Secondo DBMS and Network Constrained Moving Objects

Simone Jandt

©FernUniversität in Hagen

LG Datenbanksysteme für neue Anwendungen

June 6, 2013

Contents

1 Overview **3**

2 Original Network Data Model **4**
2.1 Introduction . . . 4
2.2 Data Types . . . 4
2.2.1 Network . . . 4
2.2.2 Single Network Position . . . 5
2.2.3 Part of the Network . . . 5
2.3 Operations . . . 5

3 Network Implementation **6**
3.1 Introduction . . . 6
3.2 Implemented Data Types . . . 6
3.2.1 The Network . . . 6
3.2.2 Single Network Position . . . 8
3.2.3 Set of Single Network Positions . . . 8
3.2.4 Network Parts . . . 8
3.2.5 Single Moving Network Positions . . . 9
3.3 Implemented Operations . . . 9
3.3.1 Network Construction . . . 10
3.3.2 Translation from 2D Space into Network Data Model . . . 11
3.3.3 Translation from Network Data Model into 2D Space . . . 14
3.3.4 Extract Attributes . . . 15
3.3.5 Bounding Boxes . . . 16
3.3.6 Property Tests . . . 18
3.3.7 Merging Data Objects . . . 20
3.3.8 Path Computing . . . 20
3.3.9 Distance Computing . . . 21
3.3.10 Network Part Around a Single Network Position . . . 23
3.3.11 Restricting Single Moving Network Positions . . . 23
3.3.12 Create Moving Value From Single Unit . . . 24
3.3.13 Static Network Position Values of Junctions . . . 24

4 JNetwork Implementation **25**
4.1 Introduction . . . 25
4.2 Implemented Data Types . . . 25
4.2.1 Basic Data Types . . . 25
4.2.2 JNetwork . . . 26
4.2.3 JNetwork Dependent Data Types . . . 27
4.3 Implemented Operations . . . 28
4.3.1 Network Creation . . . 29
4.3.2 Creation Of Data Types . . . 29
4.3.3 Translation of 2D Data Types into JNetwork Data Types . . . 30
4.3.4 Translation of JNetwork Data Types into 2D Data Types . . . 30
4.3.5 Extract Attributes . . . 31
4.3.6 Bounding Boxes . . . 34
4.3.7 Merge Data Types . . . 34
4.3.8 Restrict Data Types . . . 34
4.3.9 Comparison Operators . . . 35

4.3.10 Property Tests . . . 35
4.3.11 Path Computing . . . 36
4.3.12 Network Distances . . . 36
4.3.13 Network Parts Around a Single Network Position . . . 36
4.3.14 Alternative Route Locations for Junctions . . . 36
4.3.15 Transformation of Paths into Network Parts . . . 37

5 Scripts Using Network Implementations 38
5.1 Introduction . . . 38
5.2 BerlinMOD Benchmark . . . 38
5.2.1 Translation of Source Data . . . 38
5.2.2 Executable Query Sets . . . 45
5.2.3 Comparison of Query Run Times and Storage Space . . . 64
5.3 Open Street Map Data and Networks . . . 65
5.3.1 Network . . . 65
5.3.2 JNetwork . . . 80
5.4 Match GPS-Tracks to Networks Generated from Open Street Map Data . . . 98

6 Traffic Estimation 100
6.1 Introduction . . . 100
6.2 Data Type for Traffic Information Estimation . . . 100
6.3 Operations for Traffic Estimation . . . 100
6.3.1 Compress Data for Traffic Estimation . . . 100
6.3.2 Traffic Estimation . . . 101
6.4 Examples for Traffic Estimation . . . 101

References 103

Chapter 1

Overview

The spatio-temporal database community searches for data models that enable the user to save storage space and speedup query execution on the stored data. One idea in this field is that many moving objects, for example cars and trains, are restricted in their movement by existing networks, like streets and railway networks. Based on this idea some network data models and implementations like [4, 10, 16, 17] have been provided in the past.

Both discrete network data models implemented in Secondo DBMS [2, 8, 11] are based on the abstract network data model presented in [10]. The temporal element in this abstract network data model is a straight forward development of the time sliced representation of spatio-temporal data types in two dimensional space provided in [9] for network dependent objects.

The short introduction of the abstract network data model in Chapter 2 should help to understand the commons and differences between both network implementations provided with the extensible Secondo DBMS. The description of the data types and operations of the first network implementation can be found in Chapter 3 and the data types and operations of the second network implementation in Chapter 4.

In Chapter 5 we present executable Secondo script files which enable the user to compare the power of both network implementations by the BerlinMOD Benchmark [1] and use most of the operators described in the chapters before. In Chapter 5 we present scripts creating network objects from OSM-Data files, provided by Open-Street-Map Foundation [7], and queries and operations that enable the user to create single moving position network objects for the networks created of open street map data from data collected by GPS-Devices.

Some of the operators provided with the first and second network implementation are not used in the sample scripts provided with this guide. They have been implemented to support a planned extension of the BerlinMOD Benchmark to enable the user to compare different network data models respectively different network data model implementations with respect to network data model specific challenges like: computation of shortest and fastest paths; network distance computation; computation of network parts full filling given conditions; trip planning and trip simulation; traffic estimation; and handling of dynamic changing network properties. The extension of the second network implementation with these operators is still in progress. In this context we also implemented some simple operations supporting traffic estimation within the first network implementation and in a specialized Secondo algebra module called `TrafficAlgebra`. We describe the operators for traffic estimation in the first network implementation in Chapter 6.

Chapter 2

Original Network Data Model

2.1 Introduction

As mentioned before the central idea of the network data model presented in [10] is that movements are restricted to given networks. Cars use street networks and trains railway networks. It is natural for us to speak about the position of a place relative to the street network, instead of giving its absolute position in coordinates. In the abstract network data model provided in [10] all positions are given relative to the routes of the network which stores all the spatial information of the route curves and junctions. The temporal element is represented by a time sliced representation of the spatio-temporal elements as described in [9, 10].

In the sequel we give a very short description of the data types in the abstract network data model. Interested readers are referred to the original paper [10] for detailed information.

2.2 Data Types

The abstract network data model introduces the network (*network*) itself, single position in the network (*gpoint*), and network parts (*gline*) as data types.

The type system of [9] is extended by a new kind *GRAPH* consisting of *gpoint* and *gline*. This new kind is also defined as possible basic data type for the *TEMPORAL* type constructors *moving* and *intime* such that the authors also have defined time-dependent moving data types called *moving*(*gpoint*), *moving*(*gline*), *unit*(*gpoint*), *unit*(*gline*), *intime*(*gpoint*) and *intime*(*gline*). The time-dependent network data types are not described in detail in [10], but their definition is straightforward to the definition of the spatial and spatio-temporal data types in [9].

In the sequel we present a short overview of the static network data types. Interested users are referred to the original papers where complete formal definitions of all data types and operations are given.

2.2.1 Network

In the abstract network data model the data type *network* is defined by two sets describing the spatial structure of the represented (street) network. The first set is called *routes* and describes, for example, the roads of a street network (see Table 2.1). The second set is called *junctions* and describes, following our example, the crossing point of two roads (see Table 2.2).

Attribute	**Data Type**	**Explanation**
id	*int*	route identifier
length	*real*	length of the route
curve	*line*	spatial geometry of the route
kind	{*simple*, *dual*}	on some roads, for example motorways, we have to distinguish between the two sides of the road these routes are marked to be *dual*.
start	{*smaller*, *bigger*}	tells us if the spatial curve starts at the lexicographical smaller or bigger endpoint

Table 2.1: Attributes of the Entries in the Set of Routes

Attribute	Data Type	Explanation
routemeas1	(*int*, *real*)	pair identifying the first route of the junction and describing the position of the junction on that route[1]
routemeas2	(*int*, *real*)	pair identifying the second route of the junction and describing the position of the junction on that route
cc	*int*	connectivity code[2] telling which lanes of the roads are connected by this junction

Table 2.2: Attributes of the Entries in the Set of Junctions

In a more discrete network data model description in [10] the sets of routes and junctions are represented by two relations, and the attributes *kind* and *start* of the *routes relation* tuples are represented by two Boolean flags. The flags are set to ***true*** if the route is *dual* respectively the curve starts at the smaller end point.

2.2.2 Single Network Position

The data type *gpoint* defines a single position in the network. It consists of a route identifier which must exist in the set of routes of the network, the distance from the routes origin following the route curve, and a side value. The distance d must hold $0 \leq d \leq$ *length of route curve*. The side value may be one of $\{up, down, none\}$. Whereas *none* is always used for simple routes and means reachable from both sides of the route. According to this *up* means reachable only driving the road upwards from the origin to the end of the route curve, and *down* means reachable only driving the road downwards from the end to the origin of the route curve.

In a more discrete representation the authors of [10] extend the data type *gpoint* by an integer value identifying the network the *gpoint* is related to.

2.2.3 Part of the Network

Parts of routes are called *route intervals*. They are defined by two single network positions on the same route at the same side and can be written as $(rid, d_1, d_2, side)$, with $0 \leq d_1 \leq d_2 \leq$ *length of route* and $side \in \{up, down, none\}$, whereas the side value is defined almost analogous to Section 2.2.2.

A network part is defined by a set of *route intervals* related to the same network. [10] introduces the data type *gline* as discrete representation of network parts which consists of an integer value which identifies the network, and a set of *route intervals* describing the network part represented by this *gline*.

2.3 Operations

[10] defines syntax and semantics of different sets of operations most of them are defined straight forward to the operations on spatial and spatio-temporal data types described in [5, 6]. Others are significant to network objects, they enable the user to access the single attributes of the network data types, convert spatial data types into network data types and vice verse, compute network parts and paths from one network position to another, or simulate trips between network positions.

We omit the detailed description of the operations at this point, because we describe the existing network implementations including the operations on the implemented network data types in detail in Chapter 3 and Chapter 4.

[1] Junctions are only defined between two different routes. The first route is always the route with the smaller identifier. The position of a junction on a route is defined by the distance d of the junction from the routes origin following the route curve, which must hold the equation $0 \leq d \leq$ *length of route curve*.

[2] See [10] for a detailed explanation of the connectivity code definition.

Chapter 3

Network Implementation

3.1 Introduction

In this chapter we describe our first implementation of the abstract network data model from [10] in SECONDO. Parts of this network implementation have been done by one of our students as part of his final thesis [14].

This first implementation is parted into two SECONDO algebra modules. The first algebra module called `NetworkAlgebra` contains the data type *network* and the network dependent static data types *gpoint*, *gpoints*, and *gline* as far as the operations deal with these data types. The second algebra module called `TemporalNetAlgebra` contains the network dependent temporal data types *mgpoint*, *ugpoint* and *igpoint* and the operations dealing with these data types.

In Section 3.2 we describe the implemented data types of both algebra modules and in Section 3.3 the implemented operations on these data types in SECONDO.

3.2 Implemented Data Types

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

3.2.1 The Network

Different from the data type description in the abstract network data model (see Chapter 2) the implementation of the data type *network* consists of:

- three different relations, containing the routes (see Table 3.3), junctions (see Table 3.2), and sections (see Table 3.1)[1] data
- one *real* value, describing the maximum allowed distance from a curve it should be mapped on in **map-matching** operation
- four B-Trees, indexing the route identifier attributes in the four relations
- a two dimensional R-Tree, indexing the *ROUTE_CURVE* attribute of the routes relation
- a unique network identifier (*int*)[2]
- two sets connecting pairs of section identifiers and directions (called directed sections) of adjacent directed sections[3]

[1] A section describes the street part between two crossings, or an crossing and the dead end of a street.

[2] Each *network* object in a SECONDO database system must be labeled with a unique number to get a clear conjunction between the network dependent objects and the network they belong to.

[3] Two directed sections are adjacent, if they are connected by a junction.

Attribute	Data Type	Explanation
SECTION_SID	*int*	unique section identifier
SECTION_RID	*int*	route identifier of the route the section belongs to[4]
SECTION_MEAS1	*real*	distance of the begin of the section from the routes origin following the route curve
SECTION_MEAS2	*real*	distance of the end of the section from the routes origin following the route curve
SECTION_DUAL	*bool*	***true*** tells that the lanes of the different directions in the section are separated
SECTION_CURVE	*sline*	spatial geometry of the section curve in the two dimensional plane
SECTION_CURVE_STARTS_SMALLER	*bool*	***true*** if the section curve starts at the lexicographical smaller endpoint
SECTION_RRC	*TupleIdentifier*[5]	identifies the tuple in the routes relation the section is part of

Table 3.1: Attributes of Sections Relation in *network*

Attribute	Data Type	Explanation
JUNCTION_ROUTE1_ID	*int*	route identifier of the first[6] route of the junction.
JUNCTION_ROUTE1_MEAS	*real*	distance of the junction from the origin of the first route following the route curve
JUNCTION_ROUTE2_ID	*int*	route identifier of the second route of the junction
JUNCTION_ROUTE2_MEAS	*real*	distance of the junction from the origin of the second route following the route curve
JUNCTION_CC	*int*	connectivity code[7] tells us for which lanes of the routes are connected by this junction
JUNCTION_POS	*point*	spatial position of the junction in the two dimensional plane
JUNCTION_ROUTE1_RC	*TupleIdenitfier*	identifies the tuple of the first route in the routes relation.
JUNCTION_ROUTE2_RC	*TupleIdenitfier*	identifies the tuple of the second route in the routes relation
JUNCTION_SECTION_AUP_RC	*TupleIdentifier*	identifies the tuple of the section upwards of the junction on the first route in the sections relation
JUNCTION_SECTION_ADOWN_RC	*TupleIdentifier*	identifies the tuple of the section downwards of the junction on the first route in the sections relation
JUNCTION_SECTION_BUP_RC	*TupleIdentifier*	identifies the tuple of the section upwards of the junction on the second route in the sections relation
JUNCTION_SECTION_BDOWN_RC	*TupleIdentifier*	identifies the tuple of the section downwards of the junction on the second route in the sections relation

Table 3.2: Attributes of Junctions Relation in *network*

[4]One problem of this first network implementation is, that in real life a section may belong to more than one route. For example the motorways A1 and A61 in Germany share the sections between the motorway crossings Bliesheim and Blessem. This can not be represented in this first network implementation. If a section belongs to more than one route we have to decide before network creation to which route the section should be assigned to.

[5]A value of the data type *TupleIdentifier* identifies a single tuple in the relation it belongs to.

Attribute	Data Type	Explanation
ROUTE_ID	*int*	unique route identifier
ROUTE_LENGTH	*real*	length of the route curve
ROUTE_CURVE	*sline*[8]	spatial geometry of the route curve in the two dimensional plane
ROUTE_DUAL	*bool*	***true*** means that the lanes for the different directions of the road are separated
ROUTE_STARTSSMALLER	*bool*	***true*** means the route curve starts at the lexicographical smaller endpoint

Table 3.3: Attributes of Routes Relation in *network*

3.2.2 Single Network Position

The data type *gpoint* describes a single position in a *network*. It consists of an *int* value which identifies the network the route location belongs to, and the route location. The route location is given by the route identifier (*int*), the distance (*real*) from the origin of the route, and a parameter side(*side*).

The three possible values of *side* are *Down*, *Up*, and *None*. *Up*(*Down*) means a position can only be reached driving in up-(down-)wards the route curve. *None* means a position can be reached from both sides of the route.

3.2.3 Set of Single Network Positions

The data type *gpoints* consists of a set of *gpoint* values. It can be used to describe a collection of different places, for example all book shops in a town.

3.2.4 Network Parts

The data type *gline* describes a part of the network. The data type *gline* consists of a network identifier (*int*), a set of *route intervals*[9], the total length (*real*) of all *route interval*s in the set, and a Boolean flag telling if the *route intervals* are stored sorted or not.

We call a set of *route intervals* sorted if it fulfills the following conditions:

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

This form of sorting has been introduced, because the computation time of many algorithms dealing with *gline* values can be reduced, if the set of *route intervals* is stored sorted. In fact, not for all *gline* values the set of *route intervals* can be stored sorted. If we describe parts of the network, like districts of towns, we can store the *route intervals* sorted, because it is regardless in which sequence we read the set of *route intervals* describing the network part. But, if the *gline* value represents a path between two network positions a and b the *route intervals* must be stored in the sequence they are used in the path, which is nearly never a sorted sequence of *route intervals*.

Many algorithms can take profit from sorted *gline* values. As described in Section 3.3 we can perform a binary search on the sorted set of *route intervals* in $O(\log r)$ time, instead of performing a linear scan of the not sorted set of *route intervals* in $O(r)$ time. The algorithms check only the *sorted* flag to decide which sub algorithm must be used for further evaluation.

[6]The first route identifier of a junction will always be the lower route identifier of the two routes which are connected by the junction.

[7]See [10] for detailed information about the meaning of the different connectivity code values.

[8]The data type *sline* consists of a set of *HalfSegment*s representing the curve of the route in the two dimensional plane.

[9]The internal data type *RouteInterval* consists of an route identifier (*int*), and two distance values (*real*), defining the distance of the start and end position of the route part from the origin of the route. Different from [10] the both distances are not expected to be sorted by value and the side value is missing in this first network implementation. This is another reason, why we provided an second network implementation within SECONDO where these problems are omitted.

Sorting and compressing the set of *route intervals* needs time. We pay for the advantage of reduced computation time for many algorithms taking profit from sorted *gline* values by a higher time complexity of algorithms creating sorted *gline* values. We think this additional time is well invested, because it is needed once when we create a *gline* value, and we save computation time in nearly all operations dealing with *gline* values.

3.2.5 Single Moving Network Positions

The temporal version of the data type *gpoint* is called *mgpoint* (short form of *moving*(*gpoint*)). It is implemented in the `TemporalNetAlgebra`. The data type *mgpoint* represents the complete history of the movement of a single network position; for example it may represent a car driving around in the related network. The main parameter of an *mgpoint* value is a set of *ugpoints* with disjoint time intervals. The time intervals of the *ugpoints* in the set must be disjoint, because nothing in our known world can be at two different places at the same time. The *ugpoints* are stored in the *mgpoint* value sorted by ascending time intervals. This allows us to perform a binary search on the units of the *mgpoint* value to find the *ugpoint* containing a given time instant within the definition time of the *mgpoint*.

The data type *ugpoint* (short for *unit*(*gpoint*)) consists of a time interval, and two *gpoint* values with identical network identifier, route identifier and side values. The first *gpoint* describe the start and the second *gpoint* the end position of the *mgpoint* in the network within the given time interval. We assume that the *mgpoint* moves from the start to the end position with constant speed in the given time interval. The time interval consists of a start and a end time instant and two Boolean flags, one for each time instant. The Boolean flag tells us if the time interval is open or closed at the corresponding time instant. With help of these parameters we could compute the exact position of a *ugpoint* value at each time instant within the time interval. Assumed a *ugpoint* value passes a query *gpoint* value within the time interval, we can compute the time instant when the *ugpoint* reaches the given *gpoint*. The position of an *mgpoint* value at a given time instant is represented by an *igpoint*.

The data type *igpoint* (short for *intime*(*gpoint*)) consists of a time instant and a *gpoint* value representing the position of the *mgpoint* value at the given time instant.

In our experiments we extended the *mgpoint* from [10] with some additional parameters to speed up query execution:

- *length* (*real*): The length parameter stores the total distance driven by the *mgpoint* value.
- *trajectory* (Sorted set of *route intervals*)[10]: Represents the network part ever traversed by an *mgpoint*. This reduces the time to decide if an *mgpoint* ever passed a given place in the network (*gpoint* or *gline*) from $O(m)$ to $O(\log r)$ with $r \ll m$, because we can perform a binary search on the much lower number r of *route intervals* of the *trajectory* instead of a linear scan of the m units of the *mgpoint*. We do not maintain the *trajectory* value by every operation, therefore we introduced the next parameter *trajectory_defined*.
- *trajectory_defined* (*bool*): Tells us if the *trajectory* parameter is well defined or must be recomputed before we can use it.
- *bbox* (*rect3*[11]): The spatio-temporal bounding box of the *mgpoint* was introduced to save our computational work, because it is very expensive to get exact spatial information in network environment. All spatial information is only stored in the central network object. Although an *mgpoint* stays on the same route with the same speed the *mgpoint* might move in different spatial directions within a single unit. For example a car may drive downhill on a winding road. In this case it is not enough to get the spatial position of the start and end position of the unit to compute the spatial part of the bounding box. The complete route part passed in a unit must be inherited in the computation of the bounding box. The bounding box of an *mgpoint* is the union of the bounding boxes of its units, such that the bounding box computation is very expensive. For this reason the *bbox* value is not maintained at every change of an *mgpoint* value. It is only computed on demand and stored using the *trajectory* value of the *mgpoint* or stored if we could get it for free[12].

3.3 Implemented Operations

In this section we describe the operations provided with the first network implementation in SECONDO. We give for each operation its signature, an example call, the time complexity of the operation, and, if interesting, a description of the algorithm. The letters used in the formulas describing the time complexity of the operations have the meaning:

[10] The SECONDO DBMS does not allow us to use a *gline* value as parameter of an *mgpoint*. It is a way around to use a sorted set of *route intervals* instead.

[11] Three dimensional rectangle with real coordinates.

[12] For example, we can get the *bbox* value by a copy in O(1) time if we translate an *mpoint* to an *mgpoint* value.

- j_{net} is the number of entries in the *junctions relation* of a *network* value.
- r_{net} is the number of entries in the *routes relation* of a *network* value.
- s_{net} is the number of entries in the *sections relation* of a *network* value.
- b is the number of *bounding gpoints* of a *gline* value[13].
- c is the number of *candidate routes* resulting from a scan of the R-Tree of the *routes relation* of the network object.
- h is the number of *HalfSegments* of a *line* or *sline* value.
- m is the number of *units* of a *mgpoint* value.
- p is the number of *time intervals* in a *periods* value.
- r is the number of *route intervals* of a *gline* value or the *trajectory* of a *mgpoint* value.
- u is the number of units of an *mpoint* value.

If more than one object of a data type takes part in an operation we will write meaningful indexes to the letters to distinguish between the values of the different objects with the same data type.

Many operations are defined for more than one data type. We introduce another set of letters used in the signatures of the operations:

- $A := \{\underline{gline}, \underline{mgpoint}\}$.
- $B := \{\underline{gline}, \underline{mgpoint}, \underline{ugpoint}\}$.
- $C := \{\underline{mgpoint}, \underline{ugpoint}\}$.
- $D := \{\underline{gpoint}, \underline{gpoints}, \underline{gline}\}$
- $T := \{\underline{instant}, \underline{periods}\}$.
- $X := \{\underline{gpoint}, \underline{gline}\}$
- $Y := \{\underline{gpoint}, \underline{mgpoint}\}$.
- $Z := \{\underline{gpoint}, \underline{ugpoint}\}$.

This enables us to write **operator**: $A \rightarrow \underline{bool}$ instead of the itemization **operator**: $\underline{gline} \rightarrow \underline{bool}$, **operator**: $\underline{mgpoint} \rightarrow \underline{bool}$.

At least we define T_{sec} to be the tuple type of the *sections relation* of a *network* value (see Table 3.1).

3.3.1 Network Construction

$\underline{int} \times \underline{real} \times \underline{rel} \times \underline{rel} \rightarrow \underline{network}$ **thenetwork**(*n*, *factor*, *routes*, *junctions*)

The operator **thenetwork** constructs the new *network* object with the given network identifier n[14] from the two given relations by Algorithm 1. The two input relations are expected to have the following content, which corresponds to the routes and junction relations defined in the discrete network data model described in [10]:

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

The parameter *factor* is also stored in the resulting *network* object. It is used in map matching operations to tell the system how big the allowed tolerance is if a point does not match exact the given route curve[15].

[13] See Section 3.3.4 for an explanation of *bounding gpoints*.

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

[15] While implementing map matching algorithms it becomes clear, that we must allow some deviation from an exact hit on the line representing the spatial curve of the road, because GPS-Signals almost do not hit exactly the relatively small line representing the road. The accepted deviation may not be to big to prevent the algorithm from mapping the track to the wrong road respectively to test too much candidate routes. Now we get network data from different sources one time the coordinates are given in geographic coordinates, another time in UTM-System, such that we can not choose a fixed value of deviation for the map matching algorithms. Because a deviation of 1.0 in UTM-System is a completely other value in meters than in geographic coordinates. To enable map matching regardless of used spatial information system we introduced the *scalefactor* parameter to adjust the allowed deviation in map mapping corresponding to the spatial system the *network* is defined with.

Algorithm 1 thenetwork $(n, factor, rout, junct)$

Require: $n \geq 0$ (*int*), $factor$ (*real*), two input relations as described before.

1: Create empty network object *net* with $id := n$ and $scalefactor := scale$.
2: Copy *rout* to *routes relation* of *net*
3: Construct B-Tree indexing *route identifiers* in *routes relation* of *net*
4: Construct R-Tree indexing *route curves* in *routes relation* of *net*
5: Copy *junct* to *junctions relation* of *net* and add route *tuple identifiers* from *routes relation* of *net*
6: Construct two B-Trees indexing the first / second *route identifier* in the *junctions relation*
7: **for** Each tuple r in *routes relation* **do**
8: **for** Each junction j_i at this route **do**
9: Compute the *Up* and *Down sections*
10: Add the *sections* to the *sections relation*
11: Add the *section identifiers* to the *junctions relation*
12: **end for**
13: **end for**
14: Construct B-Tree indexing *route identifiers* in the *sections relation* of *net*
15: **for** Each junction j of the *junctions relation* **do**
16: Find *pairs of adjacent sections* and fill *adjacency lists* of *net*
17: **end for**

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

- line 1: O(1)
- line 2: O(r_{net})
- lines 3 + 4: O($r_{net} \log r_{net}$)
- line 5: O(j_{net})
- line 6: O($j_{net} \log j_{net}$)
- lines 7 - 13: O(s_{net})
- line 14: O($s_{net} \log s_{net}$)
- lines 15 - 17: O(j_{net})

Such that we get a total time complexity of

$$O(s_{net} \log s_{net}), \text{ because } r_{net}, j_{net} \leq s_{net}$$

.

3.3.2 Translation from 2D Space into Network Data Model

The Operations in Table 3.4 are used to translate spatial and spatio-temporal data types from the two dimensional plane data model [5,6,9] of the SECONDO DBMS into the network data model representation. In [10] these operations are all called **in_network** with different signatures. All translations will only be successful if the values of the two dimensional data types are aligned to the given network otherwise the network representation of the object is not defined.

Signature	Example Call
network × *point* → *gpoint*	**point2gpoint**(*network*, *point*)
network × *line* → *gline*	**line2gline**(*network*, *line*)
network × *mpoint* → *mgpoint*	**mpoint2mgpoint**(*network*, *mpoint*)
	mapmatching(*network*, *mpoint*)

Table 3.4: Operators Translating 2D Spatial to Network Objects

If possible the operation **point2gpoint** translates a *point* value into a *gpoint* value of the given *network* as described in Algorithm 2. The algorithm has a worst case time complexity from O($\log r_{net} + c + h$).

Algorithm 2 point2gpoint(*net*, *p*)

Require: *net* (*network*), *p* (*point*)

```
1: Use R-Tree of routes relation to get a set of candidate routes cr
2: for each c ∈ cr do
3:   if Distance(p, c) = 0 then
4:     pos = 0.0
5:     for each HalfSegment h of c from origin to end of c do
6:       if p is allocated on h then
7:         pos+ = Distance(h.start, p)
8:         return gpoint(net.Id, c.Id, pos, None)
9:       else
10:        pos+ = length of h
11:      end if
12:    end for
13:  end if
14: end for
15: return gpoint(undefined)
```

The operation **line2gline** translates a *line* value into a sorted *gline* value. The algorithm takes every *half segment* of the *line* value and tries to find the start and end of the *half segment* on the same route using a variant of **point2gpoint** Algorithm 2. The computed *route intervals* are sorted and merged with help of a *RITree*[16] before the resulting *gline* value is returned.

The time complexity of **line2gline** is

$$O(h \log r_{net} + \sum_{i=0}^{h} (c_i) + \sum_{j=0}^{h} (h_j)), \text{ because } h \geq r_{in} \wedge r_{net} \geq r_{out}.$$

The first network implementation provides with **mpoint2mgpoint** and **mapmatching** two different operators translating an *mpoint* value into a corresponding *mgpoint* value. Both operation have the signature *network* × *mpoint* → *mgpoint*.

The main difference is that the operator **mpoint2mgpoint** expects the *mpoint* to move exactly in the *network* and to start new units exactly at the *junctions* of the *network*. The operator **mapmatching** is tries to interpolate network movement between correct detected network positions by trip simulation using shortest path computation by A^*-Algorithm between the detected correct network positions in the *mpoint* movement.

The later developed SECONDO algebra module `MapMatchingAlgebra` provides with the operator **mapmatchmht** a much more sophisticated MapMatching-Algorithm matching GPS-Tracks into the first network representation. The usage is described in Section 5.4.

The operation **mpoint2mgpoint** translates an *mpoint* which is constrained by the *network* into an *mgpoint* value. The single steps of Algorithm 3 have the following time complexities:

- line 1 + 2: O(1)
- line 3: O($\log r_{net} + c + h$), because a variant of **point2gpoint** is used.
- line 4: The **for**-loop is executed u times and distinguishes three different cases:
  1. line 7: O(1)
  2. line 9 - 11: O(1)
  3. line 14 - 17: O(max(adj_j)) if adj_j is the number of routes connected by the crossing j_j.
- line 20 - 22: O(1)

We get a worst case time complexity of

$$O(\log r_{net} + c + \sum_{i=0}^{u} (h_i) + \sum_{j=0}^{u} adj_j).$$

We will have a much smaller computation time in the average case, because the worst case takes only place if the car changes the route at each unit.

[16] The internal data type *RITree* is a binary search tree for *route intervals*. It is implemented in the `NetworkAlgebra` of SECONDO. It sorts and merges a set of *route intervals* in O($r_{in} \log r_{out}$) time, if r_{in} is the number of inserted *route intervals* and r_{out} is the number of resulting *route intervals* and $r_{in} \geq r_{out}$.

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

Require: *net*(*network*) and *mpoint* (*mpoint*)
Initialize empty result value *mgpoint* with *net.Id*
upoint = first unit *mpoint*
Initialize *ugpoint* with *net* value of *upoint*
for Each *upoint u* in *mpoint* **do**
  if Endpoint of *u* is on same route than *ugpoint* **then**
    if Direction and speed stay the same **then**
      Extend *ugpoint* to include value of *u*
    else
      Add *ugpoint* to *mgpoint*
      Add *routeinterval* of *ugpoint* to *trajectory*
      ugpoint is *net* value of *u*
    end if
  else
    Add *ugpoint* to *mgpoint*
    Add *routeinterval* of *ugpoint* to *trajectory*
    Search *u* on adjacent sections
    ugpoint is *net* value of *upoint*
  end if
end for
Add *ugpoint* to *mgpoint*
Copy *bounding box* of *mpoint* to *mgpoint*
return *mgpoint*

Algorithm 4 mapmatching(*net*, *mpoint*)

Require: *net*(*network*) and *mpoin* (*mpoint*)
Initialize empty result value *mgpoint* with *net.Id*
start := $\perp$
end := $\perp$
while $\exists$*upoint* $\in$ *mpoint* **do**
  while *start* =$\perp$ **do**
    start := **point2gpoint**(*net*, *upoint.start*)
  end while
  while *end* =$\perp$ **do**
    end := **point2gpoint**(*net*, *upoint.end*)
  end while
  if *start* $\neq\perp$ and *end* $\neq\perp$ **then**
    if *start* is on same route as *end* **then**
      Write *ugpoint* to result *route interval* of *ugpoint* to *trajectory*
    else
      sp := **shortest_pathastar**(*start*, *end*)
      for Each *route interval* of *sp* **do**
        Write *ugpoint* to result *route interval* of *ugpoint* to *trajectory*
      end for
    end if
  end if
  start := *end*
  end := $\perp$
end while
Copy *bounding box* of *mpoint* to *mgpoint*
return *mgpoint*

The operation **mapmatching** (see Algorithm 4) uses for the *end* computation a variant of **point2gpoint** which first tries to map the new *point* to the same route as the last *point* was matched. If this is successful the time complexity for this step is $O(h_i)$ instead of $O(\log r_{net} + c_i + h_i)$. The shortest path computation with help of the A^*-Variant stops immediately if the path is found. In most cases the distance between the two network positions on different routes is very small, such that this operation is not so expensive and does not dominate the time costs of searching start and end point in the network. But it may become very expensive if

both network positions are far away from each other. But in nearly all cases the worst case time complexity of **mapmatching** is limited by

$$O(u \log r_{net} + \sum_{i=0}^{u} c_i + \sum_{j=0}^{u} h_j)$$

.

