip exactmatch[.Moid_CAR]
                filter [.Trip present attr(t, Period)]
                projectextend [; L: length(.Trip atperiods attr(t, Period))]
            sum [L], 3)]
        projectextend [; Licence: attr(t, Licence), Period: attr(t, Period), Dist: .Dist]]
consume;

let OBACRres009 =
    dataSCcar feed project [Journey] {V1}
    QueryPeriods feed {PP}
    product
    projectextend [Id_PP; Period: .Period_PP, D: length(.Journey_V1 atperiods .Period_PP)]
    sortby [Id_PP, Period, D desc]
    groupby [Id_PP, Period; Dist: round group feed max [D], 3) ]
    project [Period, Dist]
consume;

let TBACRres009 =
    QueryPeriods feed
    extend [PeriodBox: queryrect2d(minimum(.Period)) union queryrect2d(maximum(.Period))]
    loopjoin [fun (t:TUPLE) dataMCtrip_Trip_tmpuni windowintersectsS [attr(t, PeriodBox)]
        sort rdup dataMCtrip gettuples
        projectextend [Moid; TripOdo: length(.Trip atperiods attr(t,Period))]
        filter [.TripOdo > 0]
        sortby [Moid asc]
        groupby [Moid; Length: round group feed sum [TripOdo], 3)]]
    groupby [Id, Period; Dist: group feed max [Length]]
    project [Period, Dist]
consume;

let OBACRres010 =
    QueryLicences feed head [10]
```

```
        loopsel [dataSCcar_Licence_btree dataSCcar exactmatch [.Licence]
            project [Licence, Journey] {V1}]
    dataSCcar feed project [Licence, Journey] {V2}
    symmjoin [.Licence_V1 ≠ ..Licence_V2]
    filter [everNearerThan(.Journey_V1, .Journey_V2, 3.0)]
    projectextend [; QueryLicence: .Licence_V1, OtherLicence: .Licence_V2,
            Pos: .Journey_V1 atperiods deftime((distance(.Journey_V1, .Journey_V2) < 3.0) at TRUE)]
    filter [not(isempty(deftime(.Pos)))]
    project [QueryLicence, OtherLicence, Pos]
    sort rdup
consume;

let TBACRres010 =
    QueryLicences feed head [10] project [Licence] {V1}
    loopsel[fun (t:TUPLE) dataMCcar_Licence_btree dataMCcar exactmatch [attr(t, Licence_V1)]
        project[Moid]
        loopjoin[dataMCtrip_Moid_btree dataMCtrip exactmatch [.Moid] remove [Moid]] {V3}
        extend [t3bbx: bbox(.Trip_V3)]
        loopjoin [fun (u:TUPLE) dataMCtrip_Trip_sptmpuni windowintersectsS[rectangle3(
                minD(attr(u, t3bbx), 1) - 3.0, maxD(attr(u, t3bbx), 1) + 3.0, minD(attr(u, t3bbx), 2) - 3.0,
                maxD(attr(u, t3bbx), 2) + 3.0, minD(attr(u, t3bbx), 3), maxD(attr(u, t3bbx), 3))]
            sort rdup dataMCtrip gettuples
            filter [.Moid ≠ attr(u, Moid_V3)]
            filter [everNearerThan(attr(u, Trip_V3), .Trip, 3.0)]
            projectextend [Moid; Times: deftime((distance(attr(u, Trip_V3), .Trip) < 3.0) at TRUE)]
            filter [not(isempty(.Times))]
            sortby [Moid]
            groupby[Moid; Times1: group feed aggregateB[Times;
                    fun (P1:periods, P2:periods) P1 union P2; [const periods value ()]]]
            loopjoin [dataMCcar_Moid_btree dataMCcar exactmatch [.Moid]
                project[Licence]]]
        projectextend[Moid_V3, Moid, Times1, Trip_V3; QueryLicence: attr(t, Licence_V1), OtherLicence: .Licence]
        sortby [Moid_V3, Moid, QueryLicence, OtherLicence]
        groupby [Moid_V3, Moid, QueryLicence, OtherLicence;
                AllTimes: group feed aggregateB[Times1;
                        fun (P3:periods, P4:periods) P3 union P4; [const periods value ()]],
                AllTrips: group feed projectextend [Trip_V3;Start: inst(initial(.Trip_V3))]
                        sortby [Start] projecttransformstream [Trip_V3] concatS]
        projectextend [QueryLicence,OtherLicence; Pos: .AllTrips atperiods .AllTimes]]
consume;