3.3.3 Translation from Network Data Model into 2D Space

The operations in Table 3.5 translate network data types into spatial data types. In [10] these operations are called **in_space**.

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

Table 3.5: Operators Tranlsating Network into 2D Spatial Objects

The operation **gpoint2point** translates a *gpoint* value into the corresponding *point* value. The algorithm uses the B-Tree of the *routes relation* of the *network* object, the *gpoint* belongs to, to get the route curve of the *gpoint*. This takes $O(\log r_{net})$ time. The spatial position of the *gpoint* on this route is computed by searching the *HalfSegments* of this route curve for the position of the *gpoint* in worst case $O(h)$ time. Together we get a worst case time complexity of $O(h + \log r_{net})$.

The operation **gline2line** translates a *gline* value into a spatial *line* value. The algorithm uses the B-Tree index on the *routes relation* to get the corresponding route curve for every *route interval* of the *gline* value. For each *route interval* r_i the corresponding h_i segments of the route curve are computed and merged into the resulting *line* value.

We need $O(\log r_{net})$ time to get the route curve and $O(h_i)$ time to get the segments of the *route interval*. The time complexity of the complete operation is $O(r \log r_{net} + \sum_{i=1}^{r} h_i)$

Algorithm 5 mgpoint2mpoint(*mgpoint*)

Require: *mgpoint* (*mgpoint*)

```
1:  for Each ugpoint of mgpoint do
2:    if ugpoint stays on the same route as last ugpoint then
3:      use variant of gpoint2point to compute the position of end point
4:      if start and end point are not on the same HalfSegment then
5:        splitugpoint(ugpoint, routecurve, mpoint)
6:      else
7:        Add upoint to mpoint
8:      end if
9:    else
10:     Use B-Tree Index to get new route curve for the ugpoint
11:     Use variant of gpoint2point to compute the start and end point
12:     if start and end are not on the same HalfSegment then
13:       splitugpoint(ugpoint, routecurve, mpoint)
14:     else
15:       Add upoint to mpoint
16:     end if
17:   end if
18: end for
19: return mpoint
```

Algorithm 6 splitugpoint(*ugpoint*, *curve*, *mpoint*)

Require: *ugpoint* (*ugpoint*), *curve* (*sline*) passed within *ugpoint*. resulting *mpoint* (*mpoint*)
1: **if** Moving Direction = *Up* **then**
2: compute the time instant the *ugpoint* reaches the end of the segment
3: **else**
4: compute the time instant the *ugpoint* reaches the start of the segment
5: **end if**
6: Add *upoint* to *mpoint*
7: **for** Each segment of *curve* passed completely by *ugpoint* **do**
8: **if** Moving Direction = *Up* **then**
9: compute the time instant the *ugpoint* reaches the end of the segment
10: **else**
11: compute the time instant the *ugpoint* reaches the start of the segment
12: **end if**
13: Add corresponding *upoint* to *mpoint*
14: **end for**
15: **if** Moving Direction = *Up* **then**
16: Add last *upoint* from section start to the end of *ugpoint* to *mpoint*
17: **else**
18: Add last *upoint* from section end to the end of *ugpoint* to *mpoint*
19: **end if**

The operation **mgpoint2mpoint** translates an *mgpoint* value into the corresponding *mpoint* value. See Algorithms 5 and 6 for detailed description. The time complexity of Algorithm 6 depends on the number of *half segments* of the route curve passed by the *ugpoint*. It needs O(h) time.

The single steps of Algorithm 5 need the following times:

- line 1: The FOR-Loop is called m times and needs
  - in case of line 3: O($h + \log r_{net}$) plus
    - in case of line 5: O(h)
    - in case of line 7: O(1)
  - in case of line 10-11: O($h + \log r_{net}$) plus
    - in case of line 13: O(h)
    - in case of line 15: O(1)

We get a worst case time complexity of

$$O(m \log r_{net} + \sum_{i=1}^{m} h_i)$$

3.3.4 Extract Attributes

The operators of Table 3.7 return the attributes from the different data types in O(1) time, whereas the time complexity of the operators in Table 3.6 depends on the complexity of the return value.

Signature	Example Call
network → *rel*	**routes**(*network*)
	junctions(*network*)
	sections(*network*)
network × *int* × *bool* → *stream*(*tuple*(sid *int*, dir *bool*))	**getAdjacentSections**(*network*, *sid*, *up*)
	getReverseAdjacentSections(*network*, *sid*, *up*)
mgpoint → *gline*	**trajectory**(*mgpoint*)
C → *periods*	**deftime**(*mgpoint*)
mgpoint → *stream*(*ugpoint*)	**units**(*mgpoint*)
gline → *gpoints*	**getBGP**(*gline*)

Table 3.6: Operators Extracting Complex Attributes of Data Types

Signature	Operator	Explanation
$A \rightarrow$ *int*	**no_components**	Returns the number of *route intervals* respectively *units* of the argument.
$A \rightarrow$ *bool*	**isempty**	Returns ***true*** if the argument is not defined or has no components.
$B \rightarrow$ *real*	**length**	Returns the length of the *gline* respectively driven distance of the *mgpoint* or *ugpoint*
mgpoint→*igpoint*	**initial**	Returns the first position and start time of the *mgpoint* value.
	initial	Returns the last position and end time of the *mgpoint* value.
ugpoint → *real*	**unitrid**	Returns the route identifier of the *ugpoint* value.
	unitstartpos	Returns the start position of the *ugpoint* value.
	unitendpos	Returns the end position of the *ugpoint* value.
	unitstarttime	Returns the start time instant of the *ugpoint* value as *real* value.
	unitendtime	Returns the end time instant on the *ugpoint* value as *real* value.
ugpoint → *instant*	**startunitinst**	Returns the start time instant of the *ugpoint* value.
	endunitinst	Returns the end time instant on the *ugpoint* value.
igpoint → *gpoint*	**val**	Returns the *gpoint* of the *igpoint* value
igpoint → *instant*	**inst**	Returns the time instant of the *igpoint* value

Table 3.7: Operators Extracting Attributes of Data Types in O(1) Time

The operators **routes** ($O(r_{net})$), **junctions** ($O(j_{net})$), and **sections** ($O(s_{net})$) get a *network* object as parameter and return the values of the internal relations of the given *network*. The time complexity of the operators is given in brackets.

The operators **getAdjacentSections** and **getReverseAdjacentSections** get as parameters a *network* value, an *identifier* (*int*) which identifies the query section, and a Boolean value telling if we want to search in *Up*(***true***) or *Down*(***false***) direction of the section. The tuples in the return stream identify the (reverse) adjacent sections[17] of the query section by identifier and direction. The query section is searched by a binary search in the (reverse) adjacency list and the x result sections are returned. The time complexity of the operations is $O(x + \log s_{net})$.

The operation **trajectory** returns the *trajectory* of the *mgpoint* as sorted *gline* value representing all the places ever traversed by the *mgpoint*. If the *trajectory* attribute of the *mgpoint* is defined the *route intervals* are returned as *gline* value in $O(r)$ time. Otherwise the *trajectory* attribute is computed and returned as *gline* value by a linear scan of the units of the *mgpoint* value in $O(r_{out} + m \log r_{out})$ time. The latter time complexity value could be reduced to $O(m)$ if we store the computed *route intervals* immediately to the resulting *gline* value without sorting and merging. As mentioned in Section 3.2.4, we think that the overhead in computation time for sorting and merging is well invested.

The operation **deftime** is defined for *mgpoint* and *ugpoint*. It returns the *periods* representing the definition time of the the *mgpoint* value respectively the *ugpoint* value. This takes $O(1)$ time for *ugpoint* values and $O(m)$ time for *mgpoint* values.

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

The operation **getBGP** returns the *bounding gpoints*[18] of the given *gline* value as *gpoints* value. The time complexity of the operation is $O(r)$.

3.3.5 Bounding Boxes

In network environment we know two different types of bounding boxes. On the one hand the spatial and spatio-temporal bounding boxes (see Table 3.8) as they are known from the data model of [6] for spatial and spatio-temporal data types. And on the other hand network and network-temporal bounding boxes (see Table 3.9), which abuse the route identifiers and positions on the routes as x- and y-coordinates of the "network bounding rectangle".

[17] The adjacent sections are the directed sections which can be reached from the query section passing it in *Up* respectively *Down* direction. The reverse adjacent sections are the directed sections which must be passed to reach the query section.

[18] *Bounding gpoints* define the network positions, which must be passed by everyone who wants to reach the inside of the *gline* from the outside of the *gline* and vice versa. We are interested in these points, because they can be used to reduce the complexity of shortest path and Network Distance computing between *gline* values.

3.3.5.1 Spatial and Spatio-Temporal Bounding Boxes

Signature	Example Call
network → *rect2*	**netbbox**(*network*)
ugpoint → *rect3*	**unitboundingbox**(*ugpoint*)
mgpoint → *rect3*	**mgpbbox**(*mgpoint*)

Table 3.8: Operators Generating Spatial- and Spatio-Temporal Bounding Boxes

The operation **netbbox** returns the spatial bounding box of the *network* value. This is the same as the bounding box of the R-Tree of the *routes relation* of *network* and can be returned in O(1) time.

The operations **unitboundingbox** and **mgpbbox** return the spatio-temporal bounding boxes of *ugpoint* respectively *mgpoint* values as three dimensional rectangles with coordinates:

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

The spatial part of the unit bounding box is defined as the spatial bounding box of the *route interval* passed by the *ugpoint* value. The temporal part (z-coordinates) of the unit bounding box is given by the *real* values representing the start and the end time instant of the time interval of the *ugpoint* value.

To compute the spatio-temporal bounding box of a *ugpoint* the operation **unitboundingbox** needs access to the *half segments* of the corresponding route curve. It uses the B-Tree index of the *routes relation* of the *network* the *ugpoint* belongs to, to get the route curve in $O(\log r_{net})$ time. The worst case time complexity for **unitboundingbox** is $O(h + \log r_{net})$.

The spatio-temporal bounding box of a *mgpoint* value is defined by the union of the spatio-temporal bounding boxes of its *ugpoints*. The simple computation would take $O(m \log r_{net} + \sum_{i=1}^{m} h_i)$ time, which is very expensive. As mentioned before in Section 3.2.5 we introduced the parameter *bbox* to the *mgpoint* to store the spatio-temporal bounding box of an *mgpoint* value, if it has been computed once, until the *mgpoint* value changes. Otherwise we use the *trajectory* parameter of the *mgpoint* value to get the spatial part of the bounding box in much less time.

Algorithm 7 mgpbbox(*mgpoint*)

Require: *mgpoint*(*mgpoint*)
1: **if** *bbox* exists **then**
2: **return** *bbox*
3: **else**
4: **if** *trajectory* is not defined **then**
5: **trajectory**(*mpoint*)
6: **end if**
7: Compute union of *bounding boxes* of *route intervals* in *trajectory*
8: Extend the resulting box by start and end time of the *mgpoint*
9: **return** *bbox*
10: **end if**

The single steps of **mgpbbox** (see Algorithm 7) have a time complexity of:

- line 1: O(1)
- line 2: O(1)
- line 4: O(1)
- line 5: $O(m \log r_{out} + r_{out})$

- line 7: O($\sum_{i=1}^{r_{out}} h_{r_i}$)
- line 8: O(1)
- line 9: O(1)

In the worst case we get a time complexity of $O(m \log r_{out} + \sum_{i=1}^{r_{out}} h_{r_i})$, which is still better than the simple version, because $r_{out} \ll m \wedge r_{out} \ll r_{net}$.

3.3.5.2 Network and Network-Temporal Bounding Boxes

A network(-temporal) bounding box is a two (three) dimensional rectangle with coordinates (x_1, x_2, y_1, y_2) respectively $(x_1, x_2, y_1, y_2, z_1, z_2)$, where the both x-coordinates are defined by the route identifier and are always identical, the y-coordinates are given by the positions on the route, and the z-coordinates are defined as *real* values representing the start (z_1) respectively the end (z_2) time instant of the time interval of the *ugpoint*.

Signature	Example Call
gpoint → *rect*	**gpoint2rect**(*gpoint*)
gline → *stream*(*rect*)	**routeintervals**(*gline*)
ugpoint → *rect3*	**unitbox**(*ugpoint*)
ugpoint → *rect*	**unitbox2**(*ugpoint*)

Table 3.9: Operators Generating Network- and Network-Temporal Bounding Boxes

The operation **gpoint2rect** computes the network bounding box of a *gpoint* value in O(1) time. The y-coordinates are defined as $y_1 = position - 0.000001$ respectively $y_2 = position + 0.000001$. The small *real* value is added to avoid problems with the computational inaccuracy of *real* values especially at route start and end points.

The operation **routeintervals** returns a stream of network bounding boxes, one for each *route interval* of the *gline* value. The y-coordinates are defined to be $y_1 = \min$(start position, end position) and $y_2 = \max$(start position, end position). The operation needs O(r) time.

The operation **unitbox2** returns the network bounding box of the *ugpoint* value in O(1) time. The y-coordinates are given by $y_1 = \min$(start position, end position) and $y_2 = \max$(start position, end position).

The operation **unitbox** returns the network-temporal bounding box of the *ugpoint* value in O(1) time. The operation extends the two dimensional rectangle of **unitbox2** with the z-coordinates defined by the *real* values of the start and end time instants of the *ugpoint*.

3.3.6 Property Tests

The operations in this section (see Table 3.10[19]) check if the arguments fulfill given conditions and return ***true*** if this is the case. A special case is the operation **inside** for *mgpoint*, because the argument and the return value are *moving* data types.

Signature	Example Call
$X \times X \rightarrow$ *bool*	$a = b$
$Y \times Y \rightarrow$ *bool*	**intersects**(a, b)
mgpoint $\times X \rightarrow$ *bool*	*mgpoint* **passes** b
gpoint $\times$ *gline* → *bool*	a **inside** *gline*
mgpoint $\times$ *gline* → *mbool*	*mgpoint* **inside** *gline*
mgpoint $\times T \rightarrow$ *bool*	*mgpoint* **present** b

Table 3.10: Operators Checking Properties

The operation = compares the arguments and returns ***true*** if they are equal, ***false*** otherwise. For two *gpoint* values the check is done in O(1) time. For two *gline* values we have to compare all *route intervals* of both *gline* values in the positiv case, in the negativ case ***false*** is returned immediately if a difference between

[19] The data type of the first parameter must be equal to the data type of the second parameter if X or Y occur twice in a signature.

the two *gline* values is detected to save computation time. If both *gline* are sorted the comparison of the two sets of *route intervals* needs $O(r)$ time. If only one *gline* value is sorted we need $O(r \log r)$ time. If none of the *gline* values is sorted we need $O(r^2)$ time.

The operation **intersects** checks if the two arguments intersect respectively meet at least at one position in the network.

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

1. If both *gline* values are sorted, a parallel scan through the route intervals of both *gline* values is performed in $O(r_1 + r_2)$ time.

2. If only one *gline* value is sorted, a linear scan of the unsorted *gline* is performed. For each *route interval* of the unsorted *gline* a binary search for an overlapping *route interval* is performed on the *route intervals* of the sorted *gline* value. This takes $O(r_i \log r_j)$ time, with $i, j \in \{1, 2\}, i \neq j$.

3. If both *gline* values are not sorted, a linear scan of the first *gline* value is performed, and for each *route interval* a linear scan for overlapping *route intervals* on the second *gline* value is performed in $O(r_1 r_2)$.

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

For two *mgpoint* values the algorithm works almost analogous to **intersection** (see 3.3.11) but if two intersecting units are found ***true*** is returned immediately. In the worst case that no intersection is found the time complexity is equal to the time complexity of **intersection**.

The operation **passes** checks if the *mgpoint* ever passes the given network position (*gpoint*) or part (*gline*). The algorithm uses the *trajectory* parameter of the *mgpoint*. If the trajectory is not defined the trajectory is first computed using **trajectory**(*mgpoint*). In this case we must add the time complexity **trajectory** to the time complexity of **passes**. In the sequel we assume that the trajectory is already defined.

If the second argument is a *gpoint* a binary search for a *route interval* that contains the *gpoint* is performed on the *trajectory* parameter. This will take $O(\log r)$ time.

If the second argument is a *gline* the algorithm distinguishes two cases:

1. If the *gline* is sorted a parallel scan of the *route intervals* of the *gline* value and the *trajectory* of the *mgpoint* is performed to detect an intersecting pair of *route intervals* one from *gline* and one from *mgpoint*. The time complexity is $O(r_{mgpoint} + r_{gline})$

2. If the *gline* is not sorted a linear scan of the *route intervals* of the *gline* value is performed. And for every route interval a binary search for an intersecting *route interval* is performed on the *trajectory* of the *mgpoint*. The time complexity is $O(r_{gline} \log r_{mgpoint})$.

In both cases ***true*** is returned immediately if a pair of intersecting *route intervals* is detected.

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

If the second argument is a *gpoint* the algorithm distinguishes two cases:

1. If the *gline* is sorted a binary search for a *route interval* containing the *gpoint* is performed in $O(\log r)$ time.

2. If the *gline* is not sorted a linear scan of the *route intervals* of the *gline* is performed to find a route interval containing the *gpoint* in $O(r)$ time.

If the second argument is an *mgpoint* the result of **inside** *gline* is an *mbool*[20] is returned. The *mbool* is ***true*** every time interval the *mgpoint* moves inside the *gline* and ***false*** for the other time intervals of the definition time of the *mgpoint*. The algorithm checks for every unit of the *mgpoint* if there is any intersection with the *route intervals* of the *gline*. Based on these values the resulting *mbool* is computed. Again we distinguish between sorted and unsorted *gline*

- If the *gline* is sorted a binary search on the *route intervals* is performed and the operation takes $O(m \log r)$ time.

- If the *gline* is not sorted a linear scan on the *route intervals* is performed and the operation takes $O(mr)$ time.

[20] (Short form of *moving*(*bool*)). An *mbool* value changes its value within time. See [6] for more details.

The operation **present** checks the time intervals of the *mgpoint* value and returns ***true*** if the *mgpoint* value is defined at the given time *instant* or at least a part of the given *period* of time.

If the second argument is a *instant* value the algorithm performs a binary search on the units of the *mgpoint* for the given time instant in $O(\log m)$ time.

If the second argument is a *periods* value the algorithm performs a parallel scan of the units of the *mgpoint* value and the units of the *periods* value and returns ***true*** if an intersecting time interval is found. The worst case time complexity is $O(m + p)$.

3.3.7 Merging Data Objects

$A \times A \to A$ a **union** b

The operation **union** merges two argument objects into one result object of the same data type in the same network.

Algorithm 8 describes the operation for two *gline* values. The time complexity is $O(r_1 + r_2)$ if both *gline* values are sorted and $O((r_1 + r_2) \log r_{out})$ in all other cases.

Algorithm 8 union(*gline*1, *gline*2)

Require: *gline1*, *gline2* of data type *gline*
1: **if** Both *gline* are sorted **then**
2: Perform parallel scan of the *route intervals* of both *gline*
3: **if** current pair of *route intervals* intersect **then**
4: merge *route intervals* into one
5: **if** upcoming *route intervals* intersect the resulting *route interval* **then**
6: extend merged *route interval*
7: **end if**
8: Add merged *route interval* to *result* and continue scan
9: **else**
10: Add smaller *route interval* to resulting *gline* and continue scan
11: **end if**
12: **else**
13: Fill the *route intervals* of both *gline* in a common RITree to compute resulting *gline*
14: **end if**
15: **return** resulting *gline*

If we do not want to store the resulting *gline* sorted we could simply add every *route interval* of both *gline* values into the new *gline* in $O(r_1 + r_2)$ time. As mentioned before in Section 3.2.4, we think that the additional time for merging and sorting is well invested at this point.

For two *mgpoint* the algorithm performs a parallel scan through the units of both *mgpoint* and writes the units of the *mgpoint*s in ascending order of their time intervals to the resulting *mgpoint*. If there are overlapping time intervals the algorithm checks, if both *ugpoint* values define the same places for the same times. If this is the case one of them is written to the result and the other one is ignored. If the *ugpoint* values are different the computation stops immediately and the result is marked to be undefined. The worst case time complexity of this operation is $O(m_1 + m_2)$.

3.3.8 Path Computing

$X \times X \to$ *gline* **shortest_path**(a, b)
$D \times D \to$ *gline* **shortest_pathastar**(a, b)

The operation **shortest_path** computes the shortest path between the arguments using Dijkstras Algorithm of shortest paths [3], whereas the operation **shortest_pathastar** uses the A^*-Algorithm [12] for the computation. Because of the bad performance of Dijkstras Algorithm **shortest_path** is only implemented for pairs of *gpoint* and *gline*, whereas **shortest_pathastar** supports all possible combinations of static network data types.

For two *gpoint* values the worst case time complexity of **shortest_path** is $O(s_{net} + j_{net} \log j_{net})$ see [3].

For two *gline* values the operation **shortest_path** computes first the *bounding gpoint* values of both *gline* values using **getBGP** (see Section 3.3.4) followed by the computation of the shortest paths for each possible pair of bounding gpoint values one from *gline1* and one from *gline2* using Dijkstras Algorithm of shortest

paths. The shortest detected path is returned as result value. The time complexity for this operation is $O(r_1 + r_2 + b_1 b_2 (s_{net} + j_{net} \log j_{net}))$.

Different from operation **shortest_path** the operation **shortest_pathastar** uses the A^*-Variant of Dijkstras Algorithm. The A^*-Algorithm is known to have better run times in almost all cases than Dijkstras Algorithm, because it touches fewer sections than Dijkstras Algorithm. This can be checked in SECONDO comparing the results of the operation **spsearchvisited** (described below) for both algorithms.

All possible combinations of static network data types are reduced to compute the shortest path with A^*-Algorithm between two sets of *gpoints*. This is done in O(1) for *gpoint* and *gpoints* values. In case of a *gline* value the *bounding gpoints* are computed using the operation **getBGP** before the shortest path A^*-Algorithm for two *gpoints* values is used.

Our variant of the A^*-Algorithm for two sets of *gpoints* does not compute the shortest path for all pairs of *gpoint* one from *gpoints1* and one from *gpoints2*. It initializes the priority queue for A^* with all *gpoint* of *gpoints1* and uses this single priority queue to compute the shortest path to the destination set *gpoints2*. The computation stops, if the first time a *gpoint* from *gpoints2* is the minimum of the priority queue. A few additional operations ensure that the path we found first is really the shortest path between the two position sets. In our experiments we compared the brute force attempt of **shortest_path** with our **shortest_pathastar** variant. The speed up for two sets of network positions is enormous. We measured a run time of 0.17 seconds for **shortest_pathastar** computation, while the brute force attempt took 4.199 seconds in the middle for the same input and result.

As mentioned before we introduced an operation **spsearchvisited** to enable the user to compare the different shortest path algorithms. The operation has the signature: $D \times D \times \underline{bool} \rightarrow \underline{stream}(\underline{tuple}(T_{sec}))$. The result is a stream with the tuples of the internal section table, which have been visited within the shortest path search. If both arguments are of data type *gpoint* respectively *gline* the Boolean value can be used to select if Dijkstras Algorithm (***true***) or the A^*-Algorithm (***false***) should be performed for shortest path search. For all other combinations of argument data types only the A^*-Algorithm is implemented, such that only ***false*** is possible as Boolean input value. The time complexity of **spsearchvisited** is analogous to the used Algorithm for path computation.

The operation **shortestpathtree** was introduced to support network distance computation between a static *gpoint* and an *mgpoint* value. The signature of the operation is $\underline{gpoint} \times \underline{network} \rightarrow \underline{stream}(\underline{tuple}(\text{sid } \underline{int}, \text{distance } \underline{real}, \text{direction } \underline{bool}))$. The operation computes the shortest path from one source *gpoint* to all other places in the network using Dijkstras-Algorithm in $O(s_{net} + j_{net} \log j_{net})$ time. It returns a stream of tuples. Each tuple represents a network section in the shortest path tree of the network with origin *gpoint*. The sections are represented by their identifier, their distance from the source *gpoint* value and a Boolean flag telling, if the section is passed in *Up* or *Down* direction within the shortest path tree.

3.3.9 Distance Computing

There is a big difference between the Euclidean Distance and the Network Distance of two places a and b. The Euclidean Distance is given by the length of the beeline between the two places regardless from existing paths in the network between the two locations and it is the same equal if we estimate the distance from a to b or form b to a. On the contrary the Network Distance is given by the length of the shortest path between a and b in the network. According to this, and contrary to the Euclidean Distance, the Network Distance from a to b might be another than the Network Distance from b to a. Because there might be one way routes in the shortest path from a to b, which cannot be used in the shortest path from b to a or vice verse.

3.3.9.1 Euclidean Distances

$X \times X \rightarrow \underline{real}$	**distance**(a, b)
$\underline{mgpoint} \times \underline{mgpoint} \rightarrow \underline{mreal}$[21]	**distance**(*mgpoint1*, *mgpoint2*)

Although Euclidean Distances do not make much sense in a network environment, we implemented the **distance** operation which computes the Euclidean Distance between two *gpoint*, *gline*, or *mgpoint* values for network objects for convenience.

All following algorithms for Euclidean Distance computing do first a translation of the network data types into equivalent two dimensional data types using the operators of Section 3.3.3 before they use the existing distance operation of this equivalent two dimensional data types to compute the Euclidean Distance between the network data types. Therefore the time complexity is always given by the sum of the translation time and the time for the distance computation. In all cases the translation time dominates the time for distance

[21] We have moving data types as arguments such that the result is also a moving data type. See [6] for detailed explanation of *mreal*.

computing, such that the time complexity for the distance computation between two network objects is the same than for the translation of the network objects.

3.3.9.2 Network Distances

As mentioned before the Network Distance is given by the length of the shortest path between the two arguments, whereas the first argument is the source and the second argument the target of the shortest path. This definition is necessary, because the shortest path from the first to the second argument may differ from the shortest path from the second to the first argument.

We distinguish between two cases. The first simpler case computes Network Distances only between static network positions. The result is, in this case, always a single *real* value. In the second case we estimate Network Distances where at least one of the data objects is an *mgpoint*. In this cases the result is an *mreal*.

For the static case analogous to the shortest path computation we know to different operators **netdistance** and **netdistancenew**.

3.3.9.2.1 Static Network Positions

$X \times X \rightarrow \underline{real}$ **netdistance**(a, b)
$D \times D \rightarrow \underline{gline}$ **netdistancenew**(a, b)

The operator **netdistance** uses the Dijkstras Algorithm implemented in operation **shortest_path**, whereas the operator **netdistancenew** uses the A^*-Algorithm version as described in Section 3.3.8 for operation **shortest_pathastar**. The time complexity of the operations is dominated by the shortest path computation, such that the time complexity of the operation is given by the shortest path algorithm used. Algorithm 9 computes the minimum Network distance between two *gline* values.

Algorithm 9 netdistance($gline1$,$gline2$)

```
1: BGP1 = getBGP(gline1)
2: BGP2 = getBGP(gline2)
3: minDist = ∞
4: for Each pair of p1 ∈ BGP1 and p2 ∈ BGP2 do
5:    if p1 inside gline2 ∨ p2 inside gline1 then
6:       return 0.0
7:    else
8:       actDist = length(shortestpath(p1,p2))
9:       if actDist < minDist then
10:         minDist = actDist
11:      end if
12:   end if
13: end for
14: return minDist
```

3.3.9.2.2 Moving Network Positions

$Z \times Z \rightarrow \underline{ureal}$ **netdistance**(a, b)
$Y \times Y \rightarrow \underline{mreal}$ **netdistance** or **netdistancenew**(a, b)

The computation of moving Network Distances is not exact yet. Exact computation would take much more computation time and very complex operations, because the shortest path from object a to object b may change more than one time within one unit completely such that we would get more than one shortest path and resulting unit per unit.

The operation **netdistance** uses the A^*-Algorithm to estimate the Network Distance. The initial distance value for the resulting *ureal* value of the **netdistance** operation between a *gpoint* and a *ugpoint* is defined by the Network Distance between the *gpoint* and the start position of the *ugpoint* respectively vice verse. The end distance value of the result is computed by addition respectively subtraction of the distance between start and end position of *ugpoint* to start distance value[22]. The start and end values are used to compute the parameters of a *ureal* value representing the development of the distance value in the time interval defined by the *ugpoint*.

[22] If the length of the *ugpoint* values is added or subtracted depends on the direction of the movement of the *ugpoint* value relative to the shortest path.

For the Network Distance between two *ugpoint* values the start distance value is the Network Distance between the two start positions of the both *ugpoint* values. The end value is estimated by adding/subtracting the lengths of both *ugpoint* values almost analogous to the *gpoint* to *ugpoint* case.

To compute the Network Distance between a *gpoint* and a *mgpoint* or vice verse we compute first the (reverse) shortest path tree of the *gpoint*. After that we do a linear scan of the units of the *mgpoint* and compute the sections passed by this unit. We get the length of the shortest path from/to the start and end of this unit to the *gpoint* by a look up of the results of the shortest path tree computation. These length values are used to compute corresponding *ureal* values for the resulting *mreal* value representing the moving network distance between the moving and the static network position.

In all cases the time complexity is dominated by the time complexity of shortest path respectively shortest path tree computation.

The operation **netdistancenew** tries to save computation time by first estimating the set of sections ever passed by the *mgpoint* and then stopping the (reverse) shortest path tree computation when all passed sections have been reached. The rest of the algorithm is analogous to **netdistance** for *mgpoint* and *gpoint* values. The main difference is the time complexity of the (reverse) shortest path tree computation. Depending on the movement and the distance of the movement of the *mgpoint* relatively to the *gpoint* the average computation time is much better than in case of **netdistance** for the same query objects.

3.3.10 Network Part Around a Single Network Position

The three operations **circlen**, **in_circlen** and **out_circlen** with signature $\underline{gpoint} \times \underline{real} \rightarrow \underline{gline}$ and syntax **op**(*gpoint*, *dist*) return a *gline* value. In case of **out_circlen** the *gline* represents the parts of the network around the given *gpoint* which can be reached within the Network Distance given by *dist* from the *gpoint*. In case of **in_circlen** the *gline* represents the parts of the network from which the *gpoint* can be reached within *dist*. And **circlen** returns the union of the results of **out_circlen** and **in_circlen**.

The values are computed by building the (reverse) shortest path tree of *gpoint* until the given distance is smaller than the next distance coming from the priority queue.

3.3.11 Restricting Single Moving Network Positions

Signature	Example Call
$\underline{mgpoint} \times \underline{instant} \rightarrow \underline{igpoint}$	*mgpoint* **atinstant** *periods*
$\underline{mgpoint} \times \underline{periods} \rightarrow \underline{mgpoint}$	*mgpoint* **atperiods** *periods*
$\underline{mgpoint} \times X \rightarrow \underline{mgpoint}$	*mgpoint* **at** *a*
$\underline{mgpoint} \times \underline{mgpoint} \rightarrow \underline{mgpoint}$	**intersection**(*mgpoint1*, *mgpoint2*)
$\underline{mgpoint} \times \underline{real} \rightarrow \underline{mgpoint}$	**simplify**(*mgpoint*, *real*)

Table 3.11: Operators Restricting Single Moving Network Positions

The operations in this section (see Table 3.11) restrict *mgpoint* values to given times or places or reduce the number of units of the *mgpoint*.