let OBACRres011 =
    QueryPoints feed head [10] project[Pos] {PP}
    QueryInstants feed head [10] project[Instant] {II}
    product
    loopjoin [dataSCcar_Journey_sptmpuni windowintersectsS [box3d(bbox(.Pos_PP), .Instant_II)]
        sort rdup]
    dataSCcar gettuples
    projectextend [Licence, Pos_PP, Instant_II; XPos: val(.Journey atinstant .Instant_II)]
    filter [not(isempty(.XPos))]
    filter [distance(.XPos, .Pos_PP) < 0.5]
    projectextend [Licence; Pos: .Pos_PP, Instant: .Instant_II]
    sort rdup
consume;

let TBACRres011 =
    QueryPoints feed head [10] project [Pos] {PP}
    QueryInstants feed head [10] project [Instant] {II}
    product
    loopjoin [fun (t:TUPLE) dataMCtrip_Trip_sptmpuni windowintersectsS [
                box3d(bbox(attr(t, Pos_PP)), attr(t, Instant_II))]
        sort rdup dataMCtrip gettuples
        filter [.Trip present attr(t, Instant_II)]
        projectextend [Moid; XPos: val(.Trip atinstant attr(t, Instant_II))]
        filter [not(isempty(.XPos))]
        filter [distance(.XPos, attr(t, Pos_PP)) < 0.5]
        project [Moid]
        sort rdup]
    loopjoin [dataMCcar_Moid_btree dataMCcar exactmatch[.Moid] project [Licence]]
    projectextend [Licence; Pos: .Pos_PP, Instant: .Instant_II]
consume;

let OBACRres012allInstants =
    QueryInstants feed head [10]
```

```
    extend [Period: theRange(.Instant, .Instant, TRUE, TRUE)]
    aggregateB[Period; fun(I1: periods, I2:periods) I1 union I2; [const periods value ()]];
let OBACRres012 =
    QueryPoints feed head [10] project [Pos]
        loopjoin [dataSCcar_Journey_sptuni windowintersectsS [bbox(.Pos)]
            sort rdup dataSCcar gettuples
            projectextend [Licence; Journey: .Journey atperiods OBACRres012allInstants]]
        filter [.Journey passes .Pos]
        projectextend [Licence, Pos; Journey: .Journey at .Pos] {V1}
    QueryPoints feed head [10] project [Pos]
        loopjoin [dataSCcar_Journey_sptuni windowintersectsS [bbox(.Pos)]
            sort rdup dataSCcar gettuples
            projectextend [Licence; Journey: .Journey atperiods OBACRres012allInstants]]
        filter [.Journey passes .Pos]
        projectextend [Licence, Pos; Journey: .Journey at .Pos] {V2}
    symmjoin [.Licence_V1 < ..Licence_V2]
    QueryInstants feed head [10]
    symmjoin [val(.Journey_V1 atinstant ..Instant) = val(.Journey_V2 atinstant ..Instant)]
    projectextend [ Pos_V2, Instant; Licence1: .Licence_V1, Licence2: .Licence_V2]
    sort rdup
consume;
delete OBACRres012allInstants;

let TBACRres012 =
    QueryPoints feed head [10] project[Pos]
    QueryInstants feed head [10] project[Instant]
    product
    loopsel [fun(t: TUPLE)
        dataMCtrip_Trip_sptmpuni windowintersectsS [box3d(bbox(attr(t, Pos)), attr(t, Instant))] sort rdup {A}
        dataMCtrip_Trip_sptmpuni windowintersectsS [box3d(bbox(attr(t, Pos)), attr(t, Instant))] sort rdup {B}
        symmjoin [.id_A ≠ ..id_B]
        dataMCtrip gettuples2 [id_A] {C}
        dataMCtrip gettuples2 [id_B_C]
        filter [.Moid < .Moid_C]
        filter [no_components(intersection(.Trip, .Trip_C)) > 0]
        project [Moid, Moid_C]
        sort rdup
        loopjoin [dataMCcar_Moid_btree dataMCcar exactmatch [.Moid] project[Licence]]
        loopjoin [dataMCcar_Moid_btree dataMCcar exactmatch [.Moid_C] project[Licence] {C}]
        projectextend [; Pos_V2: attr(t, Pos), Instant: attr(t, Instant), Licence1: .Licence, Licence2: .Licence_C]]
consume;

let OBACRres013 =
    QueryRegions feed head [10] filter [not(isempty(.Region))] {RR}
    QueryPeriods feed head [10] filter [not(isempty(.Period))] {PP}
    product
    loopsel [fun (t:TUPLE) dataSCcar_Journey_sptmpuni windowintersectsS [
            box3d(bbox(attr(t, Region_RR)), attr(t, Period_PP))]
        sort rdup dataSCcar gettuples
        filter [(.Journey atperiods attr(t, Period_PP)) passes attr(t, Region_RR)]
        projectextend [Licence; Region: attr(t, Region_RR), Period: attr(t, Period_PP),
                Id_RR: attr(t, Id_RR), Id_PP: attr(t, Id_PP)]]
    sortby[Id_RR, Period, Licence]
    krdup[Id_RR, Period, Licence]
    project[Id_RR, Period, Licence]
consume;

let TBACRres013 =
    QueryRegions feed head [10] filter [not(isempty(.Region))] {RR}
    QueryPeriods feed head [10] filter [not(isempty(.Period))] {PP}
    product
    loopsel [fun (t:TUPLE) dataMCtrip_Trip_sptmpuni windowintersectsS [
                box3d(bbox(attr(t, Region_RR)),attr(t, Period_PP))]
        sort rdup dataMCtrip gettuples
        filter [(.Trip atperiods attr(t, Period_PP)) passes attr(t, Region_RR)]
        project [Moid]
        sort rdup
        loopjoin [dataMCcar_Moid_btree dataMCcar exactmatch [.Moid] project[Licence]]
        projectextend [; Region: attr(t, Region_RR), Period: attr(t, Period_PP), Licence: .Licence]]
consume;

let OBACRres014 =
    QueryRegions feed head [10] {RR}
    QueryInstants feed head [10] {II}
```

```
    product
    loopsel [fun(t:TUPLE) dataSCcar_Journey_sptmpuni windowintersectsS [
            box3d(bbox(attr(t, Region_RR)), attr(t, Instant_II))]
        sort rdup dataSCcar gettuples
        filter [val(.Journey atinstant attr(t, Instant_II)) inside attr(t, Region_RR)]
        projectextend [Licence; Region: attr(t, Region_RR), Instant: attr(t, Instant_II),
            Id_RR: attr(t, Id_RR), Id_II: attr(t, Id_II)]]
    sortby [Id_RR, Instant, Licence]
    krdup [Id_RR, Instant, Licence]
    project [Id_RR, Instant, Licence]
consume;

let TBACRres014 =
    QueryRegions feed head [10] {RR}
    QueryInstants feed head [10] {II}
    product
    loopsel [fun (t:TUPLE) dataMCtrip_Trip_sptmpuni windowintersectsS [
            box3d(bbox(attr(t, Region_RR)), attr(t, Instant_II))]
        sort rdup dataMCtrip gettuples
        filter [.Trip present attr(t, Instant_II)]
        filter [val(.Trip atinstant attr(t, Instant_II)) inside attr(t, Region_RR)]
        project [Moid]
        sort rdup
        loopjoin [dataMCcar_Moid_btree dataMCcar exactmatch [.Moid] project [Licence]]
        projectextend [; Region: attr(t, Region_RR), Instant: attr(t, Instant_II), Licence: .Licence]]
consume;