The operation **atinstant** restricts the *mgpoint* to the given time instant. It performs a binary search on the units of the *mgpoint* to find the unit containing the given time instant. If a corresponding unit is found the result is computed and returned, otherwise an *undefined igpoint* value is returned. The time complexity of the operation is $O(\log m)$.

The operation **atperiods** restricts the *mgpoint* to the given *periods* of time. It performs a parallel scan of the *periods* and the *mgpoint* value. The (parts) of units which are inside the *periods* value are written to the resulting *mgpoint*. The time complexity is $O(m + p)$.

The operation **at** restricts the *mgpoint* to the times and places given as *gpoint* or *gline* value.

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

For a *gline* value the operation **at** performs a linear scan on the units of the *mgpoint*. For each unit of the *mgpoint* we check if it passes any *route interval* of the *gline*. For sorted *gline* values a binary search on the *route intervals* and for unsorted *gline* values a linear scan of the *route intervals* is performed. If the *route interval* and the unit of the *mgpoint* intersect the times and places of the intersection are computed as *ugpoint* value and added to the resulting *mgpoint*. The time complexity of the operation is $O(m \log r)$ for sorted and $O(mr)$ for unsorted *gline* values.

The operation **intersection** returns the times and places where both argument *mgpoints* have been at the same time. The algorithm first computes the refinement partitions[23] of the both *mgpoint*. Then it performs a parallel scan through the refinement partitions of both *mgpoint* and checks for every pair of units if there is an intersection. If this is the case a *ugpoint* with the intersection value is computed and written to the resulting *mgpoint*. Let s be the number of units of the refinement partitions. The time complexity of the algorithm is $O(m_1 + m_2 + s)$.

The operation **simplify** reduces the number of units of the *mgpoint*, by merging the units, where the *mgpoint* moves on the same route, in the same direction, and the speed difference between two units is smaller as defined by the given *real* value. The operation performs a linear scan on the units of the *mgpoint* and checks the condition for every unit. This takes $O(m)$ time. Detailed information about the simplification can be found in [14].

3.3.12 Create Moving Value From Single Unit

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

The operation **ugpoint2mgpoint** constructs a *mgpoint* value from an single *ugpoint* value in O(1) time.

3.3.13 Static Network Position Values of Junctions

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

A problem of the network dat/a model is that junctions belong to more than one route. That means, different from the spatial case, one junction has more than one network representation. Operators like **passes** or **inside** do not check if the query *gpoint* is a junction and probably has more than one representation, because the interpretation of passing a network junction in [10] is slightly different from passing a *point* in the two dimensional space. So if, for example, an *mgpoint* passes a junction on the one route and the *gpoint* representing the junction is given related to another route we get ***false*** as result. This is correct in the network data model but does not fit to the **passes** interpretation of the data model of free movement in two dimensional space.

We introduced the operation **polygpoints** to bypass this problem in the BerlinMOD Benchmark [1]. This operation returns for every given *gpoint* a *stream* of *gpoint*. This stream contains only the *gpoint* itself if the *gpoint* is not a junction, and the *gpoint* itself and all aliases for this *gpoint* representing the same place in space if the *gpoint* is a junction.

The algorithm **polygpoints** first copies the argument *gpoint* to the output stream in O(1) time. Then it checks if the *gpoint* represents a junction by selecting all junctions from the junctions relation which are on the route with the route identifier of the *gpoint* with help of the junctions relation B-Tree in $O(\log j_{net} + k)$ time. This k junctions are checked if they are identified by the *gpoint*. In the worst case this takes $O(k)$ time. If this is the case all other *gpoint* values identifying the same junction on other routes are returned in the output stream. The complete algorithm has a worst case complexity of $O(\log j_{net} + k)$.

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

Chapter 4

JNetwork Implementation

4.1 Introduction

As mentioned before we provide two implementations of the network data model of [10] in SECONDO DBMS. The second implementation was developed because the tests with the first implementation of the network data model (see Chapter 3) in SECONDO showed that our expectations are fulfilled. The network data model needs less storage space than the spatial data model implemented in SECONDO; and the query run times compared with the BerlinMOD Benchmark are also much better for the network data model. But within the same tests we detected several problems of the first network implementation:

- The number of available sections is limited by the main memory at network creation time, such that we could not import bigger amounts of street data as single network object.
- The side value in *RouteInterval* was not implemented and could not be added easily because the implementation is not really well object oriented, such that the maintenance of the code becomes very expensive and difficult. For the extension of the *RouteInterval* we would have to change thousand lines of code at every place the *RouteInterval* occurs.
- Some data information is stored twice for example in the network relations the spatial route curve is stored once in the routes and once in the sections relation, this could be reduced to save storage space without loss of information.
- The idea to use the implemented generic mapping functions directly does not work for moving network data types, such that we could save data space by using one route interval for the spatial part of each moving unit instead of two network positions which consists except of the position value of the same data content.

Based on these experiences we implemented the network data model a second time in an improved version which supports the missing functionality, saves more storage space and has at least the same or better query run times (see Figure 5.1) than the first network implementation described in Chapter 3. Different from the first implementation the static and the temporal data types and the operations on this data types are provided in one single SECONDO algebra module called `JNetAlgebra`.

Analogous to the description of the first implementation we describe the data types of the second implementation in Section 4.2 and the operations on these data types in Section 4.3.

4.2 Implemented Data Types

Beside the central jnetwork object (see Section 4.2.2) and the jnetwork depending data types (see Section 4.2.3) we introduced some helpful other SECONDO attribute data types (see Section 4.2.1). These additional data types are used as part of network data types and as input to create jnetworks and jnetwork dependent objects.

All data types described in the sequel have an additional Boolean flag telling if the current data type value is well defined or not. We will not mention this flag again at every data type.

4.2.1 Basic Data Types

The data type *jdirection* encapsulates the enumeration of side values *Up*, *Down* and *Both*[1] to be useable as attribute data type into relations. It is used in the other data types to realize the side value as described in [10].

[1] We found it more logical to say a position is reachable from both sides of a road than to use *none* for the same sense, like it is done in the original abstract data model and in the first implementation.

The definition of the data type *rloc* (short form of route location) is based on the definition of the abstract data type *RLoc* in [10]. It consists of a route identifier (*int*), the distance of the location from the origin of the route curve (*real*) following the route curve, and a side value (*jdirection*) describing from which side of the road the location can be reached on the route.

The data type *jrint* (short form of route interval in this second network data model implementation) is based on the abstract *route interval* definition of [10]. It consists of a route identifier (*int*), two distance values (*real*), and a side value (*jdirection*). The two distance values describe the route part covered by the *jrint*. The first distance value is always smaller or equal to the second distance value. This is possible because the side value tells, depending on the usage in which direction the route part is used respectively, at which side of the road the route part is allocated.

The data type *ndg* (short for net distance group) consists of four *int* values and one *real* value. The four integer values identify the source junction, the target junction, the next junction, and the next section on the path from the source to the target junction by their identifiers. The *real* value describes the Network Distance (length of the shortest path) from the source junction to the target junction in the network.

The data type *junit* consists of a time interval[2] and a *jrint* value. Telling on which route the car drives in which direction from which position to which position in the given time interval at the same speed. It is used within the data types *mjpoint* and *ujpoint* to describe the movement in a single time interval. We can use this information to compute the exact position of the car at any time instant within the time interval. If the car passes a given route location within this *junit* the exact time the car reaches the given route location can be computed.

The next data types *listint*, *listrloc*, *listjrint*, and *listndg* organize sets of *int*, *rloc*, *jrint*, and *ndg* as sorted main memory independent lists. These sorted lists enable us to perform binary searches for values in the set that is not limited by the available main memory.

4.2.2 JNetwork

The central network data type of the JNet Implementation is the data type *jnet* which consists of:

- The database name of the network object as *string* value[3]
- A *real* value storing the allowed deviation for map matching algorithms[4]
- Relation with junctions data (see Table 4.1)
- Relation with routes data (see Table 4.3)
- Relation with sections data (see Table 4.2)
- Ordered Relation with network distance data (see Table 4.4)
- R-Tree for the section relation indexing the *Curve* attribute
- R-Tree for the junctions relation indexing the *Pos* attribute
- Three B-Trees indexing the identifiers of the junctions, routes and sections in the corresponding relations.

Attribute	Data Type	Explanation
Id	*int*	unique junction identifier
Pos	*point*	spatial position of junction in two dimensional space
ListJuncPos	*listrloc*	list of network positions of this junction[5]
ListInSections	*listint*	list of section identifiers from which the junction can be reached
ListOutSections	*listint*	list of section identifiers over which the junction can be leaved

Table 4.1: Junctions Relation of Data Type JNetwork

[2] See Section 3.2.5 for detailed information about time interval definition in SECONDO.

[3] We use the database name of the network object as *string* value instead of the *int* identifier used in the first implementation to enable a faster access to the *jnet* object for the network dependent objects.

[4] The sense of this value is manly the same than explained for the *scalefactor* attribute of *network* in the first implementation see Footnote 15 on Page 10.

[5] One spatial position has different network locations if the position is at a junction. See Section 3.3.13 for detailed problem description.

Attribute	Data Type	Explanation
Id	*int*	unique section identifier
Curve	*sline*	spatial curve representing the section curve in two dimensional space
StartJunctionId	*int*	junction identifier of the junction at the start point of the section route curve
EndJunctionId	*int*	junction identifier of the junction at the end point of the section route curve
Direction	*jdirection*	tells in which direction(s) the section can be used
VMax	*real*	maximum allowed speed on this section
Length	*real*	length of the section curve
ListSectRouteIntervals	*listjrint*	list of route intervals represented by this section[6]
ListAdjSectionsUp	*listint*	list of adjacent sections driving section up
ListAdjSectionsDown	*listint*	list of adjacent sections driving section down
ListRevAdjSectionsUp	*listint*	list of reverse adjacent sections for driving section in up direction.
ListRevAdjSectionsDown	*listint*	list of reverse adjacent sections for driving section in down direction.

Table 4.2: Sections Relation of Data Type JNetwork

Attribute	Data Type	Explanation
Id	*int*	unique route identifier
ListJunctions	*listint*	list of identifiers of junctions belonging to this route
ListSections	*listint*	list of identifiers of sections belonging to this route
Length	*real*	total length of route curve

Table 4.3: Routes Relation of Data Type JNetwork

Attribute	Data Type	Explanation
Source	*int*	junction identifier of source junction
Target	*int*	identifier of target junction
NextJunction	*int*	identifier of next junction in path from source to target
NextSection	*int*	identifier of next section on path from source to target
NetworkDistance	*real*	Network Distance from source to target junction

Table 4.4: Network Distance Relation of Data Type JNetwork

4.2.3 JNetwork Dependent Data Types

The most jnetwork dependent data types extend just one of the basic data types with a jnetwork identifier (*string*) connecting the described value with an existing *jnet* object in the database. For jnetwork dependent objects we check at creation if the jnetwork object the data type should be related to exists and if the given location exists in the connected jnetwork object.

We use the *string* with the database name of the jnetwork object as connection instead of an *int* value like it is done in the first implementation, because this speeds up the access to the *jnet* object in the operations that need access to information stored in the jnetwork object. We pay for this advantage with little additional time needed to compare two *string* values compared with the time needed to compare two *int* values. We think that this time is well invested.

The data type *jpoint* describes a single static position in the jnetwork by connecting a route location (*rloc*) to a jnetwork identifier.

[6] One section curve may be part of different roads. See Footnote 4 on Page 7 for explanation.

The data type *jpoints* connects the jnetwork identifier with a sorted set of *rloc* values. The route locations are sorted by route identifier, side value and the distance from the origin of the route. This enables us to perform a binary search on the set to find a given position in the set. An value of this data type may be used to represent the addresses of all butchers in the town.

Different from the first network implementation we provide two different data types (*jline* and *jpath*) representing network parts in the second network implementation. Both data types consist of the jnetwork identifier and a set of route intervals (*jrint*). The difference is the sequence the set of route intervals is stored. The data type *jpath* describes a path in the jnetwork and the route intervals are stored sorted by their usage in the path from the first point to the last point of the path. The data type *jline* describes arbitrary parts of the jnetwork and the route intervals are stored sorted by route identifier, side value and shorter distance from the origin of the route curve. The data type *jline* corresponds to the sorted *gline* of the first implementation, whereas the *jpath* is more like the unsorted *gline*. Using two different data types at this place frees us from handling different cases for sorted and unsorted values in each operation dealing with network parts. If needed *jpath* values can be easily translated into *jline* values using **tojline** operation as described in Section 4.3.15.

The data type *mjpoint* represents the history of movement of a single position (for example one car) in the jnetwork. It consists of the jnetwork identifier, a set of *JUnit* values with not overlapping time intervals, a set of *jrint* describing the network part ever visited by an *mjpoint* value, and the total length of the driven distances (*real*). The set of *junit* values is stored sorted by ascending time intervals[7].

The data type *ujpoint* consists of a jnetwork identifier and a single *junit* value. It describes the movement in this time interval.

The data type *ijpoint* consists of a time instant and a *jpoint*. It describes the position of a moving jnetwork position at the given time instant.

4.3 Implemented Operations

In the sequel we describe the implemented operations in the second implementation of the network data model. Analogous to the chapter before we present for each operator its signature, an example call and information about the used algorithms and if interesting the time complexity of the algorithms.

As before we define some abbreviations for the signatures and time complexity descriptions:

- j_{jnet} is the number of junctions in the *junctions relation* of a *jnet* value
- r_{jnet} is the number of routes in *routes relation* of a *jnet* value
- s_{jnet} is the number of sections in *sections relation* of a *jnet* value
- a is the number of sections of the route, and a_j the number of route intervals of section j
- c is the number of candidate sections a *point* value may be mapped to.
- e is the number of elements in a stream, and e_j is the number of list elements of the j-th stream element if the stream element is a list data type
- h is the number of *HalfSegments* in a *line* or *sline* value
- l is the number of list elements in a list data type
- m is the number of units of a *mjpoint* value
- p is number of time intervals in a *periods* value
- r is the number of route intervals of a *jline* value
- t is the number of *jrint* values in the $trajectory$ of a *mjpoint*
- u is the number of units of a *mpoint* value
- $B := \{int, rloc, jrint, ndg\}$
- $ListB := \{listint, listrloc, listjrint, listndg\}$
- $M := \{mjpoint, ujpoint\}$

[7]This implementation does not fit the generic model of moving in [5]. In the generic system of moving the data part should consist of two *jpoint* values instead of a *jrint*. The first network implementation showed that we can not use the implemented generic operators for network data types directly such that we have to do specific network implementation for all operators even if we use the generic data model for moving. We decided to use our implementation to save storage space without loss of information.

- $N := \{\underline{ujpoint}, \underline{jpoint}, \underline{jrint}\}$
- $P := \{\underline{jpoint}, \underline{jpoints}, \underline{jline}\}$
- $S := \{\underline{jpoint}, \underline{jline}\}$
- $T := \{\underline{instant}, \underline{periods}\}$
- $X := \{\underline{jdirection}, \underline{rloc}, \underline{jrint}, \underline{ndg}, \underline{jpoint}, \underline{jline}, ListB\}$
- $Y := \{\underline{mjpoint}, \underline{jline}\}$

4.3.1 Network Creation

The operator **createjnet** creates a single jnetwork object from these five arguments, which are almost analogous to the corresponding attributes in the resulting *jnet* value:

1. object name for the new jnetwork in the database (*string*)
2. tolerance value for map matching (*real*)
3. relation (*rel*) with the junctions data (see Table 4.1)
4. relation (*rel*) with the sections data (see Table 4.2)
5. relation (*rel*) with the routes data (see Table 4.3)

The operation checks if the network identifier is available as object name for the current database. If this is the case, the given network object is created and stored in the database with the given object name and ***true*** is returned, otherwise the object is not created and ***false*** is returned.

The check of the network object name and the insertion of the result object in the database is done in O(1) time. The three relations are copied in $O(r_{jnet} + s_{jnet} + j_{jnet})$ time. The network distance relation of the resulting *jnet* value is initialized in $O(j_{net} \log j_{net})$ time. The three B-Trees and the two R-Trees of the *jnet* value are created in $O(j_{jnet} \log j_{jnet})$, $O(s_{jnet} \log s_{jnet})$, respectively $O(r_{jnet} \log r_{jnet})$ time. The whole jnetwork creation is done in $O(j_{net} \log j_{net})$ time, because $1 \leq r_{jnet} \leq s_{jnet} < j_{jnet} < j_{net} \log j_{jnet}$.

4.3.2 Creation Of Data Types

The operators in Table 4.5 are used to create objects of the given data types from attribute values.

Signature	**Example Call**
$\underline{int} \times \underline{real} \times \underline{jdirection} \rightarrow \underline{rloc}$	**createrloc**(*routeid*, *distance*, *side*)
$\underline{int} \times \underline{real} \times \underline{real} \times \underline{jdirection} \rightarrow \underline{jrint}$	**createrint**(*routeid*, *start*, *end*, *side*)
$\underline{int} \times \underline{int} \times \underline{int} \times \underline{int} \times \underline{real} \rightarrow \underline{ndg}$	**createndg**(*source*, *target*, *nextjunc*, *nextsect*, *distance*)
$\underline{stream}(B) \rightarrow ListB$	*stream* (*a*) **createlist**
$\underline{stream}(ListB) \rightarrow ListB$	*stream* (*a*) **createlist**
$\underline{jnet} \times \underline{rloc} \rightarrow \underline{jpoint}$	**createjpoint**(*jnet*, *rloc*)
$\underline{jnet} \times \underline{listrloc} \rightarrow \underline{jpoints}$	**createjpoints**(*jnet*, *listrloc*)
$\underline{jnet} \times \underline{listjrint} \rightarrow \underline{jline}$	**createjline**(*jnet*, *listjrint*)
$\underline{jpoint} \times \underline{instant} \rightarrow \underline{ijpoint}$	**createijpoint**(*jpoint*, *instant*)
$\underline{jnet} \times \underline{jrint} \times \underline{instant} \times \underline{instant} \times \underline{bool} \times \underline{bool} \rightarrow \underline{ujpoint}$	**createujpoint**(*jnet*, *jrint*, *starttime*, *endtime*. *lc*, *rc*)
$\underline{ujpoint} \rightarrow \underline{mjpoint}$	**createmjpoint**(*ujpoint*)

Table 4.5: Operators Creating Data Types

All basic data types are created in O(1) time from given attribute values.

The list data types are created from a stream of input data types respectively a stream of lists of input data types. The time complexity for list creation is $O(e \log e)$, if the stream elements are simple basic data types and

$O((e + \sum_{j=1}^{e} e_j) \log(e + \sum_{j=1}^{e} e_j))$, if the stream elements are lists of basic data types, because the lists store the elements sorted.

The result of the operators creating network dependent data types is only defined, if the network positions described by the arguments exist in the given network, otherwise the result of the creation process is undefined.

For each *rloc* value and each *jrint* value the operators search the B-Tree of the routes relation if the route identifier is valid and the given distance values are between zero and the route length. The operations **creatjpoint** and **createujpoint** need $O(\log r)$ time. The operations **createjpoints** and **createjline** need $O(l \log r)$ time. Only the operations **createijpoint** and **createmjpoint** need $O(1)$ time, because their arguments have already been checked to exist in the jnetwork.

4.3.3 Translation of 2D Data Types into JNetwork Data Types

The operator **tonetwork** translates spatial and spatio-temporal data types into corresponding jnetwork data types if possible. If there is no corresponding jnetwork position the return value is undefined.

Signature	Example Call
jnet × *point* → *jpoint*	**tonetwork**(*jnet*, *point*)
jnet × *line* → *jline*	**tonetwork**(*jnet*, *line*)
jnet × *mpoint* → *mjpoint*	**tonetwork**(*jnet*, *mpoint*)

Table 4.6: Signatures and Example Calls for Operator **tonetwork**

The algorithm of the operation **tonetwork** for a *point* value works almost analogous to the operation **point2gpoint** described in Algorithm 2. The main difference is that now the R-Tree of the *sections relation* of the *jnet* is used instead of the R-Tree of the *routes relation* of the *network*. The operation needs $O(\log s_{jnet}+c+h)$ time.

The operation **tonetwork** for spatial *line* values computes for both end points of each *HalfSegment* of the *line* value the corresponding route interval using the operation **tonetwork** for *point* values. The time complexity of the operation is $O(h \log s_{jnet} + \sum_{i=0}^{h} c + \sum_{j=0}^{h} h_{c_j})$.

The operation **tonetwork** for *mpoint* is described in Algorithm 10 and sub algorithms. The operation has a worst case time complexity of

$$O(u \log s_{jnet} + \sum_{i=0}^{u} c + \sum_{j=0}^{u} h_{c_j} + \sum_{k=0}^{u} z_k)),$$

whereas z_k is the time needed for trip simulation k. If the units of the *mpoint* can be well matched to *jnet* z_k will be in $O(1)$ in the most cases. If the units of the *mpoint* can not be well matched to *jnet* z_k might become up to $O(s_{jnet} + j_{jnet} \log j_{jnet})$ in worst case because of the A^*-Algorithm of shortest path computing.

Algorithm 10 tonetwork(*jnet*, *mp*)

Require: A *jnetwork* object *jnet* and a *mpoint* value *mp*

1: Linear Scan of *mp* units *u* until first point related to network is found by **tonetwork**(*jnet*, *u.startpoint*) remember *jpoint* value of position as *a* and *starttime*
2: **for** Each remaining unit *u* of *mpoint* **do**
3: Continue scan with *u*.endpoint until second point related to network is found by **tonetwork**(*jnet*, *u.endpoint*) remember *jpoint* value of position as *b* and *endtime*
4: $result+ = simulateTrip(jnet, a, b, starttime, endtime)$
5: $a = b$ und $starttime = endtime$
6: **end for**
7: **return** *result*

4.3.4 Translation of JNetwork Data Types into 2D Data Types

The operator **fromnetwork** translates jnetwork and jnetwork-temporal data types into corresponding spatial and spatio-temporal data types.

Algorithm 11 simulateTrip$(jnet, a, b, starttime, endtime)$

Require: *jnet* (*jnetwork*), *a*, *b* (*rloc*), *starttime*, *endtime* (*instant*)
1: $sp = shortestPath(a, b)$
2: **for** Each route interval $ri \in sp$ **do**
3: Compute relative length compared with length *sp*
4: Compute corresponding time interval
5: Write unit to *result*
6: **end for**
7: **return** *result*

Algorithm 12 shortestPath$(jnet, a, b)$

Require: *jnet* (*jent*), *a*, *b*(*rloc*)
1: **if** *a* and *b* are on the same route and a direct connection exists in *jnet* **then**
2: **return** jrint(*a*,*b*) as path
3: **else**
4: **if** Section of *a* and section of *b* have a common crossing *c* **then**
5: **return** jrint(*a*,*c*), jrint (*c*,*b*) as path
6: **else**
7: **return** Result of A^*-Algorithm for shortest path (*a*, *b*)
8: **end if**
9: **end if**

Signature	Example Call
jpoint → *point*	**fromnetwork**(*point*)
jline → *line*	**fromnetwork**(*jline*)
mjpoint → *mpoint*	**fromnetwork**(*jpoint*)

Table 4.7: Signatures and Example Calls for Operator **fromnetwork**

The operation **fromnetwork** for *jpoint* is described in Algorithm 13. The operation has a time complexity of $O(\log r + c + h)$.

Algorithm 13 fromnetwork(jp)

Require: *jp* (*jpoint*)
1: Search B-Tree of routes relation for route *r* of *jp*
2: **for** Each section *s* of the sectionlist of *r* **do**
3: **if** *s* contains *jp* **then**
4: **return** Spatial position of *jp* on section curve
5: **end if**
6: **end for**

The operation **fromnetwork** for *jline* values is described in Algorithm 14. The time complexity is $O(n \log r + \sum_{i=0}^{n} c + \sum_{j=0}^{n} h_{c_j})$.

The operation **fromnetwork** for *mjpoint* is described in Algorithm 15. The FOR-Loop will be called *u* times. If line 2 is evaluated to ***true*** line 3-6 have a time complexity of $O(\log r + c + h)$. The operation in line 9 has a time complexity of O(1). Line 11+12 have a worst case time complexity of $O(h)$. We get a total time complexity of $O(n \log r + \sum_{i=0}^{n} c + \sum_{j=0}^{n} h_{c_j})$ for the worst case, but computation will be faster in most cases because the route is not changed every unit and the most units are very short such that the most units must not be divided up into pieces.

4.3.5 Extract Attributes

The operators of Table 4.8 return the simple attributes from the different data types in O(1) time, whereas the operators of Table 4.9 return the more complex attributes.

Algorithm 14 fromnetwork(*jline*)

Require: *jline* (*jline*)
1: **for** Each route interval i of *jline* **do**
2: Search B-Tree routes for route r of i
3: **for** Each section s of the sectionlist of r **do**
4: **if** s intersects i **then**
5: Add half segments belonging to the intersection to the *result*
6: **end if**
7: **end for**
8: **end for**
9: **return** *result*

Algorithm 15 fromnetwork(*mjpoint*)

Require: *mjpoint* (*mjpoint*)
1: **for** Each unit u of *mjpoint* **do**
2: **if** u is on another route than unit before **then**
3: Search B-Tree routes for route r of u
4: Compute route curve c from section curves of r
5: Compute spatial pos of *u.startpos* on c
6: Remember position detection values on c
7: **end if**
8: **if** *u.endpos* is on the same *HalfSegment* of c than *u.startpos* **then**
9: Add simple corresponding unit to *result*
10: **else**
11: Split u at the points the passed *HalfSegment* of c changes.
12: add resulting split units to *result*
13: **end if**
14: **end for**
15: **return** *result*

Operator	Signature	Explanation
isempty	$Y \rightarrow$ *bool*	Returns ***true*** if argument is defined and the set of units respectively route intervals is empty.
initial	*mjpoint*→*ijpoint*	Returns start position and time of the *mjpoint* value.
length	*mjpoint* → *real*	Returns the total driven length of an *mjpoint* value.
val	*ijpoint* → *jpoint*	Returns the network position of the *ijpoint* value.
inst	*ijpoint* → *instant*	Returns the time instant of the *ijpoint* value.

Table 4.8: Operators Returning Data Type Attributes in O(1)

Signature	Example Call
jnet → *rel*	**routes**(*jnet*)
	junctions(*jnet*)
	sections(*jnet*)
	distances(*jnet*)
jnet × *int* × *jdirection* → *jlistint*	**getAdjacentSections**(*jnet*, *sectid*, *direction*)
	getReverseAdjacentSections(*jnet*, *sectid*, *direction*)
mjpoint → *jline*	**trajectory**(*mjpoint*)
ListB → *stream*(*B*)	**createstream**(*a*)
mjpoint → *stream*(*ujpoint*)	**units**(*mjpoint*)
jline → *stream*(*jrint*)	**units**(*jline*)
jline → *jpoints*	**getBGP**(*jline*)

Table 4.9: Operators Returning Complex Attributes of Data Types

The operations **junctions** ($O(j_{jnet})$), **sections** ($O(s_{jnet})$), **routes** ($O(r_{jnet})$), and **distances** ($O(j_{jnet}^2)$)[8] return the content of the internal relations of the *jnet* value. The time complexity of the operations depends on the number of entries in the relations.

The operations **getAdjancentSections** and **getReverseAdjacentSections** return the list with the identifiers of the sections which are (reverse) adjacent[9] to the given section in the given direction in $O(l + \log s_{jnet})$ time.

The operation **trajectory** returns the *trajectory* of the *mjpoint* value as *jline* value in $O(r)$ time.

The operator **createstream** returns the elements of a list data type as stream in $O(l)$ time.

The operation **units** returns the units of a *mjpoint* respectively the route intervals of an *jline* as stream of *ujpoint* respectively *jrint* in $O(m)$ respectively $O(r)$ time.

The operation **getBGP** returns the bounding *jpoints*[10] of the given *jline* like described in Algorithm 16.

Algorithm 16 getBGP(*jline*)

Require: *jline*

```
 1: for Each route interval ri of jline do
 2:   Get list of covered sections from jnet of jline for ri
 3:   for Each Covered section cs do
 4:     if End of ri is inside cs then
 5:       Write jpoint for ri end to resulting jpoints
 6:     else
 7:       Get adjacent sections as at cs end points
 8:       if ∃as : as ∉ jline then
 9:         Write jpoint for end point of cs to resulting jpoints
10:       end if
11:     end if
12:   end for
13: end for
14: return jpoints
```

Algorithm 16 has a worst case time complexity of

$$O(r \log r_{jnet} + (l + \log s_{jnet} + l \log r) \sum_{i=1}^{r} c_i) = O(r \log r_{jnet} + (l \log r + \log s_{jnet}) \sum_{i=1}^{r} c_i),$$

because the steps have the following time complexities:

- line 1: The for-loop will be called r times. The operations inside take:
  - line 2: $O(c + \log r_{jnet})$

[8] The worst case time complexity is $O(j_{jnet}^2)$, but this holds only if all Network Distances have already been computed, which will nearly never be the case.

[9] See Footnote17 at Page 16 for description of (reverse) adjacent sections.

[10] See Footnote 18 on Page 16 for the explanation of bounding points of a network part.

  - line 3: The for-loop will be called for each candidate c received by line 2. The operations inside take:
    * line 4-6: O(1)
    * line 7: O($l + \log s_{jnet}$)
    * line 8: O($l \log r$)
    * line 9: O(1)

- line 14: O(1)

4.3.6 Bounding Boxes

Almost analogous to the first network implementation we distinguish between spatio-temporal bounding boxes (**bbox**) and network (**netbox**) and network-temporal (**tempnetbox**) bounding boxes (see Section 3.3.5). Again the network bounding boxes use the route identifier as x-coordinates and the route position(s) as y-coordinates. For all temporal data types the z-coordinates are defined by the start and end time instant. All coordinates are given as *real* values.

Signature	**Example Call**
$M \rightarrow$ *rect3*	**bbox**(a)
$N \rightarrow$ *rect2*	**netbox**(a)
ujpoint $\rightarrow$ *rect3*	**tempnetbox**(*ujpoint*)

Table 4.10: Operators Returning Bounding Boxes

The time complexity of the operations **netbox** and **tempnetbox**, which compute the jnetwork respectively jnetwork-temporal bounding box of the argument is O(1).

The operation **bbox** computes the spatio-temporal bounding box of the argument.

For *ujpoint* values the algorithm for **bbox** searches the B-Tree of the *routes relation* of the *jnet* in O($\log r$) time to get the *routes section list*. Then it uses the sections in the section list to compute the spatial simple line value for the route interval given in the *ujpoint* in O($\sum_{j=1}^{a} a_j$). At last the bounding box of this line value is extended by the temporal values of the *ujpoint* and returned in O(1) time. The total worst case time complexity for computing the **bbox** of an *ujpoint* is O($\log r + \sum_{j=1}^{a} a_j$).

For *mjpoint* values we compute the union of the spatial bounding boxes of the route intervals in the *trajectory* of the *mjpoint* to compute the spatial bounding box of the *mjpoint*. For each route interval we need O($\log r + sum_{j=1}^{a} a_j$) as explained above at **bbox**(*ujpoint*), such that we get a total time complexity of O($i(\log r + sum_{j=1}^{a} a_j)$) for the computation of the spatio-temporal bounding box of an *mjpoint* value.

4.3.7 Merge Data Types

$Y \times Y \rightarrow Y$ $\qquad\qquad$ a **union** b

The operation **union** expects two values of the same data type belonging to the same jnetwork object, and computes an single value of this data type from the two argument data types if this is possible.

For two *jline* values the operation **union** is always defined. The route intervals of both *jline* values are added to the resulting *jline* value in a parallel scan of both sets of route intervals. If overlapping route intervals are detected within the scan they are merged into one. The operation has a time complexity of O($n_1 + n_2$).

For two *mjpoint* values the result of the operation **union** is undefined if both *mjpoint* values have overlapping *junit* values describing different positions for the same time instant. In all other cases the result is defined and a parallel scan of the two sets of *junits* is performed to copy the *junits* of both *mjpoint* in correct sequence to the resulting *mjpoint* in O($m_1 + m_2$) time.

4.3.8 Restrict Data Types

The data types of Table 4.11 restrict the data types by times or places or remove elements from lists.

Signature	Example Call
mjpoint × *instant* → *ijpoint*	*mjpoint* **atinstant** *instant*
mjpoint × *periods* → *mjpoint*	*mjpoint* **atperiods** *periods*
mjpoint × *S* → *mjpoint*	*mjpoint* **at** *a*
ListB × *B* ∪ *ListB* → *ListB*	*a* **-** *b*
	restrict(*a*,*b*)

Table 4.11: Restrict Data Types

The operation **atinstant** performs a binary search after the unit of the *mjpoint* containing the given time instant. If a corresponding unit can be found, the corresponding *ijpoint* is returned, otherwise the return value is undefined. The time complexity of the operation is $O(\log m)$.

The operation **atperiods** performs a parallel linear scan of both arguments. It restricts the units of the *mjpoint* to the time intervals defined by *periods*. The time complexity is $O(m + p)$.

The operation **at** restricts the *mjpoint* to the times it was at the described jnetwork position(s). For both second argument types a linear scan of the units of the *mjpoint* is performed to find the times it passes the given places.

In case of a single network position *jpoint* the time complexity is $O(m)$.

For a network part (*jline*) for each unit of the *mjpoint* a binary search for an intersecting route interval within the set of route intervals of the *jline* is performed, therefore the time complexity in this case is $O(m \log r)$.

The operation **-** returns the values of the input list without the values of the second argument, and the operation **restrict** returns only the elements of the input list that are also in the values of the second argument. If the second argument is a single value this can be done in $O(\log l)$ time, because we perform a binary search on the list values. If the second argument is even a list the time complexity is $O(l_1 + l_2)$, because a parallel scan of both lists is performed to compute the result value.

4.3.9 Comparison Operators

For all data types X the comparison operations $=$, $<$, $>$, $\leq$, $\geq$, and $\neq$ are defined. For the simple data types *jdirection*, *rloc*, *jrint*, and *jpoint* the need $O(1)$ time. For the list data types (*ListB*) the worst case time complexity of the operations is $O(l)$ and for the data type *jline* $O(r)$, we do not need to distinguish between the first and second argument of *ListB* or *jline* values, because if the list length or the number of route intervals is different the values are different.

4.3.10 Property Tests

The operations of Table 4.12 check if the first argument fulfills properties defined by the operation and the second argument.

Signature	Example Call
jpoint × *jline* → *bool*	*jpoint* **inside** *jline*
mgpoint × *S* → *bool*	*mjpoint* **passes** *a*
mgpoint × *mjpoint* → *bool*	*mjpoint1* **intersects** *mjpoint2*
mjpoint × *T* → *bool*	*mjpoint1* **present** *a*

Table 4.12: Restrict Data Types

The operation **inside** performs a binary search in the set of route intervals of the *jline* to decide if the given *jpoint* is inside the network part described by the *jline*. The binary search needs $O(\log r)$ time.

The operation **passes** performs a binary search on the set of route intervals of the trajectory for the given *mjpoint* in $O(\log r)$ or a parallel scan of the route intervals in the trajectory of the *mjpoint* and the route intervals of the *jline* in $O(r_1 + r_2)$ time to decide, if the *mjpoint* passes the given network position(s) at least once or not.

The operation **intersects** checks if the two *mjpoint* are at least once at the same place at the same time. The algorithm first performs a parallel scan of the units of the two *mjpoint* to compute the refinement partitions[11]

[11] Building the refinement partitions means splitting up the units of both *mjpoint* such that the refinement partitions of both *mjpoint* have the same number of units and the units of both refined *mjpoint* have the same time intervals.

of both *mjpoint* in $O(m_1 + m_2)$ time. After that we check for each pair of units of the refinement partitions if the positions of both *mjpoint* within this time interval intersect or not. If an intersecting unit is found the computation is stopped immediately and ***true*** is returned. If no intersecting pair of units is found the time complexity of the whole algorithm is $O(m_1 + m_2 + m_r)$.

The operation **present** checks if the *mjpoint* is defined at the given time. In case of a single time instant as time argument a binary search for a defined unit including this time instant is performed in $O(\log m)$ time. In case of a periods value as time argument a parallel scan through both arguments is performed. The scan stops immediately if an intersecting pair of time intervals is found and true is returned. In the worst case that no intersection is found this needs $O(m + p)$ time.

4.3.11 Path Computing

$P \times P \rightarrow$ *jpath* **shortest_path**(*a*, *b*)

The operation **shortest_path** computes the shortest path in the jnetwork from the first argument to the second argument using A^*-Variant of Dijkstras Algorithm. The two arguments must not be of the same data type, but the result is only defined if both arguments belong to the same jnetwork. We omitted to implement a pure Dijkstra Version of the shortest path operation because the run times of A^*-Variant are better than of Dijkstras Algorithm (see Section 3.3.8). We always try to find out if the shortest path has been computed before by an look up of the network distance table of the jnetwork. If the network distance is already known, the computation is done by following the path from *a* to *b* in the network distance table.

4.3.12 Network Distances

$P \times P \rightarrow$ *real* **netdistance**(*a*, *b*)
jpoint $\rightarrow$ *stream*(*tuple*(jid *int*, dist *real*)) **shortestpathtree**(*jpoint*)
reverseshortestpathtree(*jpoint*)

The operation **netdistance** computes the length of the shortest path in the jnetwork from the first to the second argument[12]. The two arguments must not be of the same data type, but the result is only defined if both arguments belong to the same jnetwork. If the network distance has been computed before, we do not need to compute the complete path, we can just get the network distance by a look up of the network distance table.

The operation **shortestpathtree** computes the network distances from the given *jpoint* to all junctions of the corresponding jnetwork using Dijkstras-Algorithm of shortest paths. The operations **reverseshortestpathtree** computes the distances from all junctions of the corresponding jnetwork to the given *jpoint* using Dijkstras-Algorithm of shortest paths. The result of both operations is a *stream* of *tuple*s. Each tuple consisting of a junction identifier and the network distance of this junction. The results of both operations differ if their are oneways in the jnetwork otherwise the results should be the same.

4.3.13 Network Parts Around a Single Network Position

The three operations **circle**, **incircle** and **outcircle** with signature *jpoint* $\times$ *real* $\rightarrow$ *jline* and syntax **op**(*gpoint*, *dist*) return a *jline* value. In case of **outcircle** the *jline* represents the parts of the network around the given *jpoint* which can be reached within the Network Distance given by *dist* from the *jpoint*. In case of **incircle** the *jline* represents the parts of the network from which the *jpoint* can be reached within *dist*. And **circle** returns the union of the results of **outcircle** and **incircle**.

The values are computed almost analogous to the building the (reverse) shortest path tree of *jpoint*. The difference is the result format and the stop of the computation if the given distance is smaller than the next distances coming from the priority queue.

4.3.14 Alternative Route Locations for Junctions

jpoint $\rightarrow$ *stream*(*jpoint*) **altrlocs**(*jpoint*)

[12] For detailed explanation of Network Distance see Section 3.3.9.2.

Algorithm 17 altrlocs(*jpoint*)

Require: *jpoint*

```
 1: get section list sl of route tuple for rid of jpoint
 2: for each section s ∈ sl do
 3:    get route interval list ril of section s
 4:    for each route interval ri ∈ ril do
 5:       if rloc of jpoint inside ri then
 6:          if rloc is junction then
 7:             return all jpoint values describing this junction
 8:          else
 9:             return jpoint
10:          end if
11:       end if
12:    end for
13: end for
```

The operation **altrlocs** (see Algorithm 17) corresponds to the operation **polygpoints** from the first implementation (see Section 3.3.13). The time complexity is $O(\log r)$ for line 1 of the algorithm. The search for the correct route interval containing the route location in lines 2 - 6 has a worst case complexity of $O(\sum_{j=1}^{a} a_j)$. In case of line 9 the stream creation takes $O(1)$ time. In case of line 7 all c junction descriptions are returned in $O(c)$ time to the stream. We get a worst case complexity of $O(c + \log r + \sum_{j=1}^{a} a_j)$ for the **altrlocs** operation.

4.3.15 Transformation of Paths into Network Parts

jpath → *jline* **tojline**(*jpath*)

The operation **tojline** resorts the route intervals of the *jpath* and stores the result as *jline* value in $O(r \log r)$ time.

Chapter 5

Scripts Using Network Implementations

5.1 Introduction

In this chapter we give some examples for the usage of the operations of both network implementations described in the sections before.

In Section 5.2 we provide scripts in SECONDO executable language translating the data generated by the BerlinMOD Benchmark [1] data generator into the network representations. We also provide scripts with the queries of the BerlinMOD Benchmark for both network implementations.

In Section 5.3 we provide scripts which generate network representations from the information about street networks included in the OSM-Datafiles provided in the web by [7]. At last in Section 5.4 we present some example queries using the operators provided by the `MapMatchingAlgebra` to translate GPS-Tracks into moving network data types on networks generated from open street map data files.

5.2 BerlinMOD Benchmark

Before we can start to translate the BerlinMOD Benchmark data into network representation and run our network queries we have to generate the data provided with the BerlinMOD Benchmark presented in [1][1]. We have to run the data generation script (`BerlinMOD_DataGenerator.SEC`), which needs the three source data files (`streets.data`, `homeRegions.data`, and `workRegions.data`).

The file names and storage position of the data files relative to the `secondo/bin`-Directory on your computer has to be inserted into line 69-71 in the file `BerlinMOD_DataGenerator.SEC`. In line 83 of this file you can set the parameter *SCALEFACTOR* to control the amount of generated data generated by the script. As described in [1].

The script can be started by calling

`SecondoTTYBDB -i BerlinMOD_DataGenerator.SEC`

from the command line in the `secondo/bin`-Directory, or by starting SecondoTTYBDB first and then entering:

@BerlinMOD_DataGenerator.SEC

at the SECONDO prompt.

The script generates a new database called `berlinmod` and fills it with the data described in [1].

After this preparation step we can start to translate the generated data using the executable scripts described in this document, and run our network BerlinMOD Benchmark queries on the translated data later on.

5.2.1 Translation of Source Data

Both network implementations provide different network representations and network creation operators, such that we have to provide own scripts for each of the network implementations to translate the BerlinMOD Benchmark data into the corresponding network representation.

In Section 5.2.1.1 we comment the operations for the translation into the data types of the first network implementation and in Section 5.2.1.2 we do the same for the second network implementation.

[1] The scripts and data files of the BerlinMOD Benchmark are published in the web at `http://dna.fernuni-hagen.de/secondo/BerlinMOD`.

5.2.1.1 Network

The script in the file `Network_CreateObjects.SEC` translates the data set of the BerlinMOD Benchmark into the data types of the first network implementation.

```
# This file performs translates the spatial and spatio-temporal data objects
# of the BerlinMOD benchmark in the Secondo DBMS into their network data
# model representation and builds the according indexes.
#
# It is assumed that there is a database berlinmod with the data objects
# created by the BerlinMOD_DataGenerator.SEC script.
#
# Open Database Berlinmod

open database berlinmod;

# Build a Network From Streets Data.
#
# Because BerlinMOD streets data lacks on informations about street crossings
# we use default values for the connectivity codes enabling all connections in
# a crossing

let B_Routes =
  streets feed
    projectextendstream[; GeoData: .GeoData polylines[TRUE]]
    addcounter[Id,1]
    projectextend [Id; Lengt : size(.GeoData),
                       Geometry: fromline(.GeoData),
                       Dual: TRUE,
                       StartSmaller: TRUE]
consume;

let B_Junctions =
  B_Routes feed {r1}
  B_Routes feed {r2}
  symmjoin [(.Id_r1 < ..Id_r2) and (.Geometry_r1 intersects ..Geometry_r2)]
  projectextendstream[Id_r1, Geometry_r1, Id_r2,
                      Geometry_r2; CROSSING_POINT: components(crossings(.Geometry_r1,
                                                                        .Geometry_r2))]
  projectextend[; R1Id: .Id_r1,
                  R1meas: atpoint(.Geometry_r1, .CROSSING_POINT, TRUE),
                  R2Id: .Id_r2,
                  R2meas: atpoint(.Geometry_r2, .CROSSING_POINT, TRUE),
                  CC: 65535]
consume;

let B_NETWORK =
  thenetwork(1,
             1.0,
             B_Routes,
             B_Junctions);

# Translate Trips into Network Representation

let dataSNcar =
  dataScar feed
    projectextend[Licence, Model, Type; Trip: mpoint2mgpoint(B_NETWORK, .Trip)]
consume;

let dataMNtrip =
  dataMtrip feed
    projectextend [Moid; Trip: mpoint2mgpoint(B_NETWORK, .Trip)]
consume;

# Translate QueryPoint Set into Newtork Representation

let QueryPointsNet =
  QueryPoints feed
    projectextend[Id; Pos: point2gpoint(B_NETWORK, .Pos)]
    projectextendstream[Id;  Pos: polygpoints(.Pos, B_NETWORK)]
 consume;

let QueryPoints1Net =
  QueryPoints feed head[10]
    projectextend[Id; Pos: point2gpoint(B_NETWORK, .Pos)]
    projectextendstream[Id;  Pos: polygpoints(.Pos, B_NETWORK)]
 consume;

# Translate QueryRegions into Network Representation

let routesline =
  components(routes(B_NETWORK) feed
               projectextend[;Curve: toline(.Curve)]
               aggregateB[Curve; fun(L1: line, L2: line) union_new(L1, L2); [const line value()]])
  transformstream
  extend[NoSeg: no_segments(.Elem)]
  sortby [NoSeg desc]
extract [Elem];
```

```
let QRlines =
  QueryRegions feed
  projectextend [Id; Lr: intersection_new(.Region, routesline)]
consume;

let QueryRegionsNet =
  QRlines feed
  projectextend[Id; Region: line2gline(B_NETWORK, .Lr)]
consume;

# Build Indexes on Network Representation
#
# B-Tree indexes for licences in dataScar, and dataMtrip, and for moIds in
# dataMcar and dataMNtrip.

derive dataSNcar_Licence_btree = dataSNcar createbtree[Licence];

derive dataMcar_Licence_btree = dataMcar createbtree[Licence];

derive dataMcar_Moid_btree = dataMcar createbtree[Moid];

derive dataMNtrip_Moid_btree = dataMNtrip createbtree[Moid];

# Temporal Network Position Indexes (TNPI) and Network Position Indexes (NPI)
# for dataMNtrip and dataSNcar

derive dataSNcar_BoxNet_timespace =
  dataSNcar feed
  extend[TID: tupleid(.)]
  projectextendstream[TID; UTrip: units(.Trip)]
  extend[Box: unitbox(.UTrip)]
  sortby[Box asc]
bulkloadrtree[Box];

derive dataMNtrip_BoxNet_timespace =
  dataMNtrip feed
  extend[TID: tupleid(.)]
  projectextendstream[TID; UTrip: units(.Trip)]
  extend[Box: unitbox(.UTrip)]
  sortby[Box asc]
 bulkloadrtree[Box];

derive dataSNcar_TrajBoxNet =
  dataSNcar feed
  extend[TID: tupleid(.)]
  projectextendstream[TID; Box: routeintervals(trajectory(.Trip))]
  sortby [Box asc]
 bulkloadrtree[Box];

derive dataMNtrip_TrajBoxNet =
  dataMNtrip feed
  extend[TID: tupleid(.)]
  projectextendstream[TID; Box: routeintervals(trajectory(.Trip))]
  sortby [Box asc]
 bulkloadrtree[Box];

# Spatio-Temporal Index for dataMNtrip

derive dataMNtrip_SpatioTemp =
  dataMNtrip feed
  extend[TID: tupleid(.)]
  projectextend[TID; Box: mgpbbox(.Trip)]
  sortby [Box asc]
 bulkloadrtree[Box];

# Often used Query Object Relations

let QueryLicences1 = QueryLicences feed head[10] consume;

let QueryLicences2 = QueryLicences feed head[20] filter [.Id > 10] consume;

let QueryPeriods1 = QueryPeriods feed head[10] consume;

let QueryInstant1 = QueryInstants feed head[10] consume;

let QueryRegions1Net = QueryRegionsNet feed head[10] consume;

# Creation Finished Close Database

close database;
```

5.2.1.2 JNetwork

The script in the file `JNetwork_CreateBMODObjects.SEC` translates the data set of the BerlinMOD Benchmark into the data types of the second network implementation.

```
# This file performs translates the spatial and spatio-temporal data objects
```

```
# of the BerlinMOD benchmark in the Secondo DBMS into their network data
# model representation and builds the according indexes.
#
# It is assumed that there is a database berlinmod with the data objects
# created by the BerlinMOD_DataGenerator.SEC script.
#
# Open Database Berlinmod

open database berlinmod;

# Build a JNetwork From Streets Data
#
# collect roads data

let RoadsTmp =
  streets feed
    projectextend[GeoData; VMax: ifthenelse(.Vmax > 0.0, .Vmax, 0.01)]
    projectextendstream[VMax; RoadCurve: .GeoData polylines[FALSE]]
    projectextend[VMax; RoadC: fromline(.RoadCurve)]
    projectextend[VMax; Lenth: size(.RoadC), RoadCurve: .RoadC]
    sortby[Lenth desc, VMax asc, RoadCurve asc]
    addcounter[Rid, 1]
    project[Rid, RoadCurve, VMax, Lenth]
consume;

# Compute jnetwork junctions based on roads data

let RoadEndPoints =
  RoadsTmp feed
    projectextend[Rid, RoadCurve; StartPoint: getstartpoint(.RoadCurve),
                                  EndPoint: getendpoint(.RoadCurve)]
consume;

let RoadCrossings =
  RoadsTmp feed
    project[Rid, RoadCurve] {r1}
  RoadsTmp feed
    project[Rid, RoadCurve] {r2}
  itSpatialJoin[RoadCurve_r1, RoadCurve_r2, 4,8]
    filter [.Rid_r1 < .Rid_r2]
    filter [.RoadCurve_r1 intersects .RoadCurve_r2]
    projectextendstream[Rid_r1, Rid_r2, RoadCurve_r1,
      RoadCurve_r2; CROSSING: components(crossings(.RoadCurve_r1,
                                                   .RoadCurve_r2))]
    projectextend[;R1id: .Rid_r1,
                  R1Pos: atpoint(.RoadCurve_r1, .CROSSING),
                  R2id: .Rid_r2,
                  R2Pos: atpoint(.RoadCurve_r2, .CROSSING),
                  SpatialPos: .CROSSING]
consume;

let JunctionsTmp =
  ( ( RoadEndPoints feed
        projectextend [; Pos: .StartPoint])
    ( RoadEndPoints feed
        projectextend [; Pos: .EndPoint])
    concat)
  ( RoadCrossings feed
      projectextend[; Pos: .SpatialPos])
  concat
    sortby[Pos]
    rdup
    addcounter[Jid, 1]
    project[Jid, Pos]
consume;

# connect roads with the junctions on the road

let RoadsTmp2 =
  RoadsTmp feed
  JunctionsTmp feed
  itSpatialJoin[RoadCurve, Pos, 4,8]
    filter[.Pos inside .RoadCurve]
    projectextend[Rid, VMax, Lenth, RoadCurve,
                  Jid; SpatialPos: .Pos,
                       RoadPos: atpoint(.RoadCurve,.Pos)]
consume;

let RoadsTmp3 =
  RoadsTmp2 feed
    filter [iscycle(.RoadCurve)]
    filter [.RoadPos = 0.0]
    projectextend[Rid, VMax, Lenth, RoadCurve, Jid,
                  SpatialPos; RoadPos: size(.RoadCurve)]
consume;

let RoadsTmp4 =
  RoadsTmp2 feed
    filter [iscycle(.RoadCurve)]
    filter [.RoadPos = size(.RoadCurve)]
    projectextend[Rid, VMax, Lenth, RoadCurve, Jid, SpatialPos; RoadPos: 0.0]
```

```
consume;

let RoadsTmp5 =
  ( ( RoadsTmp3 feed)
    ( RoadsTmp4 feed)
    concat)
  ( RoadsTmp2 feed)
  concat
  sortby[Rid, RoadPos, Jid, SpatialPos, VMax, Lenth, RoadCurve]
  rdup
consume;

let RoadJuncList =
  RoadsTmp5 feed
    project[Rid, Jid]
    sortby [Rid, Jid]
    rdup
    groupby[Rid; JuncList: group feed projecttransformstream[Jid] createlist]
consume;

let JuncRLocList =
  RoadsTmp5 feed
    projectextend[Jid; RLoc: createrloc(.Rid,
                                        .RoadPos,
                                        [const jdirection value(Both)])]
    sortby[Jid, RLoc]
    rdup
    groupby[Jid; RLocList: group feed projecttransformstream[RLoc] createlist]
consume;

# compute sections

let SectTmp =
   RoadsTmp5 feed
     project[Rid, RoadCurve, SpatialPos]
     sortby[Rid, RoadCurve, SpatialPos]
     groupby[Rid, RoadCurve; SplitPoints: group feed
                                          projecttransformstream[SpatialPos]
                                          collect_points[TRUE]]
     projectextendstream[Rid; SectCurve: splitslineatpoints(.RoadCurve,
                                                            .SplitPoints)]
     extend[StartPoint: getstartpoint(.SectCurve),
            EndPoint: getendpoint(.SectCurve),
            Lenth: size(.SectCurve),
            JDir: [const jdirection value(Both)]]
   JunctionsTmp feed {j1}
   itSpatialJoin[StartPoint,Pos_j1,4,8]
     filter[.StartPoint = .Pos_j1]
     projectextend[Rid, SectCurve, EndPoint, Lenth, JDir; StartJid: .Jid_j1]
   JunctionsTmp feed {j2}
   itSpatialJoin[EndPoint,Pos_j2,4,8]
     filter[.EndPoint = .Pos_j2]
     projectextend[Rid, SectCurve, Lenth, JDir, StartJid; EndJid: .Jid_j2]
   RoadsTmp5 feed {r1}
   hashjoin[Rid, Rid_r1]
     filter [.Rid = .Rid_r1]
     filter [.StartJid = .Jid_r1]
     projectextend[Rid, SectCurve, Lenth, JDir, StartJid,
                   EndJid; VMax: .VMax_r1,
                           StartPos: .RoadPos_r1]
   RoadsTmp5 feed {r2}
   hashjoin[Rid, Rid_r2]
     filter [.Rid = .Rid_r2]
     filter [.EndJid = .Jid_r2]
     projectextend[Rid, SectCurve, Lenth, JDir, StartJid, EndJid, VMax,
                   StartPos; EndPos: .RoadPos_r2]
     sortby[Rid, StartJid, EndJid, Lenth, VMax, JDir, SectCurve, StartPos, EndPos]
     filter[.StartPos < .EndPos]
     rdup
     addcounter[Sid,1]
consume;

# compute section lists

let RoadSectList =
  SectTmp feed
    project[Rid, Sid]
    sortby[Rid, Sid]
    groupby[Rid; ListSect: group feed projecttransformstream[Sid] createlist]
consume;

let SectRouteIntervals =
  SectTmp feed
    projectextend [Sid; RInt: createrint(.Rid, .StartPos, .EndPos, .JDir)]
    sortby [Sid, RInt]
    groupby [Sid; ListRInt: group feed projecttransformstream[RInt] createlist]
consume;

let JuncInAndOutList =
  ( SectTmp feed
      projectextend [Sid; Jid: .StartJid])
```

```
  ( SectTmp feed
      projectextend [Sid; Jid: .EndJid])
  concat
    sortby [Jid, Sid]
    rdup
    groupby[Jid; ListInOutSect: group feed projecttransformstream[Sid] createlist]
consume;

# create input relations for jnetwork creation

let InJunc =
  JunctionsTmp feed
  JuncRLocList feed {r1}
  hashjoin[Jid, Jid_r1]
    filter [.Jid = .Jid_r1]
    projectextend [Jid, Pos; ListRLoc: .RLocList_r1]
  JuncInAndOutList feed {sl}
  hashjoin[Jid, Jid_sl]
    filter [.Jid = .Jid_sl]
    projectextend[Jid, Pos, ListRLoc; ListInSect: .ListInOutSect_sl,
                                      ListOutSect: .ListInOutSect_sl]
    sortby[Jid, Pos, ListRLoc, ListInSect, ListOutSect]
    rdup
consume;

let InSect =
  SectTmp feed
  SectRouteIntervals feed {ri}
  hashjoin[Sid, Sid_ri]
    filter [.Sid = .Sid_ri]
    projectextend[Sid, SectCurve, StartJid, EndJid, JDir, VMax,
                  Lenth; ListRint: .ListRInt_ri]
  InJunc feed {j1}
  hashjoin[StartJid, Jid_j1]
    filter[.StartJid = .Jid_j1]
    projectextend[Sid, SectCurve, StartJid, EndJid, JDir, VMax, Lenth,
                  ListRint; ListAdjSectDown: .ListOutSect_j1,
                            ListRevAdjSectUp: .ListInSect_j1]
  InJunc feed {j2}
  hashjoin[EndJid, Jid_j2]
    filter[.EndJid = .Jid_j2]
    projectextend[Sid, SectCurve, StartJid, EndJid, JDir, VMax, Lenth,
                  ListRint, ListAdjSectDown,
                  ListRevAdjSectUp; ListAdjSectUp: .ListOutSect_j2,
                                    ListRevAdjSectDown: .ListInSect_j2]
    project[Sid, SectCurve, StartJid, EndJid, JDir, VMax, Lenth, ListRint,
            ListAdjSectUp, ListAdjSectDown, ListRevAdjSectUp, ListRevAdjSectDown]
    sortby [Sid, StartJid, EndJid, SectCurve, JDir, VMax, Lenth, ListRint,
            ListAdjSectUp, ListAdjSectDown, ListRevAdjSectUp, ListRevAdjSectDown]
    rdup
consume;

let InRoad =
  RoadsTmp feed
  RoadJuncList feed {j1}
  hashjoin[Rid, Rid_j1]
    filter [.Rid = .Rid_j1]
    projectextend[Rid, Lenth; ListJunc: .JuncList_j1]
  RoadSectList feed {s1}
  hashjoin[Rid, Rid_s1]
    projectextend[Rid, ListJunc, Lenth; ListSect: .ListSect_s1]
    project[Rid, ListJunc, ListSect, Lenth]
    sortby [Rid, ListJunc, ListSect, Lenth]
    rdup
consume;

# create jnetwork

query createjnet("JBNet",
                 1.0,
                 InJunc,
                 InSect,
                 InRoad);

# Translate QueryPoint Set into JNewtork Representation

let QueryPointsJNet =
  QueryPoints feed
    projectextend[Id; Pos: tonetwork(JBNet, .Pos)]
 consume;

let QueryPointsJNetAll =
  QueryPointsJNet feed
    projectextendstream[Id; NPos: altrlocs(.Pos)]
    projectextend[Id; Pos: .NPos]
consume;

let QueryPoints1JNet =
  QueryPoints feed head[10]
    projectextend[Id; Pos: tonetwork(JBNet, .Pos)]
 consume;
```

```
let QueryPoints1JNetAll =
  QueryPoints1JNet feed
    projectextendstream[Id; NPos: altrlocs(.Pos)]
    projectextend[Id; Pos: .NPos]
consume;


# Translate QueryRegions into JNetwork Representation

let routeslinej =
  components(sections(JBNet) feed
                              projectextend[;Curve: toline(.Curve)]
                              aggregateB[Curve; fun(L1: line, L2: line)
                                        union_new(L1, L2); [const line value()]])
    transformstream
    extend[NoSeg: no_segments(.Elem)]
    sortby [NoSeg desc]
extract [Elem];

let QRlinesj =
  QueryRegions feed
    projectextend [Id; Lr: intersection_new(.Region, routeslinej)]
 consume;

let QueryRegionsJNet =
  QRlinesj feed
  projectextend[Id; Region: tonetwork(JBNet, .Lr)]
 consume;

# Translate Trips into JNetwork Representation

let dataSJcar =
  dataScar feed
    projectextend[Licence, Model, Type; Trip: tonetwork(JBNet, .Trip)]
consume;

let dataMJtrip =
  dataMtrip feed
    projectextend [Moid; Trip: tonetwork(JBNet, .Trip)]
consume;


# Build Indexes on JNetwork representation
#
# B-Tree indexes for licences in dataScar, and dataMtrip, and for moids in
# dataMcar and dataMJtrip.

derive dataSJcar_Licence_btree = dataSJcar createbtree[Licence];

derive dataMcar_Licence_btree = dataMcar createbtree[Licence];

derive dataMcar_Moid_btree = dataMcar createbtree[Moid];

derive dataMJtrip_Moid_btree = dataMJtrip createbtree[Moid];


# Temporal JNetwork Position Indexes (TNPI) and JNetwork Position Indexes (NPI)
# for dataMNtrip and dataSNcar

derive dataSJcar_BoxNet_timespace =
  dataSJcar feed
    extend[TID: tupleid(.)]
    projectextendstream[TID; UTrip: units(.Trip)]
    extend[Box: tempnetbox(.UTrip)]
    sortby[Box asc]
bulkloadrtree[Box];

derive dataMJtrip_BoxNet_timespace =
  dataMJtrip feed
    extend[TID: tupleid(.)]
    projectextendstream[TID; UTrip: units(.Trip)]
    extend[Box: tempnetbox(.UTrip)]
    sortby[Box asc]
bulkloadrtree[Box];

derive dataSJcar_BoxNet =
  dataSJcar feed
    extend[TID: tupleid(.)]
    projectextendstream[TID; UTrip: units(.Trip)]
    extend[Box: netbox(.UTrip)]
    sortby [Box asc]
bulkloadrtree[Box];

derive dataMJtrip_BoxNet =
  dataMJtrip feed
    extend[TID: tupleid(.)]
    projectextendstream[TID; UTrip: units(.Trip)]
    extend[Box: netbox(.UTrip)]
    sortby [Box asc]
bulkloadrtree[Box];
```

```
derive dataSJcar_TrajBoxNet =
  dataSJcar feed
    extend[TID: tupleid(.)]
    projectextendstream[TID; RInt: units(trajectory(.Trip))]
    projectextend[TID; Box: netbox(.RInt)]
    sortby[Box asc]
bulkloadrtree[Box];

derive dataMJtrip_TrajBoxNet =
  dataMJtrip feed
    extend[TID: tupleid(.)]
    projectextendstream[TID; RInt: units(trajectory(.Trip))]
    projectextend[TID; Box: netbox(.RInt)]
    sortby[Box asc]
bulkloadrtree[Box];

# Spatio-Temporal Index for dataMJtrip

derive dataMJtrip_SpatioTemp =
  dataMJtrip feed
    extend[TID: tupleid(.)]
    projectextend[TID; Box: bbox(.Trip)]
    sortby [Box asc]
 bulkloadrtree[Box];

# Often used Query Object Relations

let QueryLicences1 = QueryLicences feed head[10] consume;

let QueryLicences2 = QueryLicences feed head[20] filter [.Id > 10] consume;

let QueryPeriods1 = QueryPeriods feed head[10] consume;

let QueryInstant1 = QueryInstants feed head[10] consume;

let QueryRegions1JNet = QueryRegionsJNet feed head[10] consume;

# Finished Close Database

close database;
```

5.2.2 Executable Query Sets

We provide for both network implementations executable Secondo scripts with the 17 queries for the object and the trip based approach of the BerlinMOD Benchmark.