let OBACRres015 =
    QueryPoints feed head[10] {PO}
    QueryPeriods feed head[10] {PR}
    product
    loopsel [fun (t:TUPLE) dataSCcar_Journey_sptmpuni windowintersectsS [
            box3d(bbox(attr(t, Pos_PO)), attr(t, Period_PR))]
        sort rdup dataSCcar gettuples
        filter [(.Journey atperiods attr(t,Period_PR)) passes attr(t, Pos_PO)]
        projectextend [Licence; Point: attr(t, Pos_PO), Period: attr(t, Period_PR),
            Id_PO: attr(t, Id_PO), Id_PR: attr(t, Id_PR)]]
    sortby [Id_PO, Period, Licence]
    krdup [Id_PO, Period, Licence]
    project [Id_PO, Period, Licence]
consume;

let TBACRres015 =
    QueryPoints feed head [10] {PO}
    QueryPeriods feed head [10] {PR}
    product
    loopsel [fun (t:TUPLE) dataMCtrip_Trip_sptmpuni windowintersectsS [
            box3d(bbox(attr(t, Pos_PO)),attr(t, Period_PR))]
        sort rdup dataMCtrip gettuples
        filter [.Trip present attr(t, Period_PR)]
        filter [(.Trip atperiods attr(t, Period_PR)) passes attr(t, Pos_PO)]
        project [Moid]
        sort rdup
        loopjoin [dataMCcar_Moid_btree dataMCcar exactmatch [.Moid] project [Licence]]
        projectextend [; Point: attr(t, Pos_PO), Period: attr(t, Period_PR),Licence: .Licence]]
consume;

let OBACRres016Candidates1 =
    QueryLicences feed head[10]
          loopsel [fun (t:TUPLE) dataSCcar_Licence_btree dataSCcar exactmatch [attr(t, Licence)]]
        QueryPeriods feed head[10] {PP}
        QueryRegions feed head[10] {RR}
        product
    product
    projectextend [Licence, Region_RR, Period_PP, Id_RR, Id_PP;
          Journey: (.Journey atperiods .Period_PP) at .Region_RR]
    filter [no_components(.Journey) > 0]
consume;
let OBACRres016Candidates2 =
    QueryLicences feed head [20] filter [.Id > 10]
    loopsel [fun (t:TUPLE) dataSCcar_Licence_btree dataSCcar exactmatch [attr(t, Licence)]]
        QueryPeriods feed head [10] {PP}
        QueryRegions feed head [10] {RR}
        product
```

```
    product
    projectextend [Licence, Region_RR, Period_PP, Id_RR, Id_PP;
            Journey: (.Journey atperiods .Period_PP) at .Region_RR]
    filter [no_components(.Journey) > 0]
consume;
let OBACRres016 =
    OBACRres016Candidates1 feed {C1}
    OBACRres016Candidates2 feed {C2}
    symmjoin [(.Licence_C1 ≠ ..Licence_C2) and (.Id_RR_C1 = ..Id_RR_C2) and (.Id_PP_C1 = ..Id_PP_C2)]
    filter [not(everNearerThan(.Journey_C1, .Journey_C2, 0.1))]
    projectextend [; Licence1: .Licence_C1, Licence2: .Licence_C2, Region: .Region_RR_C1,
            Period: .Period_PP_C1, Id_RR: .Id_RR_C1, Id_PP: .Id_PP_C1]
    sortby [Id_RR, Id_PP, Licence1, Licence2]
    project [Id_RR, Period, Licence1, Licence2]
consume;
delete OBACRres016Candidates1;
delete OBACRres016Candidates2;