5.2.2.1 Network

The script in the file `Network_OBA-Queries.SEC` executes the 17 queries of the object based approach of the BerlinMOD Benchmark using the first network implementation.

```
# Network queries for the object based approach of the BerlinMOD Benchmark.
#
# The script assumes that there is a database berlinmod with a network data
# model representation of the BerlinMOD Benchmark data.
#
# This database can be generated by the script 'BerlinMOD_DataGenerator.SEC'.
# The network data model representation and accroding indexes can be generated
# with the script 'Network_CreateObjects.SEC'
#
# Start Script Opening the Database

open database berlinmod;

# Query 1: What are the models of the vehicles with license plate numbers from
#          QueryLicence?

let OBANres001 =
  QueryLicences feed {l}
    loopjoin [dataSNcar_Licence_btree dataSNcar exactmatch [.Licence_l]]
    project [Licence, Model]
consume;


# Query 2: How many vehicles exist that are passenger cars?

let OBANres002 =
  dataSNcar feed
    filter [.Type = "passenger"]
count;


# Query 3: Where have the vehicles with licenses from QueryLicence1 been at
#          each instant from QueryInstant1?
```

```
let OBANres003 =
  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;


# Query 4: Which license plate numbers belong to vehicles that have passed the
#          points from QueryPoints?

let OBANres004 =
  QueryPointsNet feed
    projectextend[Id, Pos; Prect: gpoint2rect(.Pos)]
    loopjoin[dataSNcar_TrajBoxNet windowintersectsS[.Prect]
             sort rdup dataSNcar gettuples]
    filter [.Trip passes .Pos]
    project [Id, Licence]
    sortby [Id asc, Licence asc]
    krdup [Id, Licence]
consume;

# Query 5: What is the minimum distance between places, where a vehicle with a
#          license from QueryLicences1 and a vehicle with licenses from
#          QueryLicence2 have been?

let OBANres005tmp1 =
  QueryLicences1 feed {l1}
    loopsel[dataSNcar_Licence_btree dataSNcar exactmatch [.Licence_l1]
            projectextend[Licence; TrajLine: gline2line(trajectory(.Trip))]]
consume;

let OBANres005tmp2 =
  QueryLicences2 feed {l2}
    loopsel[dataSNcar_Licence_btree dataSNcar exactmatch [.Licence_l2]
            projectextend[Licence; TrajLine: gline2line(trajectory(.Trip))]]
consume;

let OBANres005 =
  OBANres005tmp1 feed {c1}
  OBANres005tmp2 feed {c2}
  product
    projectextend [Licence_c1,
                   Licence_c2; Distance: round(distance(.TrajLine_c1,
                                                        .TrajLine_c2),3)]
    sortby[Licence_c1, Licence_c2]
consume;

# delete temporary objects

delete OBANres005tmp1;
delete OBANres005tmp2;

# Query 6: What are the pairs of license plate numbers of "trucks", that have
#          been as close as 10m or less to each other?

let OBANres006tmp1 =
  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 OBANres006 =
  OBANres006tmp1 feed {a}
  OBANres006tmp1 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 temporary object

delete OBANres006tmp1;

# Query 7: What are the license plate numbers of the "passenger" cars that
#          have reached points from QueryPoints first of all "passenger" cars
#          during the complete observation period?

let OBANres007tmp1 =
```

```
  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 OBANres007 =
  OBANres007tmp1 feed
    groupby[Id; FirstTime: group feed min[Instant]]{b}
  OBANres007tmp1 feed {a}
  symmjoin [..Id_a = .Id_b]
    filter[.Instant_a <= .FirstTime_b]
    project[Id_a, Licence_a]
    sortby[Id_a, Licence_a]
consume;

# delete temporary object

delete OBANres007tmp1;

# Query 8: What are the overall traveled distances of the vehicles with
#          license plate numbers from QueryLicences1 during the periods from
#          QueryPeriods1?

let OBANres008 =
  QueryLicences1 feed {l}
    loopsel [dataSNcar_Licence_btree dataSNcar exactmatch [.Licence_l]]
  QueryPeriods1 feed
    filter[not(isempty(.Period))] {p}
  product
    projectextend [Licence,
                   Period_p; Distance: round(length(.Trip atperiods .Period_p),3)]
    sortby[Licence, Period_p]
consume;

# Query 9: What is the longest distance that was traveled by a vehicle during
#          each of the periods from QueryPeriods?

let OBANres009 =
  dataSNcar feed {c}
  QueryPeriods feed
    filter[not(isempty(.Period))]{p}
  product
    projectextend [Id_p, Period_p,
                   Licence_c; Dist: round(length(.Trip_c atperiods .Period_p),3)]
    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]
    project [Period_p, Distance]
consume;


# Query 10: When and where did the vehicles with license plate numbers from
#           QueryLicences1 meet other vehicles (distance < 3m) and what are
#           the latter licenses?

let OBANres010 =
  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;

# Query 11: Which vehicles passed a point from QueryPoints1 at one of the
#           instants from QueryInstant1?

let OBANres011 =
```

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

# Query 12: Which vehicles met at a point from QueryPoints1 at an instant from
#           QueryInstant1?

let OBANres012tmp1 =
  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
            filter [.Trip passes (attr(t,Pos_p))]
            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 OBANres012 =
  OBANres012tmp1 feed {c1}
  OBANres012tmp1 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 temporary objects

delete OBANres012tmp1;

# Query 13: Which vehicles traveled within one of the regions from
#           QueryRegions1 during the periods from QueryPeriods1?

let OBANres013 =
  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;

# Query 14: Which vehicles traveled within one of the regions from
#           QueryRegions1 at one of the instants from QueryInstant1?

let OBANres014 =
  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;

#  Query 15: Which vehicles passed a point from QueryPoints1 during a period
#            from QueryPeriods1?

let OBANres015 =
  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;

# Query 16: List the pairs of licenses for vehicles the first from
#           QueryLicences1, the second from QueryLicences2, where the
#           corresponding vehicles are both present within a region from
#           QueryRegions1 during a period from QueryPeriod1, but do not meet
#           each other there and then.

let OBANres016 =
  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]
    sortby[Id_r_a, Period_p_a, Licence_a, Licence_b]
consume;

# Query 17: Which points from QueryPoints have been visited by a maximum
#           number of different vehicles?

let OBANres017tmp1 =
  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 OBANres017 =
  OBANres017tmp1 feed
    filter [.Hits = (OBANres017tmp1 feed max[Hits])]
    project [Id_p, Hits]
consume;

#delete temporary object

delete OBANres017tmp1;

# Save runtime information

let QRT_NET_OBA = SEC2COMMANDS feed consume;

# To save runtime information on hard disk uncomment next line
# save QRT_NET_OBA to 'NetworkOBARunTimes.DAT';

# Finish Script and Close Database

close database;
```

The script in the file `Network_TBA-Queries.SEC` executes the 17 queries of the trip based approach of the BerlinMOD Benchmark using the first network implementation.

```
# Network queries for the trip based approach of the BerlinMOD Benchmark.
#
# The script assumes that there is a database berlinmod with a network data
```

```
# model representation of the BerlinMOD Benchmark data.
#
# This database can be generated by the script 'BerlinMOD_DataGenerator.SEC'.
# The network data model representation and accroding indexes can be generated
# with the script 'Network_CreateObjects.SEC'
#
# Start Script Opening the Database

open database berlinmod;

# Query 1: What are the models of the vehicles with license plate numbers from
#          QueryLicence?

let TBANres001 =
  QueryLicences feed {l}
    loopjoin [dataMcar_Licence_btree dataMcar exactmatch[.Licence_l]]
    project [Licence, Model]
consume;

# Query 2: How many vehicles exit that are passenger cars?

let TBANres002 =
  dataMcar feed
    filter [.Type = "passenger"]
count;

# Query 3: Where have the vehicles with licenses from QueryLicence1 been at
#          each  instant from QueryInstant1?

let TBANres003 =
  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)]
    sortby[Instant_i, Licence_ll]
consume;

# Query 4: Which license plate numbers belong to vehicles that have passed the
#          points from QueryPoints?

let TBANres004 =
  QueryPointsNet feed
    projectextend[Id, Pos; Elem: gpoint2rect(.Pos)]
    loopjoin[dataMNtrip_TrajBoxNet windowintersectsS[.Elem]
             sort rdup dataMNtrip gettuples]
      filter[.Trip passes .Pos]
      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;


# Query 5: What is the minimum distance between places, where a vehicle with a
#          license from QueryLicences1 and a vehicle with Licenses from
#          QueryLicence2 have been?

let TBANres005 =
  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: round(distance(.Trax_c1, .Trax_c2),3)]
    sortby[Licence_c1, Licence_c2]
consume;

# Query 6: What are the pairs of license plate numbers of "trucks", that have
#          been as close as 10m or less to each other?

let TBANres006tmp1 =
  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 TBANres006 =
  TBANres006tmp1 feed {c1}
  TBANres006tmp1 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 temporary objects

delete TBANres006tmp1;


# Query 7: What are the license plate numbers of the "passenger" cars that
#          have reached points from QueryPoints first of all "passenger" cars
#          during the complete observation period?

let TBANres007tmp1 =
  QueryPointsNet feed
    projectextend[Id, Pos; Prect: gpoint2rect(.Pos)]
    loopsel[fun(t:TUPLE) dataMNtrip_TrajBoxNet windowintersectsS[attr(t,Prect)]
            sort rdup dataMNtrip gettuples
            filter[.Trip passes (attr(t,Pos))]
            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 TBANres007 =
  TBANres007tmp1 feed
    groupby [Id; FirstTime: group feed min[TimeAtPos]]{b}
  TBANres007tmp1 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 temporary object

delete TBANres007tmp1;


# Query 8: What are the overall traveled distances of the vehicles with
#          license plate numbers from QueryLicences1 during the periods from
#          QueryPeriods1?

let TBANres008 =
  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: round(group feed sum[Distance],3)]
    project[Licence, Period, Dist]
    sortby[Licence, Period, Dist]
consume;

# Query 9: What is the longest distance that was traveled by a vehicle during
#          each of the periods from QueryPeriods?

let TBANres009 =
  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, Period_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: round(group feed max[Dist],3)]
    filter[.Dista > 0.0]
    project[Period_p, Dista]
    sortby[Period_p, Dista]
consume;

# Query 10: When and where did the vehicles with license plate numbers from
#           QueryLicences1 meet other vehicles (distance < 3m) and what are
#           the latter licenses?

let TBANres010 =
  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[; QueryLicence: attr(t, Licence_V1),
                              OtherLicence: .Licence,
                              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]
      sortby[QueryLicence, OtherLicence, AllPos]
consume;

# Query 11: Which vehicles passed a point from QueryPoints1 at one of the
#           instants from QueryInstant1?

let TBANres011 =
  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
            filter[.Trip passes (attr(t,Pos_p))]
            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;

# Query 12: Which vehicles met at a point from QueryPoints1 at an instant from
#           QueryInstant1?

let TBANres012tmp1 =
  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
            filter[.Trip passes (attr(t,Pos_p))]
            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 TBANres012 =
  TBANres012tmp1 feed {A}
```

```
  TBANres012tmp1 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 temporary object

delete TBANres012tmp1;

# Query 13: Which vehicles traveled within one of the regions from
#           QueryRegions1 during the periods from QueryPeriods1?

let TBANres013 =
  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 [Licence asc, Id_r asc, Period_p asc]
      krdup [Licence, Id_r, Period_p]
consume;

# Query 14: Which vehicles traveled within one of the regions from
#           QueryRegions1 at one of the instants from QueryInstant1?

let TBANres014 =
  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;

# Query 15: Which vehicles passed a point from QueryPoints1 during a period
#           from QueryPeriods1?

let TBANres015 =
  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;

# Query 16: List the pairs of licenses for vehicles the first from
#           QueryLicences1, the second from QueryLicences2, where the
#           corresponding vehicles are both present within a Region from
#           QueryRegions1 during a period from QueryPeriod1, but do not meet
#           each other there and then.

let TBANres016 =
  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, 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;

# Query 17: Which points from QueryPoints have been visited by a maximum
#           number of different vehicles?

let TBANres017tmp1 =
  QueryPointsNet feed
    projectextend[Id, Pos; Elem: gpoint2rect(.Pos)]
    loopsel[fun(t:TUPLE) dataMNtrip_TrajBoxNet windowintersectsS[attr(t,Elem)]
            sort rdup dataMNtrip gettuples
            filter [.Trip passes (attr(t,Pos))]
            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 TBANres017 =
  TBANres017tmp1 feed
    filter [.Hits = (TBANres017tmp1 feed max[Hits])]
    project [Id_p, Hits]
consume;

# delete temporary object

delete TBANres017tmp1;

# Save query runtimes

let QRT_NET_TBA = SEC2COMMANDS feed  consume;

# Uncomment the next line if you want to save run time information on disk
# save QRT_NET_TBA to 'NetworkTBARunTimes.DAT';

# Finish Script and Close Database

close database;
```

5.2.2.2 JNetwork

The script in the file `JNetwork_OBA-Queries.SEC` executes the 17 queries of the object based approach of the BerlinMOD Benchmark using the second network implementation.

```
# This file performs the OBA-Queries of the BerlinMOD benchmark on the
# JNetwork Representation Secondo DBMS. Created by 'BerlinMOD_DataGenerator.SEC'
# and 'JNetwork_CreateBMODObjects.SEC'

open database berlinmod;

# Query 1: What are the models of the vehicles with license plate numbers from
#          QueryLicence?

let OBAJNres001 =
  QueryLicences feed {l}
    loopjoin[ dataSJcar_Licence_btree dataSJcar exactmatch[.Licence_l]]
      project[Licence, Model]
consume;

# Query 2: How many vehicles exist that are passenger cars?

let OBAJNres002 =
  dataSJcar feed
    filter [.Type = "passenger"]
count;

# Query 3: Where have the vehicles with licenses from QueryLicence1 been at
#          each instant from QueryInstant1?
```

```
let OBAJNres003 =
  QueryLicences1 feed {l}
    loopjoin[dataSJcar_Licence_btree dataSJcar exactmatch[.Licence_l]]
  QueryInstants feed {i} head[10]
  product
    projectextend[; Licence: .Licence_l,
                    Instant: .Instant_i,
                    Pos: val(.Trip atinstant .Instant_i)]
consume;

# Query 4: Which license plate numbers belong to vehicles that have passed the
#          points from QueryPoints?

let OBAJNres004 =
  QueryPointsJNetAll feed
    extend[Prect: netbox(.Pos)]
    projectextend[Id, Pos; NBox: rectangle2(minD(.Prect,1), maxD(.Prect, 1),
                                            minD(.Prect,2) - 0.00001,
                                            maxD(.Prect,2) + 0.00001)]
    loopjoin[dataSJcar_TrajBoxNet windowintersectsS[.NBox]
             sort rdup dataSJcar gettuples]
      filter [.Trip passes .Pos]
      project[Id, Licence]
      sortby[Id asc, Licence asc]
      krdup[Id, Licence]
consume;

# Query 5: What is the minimum distance between places, where a vehicle with a
#          license from QueryLicences1 and a vehicle with licenses from
#          QueryLicence2 have been?

let OBAJNres005tmp1 =
  QueryLicences1 feed
    loopsel[dataSJcar_Licence_btree dataSJcar exactmatch[.Licence]]
      projectextend[Licence; Traj: fromnetwork(trajectory(.Trip))]
consume;

let OBAJNres005 =
  QueryLicences2 feed
    loopsel[ dataSJcar_Licence_btree dataSJcar exactmatch[.Licence] ]
      projectextend[Licence; Traj: fromnetwork(trajectory(.Trip))]{t2}
  OBAJNres005tmp1 feed {t1}
  product
    projectextend[ ; Licence1: .Licence_t1,
                     Licence2: .Licence_t2,
                     Dist: round(distance(.Traj_t1, .Traj_t2),3)]
    sortby [Licence1, Licence2]
consume;

#delete temporary object

delete OBAJNres005tmp1;

# Query 6: What are the pairs of license plate numbers of "trucks", that have
#          been as close as 10m or less to each other?

let OBAJNres006tmp1 =
  dataSJcar feed
    filter [.Type = "truck"]
    projectextend [Licence; Ptrip: fromnetwork(.Trip), BBox: bbox(.Trip)]
    extend [Box: rectangle2(minD(.BBox,1) - 5.0, maxD(.BBox,1) + 5.0,
                            minD(.BBox,2) - 5.0, maxD(.BBox,2) + 5.0)]
 consume;

let OBAJNres006 =
  OBAJNres006tmp1 feed {a}
  OBAJNres006tmp1 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 temporary object

delete OBAJNres006tmp1;

# Query 7: What are the license plate numbers of the "passenger" cars that
#          have reached points from QueryPoints first of all "passenger" cars
#          during the complete observation period?

let OBAJNres007tmp1 =
  QueryPointsJNetAll feed
    extend[Prect: netbox(.Pos)]
    projectextend[Id, Pos; NBox: rectangle2(minD(.Prect,1), maxD(.Prect, 1),
                                            minD(.Prect,2) - 0.00001,
                                            maxD(.Prect,2) + 0.00001)]
    loopsel[fun(t:TUPLE)
            dataSJcar_TrajBoxNet windowintersectsS[attr(t,NBox)]
```

```
            sort rdup dataSJcar 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 OBAJNres007 =
  OBAJNres007tmp1 feed
    groupby[Id; FirstTime: group feed min[Instant]]{b}
  OBAJNres007tmp1 feed {a}
  hashjoin[Id_b,Id_a]
    filter [(.Id_a = .Id_b) and (.Instant_a <= .FirstTime_b)]
    project[Id_a, Licence_a]
    sortby[Id_a, Licence_a]
    rdup
consume;

# delete temporary object

delete OBAJNres007tmp1;

# Query 8: What are the overall traveled distances of the vehicles with
#          license plate numbers from QueryLicences1 during the periods from
#          QueryPeriods1?

let OBAJNres008 =
  QueryLicences1 feed {l}
    loopsel[ dataSJcar_Licence_btree dataSJcar exactmatch[.Licence_l]]
  QueryPeriods1 feed
    filter [not(isempty(.Period))]{p}
  product
    projectextend[Licence; Period: .Period_p,
                           Dist: round(length(.Trip atperiods .Period_p),3)]
    project[Licence, Period, Dist]
    sortby [Licence asc, Period asc]
consume;


# Query 9: What is the longest distance that was traveled by a vehicle during
#          each of the periods from QueryPeriods?

let OBAJNres009 =
  dataSJcar feed {c}
  QueryPeriods feed
    filter[not(isempty(.Period))]{p}
  product
    projectextend [Id_p, Period_p,
                   Licence_c; Dist: round(length(.Trip_c atperiods .Period_p),3)]
    sortby [Id_p asc, Period_p asc, Dist desc]
    groupby [Id_p, Period_p; Distance: group feed max[Dist]]
    project [Period_p, Distance]
consume;


# Query 10: When and where did the vehicles with license plate numbers from
#           QueryLicences1 meet other vehicles (distance < 3m) and what are
#           the latter licenses?

let OBAJNres010tmp1 =
   QueryLicences1 feed
    loopsel[dataSJcar_Licence_btree dataSJcar exactmatch[.Licence]]
      projectextend[Licence, Trip; TripA: fromnetwork(.Trip), BBox: bbox(.Trip)]
      projectextend[Licence, Trip, TripA; Box: rectangle2((minD(.BBox,1) - 1.5),
                                                          (maxD(.BBox,1) + 1.5),
                                                          (minD(.BBox,2) - 1.5),
                                                          (maxD(.BBox,2) + 1.5))]
consume;

let OBAJNres010 =
  dataSJcar feed
    projectextend[Licence; TripA: fromnetwork(.Trip), BBox: bbox(.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}
  OBAJNres010tmp1 feed {c2}
  symmjoin[((.Box_c1 intersects ..Box_c2) and
            (.Licence_c1 # ..Licence_c2)) and
           (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;

# delete temporary object
```

```
delete OBAJNres010tmp1;

# Query 11: Which vehicles passed a point from QueryPoints1 at one of the
#           instants from QueryInstant1?

let OBAJNres011 =
  QueryPoints1JNetAll feed
    extend[Prect: netbox(.Pos)]
    projectextend[Id, Pos; NBox: rectangle2(minD(.Prect,1), maxD(.Prect, 1),
                                            minD(.Prect,2) - 0.00001,
                                            maxD(.Prect,2) + 0.00001)]
    loopsel[fun(t:TUPLE) dataSJcar_TrajBoxNet windowintersectsS[attr(t,NBox)]
            sort rdup dataSJcar gettuples
            filter [.Trip passes attr(t,Pos)]
            projectextend[Licence; Id: (attr(t,Id)),
                                   TripN: .Trip at attr(t,Pos)]]
  QueryInstant1 feed {i}
  symmjoin[.TripN present ..Instant_i]
    project[Licence, Id, Instant_i]
    sortby[Id, Licence,  Instant_i]
consume;

# Query 12: Which vehicles met at a point from QueryPoints1 at an instant from
#           QueryInstant1?

let OBAJNres012tmp1 =
  QueryInstant1 feed {i}
  QueryPoints1JNetAll feed
    extend[Prect: netbox(.Pos)]
    projectextend[Id, Pos; NBox: rectangle2(minD(.Prect,1), maxD(.Prect,1),
                                            minD(.Prect,2) - 0.00001,
                                            maxD(.Prect,2) + 0.00001)]{p}
  product
    projectextend[Instant_i, Id_p, Pos_p; Box: box3d(.NBox_p, .Instant_i)]
    loopsel[fun(t:TUPLE)
            dataSJcar_BoxNet_timespace windowintersectsS[attr(t,Box)]
            sort rdup dataSJcar gettuples
            projectextend [Licence; Id: attr(t,Id_p),
                                    Instant: attr(t,Instant_i),
                                    Pos: attr(t,Pos_p)]]
      sortby [Id asc, Instant asc, Licence asc, Pos asc]
      rdup
consume;

let OBAJNres012 =
  OBAJNres012tmp1 feed {c1}
  OBAJNres012tmp1 feed {c2}
  symmjoin[((.Licence_c1 < ..Licence_c2) and
           (.Id_c1 = ..Id_c2)) and
           (.Instant_c1 = ..Instant_c2)]
    project[Id_c1, Pos_c1, Instant_c1, Licence_c1, Licence_c2]
    sortby[Id_c1 asc, Instant_c1 asc, Licence_c2 asc]
    rdup
consume;

# delete temporary object

delete OBAJNres012tmp1;

# Query 13: Which vehicles traveled within one of the regions from
#           QueryRegions1 during the periods from QueryPeriods1?

let OBAJNres013 =
  dataSJcar feed {c}
  QueryRegions1JNet 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]
    filter[not(isempty(.Trip))]
  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 [not(isempty(.Trip))]
    project[Id_r, Period_p, Licence]
    sortby[Id_r asc, Period_p asc, Licence asc]
consume;

# Query 14: Which vehicles traveled within one of the regions from
#           QueryRegions1 at one of the instants from QueryInstant1?

let OBAJNres014 =
  QueryRegions1JNet feed
    filter [not(isempty(.Region))]
    projectextendstream[Id, Region; Box: units(.Region)]
    projectextend[Id, Region; BBox: netbox(.Box)]{r}
  QueryInstant1 feed {i}
  product
  loopsel [fun (t:TUPLE)
```

```
          dataSJcar_BoxNet_timespace windowintersectsS[box3d(attr(t,BBox_r),
                                                             attr(t,Instant_i))]
          sort rdup dataSJcar gettuples
          filter[val(.Trip atinstant attr(t,Instant_i)) inside attr(t,Region_r)]
          projectextend[Licence; Instant: attr(t,Instant_i), Id_r: attr(t,Id_r)]]
    sortby[Id_r, Instant, Licence]
    krdup[Id_r, Instant, Licence]
    project[Id_r, Instant, Licence]
consume;

#  Query 15: Which vehicles passed a point from QueryPoints1 during a period
#            from QueryPeriods1?

let OBAJNres015 =
  QueryPoints1JNetAll feed
    extend[Prect: netbox(.Pos)]
    projectextend[Id, Pos; NBox: rectangle2(minD(.Prect,1), maxD(.Prect, 1),
                                            minD(.Prect,2) - 0.00001,
                                            maxD(.Prect,2) + 0.00001)] {p}
  QueryPeriods1 feed filter [not(isempty(.Period))] {t}
  product
  loopsel[fun(t:TUPLE)
          dataSJcar_BoxNet_timespace windowintersectsS[box3d(attr(t,NBox_p),
                                                             attr(t,Period_t))]
          sort rdup dataSJcar gettuples
          filter [.Trip passes attr(t,Pos_p)]
          filter [(.Trip at attr(t,Pos_p)) present attr(t,Period_t)]
          projectextend[Licence; Id_pos: attr(t,Id_p), Times: attr(t,Period_t)]]
    project[Id_pos, Times, Licence]
    sortby[Id_pos asc, Times asc, Licence asc]
    krdup[Id_pos, Times, Licence]
consume;

# Query 16: List the pairs of licenses for vehicles the first from
#           QueryLicences1, the second from QueryLicences2, where the
#           corresponding vehicles are both present within a region from
#           QueryRegions1 during a period from QueryPeriod1, but do not meet
#           each other there and then.

let OBAJNres016 =
  QueryLicences1 feed {l}
      loopjoin [dataSJcar_Licence_btree dataSJcar 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 [not(isempty(.Trip))]
    QueryRegions1JNet 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 [not(isempty(.Trip))]{a}
  QueryLicences2 feed {l}
      loopjoin [dataSJcar_Licence_btree dataSJcar 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 [not(isempty(.Trip))]
    QueryRegions1JNet 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 [not(isempty(.Trip))]{b}
  symmjoin[(((.Id_r_a = ..Id_r_b) and
            (.Id_p_a = ..Id_p_b)) and
            (.Licence_a # ..Licence_b)) and
            (not(.Trip_a intersects ..Trip_b))]
  project [Id_r_a, Period_p_a, Licence_a, Licence_b]
  sortby[Id_r_a, Period_p_a, Licence_a, Licence_b]
consume;


# Query 17: Which points from QueryPoints have been visited by a maximum
#           number of different vehicles?

let OBAJNres017tmp1 =
  QueryPointsJNetAll feed
    extend[Prect: netbox(.Pos)]
    projectextend[Id, Pos; NBox: rectangle2(minD(.Prect,1), maxD(.Prect, 1),
                                            minD(.Prect,2) - 0.00001,
                                            maxD(.Prect,2) + 0.00001)]
    loopjoin[dataSJcar_TrajBoxNet windowintersectsS[.NBox]
             sort rdup dataSJcar gettuples]
      project [Id, Licence]
      sortby [Id, Licence]
      krdup [Id, Licence]
```

```
      groupby[Id; Hits: group feed count]
consume;

let OBAJNres017 =
  OBAJNres017tmp1 feed
    filter [.Hits = (OBAJNres017tmp1 feed max[Hits])]
    project [Id, Hits]
consume;

# delete temporary object

delete OBAJNres017tmp1;

# Store Query Run Times

let QRT_JNET_OBA = SEC2COMMANDS feed consume;

# Uncomment next line to save runtimes on hard disk
#save QRT_JNET_OBA to 'JNetworkOBARunTimes.DAT';

# finsihed close database

close database;
```

The script in the file `JNetwork_TBA-Queries.SEC` executes the 17 queries of the trip based approach of the BerlinMOD Benchmark using the first network implementation.

```
# JNetwork queries for the trip based approach of the BerlinMOD Benchmark.
#
# The script assumes that there is a database berlinmod with a jnetwork data
# model representation of the BerlinMOD Benchmark data.
#
# This database can be generated by the script 'BerlinMOD_DataGenerator.SEC'.
# The network data model representation and accroding indexes can be generated
# with the script 'JNetwork_CreateObjects.SEC'
#
# Start Script Opening the Database#

open database berlinmod;

# Query 1: What are the models of the vehicles with license plate numbers from
#          QueryLicence?

let TBAJNres001 =
  QueryLicences feed {l}
    loopjoin [dataMcar_Licence_btree dataMcar exactmatch[.Licence_l]]
      project [Licence, Model]
consume;


# Query 2: How many vehicles exit that are passenger cars?

let TBAJNres002 =
  dataMcar feed
    filter [.Type = "passenger"]
count;

# Query 3: Where have the vehicles with licenses from QueryLicence1 been at
#          each  instant from QueryInstant1?

let TBAJNres003 =
  QueryLicences1 feed {l}
    loopsel[dataMcar_Licence_btree dataMcar exactmatch[.Licence_l] {ll}]
      loopjoin[dataMJtrip_Moid_btree dataMJtrip exactmatch[.Moid_ll]]
  QueryInstant1 feed {i}
  symmjoin [.Trip present ..Instant_i]
    projectextend[Instant_i, Licence_ll; Pos: val(.Trip atinstant .Instant_i)]
    sortby[Instant_i, Licence_ll]
consume;

# Query 4: Which license plate numbers belong to vehicles that have passed the
#          points from QueryPoints?

let TBAJNres004 =
  QueryPointsJNetAll feed
    extend[Prect: netbox(.Pos)]
    projectextend[Id, Pos; NBox: rectangle2(minD(.Prect,1), maxD(.Prect, 1),
                                            minD(.Prect,2) - 0.00001,
                                            maxD(.Prect,2) + 0.00001)]
    loopjoin[dataMJtrip_TrajBoxNet windowintersectsS[.NBox]
             sort rdup dataMJtrip 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;

# Query 5: What is the minimum distance between places, where a vehicle with a
```

```
#          license from QueryLicences1 and a vehicle with Licenses from
#          QueryLicence2 have been?

let TBAJNres005tmp1 =
  QueryLicences1 feed project[Licence] {LL1}
    loopsel[fun(t:TUPLE)
            dataMcar_Licence_btree dataMcar exactmatch[attr(t,Licence_LL1)] {CAR}
            loopsel[dataMJtrip_Moid_btree dataMJtrip exactmatch[.Moid_CAR]]
              projectextend[;Traj: trajectory(.Trip)]
              aggregateB[Traj; fun(L1: jline, L2: jline) L1 union L2; [const jline value ("JBNet" ())]]
              feed namedtransformstream[Traxj]
              extend[Licence: attr(t,Licence_LL1)]]
      projectextend[Licence; Trax: fromnetwork(.Traxj)]
consume;

let TBAJNres005 =
  QueryLicences2 feed
    project[Licence] {LL2}
    loopsel[fun(s:TUPLE)
            dataMcar_Licence_btree dataMcar exactmatch[attr(s,Licence_LL2)] {CAR}
            loopsel[dataMJtrip_Moid_btree dataMJtrip exactmatch[.Moid_CAR]]
              projectextend[;Traj: trajectory(.Trip)]
              aggregateB[Traj; fun(L3: jline, L4: jline) L3 union L4; [const jline value ("JBNet"())]]
              feed namedtransformstream[Traxj]
              extend[Licence: attr(s,Licence_LL2)]]
      projectextend[Licence; Trax: fromnetwork(.Traxj)]{c2}
  TBAJNres005tmp1 feed {c1}
  product
    projectextend [Licence_c1, Licence_c2; Distance: round(distance(.Trax_c1,
                                                                    .Trax_c2),3)]
    sortby[Licence_c1, Licence_c2]
consume;

# delete temporary object

delete TBAJNres005tmp1;

# Query 6: What are the pairs of license plate numbers of "trucks", that have
#          been as close as 10m or less to each other?

let TBAJNres006tmp1 =
  dataMcar feed filter [.Type = "truck"]
  project [Licence, Moid] {c}
  loopjoin[dataMJtrip_Moid_btree dataMJtrip exactmatch[.Moid_c]]
    projectextend [;Licence: .Licence_c, Ptrip: fromnetwork(.Trip)]
    extend[BBox: bbox(.Ptrip)]
    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 TBAJNres006 =
  TBAJNres006tmp1 feed {c1}
  TBAJNres006tmp1 feed {c2}
  symmjoin[((.Box_c1 intersects ..Box_c2) and
           (.Licence_c1 < ..Licence_c2)) and
           (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 intermediate result

delete TBAJNres006tmp1;

# Query 7: What are the license plate numbers of the "passenger" cars that
#          have reached points from QueryPoints first of all "passenger" cars
#          during the complete observation period?

let TBAJNres007tmp1 =
  QueryPointsJNetAll feed
    extend[Prect: netbox(.Pos)]
    projectextend[Id, Pos; NBox: rectangle2(minD(.Prect,1), maxD(.Prect, 1),
                                            minD(.Prect,2) - 0.00001,
                                            maxD(.Prect,2) + 0.00001)]
    loopsel[fun(t:TUPLE) dataMJtrip_TrajBoxNet windowintersectsS[attr(t,NBox)]
            sort rdup dataMJtrip 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)]]
      filter[not(isempty(.TimeAtPos))]
      sortby [Id asc, TimeAtPos asc]
consume;

let TBAJNres007 =
```

```
  TBAJNres007tmp1 feed
    groupby [Id; FirstTime: group feed min[TimeAtPos]]{b}
  TBAJNres007tmp1 feed {a}
  symmjoin[(..Id_a = .Id_b) and (..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 intermediate result

delete TBAJNres007tmp1;

# Query 8: What are the overall traveled distances of the vehicles with
#          license plate numbers from QueryLicences1 during the periods from
#          QueryPeriods1?

let TBAJNres008 =
  QueryLicences1 feed {l}
    loopjoin[dataMcar_Licence_btree dataMcar exactmatch[.Licence_l]]
      project[Licence, Moid]
      loopsel[fun(t:TUPLE) dataMJtrip_Moid_btree dataMJtrip 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: round(group feed sum[Distance],3)]
    project[Licence, Period, Dist]
    sortby[Licence, Period, Dist]
consume;

# Query 9: What is the longest distance that was traveled by a vehicle during
#          each of the periods from QueryPeriods?

let TBAJNres009 =
  dataMJtrip 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, Period_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: round(group feed max[Dist],3)]
    filter[.Dista > 0.0]
    project[Period_p, Dista]
    sortby[Period_p, Dista]
consume;

# Query 10: When and where did the vehicles with license plate numbers from
#           QueryLicences1 meet other vehicles (distance < 3m) and what are
#           the latter licenses?

let TBAJNres010 =
  QueryLicences1 feed project[Licence] {V1}
    loopsel[fun(t:TUPLE)
            dataMcar_Licence_btree dataMcar exactmatch[attr(t,Licence_V1)]
            project[Moid]
            loopjoin[dataMJtrip_Moid_btree dataMJtrip exactmatch[.Moid] remove[Moid]] {V3}
              extend[T3bbx: bbox(.Trip_V3)]
              extend[PtripA: fromnetwork(.Trip_V3)]
              loopjoin[fun(u:TUPLE) dataMJtrip_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 dataMJtrip gettuples
                       filter[.Moid # attr(u, Moid_V3)]
                       projectextend[Moid; PtripB: fromnetwork(.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[; QueryLicence: attr(t, Licence_V1),
                                OtherLicence: .Licence,
                                Pos: .Trip_V3 atperiods .Times]
                filter[not(isempty(.Pos))]]
      sortby[QueryLicence asc, OtherLicence asc]
      groupby [QueryLicence,
               OtherLicence; AllPos: group feed
                                     aggregateB[Pos; fun(M1:mjpoint, M2:mjpoint)
                                     M1 union M2; [const mjpoint value("JBNet")()]]]
      project[QueryLicence, OtherLicence, AllPos]
      sortby[QueryLicence, OtherLicence, AllPos]
consume;
```

```
# Query 11: Which vehicles passed a point from QueryPoints1 at one of the
#           instants from QueryInstant1?

let TBAJNres011 =
  QueryInstant1 feed {i}
  QueryPoints1JNetAll feed
    extend[Prect: netbox(.Pos)]
    projectextend[Id, Pos; NBox: rectangle2(minD(.Prect,1), maxD(.Prect, 1),
                                            minD(.Prect,2) - 0.00001,
                                            maxD(.Prect,2) + 0.00001)]{p}
  product
    loopsel[fun(t:TUPLE)
            dataMJtrip_BoxNet_timespace windowintersectsS[box3d(attr(t,NBox_p),
                                                                attr(t,Instant_i))]
            sort rdup dataMJtrip 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;

# Query 12: Which vehicles met at a point from QueryPoints1 at an instant from
#           QueryInstant1?

let TBAJNres012tmp1 =
  QueryPoints1JNetAll feed
    extend[Prect: netbox(.Pos)]
    projectextend[Id, Pos; NBox: rectangle2(minD(.Prect,1), maxD(.Prect, 1),
                                            minD(.Prect,2) - 0.00001,
                                            maxD(.Prect,2) + 0.00001)]{p}
  QueryInstant1 feed {i}
  product
    projectextend [Id_p, Pos_p, Instant_i; Box: box3d(.NBox_p, .Instant_i)]
    loopsel[fun(t:TUPLE) dataMJtrip_BoxNet_timespace windowintersectsS[attr(t,Box)]
            sort rdup dataMJtrip 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 TBAJNres012 =
  TBAJNres012tmp1 feed {A}
  TBAJNres012tmp1 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 intermediate result

delete TBAJNres012tmp1;

# Query 13: Which vehicles traveled within one of the regions from
#           QueryRegions1 during the periods from QueryPeriods1?

let TBAJNres013 =
  dataMJtrip feed {c}
  QueryRegions1JNet 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]
    projectextend[Moid_c, Id_r, Period_p; TripA: .Trip atperiods .Period_p]
    filter[not(isempty(.TripA))]
    loopjoin [dataMcar_Moid_btree dataMcar exactmatch[.Moid_c]]
      project[Licence, Id_r, Period_p]
      sortby [Licence asc, Id_r asc, Period_p asc]
      krdup [Licence, Id_r, Period_p]
consume;


# Query 14: Which vehicles traveled within one of the regions from
#           QueryRegions1 at one of the instants from QueryInstant1?

let TBAJNres014 =
  QueryRegions1JNet feed
    filter[not(isempty(.Region))]
    projectextendstream[Id, Region; UReg: units(.Region)]
    extend[Brect: netbox(.UReg)]{r}
  QueryInstant1 feed {i}
  product
  loopsel[fun(t:TUPLE)
          dataMJtrip_BoxNet_timespace windowintersectsS[box3d(attr(t,Brect_r),
                                                              attr(t,Instant_i))]
          sort rdup dataMJtrip 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;

# Query 15: Which vehicles passed a point from QueryPoints1 during a period
#           from QueryPeriods1?

let TBAJNres015 =
  QueryPoints1JNetAll feed
    extend[Prect: netbox(.Pos)]
    projectextend[Id, Pos; NBox: rectangle2(minD(.Prect,1), maxD(.Prect,1),
                                            minD(.Prect,2) - 0.00001,
                                            maxD(.Prect,2) + 0.00001)] {p}
  QueryPeriods1 feed filter[not(isempty(.Period))]{t}
  product
  loopsel[fun(t:TUPLE)
          dataMJtrip_BoxNet_timespace windowintersectsS[box3d(attr(t,NBox_p),
                                                              attr(t,Period_t))]
          sort rdup dataMJtrip 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;

# Query 16: List the pairs of licenses for vehicles the first from
#           QueryLicences1, the second from QueryLicences2, where the
#           corresponding vehicles are both present within a Region from
#           QueryRegions1 during a period from QueryPeriod1, but do not meet
#           each other there and then.

let TBAJNres016 =
  QueryLicences1 feed {l}
    loopjoin [dataMcar_Licence_btree dataMcar exactmatch[.Licence_l]] {a}
      loopjoin[dataMJtrip_Moid_btree dataMJtrip 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[not(isempty(.Trip))]
    QueryRegions1JNet 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 [not(isempty(.Trip))]{a}
  QueryLicences2 feed {l}
    loopjoin [dataMcar_Licence_btree dataMcar exactmatch[.Licence_l]]{a}
      loopjoin[dataMJtrip_Moid_btree dataMJtrip 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[not(isempty(.Trip))]
    QueryRegions1JNet feed filter[not(isempty(.Region))]{r}
    symmjoin [.Trip passes ..Region_r]
      projectextend[Licence, Id_p, Id_r; Trip: .Trip at .Region_r]
      filter [not(isempty(.Trip))]{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, 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;

# Query 17: Which points from QueryPoints have been visited by a maximum
#           number of different vehicles?

let TBAJNres017tmp1 =
  QueryPointsJNetAll feed
    extend[Prect: netbox(.Pos)]
    projectextend[Id, Pos; NBox: rectangle2(minD(.Prect,1), maxD(.Prect, 1),
                                            minD(.Prect,2) - 0.00001,
                                            maxD(.Prect,2) + 0.00001)]
    loopsel[fun(t:TUPLE) dataMJtrip_TrajBoxNet windowintersectsS[attr(t,NBox)]
            sort rdup dataMJtrip 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 TBAJNres017 =
  TBAJNres017tmp1 feed
    filter [.Hits = (TBAJNres017tmp1 feed max[Hits])]
    project [Id_p, Hits]
consume;
```

```
# delete temporary object

delete TBAJNres017tmp1;

# save query runtimes

let QRT_JNET_TBA = SEC2COMMANDS feed  consume;

# Uncomment the next line if you want to save run time information on disk
# save QRT_JNET_TBA to 'JNetworkTBARunTimes.DAT';

# Finish Script and Close Database

close database;
```

5.2.3 Comparison of Query Run Times and Storage Space

Figure 5.1 shows a competition between the query run times of BerlinMOD Benchmark standard version and both network implementations on a workstation with 2 GB main memory, with 2.4 GHz CPU and 1 TB hard disk for different amounts of data (see Table 5.1). The single query run times are marked by different colors query 1 is at the bottom and query 17 on the top of the stack. The queries with remarkable run times for the bigger scalefactors are the queries 4 (fuchsia), 6 (yellow), 7 (black), 9 (grey), 10 (red), 13 (fuchsia), and 17 (orange). Table 5.2 shows the storage space needed by the different data representations for *scalefactor 1.0*.

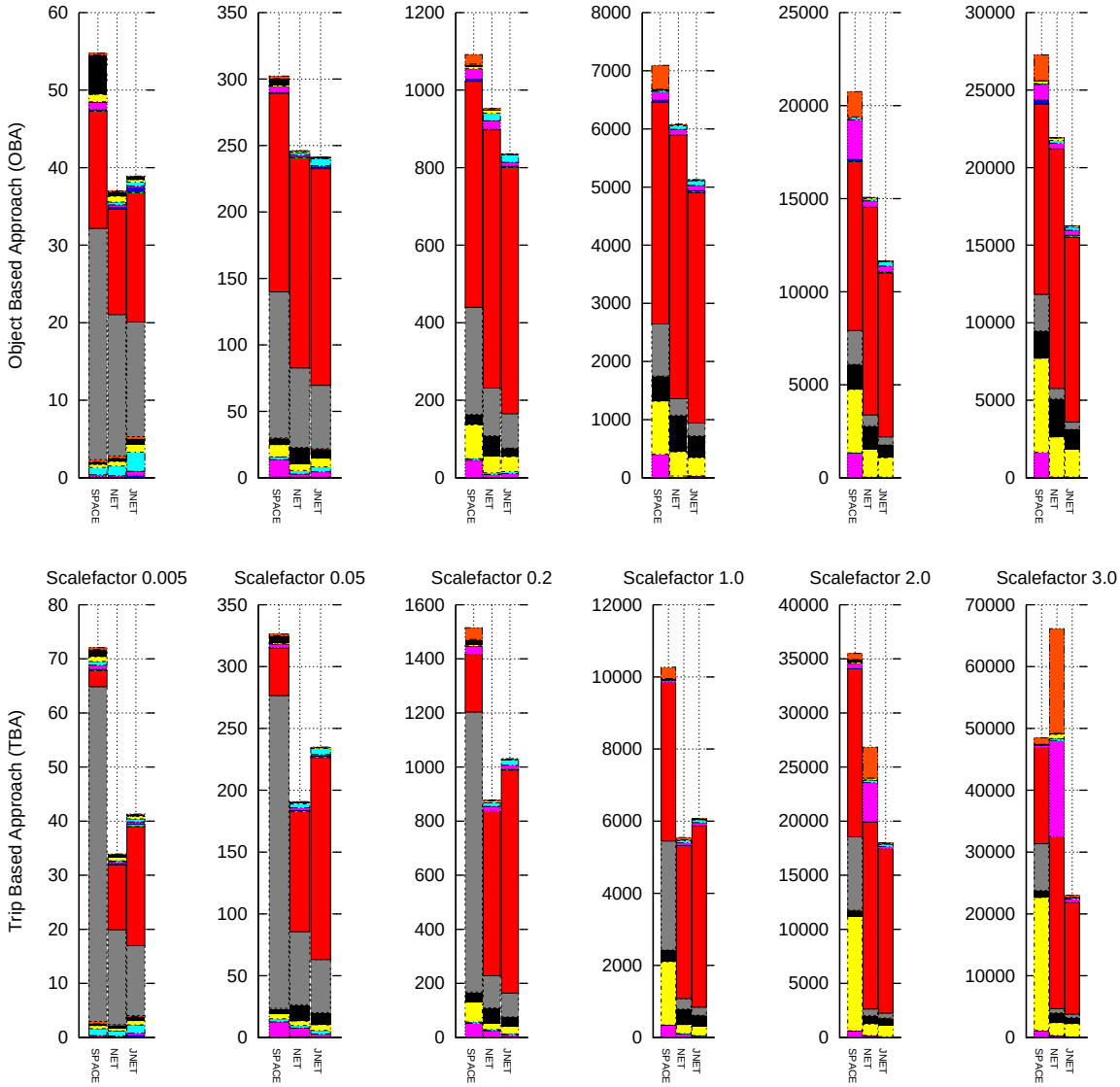

Figure 5.1: Query Run Times for Different Approaches and Scalefactors in Seconds

Scalefactor	Cars	Days	Scalefactor	Cars	Days	Scalefactor	Cars	Days
0.005	141	1	0.2	894	12	2.0	2828	39
0.05	447	6	1.0	2000	28	3.0	3464	48

Table 5.1: Amount of Cars and Days for Different Scalefactors

Object	Space	Net	JNet
network	-	11.1 MiB	35.2 MiB
dataScar	6.9 GiB	7.7 GiB	3.5 GiB
dataMtrip	7.2 GiB	8.0 GiB	4.0 GiB
QueryPoints	16 KiB	20.0 KiB	56 KiB
QueryRegions	1.1 MiB	108 KiB	152 KiB
Data	*14.1 GiB*	*15.7 GiB*	*7.5 GiB*
dataScar_Journey_sptuni	3.4 GiB	-	-
dataScar_Journey_tmpuni	4.1 GiB	-	-
dataScar_Journey_spttmpuni	6.2 GiB	-	-
dataMtrip_sptuni	3.4 GiB	-	-
dataMtrip_tmpuni	4.1 GiB	-	-
dataMtrip_spttmpuni	6.2 GiB	-	-
dataScar_BoxNet_timespace	-	8.2 GiB	4.0 GiB
dataMtrip_BoxNet_timespace	-	8.2 GiB	4.0 GiB
dataScar_TrajBoxNet	-	22.9 MiB	42.2 MiB
dataMtrip_TrajBoxNet	-	151.3 MiB	162.9 MiB
dataMtrip_SpatioTemp	-	36.2 MiB	36.3 MiB
Indexes	*24.4 GiB*	*16.6 GiB*	*8.2 GiB*
Total Storage Space	**38.5 GiB**	**32.3 GiB**	**15.7 GiB**

Table 5.2: Storage Space for Scalefactor 1.0

5.3 Open Street Map Data and Networks

For each network representation we have a script importing street networks provided by [7] in OSM-Format using the operator **fullosmimport** of the SECONDO algebra module `OSMAlgebra`. This import is defined in line 12 of both scripts. The operator gets the name of the file[2] containing the Open Street Map data and a string defining a common prefix for the six relations created by the import operation.

The six relations contain the information of nodes, node tags, ways, way tags, relations and relation tags provided by the OSM-File. These relations are used by the scripts to create the corresponding network respectively jnetwork objects. In both scripts the name of the new database can be defined by the user in line 6, which must correspond to line 8 where the new created database is opened.

5.3.1 Network

`NetworkFromFullOSMImport.SEC` The name of the resulting network object can be defined by the user of the script by editing the last **let** command in the script before the database is closed.

```
# The script imports Openstreetmap data from osm-File and creates a network
# object from this data source
#
# Create and open database

create database keDB;

open database keDB;
```

[2]If the file is not allocated in the `secondo/bin` directory the full qualified path is needed together with the file name.

```
# Define source Thfile for import and network creation

let SOURCEFILE = 'KL_Enkenbach.osm';

# import osm data from file

query fullosmimport(SOURCEFILE, "Osm");

# The tag layer must not be set in osm data if the way is on layer 0.
# Because we decide based on the layer tag if two ways intersect we extend
# the missing layer tag for all ways where no layer is given with value 0.

let WayIdsWithoutTags =
  OsmWays feed
    project [WayId]
    sortby[WayId]
  OsmWayTags feed
    projectextend[;WayId: .WayIdInTag]
    sortby [WayId]
    rdup
  mergediff
consume;

let OsmWayTagsLayerExtended =
  OsmWayTags feed
    sortby [WayIdInTag]
    groupby [WayIdInTag; C: group feed
                                filter [.WayTagKey = "layer"]
                                count]
    filter [.C = 0]
    projectextend [WayIdInTag; WayTagKey: 'layer',
                               WayTagValue: '0']
consume;

let LayerTagForWayIdsWithoutTag =
  WayIdsWithoutTags feed
    projectextend[; WayIdInTag: .WayId,
                    WayTagKey: 'layer',
                    WayTagValue: '0']
consume;

let OsmWayTagNew =
  ( ( OsmWayTags feed)
    ( OsmWayTagsLayerExtended feed)
    concat)
  ( LayerTagForWayIdsWithoutTag feed)
    concat
consume;

# Connect Spatial Information for nodes and ways
# We distinguish between different forms of way curves, because the network
# knows only curves of type sline which may only have one start and end point,
# and may not cross themself. Therefore not curve values values which do not
# fit in this system have to be splitted into disjoint sline values.

let SpatialPosOfNodes =
   OsmNodes feed
     projectextend [NodeId; NodePos: makepoint(.Lon,.Lat)]
consume;

let SpatialWayCurveSimple =
  OsmWays feed
  SpatialPosOfNodes feed
  hashjoin[NodeRef, NodeId, 99997]
    project[WayId, NodeCounter, NodePos]
    sortby [WayId, NodeCounter]
    groupby[WayId; WayCurve: group feed
                                projecttransformstream[NodePos]
                                collect_sline[TRUE]]
consume;

let SpatialWayCurveComplex =
  OsmWays feed
  SpatialWayCurveSimple feed
    filter[not(isdefined(.WayCurve))] {s}
  hashjoin[WayId, WayId_s]
    project[WayId, NodeCounter, NodeRef]
  SpatialPosOfNodes feed
  hashjoin[NodeRef, NodeId, 99997]
    project[WayId, NodeCounter, NodePos]
    sortby [WayId, NodeCounter]
    groupby[WayId; WayCurve: group feed
                                projecttransformstream[NodePos]
                                collect_line[TRUE],
                  StartPointCurve: group feed head[1] extract[NodePos]]
    projectextendstream[WayId, StartPointCurve; WayC: .WayCurve longlines]
    addcounter[PartNo,1]
    projectextend[WayId, PartNo, StartPointCurve,
                  WayC; StartPoint: getstartpoint(.WayC),
                        EndPoint: getendpoint(.WayC)]
    sortby[WayId, PartNo]
```

```
    extend_last[PrevEndPoint: ..EndPoint :: [const point value(0.0 0.0)]]
    sortby[WayId, PartNo]
    projectextend[WayId; WayCurve: ifthenelse(.StartPointCurve = .StartPoint,
                                              .WayC,
                                       ifthenelse(.StartPoint = .PrevEndPoint,
                                                  .WayC,
                                                  set_startsmaller(.WayC,
                                                                   not(get_startsmaller(.WayC))))))]
consume;

let SpatialWayCurve =
  ( SpatialWayCurveSimple feed filter [isdefined(.WayCurve)])
  ( SpatialWayCurveComplex feed)
  concat
    filter [isdefined(.WayCurve)]
consume;

# Collect tag information by identfier

let NestedNodeRel =
  SpatialPosOfNodes feed
  OsmNodeTags feed
  hashjoin[NodeId,NodeIdInTag]
    project[NodeId, NodePos, NodeTagKey, NodeTagValue]
    sortby[NodeId, NodePos, NodeTagKey, NodeTagValue]
    rdup
    nest [NodeId, NodePos; NodeInfo]
consume;

let NestedWayRel =
  SpatialWayCurve feed
  OsmWayTagNew feed
  hashjoin [WayId, WayIdInTag]
    project[WayId, WayCurve, WayTagKey, WayTagValue]
    filter[not(((.WayTagKey = "oneway") and
           ((.WayTagValue = "no") or
            (.WayTagValue = "false") or
            (.WayTagValue = "0"))))]
    sortby[WayId, WayCurve, WayTagKey, WayTagValue]
    rdup
    nest[WayId, WayCurve; WayInfo]
    projectextend[WayId, WayInfo; WayC: .WayCurve,
                  ChangeDirection: ifthenelse(.WayInfo afeed
                                                filter [.WayTagKey = "oneway"]
                                                filter [(.WayTagValue = "-1") or
                                                        (.WayTagValue = "reverse")]
                                              count > 0,
                                              TRUE, FALSE)]
    projectextend[WayId, WayInfo; WayCurve: ifthenelse(.ChangeDirection,
                                                set_startsmaller(.WayC,
                                                      not(get_startsmaller(.WayC))),
                                                .WayC)]
consume;

let NestedRelationRel =
  OsmRelations feed
  OsmRelationTags feed
  hashjoin[RelId, RelIdInTag]
    project[RelId, RefCounter, MemberRef, MemberType, MemberRole, RelTagKey, RelTagValue]
    sortby[RelId, RefCounter,  MemberRef, MemberType, MemberRole, RelTagKey, RelTagValue]
    rdup
    nest [RelId, RefCounter, MemberRef, MemberType, MemberRole; RefInfo]
    nest [RelId; RelInfo]
consume;

# Build roads defined by way relation of osm
# select highway data

let RoadParts =
  NestedWayRel feed
    filter [.WayInfo afeed
                    filter[.WayTagKey = "highway"]
                    filter[(.WayTagValue contains "living") or
                           (.WayTagValue contains "motorway") or
                           (.WayTagValue contains "path") or
                           (.WayTagValue contains "primary") or
                           (.WayTagValue contains "residential") or
                           (.WayTagValue contains "road") or
                           (.WayTagValue contains "secondary") or
                           (.WayTagValue contains "service") or
                           (.WayTagValue contains "tertiary") or
                           (.WayTagValue contains "trunk") or
                           (.WayTagValue contains "track") or
                           (.WayTagValue contains "unclassified") or
                           (.WayTagValue contains "pedestrian")]
            count > 0]
    filter [isdefined(.WayCurve)]
    filter [not(isempty(.WayCurve))]
consume;

# Roads may consists of more than one osm way. The osm ways of very long roads
```

```
# are connectedof by a ref tag which is equal for all ways belonging to the same
# road.
# The roads created by this may not consist of sline values only such that we
# have to distinguish between simple and complex road curves when we generate
# the resulting road curves.

let RoadsByRefH1 =
  RoadParts feed
    filter[not(iscycle(.WayCurve))]
    filter[.WayInfo afeed
                    filter[.WayTagKey = "oneway"]
           count = 0]
    filter[.WayInfo afeed
                    filter[.WayTagKey = "ref"]
          count > 0]
    filter[.WayInfo afeed
                    filter[.WayTagKey = "highway"]
                    filter[not(.WayTagValue contains "link")]
          count > 0]
    unnest[WayInfo]
      filter[.WayTagKey = "ref"]
      projectextendstream[WayId, WayCurve; RefToken: tokenize(' '+.WayTagValue,";/")]
      projectextend[WayId, WayCurve; Ref: trim(toObject('"'+.RefToken +'"',"a"))]
      sortby[Ref, WayCurve]
consume;

let RoadsByRefSimpleH1 =
  RoadsByRefH1 feed
    sortby[Ref, WayCurve]
    groupby[Ref; C: group feed count]
consume;

let RoadsByRefSimpleH2 =
  RoadsByRefSimpleH1 feed
    filter [.C = 1] {r1}
  RoadsByRefH1 feed {r2}
  hashjoin[Ref_r1, Ref_r2]
    projectextend[; Ref: .Ref_r1,
                    RoadCurve: .WayCurve_r2]
consume;

let RoadsByRefSimpleH3 =
  RoadsByRefSimpleH1 feed
    filter [.C > 1] {r1}
  RoadsByRefH1 feed {r2}
  hashjoin[Ref_r1, Ref_r2]
    projectextend[; Ref: .Ref_r2,
                    WayCurve: .WayCurve_r2]
    sortby[Ref, WayCurve]
    groupby[Ref; RoadC: group feed projecttransformstream[WayCurve]
                                   collect_sline[TRUE]]
    projectextend[Ref; RoadCurve: .RoadC]
consume;

let RoadsByRefSimple =
  ( RoadsByRefSimpleH2 feed)
  ( RoadsByRefSimpleH3 feed)
  concat
    sortby[Ref, RoadCurve]
consume;

let RoadsByRefComplex =
  RoadsByRefH1 feed
    sortby[Ref, WayCurve]
  RoadsByRefSimple feed
    filter[not(isdefined(.RoadCurve))] {s}
  hashjoin[Ref, Ref_s]
    projectextend[Ref, WayCurve; StartPoint: getstartpoint(.WayCurve)]
    sortby[Ref, WayCurve, StartPoint]
    groupby[Ref; RoadC: group feed projecttransformstream[WayCurve]
                                   collect_line[TRUE],
                  StartPointCurve: group feed head[1] extract[StartPoint]]
    projectextendstream[Ref, StartPointCurve; RoadCur: .RoadC longlines]
    addcounter[PartNo,1]
    projectextend[Ref, PartNo, StartPointCurve,
                  RoadCur; StartPoint: getstartpoint(.RoadCur),
                           EndPoint: getendpoint(.RoadCur)]
    sortby[Ref, PartNo]
    extend_last[PrevEndPoint: ..EndPoint :: [const point value(0.0 0.0)]]
    sortby[Ref, PartNo]
    projectextend[Ref; RoadCurve: ifthenelse(.StartPointCurve = .StartPoint,
                                             .RoadCur,
                                             ifthenelse(.StartPoint = .PrevEndPoint,
                                                        .RoadCur,
                                                        set_startsmaller(.RoadCur,
                                                                         not(get_startsmaller(.RoadCur)))))]
consume;

let RoadsByRef =
  ( RoadsByRefSimple feed
      filter [isdefined(.RoadCurve)])
```

```
  ( RoadsByRefComplex feed)
  concat
    filter [isdefined(.RoadCurve)]
consume;

# Another form of connecting way segments to long roads is the name tag which
# is equal for all ways belonging to the same road.
# Again we have to distinguish between simple and complex road curves generating
# the resulting road curves.

let RoadsByNameH1 =
  RoadParts feed
    filter[not(iscycle(.WayCurve))]
    filter[.WayInfo afeed
                   filter[.WayTagKey = "oneway"]
            count = 0]
    filter[.WayInfo afeed
                   filter[.WayTagKey = "name"]
            count > 0]
    filter[.WayInfo afeed
                   filter[.WayTagKey = "ref"]
          count = 0]
    filter[.WayInfo afeed
                   filter[.WayTagKey = "highway"]
                   filter[not(.WayTagValue contains "link")]
          count > 0]
    unnest[WayInfo]
      filter[.WayTagKey = "name"]
      projectextend[WayId, WayCurve; Name: trim(toObject('"'+.WayTagValue +'"','a"))]
      sortby[Name, WayCurve]
consume;

let RoadsByNameSimpleH1 =
  RoadsByNameH1 feed
    sortby[Name, WayCurve]
    groupby[Name; C: group feed count]
consume;

let RoadsByNameSimpleH2 =
  RoadsByNameSimpleH1 feed
    filter [.C = 1] {r1}
  RoadsByNameH1 feed {r2}
  hashjoin[Name_r1, Name_r2]
    projectextend[;Name: .Name_r1, RoadCurve: .WayCurve_r2]
consume;

let RoadsByNameSimpleH3 =
  RoadsByNameSimpleH1 feed
    filter [.C > 1] {r1}
  RoadsByNameH1 feed {r2}
  hashjoin[Name_r1, Name_r2]
    projectextend[; Name: .Name_r1, WayCurve: .WayCurve_r2]
    sortby[Name, WayCurve]
    groupby[Name; RoadC: group feed projecttransformstream[WayCurve]
                                   collect_sline[TRUE]]
    projectextend[Name; RoadCurve: .RoadC]
consume;

let RoadsByNameSimple =
  ( RoadsByNameSimpleH2 feed)
  ( RoadsByNameSimpleH3 feed)
  concat
    sortby[Name, RoadCurve]
consume;

let RoadsByNameComplex =
  RoadsByNameH1 feed
  RoadsByNameSimple feed
    filter[not(isdefined(.RoadCurve))] {s}
  hashjoin[Name, Name_s]
    projectextend[Name, WayCurve; StartPoint: getstartpoint(.WayCurve)]
    sortby[Name, WayCurve, StartPoint]
    groupby[Name; RoadC: group feed projecttransformstream[WayCurve]
                                   collect_line[TRUE],
                    StartPointCurve: group feed head[1] extract[StartPoint]]
    projectextendstream[Name, StartPointCurve; RoadCur: .RoadC longlines]
    addcounter[PartNo,1]
    projectextend[Name, PartNo, StartPointCurve,
                  RoadCur; StartPoint: getstartpoint(.RoadCur),
                           EndPoint: getendpoint(.RoadCur)]
    sortby[Name, PartNo]
    extend_last[PrevEndPoint: ..EndPoint :: [const point value(0.0 0.0)]]
    sortby[Name, PartNo]
    projectextend[Name; RoadCurve: ifthenelse(.StartPointCurve = .StartPoint,
                                      .RoadCur,
                                      ifthenelse(.StartPoint = .PrevEndPoint,
                                                 .RoadCur,
                                                 set_startsmaller(.RoadCur,
                                                                  not(get_startsmaller(.RoadCur)))))]
consume;
```

```
let RoadsByName =
  ( RoadsByNameSimple feed filter [isdefined(.RoadCurve)])
  ( RoadsByNameComplex feed)
  concat
    filter [isdefined(.RoadCurve)]
consume;

# road links connect bigger roads for example two highways are connected by
# several links.

let RoadLinksH1 =
  RoadParts feed
    filter[not(iscycle(.WayCurve))]
    filter[.WayInfo afeed
                    filter[.WayTagKey = "oneway"]
           count = 0]
    filter[.WayInfo afeed
                    filter[.WayTagKey = "highway"]
                    filter[.WayTagValue contains "link"]
          count > 0]
  unnest[WayInfo]
    filter[.WayTagKey = "highway"]
    filter[.WayTagValue contains "link"]
    project[WayId, WayCurve]
    sortby[WayId, WayCurve]
consume;

let RoadLinksSimpleH1 =
  RoadLinksH1 feed
    sortby[WayId, WayCurve]
    groupby[WayId; C: group feed count]
consume;

let RoadLinksSimpleH2 =
  RoadLinksSimpleH1 feed
    filter [.C = 1] {r1}
  RoadLinksH1 feed {r2}
  hashjoin [WayId_r1, WayId_r2]
    projectextend[; WayId: .WayId_r1, RoadCurve: .WayCurve_r2]
consume;

let RoadLinksSimpleH3 =
  RoadLinksSimpleH1 feed
    filter [.C > 1] {r1}
  RoadLinksH1 feed {r2}
  hashjoin [WayId_r1, WayId_r2]
    projectextend[; WayId: .WayId_r1, WayCurve: .WayCurve_r2]
    sortby[WayId, WayCurve]
    groupby[WayId; RoadC: group feed projecttransformstream[WayCurve]
                                collect_sline[TRUE]]
    projectextend[WayId; RoadCurve: .RoadC]
consume;

let RoadLinksSimple =
  ( RoadLinksSimpleH2 feed)
  ( RoadLinksSimpleH3 feed)
  concat
    sortby[WayId, RoadCurve]
consume;

let RoadLinksComplex =
  RoadLinksH1 feed
  RoadLinksSimple feed
    filter[not(isdefined(.RoadCurve))] {s}
  hashjoin[WayId, WayId_s]
    projectextend[WayId, WayCurve; StartPoint: getstartpoint(.WayCurve)]
    sortby[WayId, WayCurve, StartPoint]
    groupby[WayId; RoadC: group feed projecttransformstream[WayCurve]
                                collect_line[TRUE],
                  StartPointCurve: group feed head[1] extract[StartPoint]]
    projectextendstream[WayId, StartPointCurve; RoadCur: .RoadC longlines]
    addcounter[PartNo,1]
    projectextend[WayId, PartNo, StartPointCurve,
                  RoadCur; StartPoint: getstartpoint(.RoadCur),
                           EndPoint: getendpoint(.RoadCur)]
    sortby[WayId, PartNo]
    extend_last[PrevEndPoint: ..EndPoint :: [const point value(0.0 0.0)]]
    sortby[WayId, PartNo]
    projectextend[WayId; RoadCurve: ifthenelse(.StartPointCurve = .StartPoint,
                                    .RoadCur,
                                    ifthenelse(.StartPoint = .PrevEndPoint,
                                       .RoadCur,
                                       set_startsmaller(.RoadCur,
                                                        not(get_startsmaller(.RoadCur)))))]
consume;

let RoadLinks =
  ( RoadLinksSimple feed
      filter [isdefined(.RoadCurve)])
  ( RoadLinksComplex feed)
  concat
```