let TBACRres016CandidateTrips1 =
    QueryRegions feed head[10] {RR}
    QueryPeriods feed head[10] {PP}
    product
        extend[QBox: box3d(bbox(.Region_RR), .Period_PP)]
    QueryLicences feed head[10] {LL}
    product
    loopjoin [fun (tt1:TUPLE) dataMCcar_Licence_btree dataMCcar exactmatch [attr(tt1, Licence_LL)]
        project [Moid]
        loopsel [dataMCtrip_Moid_btree exactmatchS[.Moid]]
        sort {L}
        dataMCtrip_Trip_sptmpuni windowintersectsS [attr(tt1, QBox)] sort rdup {W}
        mergejoin[id_L, id_W]
        dataMCtrip gettuples2 [id_L]
        filter [.Trip present attr (tt1,Period_PP)]
        filter [.Trip passes attr (tt1,Region_RR)]
        projectextend [Moid; Trip: (.Trip atperiods attr(tt1, Period_PP)) at attr(tt1, Region_RR)]
        filter [no_components(.Trip) > 0]]
    projectextend [Id_RR, Id_PP, Region_RR, Period_PP, Trip, Moid; Licence: .Licence_LL]
consume;
let TBACRres016CandidateTrips2 =
    QueryRegions feed head[10] {RR}
    QueryPeriods feed head[10] {PP}
    product
        extend [QBox: box3d(bbox(.Region_RR), .Period_PP)]
    QueryLicences feed head [20] filter [.Id > 10] {LL}
    product
    loopjoin [fun (tt1:TUPLE) dataMCcar_Licence_btree dataMCcar exactmatch [attr(tt1, Licence_LL)]
        project [Moid]
        loopsel [dataMCtrip_Moid_btree exactmatchS[.Moid]] sort {L}
        dataMCtrip_Trip_sptmpuni windowintersectsS [attr(tt1, QBox)] sort rdup {W}
        mergejoin[id_L, id_W]
        dataMCtrip gettuples2 [id_L]
        filter [.Trip present attr(tt1, Period_PP)]
        filter [.Trip passes attr(tt1, Region_RR)]
        projectextend [Moid; Trip: (.Trip atperiods attr(tt1, Period_PP)) at attr(tt1, Region_RR)]
        filter [no_components(.Trip) > 0]]
    projectextend [Id_RR, Id_PP, Region_RR, Period_PP, Trip, Moid; Licence: .Licence_LL]
consume;
let TBACRres016 =
    TBACRres016CandidateTrips1 feed {C1}
    TBACRres016CandidateTrips2 feed {C2}
    symmjoin [(.Moid_C1 ≠ ..Moid_C2) and (.Id_RR_C1 = ..Id_RR_C2) and (.Id_PP_C1 = ..Id_PP_C2)]
    filter [not(everNearerThan(.Trip_C1, .Trip_C2, 0.1))]
    projectextend [Moid_C1, Moid_C2 ; Licence1: .Licence_C1, Licence2: .Licence_C2,
            Region: .Region_RR_C1, Period: .Period_PP_C1, Id_RR: .Id_RR_C1, Id_PP: .Id_PP_C1 ]
    sortby [Id_RR, Id_PP, Moid_C1, Moid_C2]
    krdup [Id_RR, Id_PP, Moid_C1, Moid_C2]
    project [Region, Period, Licence1, Licence2]
consume;
delete TBACRres016CandidateTrips1;
delete TBACRres016CandidateTrips2;

let OBACRres017PosCount =
    QueryPoints feed project [Pos] {PP}
    loopjoin [fun (t:TUPLE) dataSCcar_Journey_sptuni windowintersectsS [bbox(attr(t, Pos_PP))]
```

```
        sort rdup dataSCcar gettuples
        filter [.Journey passes attr(t, Pos_PP)]
        project [Licence]]
    projectextend [Licence; Pos: .Pos_PP]
    sortby [Pos asc, Licence asc]
    groupby [Pos; Hits: group feed rdup count]
consume;
let OBACRres017 =
    OBACRres017PosCount feed
    filter [.Hits = (OBACRres017PosCount feed max [Hits])]
    project [Pos, Hits]
consume;
delete OBACRres017PosCount;

let TBACRres017PosCount2 =
    QueryPoints feed project [Pos]
    loopjoin [fun (t:TUPLE) dataMCtrip_Trip_sptuni windowintersectsS [bbox(attr(t, Pos))]
        sort rdup dataMCtrip gettuples
        filter [.Trip passes attr(t, Pos)]
        project [Moid]
        sort rdup count feed namedtransformstream [Hits]]
    sortby [Hits desc, Pos asc]
consume;
let TBACRres017 =
    (TBACRres017PosCount2 feed head [1] extract [Hits])
        within [fun (MaxHits: int) TBACRres017PosCount2 feed filter [.Hits = MaxHits] tconsume]
    feed
consume;
delete TBACRres017PosCount2;
```