```
    filter [isdefined(.RoadCurve)]
consume;

# all ways not selected before build also valid roads if they are not marked
# to be oneways or roundabouts.

let RoadRestH1 =
  RoadParts feed
    filter[not(iscycle(.WayCurve))]
    filter[.WayInfo afeed
                    filter[.WayTagKey = "oneway"]
           count = 0]
    filter[.WayInfo afeed
                    filter[.WayTagKey contains "ref"]
           count = 0]
    filter[.WayInfo afeed
                    filter[.WayTagKey contains "name"]
           count = 0]
    filter[.WayInfo afeed
                    filter[.WayTagKey = "highway"]
                    filter[.WayTagValue contains "link"]
           count = 0]
    unnest[WayInfo]
      filter[.WayTagKey = "highway"]
      project[WayId, WayCurve]
      sortby[WayId, WayCurve]
consume;

let RoadRestSimpleH1 =
  RoadRestH1 feed
    sortby[WayId, WayCurve]
    groupby[WayId; C: group feed count]
consume;

let RoadRestSimpleH2 =
  RoadRestSimpleH1 feed
    filter [.C = 1] {r1}
  RoadRestH1 feed {r2}
  hashjoin[WayId_r1, WayId_r2]
    projectextend[; WayId: .WayId_r1, RoadCurve: .WayCurve_r2]
consume;

let RoadRestSimpleH3 =
  RoadRestSimpleH1 feed
    filter [.C > 1] {r1}
  RoadRestH1 feed {r2}
  hashjoin[WayId_r1, WayId_r2]
    projectextend[; WayId: .WayId_r1, WayCurve: .WayCurve_r2]
    sortby[WayId, WayCurve]
    groupby[WayId; RoadC: group feed projecttransformstream[WayCurve]
                                collect_sline[TRUE]]
    projectextend[WayId; RoadCurve: .RoadC]
consume;

let RoadRestSimple =
  ( RoadRestSimpleH2 feed)
  ( RoadRestSimpleH3 feed)
  concat
    sortby[WayId, RoadCurve]
consume;

let RoadRestComplex =
  RoadRestH1 feed
  RoadRestSimple feed
    filter[not(isdefined(.RoadCurve))] {s}
  hashjoin[WayId, WayId_s]
    projectextend[WayId, WayCurve; StartPoint: getstartpoint(.WayCurve)]
    sortby[WayId, WayCurve, StartPoint]
    groupby[WayId; RoadC: group feed projecttransformstream[WayCurve]
                                collect_line[TRUE],
                     StartPointCurve: group feed head[1] extract[StartPoint]]
    projectextendstream[WayId, StartPointCurve; RoadCur: .RoadC longlines]
    addcounter[PartNo,1]
    projectextend[WayId, PartNo, StartPointCurve,
                  RoadCur; StartPoint: getstartpoint(.RoadCur),
                           EndPoint: getendpoint(.RoadCur)]
    sortby[WayId, PartNo]
    extend_last[PrevEndPoint: ..EndPoint :: [const point value(0.0 0.0)]]
    sortby[WayId, PartNo]
    projectextend[WayId; RoadCurve: ifthenelse(.StartPointCurve = .StartPoint,
                                      .RoadCur,
                                      ifthenelse(.StartPoint = .PrevEndPoint,
                                                 .RoadCur,
                                                 set_startsmaller(.RoadCur,
                                                                  not(get_startsmaller(.RoadCur)))))]
consume;

let RoadRest =
  ( RoadRestSimple feed
      filter [isdefined(.RoadCurve)])
  ( RoadRestComplex feed)
```

```
  concat
    filter [isdefined(.RoadCurve)]
consume;

# one way streets

let RoadsByOneway =
  RoadParts feed
    filter [not(iscycle(.WayCurve))]
    filter [.WayInfo afeed
                    filter [.WayTagKey = "oneway"]
            count > 0]
    projectextend[; Name: num2string(.WayId), Curve: .WayCurve]
consume;

# road cycles

let RoadsByCycle =
  RoadParts feed
    filter[iscycle(.WayCurve)]
    projectextend[;Name: num2string(.WayId), Curve: set_startsmaller(.WayCurve, TRUE)]
consume;

# concat all forms of roads to roads relation

let Roads =
  ( ( ( RoadsByRef feed
          projectextend[;Name: .Ref, Curve: .RoadCurve])
      ( RoadsByName feed
          projectextend[;Name: .Name, Curve: .RoadCurve])
      concat)
    ( ( RoadLinks feed
          projectextend[;Name: num2string(.WayId), Curve: .RoadCurve])
      ( RoadRest feed
          projectextend[;Name: num2string(.WayId), Curve: .RoadCurve])
      concat)
    concat)
  ( (RoadsByCycle feed)
    (RoadsByOneway feed)
    concat)
  concat
    filter [isdefined(.Curve)]
    extend[CurvLength: size(.Curve)]
    sortby [CurvLength desc]
    addcounter[Rid,1]
consume;

# Build Junctions
# Junction are defined to be crossings between way curves or death ends of a
# road

let ExtendedRoadParts =
  RoadParts feed
    projectextend[WayId; StartPoint: getstartpoint(.WayCurve),
                         EndPoint: getendpoint(.WayCurve)]
consume;

let CrossingPtsTmpH1 =
   RoadParts feed
     unnest[WayInfo]
       filter[.WayTagKey = "layer"]
       projectextend[WayId, WayCurve; Layer: .WayTagValue]
consume;

let CrossingPtsTmp =
   CrossingPtsTmpH1 feed {s1}
   CrossingPtsTmpH1 feed {s2}
   itSpatialJoin [WayCurve_s1, WayCurve_s2, 4 ,8]
     filter[(.Layer_s1 = .Layer_s2)]
     filter[.WayId_s1 < .WayId_s2]
     filter[.WayCurve_s1 intersects .WayCurve_s2]
     projectextendstream[WayId_s1,
                         WayId_s2; Pt: components(crossings(.WayCurve_s1,
                                                            .WayCurve_s2))]
     filter[isdefined(.Pt)]
     projectextend[Pt; WayId1: .WayId_s1, WayId2: .WayId_s2]
consume;

let CrossingsAndRoadPartEndPoints =
  ( ( ExtendedRoadParts feed
        projectextend[WayId; Point: .StartPoint])
    ( ExtendedRoadParts feed
        projectextend[WayId; Point: .EndPoint])
    concat)
  ( ( CrossingPtsTmp feed
        projectextend[; WayId: .WayId1, Point: .Pt])
    ( CrossingPtsTmp feed
        projectextend[; WayId: .WayId2, Point: .Pt])
    concat)
  concat
    sortby[WayId, Point]
```

```
    krduph[WayId, Point]
consume;

let RoadEnds =
  Roads feed
    projectextend[Rid; StartPoint: getstartpoint(.Curve),
                       EndPoint: getendpoint(.Curve)]
consume;

let RoadEndPointsA =
  ( RoadEnds feed
    projectextend[Rid; Point: .StartPoint])
  ( RoadEnds feed
    projectextend[Rid; Point: .EndPoint])
  concat
    sortby[Rid, Point]
    rdup
consume;

let AddJuncs =
  RoadEndPointsA feed
    project[Point]
    sortby[Point]
  CrossingsAndRoadPartEndPoints feed
    project[Point]
    sortby[Point]
  mergediff
consume;

let JunctionIds =
  ( CrossingsAndRoadPartEndPoints feed project[Point])
  ( AddJuncs feed)
  concat
    sortby [Point]
    rdup
    filter [isdefined(.Point)]
    addcounter[Jid,1]
consume;

# The junctions have more than one road position if they are not a death end
# of a road

let JunctionPositionsOnRoads1 =
  CrossingsAndRoadPartEndPoints feed
    filter[isdefined(.Point)]{p1}
  JunctionIds feed
    filter[isdefined(.Point)]{j}
  itSpatialJoin[Point_p1,Point_j,4,8]
    filter [.Point_p1 = .Point_j]
    projectextend[;Jid: .Jid_j, Point: .Point_j, WayId: .WayId_p1]
  RoadParts feed {w}
  hashjoin[WayId, WayId_w]
    projectextend[Jid, Point,
                  WayId; WayCurve: .WayCurve_w,
                         WayStartPoint: getstartpoint(.WayCurve_w),
                         WayEndPoint: getendpoint(.WayCurve_w)]
  Roads feed {r1}
  itSpatialJoin[Point,Curve_r1,4,8]
    filter[.Point inside .Curve_r1]
    filter[.WayStartPoint inside .Curve_r1]
    filter[.WayEndPoint inside .Curve_r1]
    filter[.WayCurve inside .Curve_r1]
    projectextend[Jid, Point,
                  WayId; Rid: .Rid_r1,
                         RMeas: atpoint(.Curve_r1,.Point),
                         WayStartPosOnRoute: atpoint(.Curve_r1, .WayStartPoint),
                         WayEndPosOnRoute: atpoint(.Curve_r1, .WayEndPoint)]
    sortby[Jid, Point, WayId, Rid, RMeas, WayStartPosOnRoute, WayEndPosOnRoute]
    rdup
consume;

let JunctionPositionsOnRoads2 =
  CrossingsAndRoadPartEndPoints feed
    filter[isdefined(.Point)]{p1}
  JunctionIds feed
    filter[isdefined(.Point)]{j}
  itSpatialJoin[Point_p1,Point_j,4,8]
    filter [.Point_p1 = .Point_j]
    projectextend[;Jid: .Jid_j, Point: .Point_j, WayId: .WayId_p1]
    sortby[Jid, Point, WayId]
  JunctionPositionsOnRoads1 feed
    project[Jid, Point, WayId]
    sortby[Jid, Point, WayId]
  mergediff
  RoadParts feed {w}
  hashjoin[WayId, WayId_w]
    projectextend[Jid, Point,
                  WayId; WayCurve: .WayCurve_w,
                         WayStartPoint: getstartpoint(.WayCurve_w),
                         WayEndPoint: getendpoint(.WayCurve_w)]
  Roads feed {r1}
```

```
  itSpatialJoin[Point,Curve_r1,4,8]
    filter[.Point inside .Curve_r1]
    projectextend[Jid, Point,
                  WayId; Rid: .Rid_r1,
                  RMeas: atpoint(.Curve_r1,.Point),
                  WayStartPosOnRoute: atpoint(.Curve_r1,
                                              getstartpoint(intersection(.WayCurve,
                                                                         .Curve_r1))),
                  WayEndPosOnRoute: atpoint(.Curve_r1,
                                            getendpoint(intersection(.WayCurve,
                                                                     .Curve_r1)))]
    sortby[Jid, Point, WayId, Rid, RMeas, WayStartPosOnRoute, WayEndPosOnRoute]
    rdup
consume;

let JunctionPositionsOnRoads =
  (JunctionPositionsOnRoads1 feed)
  (JunctionPositionsOnRoads2 feed)
  concat
consume;

let JunctionsAtRoadEnds =
  RoadEndPointsA feed
    filter[isdefined(.Point)] {r}
  JunctionIds feed
    filter [isdefined(.Point)] {j}
  itSpatialJoin[Point_r,Point_j,4,8]
    filter [.Point_r = .Point_j]
    projectextend[; Jid: .Jid_j, Point: .Point_j, Rid: .Rid_r]
  Roads feed {r1}
  hashjoin[Rid, Rid_r1]
    projectextend[Jid, Point, Rid; RMeas: atpoint(.Curve_r1,.Point)]
    sortby [Jid, Point, Rid, RMeas]
    rdup
consume;

let JunctionsAtRoadEndPairs =
  JunctionsAtRoadEnds feed {j1}
  JunctionsAtRoadEnds feed {j2}
  hashjoin[Jid_j1, Jid_j2]
    filter  [.Rid_j1 <= .Rid_j2]
    filter [.RMeas_j1 # .RMeas_j2]
    projectextend [;Jid: .Jid_j1,
                    R1id: .Rid_j1,
                    R1Meas: .RMeas_j1,
                    R2id: .Rid_j2,
                    R2Meas: .RMeas_j2,
                    NewCC: 65535]
    sortby [Jid, R1id, R1Meas, R2id, R2Meas, NewCC]
    krduph[Jid, R1id, R1Meas, R2id, R2Meas, NewCC]
consume;

let JunctionRoadPairs =
  JunctionPositionsOnRoads feed {j1}
  JunctionPositionsOnRoads feed {j2}
  hashjoin[Jid_j1, Jid_j2]
    filter[.Rid_j1 <= .Rid_j2]
    projectextend[; Jid: .Jid_j1,
                    Point: .Point_j1,
                    R1id: .Rid_j1,
                    R1Meas: .RMeas_j1,
                    R2id: .Rid_j2,
                    R2Meas: .RMeas_j2,
                    CC: 65535,
                    WayId1: .WayId_j1,
                    Way1StartPosOnRoute: .WayStartPosOnRoute_j1,
                    Way1EndPosOnRoute: .WayEndPosOnRoute_j1,
                    WayId2: .WayId_j2,
                    Way2StartPosOnRoute: .WayStartPosOnRoute_j2,
                    Way2EndPosOnRoute: .WayEndPosOnRoute_j2]
    sortby [Jid, R1id, R1Meas, R2id, R2Meas, CC, Way1StartPosOnRoute,
            Way1EndPosOnRoute, Way2StartPosOnRoute, Way2EndPosOnRoute,
            WayId1, WayId2]
    krduph[Jid, R1id, R1Meas, R2id, R2Meas, CC, Way1StartPosOnRoute,
           Way1EndPosOnRoute, Way2StartPosOnRoute, Way2EndPosOnRoute]
consume;

# Compute correct Connectivity Codes for all junctions
#
# Select Road End Points

let RoadEndPoints =
  JunctionRoadPairs feed
    sortby [Jid, R1id, R1Meas, R2id, R2Meas]
    groupby [Jid; C: group feed count]
     filter [.C = 1] {s}
  JunctionRoadPairs feed
  hashjoin[Jid_s, Jid]
    filter[.R1id = .R2id]
    filter[.WayId1 = .WayId2]
    projectextend[Jid, R1id, R1Meas, R2id,
```

```
                R2Meas; NewCC: ifthenelse(.R1Meas = 0, 16, 2)]
consume;

# Select One Way Curves

let OneWayCurveIds =
  NestedWayRel feed
    filter[.WayInfo afeed
                    filter[.WayTagKey = "oneway"]
                    filter[not((.WayTagValue = "no") or
                               (.WayTagValue = "false") or
                               (.WayTagValue = "0"))]
           count > 0]
    project[WayId]
consume;

# motorways are also oneways

let MotorWayCurveIds =
  NestedWayRel feed
    filter[.WayInfo afeed
                    filter [.WayTagKey = "highway"]
                    filter [.WayTagValue = "motorway"]
            count > 0]
    project [WayId]
consume;

# roundabout are also oneways

let RoundaboutWayCurveIds =
  NestedWayRel feed
    filter [.WayInfo afeed
                     filter [.WayTagKey = "junction"]
                     filter [.WayTagValue = "roundabout"]
            count > 0]
    project [WayId]
consume;

let OneWayIds =
  ( ( OneWayCurveIds feed)
    ( RoundaboutWayCurveIds feed)
    concat)
  ( MotorWayCurveIds feed)
  concat
    sortby [WayId]
    rdup
consume;

# set connectivity code for junctions of roundabouts

let JunctionsOfCycles =
  JunctionRoadPairs feed
    filter [.R1id = .R2id]
    filter [.WayId1 = .WayId2]
  RoundaboutWayCurveIds feed
  hashjoin[WayId1, WayId]
    projectextend[Jid, R1id, R1Meas, R2id, R2Meas; NewCC: 1285]
consume;

# set connectivity codes for oneway junctions

let JunctionsOfOnewaysSameRID1 =
  JunctionRoadPairs feed
    filter[.R1id = .R2id]
    filter[.WayId1 # .WayId2]
  OneWayIds feed
  hashjoin[WayId1, WayId]
  projectextend[Jid, R1id, R1Meas, R2id,
                R2Meas; NewCC: ifthenelse(.Way1StartPosOnRoute < .Way1EndPosOnRoute,
                                          ifthenelse(.R1Meas = .Way1EndPosOnRoute,
                                                     .CC binand 21845,
                                                     ifthenelse(.R1Meas = .Way1StartPosOnRoute,
                                                                .CC binand 3855,
                                                                .CC binand 1285)),
                                          ifthenelse(.R1Meas = .Way1EndPosOnRoute,
                                                     .CC binand 43690,
                                                     ifthenelse(.R1Meas = .Way1StartPosOnRoute,
                                                                .CC binand 61680,
                                                                .CC binand 41120)))]
consume;

let JunctionsOfOnewaysSameRID2 =
  JunctionRoadPairs feed
    filter [.R1id = .R2id]
    filter [.WayId1 # .WayId2]
  OneWayIds feed
  hashjoin[WayId2, WayId]
    projectextend[Jid, R1id, R1Meas, R2id,
                  R2Meas; NewCC: ifthenelse(.Way2StartPosOnRoute < .Way2EndPosOnRoute,
                                            ifthenelse(.R2Meas = .Way2EndPosOnRoute,
                                                       .CC binand 21845,
```

```
                                                    ifthenelse(.R2Meas = .Way2StartPosOnRoute,
                                                               .CC binand 3855,
                                                               .CC binand 1285)),
                                           ifthenelse(.R2Meas = .Way2EndPosOnRoute,
                                                      .CC binand 43690,
                                                      ifthenelse(.R2Meas = .Way2StartPosOnRoute,
                                                                 .CC binand 61680,
                                                                 .CC binand 41120)))]
 consume;

let JunctionsWithOneWayOnRouteA =
  JunctionRoadPairs feed
    filter[.R1id < .R2id]
    filter[.WayId1 # .WayId2]
  OneWayIds feed
  hashjoin[WayId1, WayId]
    projectextend[Jid, R1id, R1Meas, R2id,
                  R2Meas; NewCC: ifthenelse(.Way1StartPosOnRoute <= .Way1EndPosOnRoute,
                                            ifthenelse(.R1Meas = .Way1EndPosOnRoute,
                                                       .CC binand 56797,
                                                       ifthenelse(.R1Meas = .Way1StartPosOnRoute,
                                                                  .CC binand 65295,
                                                                  .CC binand 56589)),
                                            ifthenelse(.R1Meas = .Way1EndPosOnRoute,
                                                       .CC binand 61166,
                                                       ifthenelse(.R1Meas = .Way1StartPosOnRoute,
                                                                  .CC binand 65520,
                                                                  .CC binand 61152)))]
consume;

let JunctionsWithOneWayOnRouteB =
  JunctionRoadPairs feed
    filter[.R1id < .R2id]
    filter [.WayId1 # .WayId2]
  OneWayIds feed
  hashjoin[WayId2, WayId]
    projectextend[Jid, R1id, R1Meas, R2id,
                  R2Meas; NewCC: ifthenelse(.Way2StartPosOnRoute <= .Way2EndPosOnRoute,
                                            ifthenelse(.R2Meas = .Way2EndPosOnRoute,
                                                       .CC binand 30583,
                                                       ifthenelse(.R2Meas = .Way2StartPosOnRoute,
                                                                  .CC binand 4095,
                                                                  .CC binand 1911)),
                                            ifthenelse(.R2Meas = .Way2EndPosOnRoute,
                                                       .CC binand 48095,
                                                       ifthenelse(.R2Meas = .Way2StartPosOnRoute,
                                                                  .CC binand 61695,
                                                                  .CC binand 45243)))]
consume;

# Build Relation of Restrictions for way connections

let ViaNodesRel =
  NestedRelationRel feed
    filter [.RelInfo afeed
                    filter [.RefInfo afeed
                                     filter [.RelTagKey contains "restriction"]
                            count > 0]
            count > 0]
    unnest [RelInfo]
      filter[.MemberRole = "via"]
      filter [.MemberType = "node"]
  SpatialPosOfNodes feed
  hashjoin[MemberRef,NodeId]
    projectextend[RelId, NodeId, NodePos; RelTagVal: .RefInfo afeed extract [RelTagValue]]
consume;

let FromWaysRel =
  NestedRelationRel feed
    filter [.RelInfo afeed
                     filter [.RefInfo afeed
                                      filter [.RelTagKey contains "restriction"]
                             count > 0]
            count > 0]
    unnest [RelInfo]
      filter [.MemberRole = "from"]
  SpatialWayCurve feed
  hashjoin[MemberRef, WayId]
    project[RelId, WayId, WayCurve]
consume;

let ToWaysRel =
  NestedRelationRel feed
    filter [.RelInfo afeed
                     filter [.RefInfo afeed
                                      filter [.RelTagKey contains "restriction"]
                             count > 0]
            count > 0]
    unnest [RelInfo]
      filter [.MemberRole = "to"]
  SpatialWayCurve feed
```

```
  hashjoin[MemberRef, WayId]
    project[RelId, WayId, WayCurve]
consume;

let NodeRestrictions =
  ViaNodesRel feed {v}
  FromWaysRel feed {f}
  hashjoin[RelId_v, RelId_f]
    project [RelId_v, NodeId_v, NodePos_v, RelTagVal_v, WayId_f]
  ToWaysRel feed {t}
  hashjoin[RelId_v, RelId_t]
    project[NodeId_v, NodePos_v, RelTagVal_v, WayId_f, WayId_t]
    sortby [NodeId_v, NodePos_v, WayId_f, WayId_t, RelTagVal_v]
    rdup
consume;

# Compute connectivity codes for known restrictions on single junctions

let RestrictedJunctionsAA1 =
  JunctionRoadPairs feed
  NodeRestrictions feed
  itSpatialJoin[Point, NodePos_v,4,8]
    filter[.Point = .NodePos_v]
    filter[.WayId1 = .WayId_f]
    filter[.WayId2 = .WayId_t]
    filter [.R1id = .R2id]
    projectextend [Jid, R1id, R1Meas, R2id,
                   R2Meas;NewCC: ifthenelse(.WayId1 = .WayId2,
                                            ifthenelse(.Way1StartPosOnRoute < .Way1EndPosOnRoute,
                                                       ifthenelse(.Way1StartPosOnRoute = .R1Meas,
                                                                  .CC binand 65519,
                                                                  .CC binand 65533),
                                                       ifthenelse(.Way1StartPosOnRoute = .R1Meas,
                                                                  .CC binand 65533,
                                                                  .CC binand 65519)),
                                            ifthenelse(.Way1StartPosOnRoute < .Way1EndPosOnRoute,
                                                        ifthenelse(.Way1StartPosOnRoute = .R1Meas,
                                                                   ifthenelse(.RelTagVal_v contains "no",
                                                                              .CC binand 65503,
                                                                              .CC binand 65327),
                                                                   ifthenelse(.RelTagVal_v contains "no",
                                                                              .CC binand 65534,
                                                                              .CC binand 65521)),
                                                         ifthenelse(.Way1StartPosOnRoute = .R1Meas,
                                                                   ifthenelse(.RelTagVal_v contains "no",
                                                                              .CC binand 65534,
                                                                              .CC binand 65521),
                                                                   ifthenelse(.RelTagVal_v contains "no",
                                                                              .CC binand 65503,
                                                                              .CC binand 65327))))]
    sortby[Jid, R1id, R1Meas, R2id, R2Meas, NewCC]
consume;

let RestrictedJunctionsAA2 =
  JunctionRoadPairs feed
  NodeRestrictions feed
  itSpatialJoin[Point, NodePos_v,4,8]
    filter[.Point = .NodePos_v]
    filter[.WayId2 = .WayId_f]
    filter[.WayId1 = .WayId_t]
    filter [.R1id = .R2id]
    projectextend [Jid, R1id, R1Meas, R2id,
                   R2Meas;NewCC: ifthenelse(.WayId1 = .WayId2,
                                            ifthenelse(.Way2StartPosOnRoute < .Way2EndPosOnRoute,
                                                       ifthenelse(.Way2StartPosOnRoute = .R2Meas,
                                                                  .CC binand 65519,
                                                                  .CC binand 65533),
                                                       ifthenelse(.Way2StartPosOnRoute = .R2Meas,
                                                                  .CC binand 65533,
                                                                  .CC binand 65519)),
                                            ifthenelse(.Way2StartPosOnRoute < .Way2EndPosOnRoute,
                                                       ifthenelse(.Way2StartPosOnRoute = .R2Meas,
                                                                  ifthenelse(.RelTagVal_v contains "no",
                                                                             .CC binand 65503,
                                                                             .CC binand 65327),
                                                                  ifthenelse(.RelTagVal_v contains "no",
                                                                             .CC binand 65534,
                                                                             .CC binand 65521)),
                                            ifthenelse(.Way2StartPosOnRoute = .R2Meas,
                                                       ifthenelse(.RelTagVal_v contains "no",
                                                                  .CC binand 65534,
                                                                  .CC binand 65521),
                                                       ifthenelse(.RelTagVal_v contains "no",
                                                                  .CC binand 65503,
                                                                  .CC binand 65327))))]
    sortby[Jid, R1id, R1Meas, R2id, R2Meas, NewCC]
consume;

let RestrictedJunctionsAB =
  JunctionRoadPairs feed
  NodeRestrictions feed
```

```
  itSpatialJoin[Point, NodePos_v,4,8]
    filter[.Point = .NodePos_v]
    filter[.WayId1 = .WayId_f]
    filter[.WayId2 = .WayId_t]
    filter[.R1id < .R2id]
    projectextend [Jid, R1id, R1Meas, R2id,
                   R2Meas; NewCC: ifthenelse(.WayId1 = .WayId2,
                                          0,
                                          ifthenelse(.Way1StartPosOnRoute < .Way1EndPosOnRoute,
                                                ifthenelse(.Way2StartPosOnRoute < .Way1EndPosOnRoute,
                                                      ifthenelse(.Way1StartPosOnRoute = .R1Meas,
                                                            ifthenelse(.Way2StartPosOnRoute = .R2Meas,
                                                                  ifthenelse(.RelTagVal_v contains "no",
                                                                        .CC binand 65471,
                                                                        .CC binand 65359),
                                                                  ifthenelse(.RelTagVal_v contains "no",
                                                                        .CC binand 65407,
                                                                        .CC binand 65423)),
                                                            ifthenelse(.Way2StartPosOnRoute = .R2Meas,
                                                                  ifthenelse(.RelTagVal_v contains "no",
                                                                        .CC binand 65531,
                                                                        .CC binand 65524),
                                                                  ifthenelse(.RelTagVal_v contains "no",
                                                                        .CC binand 65527,
                                                                        .CC binand 65528))),
                                                      ifthenelse(.Way1StartPosOnRoute = .R1Meas,
                                                            ifthenelse(.Way2StartPosOnRoute = .R2Meas,
                                                                  ifthenelse(.RelTagVal_v contains "no",
                                                                        .CC binand 65407,
                                                                        .CC binand 65423),
                                                                  ifthenelse(.RelTagVal_v contains "no",
                                                                        .CC binand 65471,
                                                                        .CC binand 65359)),
                                                            ifthenelse(.Way2StartPosOnRoute = .R2Meas,
                                                                  ifthenelse(.RelTagVal_v contains "no",
                                                                        .CC binand 65527,
                                                                        .CC binand 65528),
                                                                  ifthenelse(.RelTagVal_v contains "no",
                                                                        .CC binand 65531,
                                                                        .CC binand 65524)))),
                                                ifthenelse(.Way2StartPosOnRoute < .Way1EndPosOnRoute,
                                                      ifthenelse(.Way1StartPosOnRoute = .R1Meas,
                                                            ifthenelse(.Way2StartPosOnRoute = .R2Meas,
                                                                  ifthenelse(.RelTagVal_v contains "no",
                                                                        .CC binand 65531,
                                                                        .CC binand 655524),
                                                                  ifthenelse(.RelTagVal_v contains "no",
                                                                        .CC binand 65527,
                                                                        .CC binand 65528)),
                                                            ifthenelse(.Way2StartPosOnRoute = .R2Meas,
                                                                  ifthenelse(.RelTagVal_v contains "no",
                                                                        .CC binand 65471,
                                                                        .CC binand 65359),
                                                                  ifthenelse(.RelTagVal_v contains "no",
                                                                        .CC binand 65407,
                                                                        .CC binand 65423))),
                                                      ifthenelse(.Way1StartPosOnRoute = .R1Meas,
                                                            ifthenelse(.Way2StartPosOnRoute = .R2Meas,
                                                                  ifthenelse(.RelTagVal_v contains "no",
                                                                        .CC binand 65527,
                                                                        .CC binand 65528),
                                                                  ifthenelse(.RelTagVal_v contains "no",
                                                                        .CC binand 65531,
                                                                        .CC binand 65524)),
                                                            ifthenelse(.Way2StartPosOnRoute = .R2Meas,
                                                                  ifthenelse(.RelTagVal_v contains "no",
                                                                        .CC binand 65407,
                                                                        .CC binand 65423),
                                                                  ifthenelse(.RelTagVal_v contains "no",
                                                                        .CC binand 65471,
                                                                        .CC binand 65359))))))]
    sortby[Jid, R1id, R1Meas, R2id, R2Meas, NewCC]
consume;

let RestrictedJunctionsBA =
  JunctionRoadPairs feed
  NodeRestrictions feed
  itSpatialJoin[Point, NodePos_v,4,8]
  filter[.Point = .NodePos_v]
  filter[.WayId2 = .WayId_f]
  filter[.WayId1 = .WayId_t]
  filter[.R1id < .R2id]
  projectextend [Jid, R1id, R1Meas, R2id,
                 R2Meas; NewCC: ifthenelse(.WayId1 = .WayId2,
                                        0,
                                        ifthenelse(.Way1StartPosOnRoute < .Way1EndPosOnRoute,
                                              ifthenelse(.Way2StartPosOnRoute < .Way1EndPosOnRoute,
                                                    ifthenelse(.Way1StartPosOnRoute = .R1Meas,
                                                          ifthenelse(.Way2StartPosOnRoute = .R2Meas,
                                                                ifthenelse(.RelTagVal_v contains "no",
                                                                      .CC binand 61439,
```

```
                                                                .CC binand 8191),
                                                          ifthenelse(.RelTagVal_v contains "no",
                                                                .CC binand 65279,
                                                                .CC binand 61951)),
                                                    ifthenelse(.Way2StartPosOnRoute = .R2Meas,
                                                          ifthenelse(.RelTagVal_v contains "no",
                                                                .CC binand 57343,
                                                                .CC binand 12287),
                                                          ifthenelse(.RelTagVal_v contains "no",
                                                                .CC binand 65023,
                                                                .CC binand 62207))),
                                              ifthenelse(.Way1StartPosOnRoute = .R1Meas,
                                                    ifthenelse(.Way2StartPosOnRoute = .R2Meas,
                                                          ifthenelse(.RelTagVal_v contains "no",
                                                                .CC binand 65279,
                                                                .CC binand 61951),
                                                          ifthenelse(.RelTagVal_v contains "no",
                                                                .CC binand 61439,
                                                                .CC binand 8191)),
                                                    ifthenelse(.Way2StartPosOnRoute = .R2Meas,
                                                          ifthenelse(.RelTagVal_v contains "no",
                                                                .CC binand 65023,
                                                                .CC binand 62207),
                                                          ifthenelse(.RelTagVal_v contains "no",
                                                                .CC binand 57343,
                                                                .CC binand 12287)))),
                                        ifthenelse(.Way2StartPosOnRoute < .Way1EndPosOnRoute,
                                              ifthenelse(.Way1StartPosOnRoute = .R1Meas,
                                                    ifthenelse(.Way2StartPosOnRoute = .R2Meas,
                                                          ifthenelse(.RelTagVal_v contains "no",
                                                                .CC binand 57343,
                                                                .CC binand 12287),
                                                          ifthenelse(.RelTagVal_v contains "no",
                                                                .CC binand 65023,
                                                                .CC binand 62207)),
                                                    ifthenelse(.Way2StartPosOnRoute = .R2Meas,
                                                          ifthenelse(.RelTagVal_v contains "no",
                                                                .CC binand 61439,
                                                                .CC binand 8191),
                                                          ifthenelse(.RelTagVal_v contains "no",
                                                                .CC binand 65279,
                                                                .CC binand 61951))),
                                              ifthenelse(.Way1StartPosOnRoute = .R1Meas,
                                                    ifthenelse(.Way2StartPosOnRoute = .R2Meas,
                                                          ifthenelse(.RelTagVal_v contains "no",
                                                                .CC binand 65023,
                                                                .CC binand 62207),
                                                          ifthenelse(.RelTagVal_v contains "no",
                                                                .CC binand 57343,
                                                                .CC binand 12287)),
                                                    ifthenelse(.Way2StartPosOnRoute = .R2Meas,
                                                          ifthenelse(.RelTagVal_v contains "no",
                                                                .CC binand 65279,
                                                                .CC binand 61951),
                                                          ifthenelse(.RelTagVal_v contains "no",
                                                                .CC binand 61439,
                                                                .CC binand 8191)))))]
    sortby[Jid, R1id, R1Meas, R2id, R2Meas, NewCC]
consume;

# compute resulting connectivity code respecting all restrictions computed before
# for each junction. This means a junction may be a oneway and also have
# additional restrictions, such that the connectivity values must be combined.

let Junctions =
  ( ( ( ( JunctionsWithOneWayOnRouteA feed)
        ( JunctionsWithOneWayOnRouteB feed)
        concat)
      ( ( RestrictedJunctionsAA1 feed)
        ( RestrictedJunctionsAA2 feed)
        concat)
      concat)
    ( ( ( RestrictedJunctionsAB feed)
        ( RestrictedJunctionsBA feed)
        concat)
      ( ( JunctionsOfOnewaysSameRID1 feed)
        ( JunctionsOfOnewaysSameRID2 feed)
        concat)
      concat)
    concat)
  ( ( ( RoadEndPoints feed)
      ( JunctionsAtRoadEndPairs feed)
      concat)
    ( ( JunctionRoadPairs feed projectextend[Jid, R1id, R1Meas, R2id, R2Meas; NewCC: .CC])
      ( JunctionsOfCycles feed)
      concat)
    concat)
  concat
    project[Jid, R1id, R1Meas, R2id, R2Meas, NewCC]
    sortby[Jid, R1id, R1Meas, R2id, R2Meas]
    groupby[Jid, R1id, R1Meas, R2id, R2Meas; CC: group feed
```

```
                                                    projecttransformstream[NewCC]
                                                    binands]
consume;

# Build network with all possible restrictions

let KENetwork =
  thenetwork(1,
             0.0001,
             Roads feed
                projectextend[Rid, CurvLength,
                              Curve; Dual: FALSE,
                              StartS: get_startsmaller(.Curve)]
             consume,
             Junctions feed
                project [R1id, R1Meas, R2id, R2Meas, CC]
             consume);

# Script finished close database

close database;
```

5.3.2 JNetwork

`JNetworkFromFullOSMImport.SEC` The name of the resulting jnetwork object can be defined in the query with operation **createjnet** before the database is closed by changing the first parameter.

```
# The script imports Openstreetmap data from osm-File and creates a jnetwork
# object from this data source
#
# Create and open database

create database testdb;

open database testdb;

# Define source file with complete path for import and jnetwork creation

let SOURCEFILE = '/home/jandt/Downloads/OSM-Dateien/MapMatchTest.osm';

# import osm data form file into sisx relations described in OSMAlgebra
# for operator fullosmimport

query fullosmimport(SOURCEFILE, "Osm");

# Extend taginformation with default values for needed but not yet set tags
#
# Set Oneway-Tag for motorways and roundabouts because tag motorway and tag
# roundabout imply oneway definiton in osm

let AddMissingOnewayHighway =
  OsmWayTags feed
  sortby[WayIdInTag]
  groupby[WayIdInTag; Motorway: group feed
                                      filter [.WayTagKey = "highway"]
                                      filter [.WayTagValue = "motorway"]
                                count,
                      Oneway: group feed
                                 filter[.WayTagKey = "oneway"]
                                count]
  filter [.Motorway > 0]
  filter [.Oneway = 0]
  projectextend[WayIdInTag; WayTagKey: 'oneway', WayTagValue: 'yes']
consume;

let AddMissingOnewayRoundabout =
  OsmWayTags feed
  sortby[WayIdInTag]
  groupby[WayIdInTag; Roundabout: group feed
                                    filter [.WayTagKey = "junction"]
                                    filter [.WayTagValue = "roundabout"]
                                  count,
                      Oneway: group feed
                                   filter[.WayTagKey = "oneway"]
                                  count]
  filter [.Roundabout > 0]
  filter [.Oneway = 0]
  projectextend[WayIdInTag; WayTagKey: 'oneway', WayTagValue: 'yes']
consume;

let OsmWayTagNew2 =
  ((AddMissingOnewayRoundabout feed)
   (AddMissingOnewayHighway feed)
   concat)
  (OsmWayTags feed)
  concat
consume;
```

```
# add layertag where it is missing because we need it for crossing computation

let WayIdsWithoutTags =
  OsmWays feed
  project [WayId]
  sortby[WayId]
  OsmWayTags feed
  projectextend[;WayId: .WayIdInTag]
  sortby [WayId]
  rdup
  mergediff
consume;

let OsmWayTagsLayerExtended =
  OsmWayTags feed
  sortby [WayIdInTag]
  groupby [WayIdInTag; C: group feed
                            filter [.WayTagKey = "layer"]
                          count]
  filter [.C = 0]
  projectextend [WayIdInTag; WayTagKey: 'layer',
                 WayTagValue: '0']
consume;

let LayerTagForWayIdsWithoutTag =
  WayIdsWithoutTags feed
  projectextend[; WayIdInTag: .WayId,
    WayTagKey: 'layer',
    WayTagValue: '0']
consume;

let OsmWayTagNew1 =
  ((OsmWayTagNew2 feed)
  (OsmWayTagsLayerExtended feed)
  concat)
  (LayerTagForWayIdsWithoutTag feed)
  concat
consume;

# vmax speed is needed for jnetwork creation to support fastest path computation
# later on

let OsmWayTagsVMaxExtended =
  OsmWayTags feed
  sortby [WayIdInTag]
  groupby [WayIdInTag; C: group feed
                            filter [.WayTagKey = "maxspeed"]
                          count]
  filter [.C = 0]
  projectextend [WayIdInTag; WayTagKey: 'maxspeed',
                 WayTagValue: '0.0']
consume;

let VMaxForWayIdsWithoutTag =
  WayIdsWithoutTags feed
  projectextend[; WayIdInTag: .WayId,
     WayTagKey: 'maxspeed',
     WayTagValue: '0.0']
consume;

let OsmWayTagNew =
  ((OsmWayTagNew1 feed)
  (OsmWayTagsVMaxExtended feed)
  concat)
  (VMaxForWayIdsWithoutTag feed)
  concat
consume;

# Connect Spatial Information from nodes and ways
# We have to distinguish between simple and complex curves because the
# defined way curves may have more than two endpoints or cross themself, what
# is not allowed for jnetwork sline values.

let SpatialPosOfNodes =
   OsmNodes feed
   projectextend [NodeId; NodePos: makepoint(.Lon,.Lat)]
consume;

let SpatialWayCurveSimple =
  OsmWays feed
  SpatialPosOfNodes feed
  hashjoin[NodeRef, NodeId]
  filter[.NodeRef = .NodeId]
  project[WayId, NodeCounter, NodePos]
  sortby [WayId, NodeCounter]
  groupby[WayId; WayCurve: group feed projecttransformstream[NodePos] collect_sline[TRUE]]
consume;

let SpatialWayCurveComplex =
  OsmWays feed
  SpatialWayCurveSimple feed
```

```
    filter[not(isdefined(.WayCurve))] {s}
  hashjoin[WayId, WayId_s]
  filter[.WayId = .WayId_s]
  project[WayId, NodeCounter, NodeRef]
  SpatialPosOfNodes feed
  hashjoin[NodeRef, NodeId]
  filter[.NodeRef = .NodeId]
  project[WayId, NodeCounter, NodePos]
  sortby [WayId, NodeCounter]
  groupby[WayId; WayCurve: group feed projecttransformstream[NodePos] collect_line[TRUE],
                StartPointCurve: group feed head[1] extract[NodePos]]
  projectextendstream[WayId, StartPointCurve; WayC: .WayCurve longlines]
  addcounter[PartNo,1]
  projectextend[WayId, PartNo, StartPointCurve,
    WayC; StartPoint: getstartpoint(.WayC),
      EndPoint: getendpoint(.WayC)]
  sortby[WayId, PartNo]
  extend_last[PrevEndPoint: ..EndPoint :: [const point value(0.0 0.0)]]
  sortby[WayId, PartNo]
  projectextend[WayId; WayCurve: ifthenelse(.StartPointCurve = .StartPoint, .WayC,
                                 ifthenelse(.StartPoint = .PrevEndPoint, .WayC,
                                     set_startsmaller(.WayC, not(get_startsmaller(.WayC)))))]
consume;

let SpatialWayCurve =
  (SpatialWayCurveSimple feed filter [isdefined(.WayCurve)])
  (SpatialWayCurveComplex feed)
  concat
  filter [isdefined(.WayCurve)]
consume;

# Collect tag information by identfier

let NestedNodeRel =
  SpatialPosOfNodes feed
  OsmNodeTags feed
  hashjoin[NodeId,NodeIdInTag]
  filter[.NodeId = .NodeIdInTag]
  project[NodeId, NodePos, NodeTagKey, NodeTagValue]
  sortby[NodeId, NodePos, NodeTagKey, NodeTagValue]
  rdup
  nest [NodeId, NodePos; NodeInfo]
consume;

let NestedWayRel =
  SpatialWayCurve feed
  OsmWayTagNew feed
  hashjoin [WayId, WayIdInTag]
  filter[.WayId = .WayIdInTag]
  project[WayId, WayCurve, WayTagKey, WayTagValue]
  filter[not(((.WayTagKey = "oneway") and ((.WayTagValue = "no") or
                                           (.WayTagValue = "false") or
                                           (.WayTagValue = "0"))))]
  sortby[WayId, WayCurve, WayTagKey, WayTagValue]
  rdup
  nest[WayId, WayCurve; WayInfo]
  projectextend[WayId, WayInfo; WayC: .WayCurve,
    ChangeDirection: ifthenelse(.WayInfo afeed filter [.WayTagKey = "oneway"]
                                              filter [(.WayTagValue = "-1") or
                                                      (.WayTagValue = "reverse")]
                                       count > 0,
                                TRUE, FALSE)]
  projectextend[WayId, WayInfo; WayCurve: ifthenelse(.ChangeDirection,
                        set_startsmaller(.WayC, not(get_startsmaller(.WayC))),
                        .WayC)]
consume;

let NestedRelationRel =
  OsmRelations feed
  OsmRelationTags feed
  hashjoin[RelId, RelIdInTag]
  filter[.RelId = .RelIdInTag]
  project[RelId, RefCounter, MemberRef, MemberType, MemberRole, RelTagKey, RelTagValue]
  sortby[RelId, RefCounter,  MemberRef, MemberType, MemberRole, RelTagKey, RelTagValue]
  rdup
  nest [RelId, RefCounter, MemberRef, MemberType, MemberRole; RefInfo]
  nest [RelId; RelInfo]
consume;


# Select interesting WayCurves for street network


let RoadParts =
  NestedWayRel feed
  filter [.WayInfo afeed
    filter[.WayTagKey = "highway"]
    filter[(.WayTagValue contains "living") or
           (.WayTagValue contains "motorway") or
           (.WayTagValue contains "path") or
           (.WayTagValue contains "primary") or
```

```
            (.WayTagValue contains "residential") or
            (.WayTagValue contains "road") or
            (.WayTagValue contains "secondary") or
            (.WayTagValue contains "service") or
            (.WayTagValue contains "tertiary") or
            (.WayTagValue contains "trunk") or
            (.WayTagValue contains "track") or
            (.WayTagValue contains "unclassified") or
            (.WayTagValue contains "pedestrian")]
    count > 0]
  filter [isdefined(.WayCurve)]
  filter [not(isempty(.WayCurve))]
consume;

# Build Junctions


let ExtendedRoadParts =
  RoadParts feed
  unnest[WayInfo]
  filter[.WayTagKey = "layer"]
  projectextend[WayId; StartPoint: getstartpoint(.WayCurve),
     EndPoint: getendpoint(.WayCurve)]
consume;

let WayEndPoints =
  (ExtendedRoadParts feed projectextend[WayId; Point: .StartPoint])
  (ExtendedRoadParts feed projectextend[WayId; Point: .EndPoint])
  concat
consume;

let CrossingPtsTmpH1 =
   RoadParts feed
   unnest[WayInfo]
   filter[.WayTagKey = "layer"]
   projectextend[WayId, WayCurve; Layer: .WayTagValue]
consume;

let CrossingPtsTmp =
   CrossingPtsTmpH1 feed {s1}
   CrossingPtsTmpH1 feed {s2}
   itSpatialJoin [WayCurve_s1, WayCurve_s2, 4 ,8]
   filter[(.Layer_s1 = .Layer_s2)]
   filter[.WayId_s1 < .WayId_s2]
   filter[.WayCurve_s1 intersects .WayCurve_s2]
   projectextendstream[WayId_s1,
      WayId_s2; Pt: components(crossings(.WayCurve_s1, .WayCurve_s2))]
   filter[isdefined(.Pt)]
   projectextend[Pt; WayId1: .WayId_s1,
    WayId2: .WayId_s2]
consume;

let CrossingPts =
  (CrossingPtsTmp feed projectextend[; WayId: .WayId1, Point: .Pt])
  (CrossingPtsTmp feed projectextend[; WayId: .WayId2, Point: .Pt])
  concat
  sortby [WayId, Point]
  rdup
consume;

let CrossingsAndRoadPartEndPoints =
  (WayEndPoints feed)
  (CrossingPts feed)
  concat
  sortby[WayId, Point]
  krduph[WayId, Point]
consume;

let DeadEndCrossings =
  CrossingsAndRoadPartEndPoints feed
  sortby[WayId, Point]
  rdup
  CrossingPts feed
  sortby[WayId, Point]
  rdup
  mergediff
  projectextend[;Pt: .Point, WayId1: .WayId, WayId2: .WayId]
consume;

let AllCrossings =
  (CrossingPtsTmp feed)
  (DeadEndCrossings feed)
  concat
  sortby[Pt, WayId1, WayId2]
  rdup
consume;

let JunctionIds =
  AllCrossings feed project[Pt]
  sortby [Pt]
  rdup
```

```
  filter [isdefined(.Pt)]
  addcounter[Jid,1]
  projectextend[Jid; Point: .Pt]
consume;

# Split osm ways into network sections at junction points

let RoadPartSectionsTmp1 =
  RoadParts feed
  JunctionIds feed
  itSpatialJoin[WayCurve, Point,4,8]
  filter [.Point inside .WayCurve]
  project[WayId, WayCurve, Point]
  sortby [WayId, WayCurve, Point]
  groupby[WayId, WayCurve; Splitpoints: group feed projecttransformstream[Point] collect_points[TRUE]]
  projectextendstream[WayId; SectCurve: splitslineatpoints(.WayCurve, .Splitpoints)]
   extend[StartPoint: getstartpoint(.SectCurve),
          EndPoint: getendpoint(.SectCurve),
          Lenth: size(.SectCurve)]
  RoadParts feed {r1}
  hashjoin[WayId, WayId_r1]
  filter[.WayId = .WayId_r1]
  projectextend[WayId, SectCurve, StartPoint, EndPoint,
    Lenth; WayCurve: .WayCurve_r1,
      Oneway: ifthenelse(.WayInfo_r1 afeed
                                 filter[.WayTagKey_r1 = "oneway"]
                                 count > 0,TRUE,FALSE),
      RoadType: .WayInfo_r1 afeed filter[.WayTagKey_r1 = "highway"] extract [WayTagValue_r1],
      Speed:  str2real(.WayInfo_r1 afeed filter[.WayTagKey_r1 = "maxspeed"] extract [WayTagValue_r1])]
  extend[VMax: ifthenelse(.Speed > 0.0, .Speed,
        ifthenelse(.RoadType contains "living", 10.0,
        ifthenelse(.RoadType contains "motorway", 200.0,
        ifthenelse(.RoadType contains "path", 5.0,
        ifthenelse(.RoadType contains "primary", 100.0,
        ifthenelse(.RoadType contains "residential", 30.0,
        ifthenelse(.RoadType contains "road", 50.0,
        ifthenelse(.RoadType contains "secondary", 70.0,
        ifthenelse(.RoadType contains "service", 30.0,
        ifthenelse(.RoadType contains "tertiary", 50.0,
        ifthenelse(.RoadType contains "trunk", 130.0,
        ifthenelse(.RoadType contains "track", 10.0,
        ifthenelse(.RoadType contains "unclassified", 50.0,
        ifthenelse(.RoadType contains "pedestrian",5.0,30.0))))))))))))),
    Side: ifthenelse(.Oneway,[const jdirection value(Up)],[const jdirection value(Both)])]
  JunctionIds feed {j1}
  itSpatialJoin[StartPoint, Point_j1,4,8]
  filter [.StartPoint = .Point_j1]
  projectextend[WayId, SectCurve, StartPoint, EndPoint, Lenth, Side, VMax,
    WayCurve; StartJid: .Jid_j1]
  JunctionIds feed {j2}
  itSpatialJoin[EndPoint, Point_j2,4,8]
  filter [.EndPoint = .Point_j2]
  projectextend[WayId, SectCurve, StartPoint, EndPoint, WayCurve, Lenth, Side,
    VMax, StartJid; EndJid: .Jid_j2]
  addcounter[Sid, 1]
consume;

# junctions

let JunctionsAndWayCrossings =
  AllCrossings feed filter[isdefined(.Pt)]
  JunctionIds feed
  itSpatialJoin[Pt, Point,4,8]
  filter[.Pt = .Point]
  project[WayId1, WayId2, Point, Jid]
  RoadPartSectionsTmp1 feed {r1}
  hashjoin[WayId1, WayId_r1]
  filter[.WayId1 = .WayId_r1]
  filter[(.Jid = .StartJid_r1) or (.Jid = .EndJid_r1)]
  projectextend[WayId1, WayId2, Point, Jid; Sid1: .Sid_r1,
    S1SectCurve: .SectCurve_r1,
    S1StartPoint: .StartPoint_r1,
    S1EndPoint: .EndPoint_r1,
    S1StartJid: .StartJid_r1,
    S1EndJid: .EndJid_r1,
    S1Lenth: .Lenth_r1,
    S1Side: .Side_r1,
    S1JuncAtStart: ifthenelse(.StartJid_r1 = .Jid, TRUE, FALSE)]
  RoadPartSectionsTmp1 feed {r2}
  hashjoin[WayId2, WayId_r2]
  filter[.WayId2 = .WayId_r2]
  filter[(.Jid = .StartJid_r2) or (.Jid = .EndJid_r2)]
  projectextend[WayId1, WayId2, Point, Jid, Sid1, S1SectCurve,
    S1StartPoint, S1EndPoint, S1StartJid, S1EndJid, S1Lenth,
    S1JuncAtStart, S1Side; Sid2: .Sid_r2,
    S2SectCurve: .SectCurve_r2,
    S2StartPoint: .StartPoint_r2,
    S2EndPoint: .EndPoint_r2,
    S2StartJid: .StartJid_r2,
    S2EndJid: .EndJid_r2,
    S2Lenth: .Lenth_r2,
```

```
    S2Side: .Side_r2,
    S2JuncAtStart: ifthenelse(.StartJid_r2 = .Jid, TRUE, FALSE)]
consume;

# build roads
#
# by ref

let RoadsByRefH1 =
  RoadParts feed
  filter[not(iscycle(.WayCurve))]
  filter[.WayInfo afeed
    filter[.WayTagKey = "oneway"]
    count = 0]
  filter[.WayInfo afeed
    filter[.WayTagKey = "ref"]
    count > 0]
  filter[.WayInfo afeed
    filter[.WayTagKey = "highway"]
    filter[not(.WayTagValue contains "link")]
    count > 0]
  unnest[WayInfo]
  filter[.WayTagKey = "ref"]
  projectextendstream[WayId, WayCurve; RefToken: tokenize(' '+.WayTagValue,";/")]
  projectextend[WayId, WayCurve; Ref: trim(toObject('"'+.RefToken +'"',"a"))]
  sortby[Ref, WayCurve]
consume;

let RoadsByRefSimpleH1 =
  RoadsByRefH1 feed
  sortby[Ref, WayCurve]
  groupby[Ref; C: group feed count]
consume;

let RoadsByRefSimpleH2 =
  RoadsByRefSimpleH1 feed
  filter [.C = 1] {r1}
  RoadsByRefH1 feed {r2}
  hashjoin[Ref_r1, Ref_r2]
  filter[.Ref_r1 = .Ref_r2]
  projectextend[; Ref: .Ref_r1,
    RoadCurve: .WayCurve_r2]
consume;

let RoadsByRefSimpleH3 =
  RoadsByRefSimpleH1 feed
  filter [.C > 1] {r1}
  RoadsByRefH1 feed {r2}
  hashjoin[Ref_r1, Ref_r2]
  filter[.Ref_r1 = .Ref_r2]
  projectextend[; Ref: .Ref_r2,
    WayCurve: .WayCurve_r2,
    SegStart: getstartpoint(.WayCurve_r2)]
  sortby[Ref, SegStart, WayCurve]
  groupby[Ref; RoadC: group feed projecttransformstream[WayCurve] collect_sline[TRUE],
              StartPointCurve: group feed head[1] extract[SegStart]]
  extend[StartRoadC: getstartpoint(.RoadC)]
  projectextend[Ref; RoadCurve: ifthenelse(.StartPointCurve = .StartRoadC,
                                           .RoadC,
                                           set_startsmaller(.RoadC,
                                               not(get_startsmaller(.RoadC))))]
consume;

let RoadsByRefSimple =
  (RoadsByRefSimpleH2 feed)
  (RoadsByRefSimpleH3 feed)
  concat
  sortby[Ref, RoadCurve]
consume;

let RoadsByRefComplex =
  RoadsByRefH1 feed
  sortby[Ref, WayCurve]
  RoadsByRefSimple feed
    filter[not(isdefined(.RoadCurve))] {s}
  hashjoin[Ref, Ref_s]
  filter[.Ref = .Ref_s]
  projectextend[Ref, WayCurve; StartPoint: getstartpoint(.WayCurve)]
  sortby[Ref, WayCurve, StartPoint]
  groupby[Ref; RoadC: group feed projecttransformstream[WayCurve] collect_line[TRUE],
              StartPointCurve: group feed head[1] extract[StartPoint]]
  projectextendstream[Ref, StartPointCurve; RoadCur: .RoadC longlines]
  addcounter[PartNo,1]
  projectextend[Ref, PartNo, StartPointCurve,
    RoadCur; StartPoint: getstartpoint(.RoadCur),
      EndPoint: getendpoint(.RoadCur)]
  sortby[Ref, PartNo]
  extend_last[PrevEndPoint: ..EndPoint :: [const point value(0.0 0.0)]]
  sortby[Ref, PartNo]
  projectextend[Ref; RoadCurve: ifthenelse(.StartPointCurve = .StartPoint, .RoadCur,
                                ifthenelse(.StartPoint = .PrevEndPoint, .RoadCur,
```

```
                                        set_startsmaller(.RoadCur,
                                            not(get_startsmaller(.RoadCur)))))]
consume;

let RoadsByRef =
  (RoadsByRefSimple feed filter [isdefined(.RoadCurve)])
  (RoadsByRefComplex feed)
  concat
  filter [isdefined(.RoadCurve)]
consume;

# by name

let RoadsByNameH1 =
  RoadParts feed
  filter[not(iscycle(.WayCurve))]
  filter[.WayInfo afeed
    filter[.WayTagKey = "oneway"]
    count = 0]
  filter[.WayInfo afeed
    filter[.WayTagKey = "name"]
    count > 0]
  filter[.WayInfo afeed
    filter[.WayTagKey = "highway"]
    filter[not(.WayTagValue contains "link")]
    count > 0]
  unnest[WayInfo]
  filter[.WayTagKey = "name"]
  projectextend[WayId, WayCurve; Name: trim(toObject('"'+.WayTagValue +'" ',"a"))]
  sortby[Name, WayCurve]
consume;

let RoadsByNameSimpleH1 =
  RoadsByNameH1 feed
  sortby[Name, WayCurve]
  groupby[Name; C: group feed count]
consume;

let RoadsByNameSimpleH2 =
  RoadsByNameSimpleH1 feed
  filter [.C = 1] {r1}
  RoadsByNameH1 feed {r2}
  hashjoin[Name_r1, Name_r2]
  filter[.Name_r1 = .Name_r2]
  projectextend[;Name: .Name_r1,
    RoadCurve: .WayCurve_r2]
consume;

let RoadsByNameSimpleH3 =
  RoadsByNameSimpleH1 feed
  filter [.C > 1] {r1}
  RoadsByNameH1 feed {r2}
  hashjoin[Name_r1, Name_r2]
  filter[.Name_r2 = .Name_r2]
  projectextend[; Name: .Name_r1,
                  WayCurve: .WayCurve_r2,
                SegStartPoint: getstartpoint(.WayCurve_r2)]
  sortby[Name, SegStartPoint, WayCurve]
  groupby[Name; RoadC: group feed projecttransformstream[WayCurve] collect_sline[TRUE],
                StartRoadPoint: group feed head[1] extract [SegStartPoint]]
  extend [StartRoadC: getstartpoint(.RoadC)]
  projectextend[Name; RoadCurve: ifthenelse(.StartRoadPoint = .StartRoadC,
                                            .RoadC,
                                            set_startsmaller(.RoadC,
                                               not(get_startsmaller(.RoadC))))]
consume;

let RoadsByNameSimple =
  (RoadsByNameSimpleH2 feed)
  (RoadsByNameSimpleH3 feed)
   concat
   sortby[Name, RoadCurve]
consume;

let RoadsByNameComplex =
  RoadsByNameH1 feed
  RoadsByNameSimple feed
    filter[not(isdefined(.RoadCurve))] {s}
  hashjoin[Name, Name_s]
  filter[.Name = .Name_s]
  projectextend[Name, WayCurve; StartPoint: getstartpoint(.WayCurve)]
  sortby[Name, WayCurve, StartPoint]
  groupby[Name; RoadC: group feed projecttransformstream[WayCurve] collect_line[TRUE],
               StartPointCurve: group feed head[1] extract[StartPoint]]
  projectextendstream[Name, StartPointCurve; RoadCur: .RoadC longlines]
  addcounter[PartNo,1]
  projectextend[Name, PartNo, StartPointCurve,
    RoadCur; StartPoint: getstartpoint(.RoadCur),
      EndPoint: getendpoint(.RoadCur)]
  sortby[Name, PartNo]
  extend_last[PrevEndPoint: ..EndPoint :: [const point value(0.0 0.0)]]
```

```
  sortby[Name, PartNo]
  projectextend[Name; RoadCurve: ifthenelse(.StartPointCurve = .StartPoint, .RoadCur,
                                    ifthenelse(.StartPoint = .PrevEndPoint, .RoadCur,
                                      set_startsmaller(.RoadCur,
                                              not(get_startsmaller(.RoadCur))))))]
consume;

let RoadsByName =
  (RoadsByNameSimple feed filter [isdefined(.RoadCurve)])
  (RoadsByNameComplex feed)
  concat
  filter [isdefined(.RoadCurve)]
consume;

# road links

let RoadLinksH1 =
  RoadParts feed
  filter[not(iscycle(.WayCurve))]
  filter[.WayInfo afeed
    filter[.WayTagKey = "oneway"]
    count = 0]
  filter[.WayInfo afeed
    filter[.WayTagKey = "highway"]
    filter[.WayTagValue contains "link"]
    count > 0]
  unnest[WayInfo]
  filter[.WayTagKey = "highway"]
  filter[.WayTagValue contains "link"]
  project[WayId, WayCurve]
  sortby[WayId, WayCurve]
consume;

let RoadLinksSimpleH1 =
  RoadLinksH1 feed
  sortby[WayId, WayCurve]
  groupby[WayId; C: group feed count]
consume;

let RoadLinksSimpleH2 =
  RoadLinksSimpleH1 feed
  filter [.C = 1] {r1}
  RoadLinksH1 feed {r2}
  hashjoin [WayId_r1, WayId_r2]
  filter[.WayId_r1 = .WayId_r2]
  projectextend[; WayId: .WayId_r1,
    RoadCurve: .WayCurve_r2]
consume;

let RoadLinksSimpleH3 =
  RoadLinksSimpleH1 feed
  filter [.C > 1] {r1}
  RoadLinksH1 feed {r2}
  hashjoin [WayId_r1, WayId_r2]
  filter[.WayId_r1 = .WayId_r2]
  projectextend[;  WayId: .WayId_r1,
                   WayCurve: .WayCurve_r2,
                   SegStart: getstartpoint(.WayCurve_r2)]
  sortby[WayId, SegStart, WayCurve]
  groupby[WayId; RoadC: group feed projecttransformstream[WayCurve] collect_sline[TRUE],
                 StartRoad: group feed head[1] extract[SegStart]]
  extend[StartRoadC: getstartpoint(.RoadC)]
  projectextend[WayId; RoadCurve: ifthenelse(.StartRoad = .StartRoadC,
                                          .RoadC,
                                          set_startsmaller(.RoadC, not(
                                                 get_startsmaller(.RoadC))))]
consume;

let RoadLinksSimple =
  (RoadLinksSimpleH2 feed)
  (RoadLinksSimpleH3 feed)
  concat
  sortby[WayId, RoadCurve]
consume;

let RoadLinksComplex =
  RoadLinksH1 feed
  RoadLinksSimple feed
    filter[not(isdefined(.RoadCurve))] {s}
  hashjoin[WayId, WayId_s]
  filter[.WayId = .WayId_s]
  projectextend[WayId, WayCurve; StartPoint: getstartpoint(.WayCurve)]
  sortby[WayId, WayCurve, StartPoint]
  groupby[WayId; RoadC: group feed projecttransformstream[WayCurve] collect_line[TRUE],
              StartPointCurve: group feed head[1] extract[StartPoint]]
  projectextendstream[WayId, StartPointCurve; RoadCur: .RoadC longlines]
  addcounter[PartNo,1]
  projectextend[WayId, PartNo, StartPointCurve,
    RoadCur; StartPoint: getstartpoint(.RoadCur),
      EndPoint: getendpoint(.RoadCur)]
  sortby[WayId, PartNo]
```

```
  extend_last[PrevEndPoint: ..EndPoint :: [const point value(0.0 0.0)]]
  sortby[WayId, PartNo]
  projectextend[WayId; RoadCurve: ifthenelse(.StartPointCurve = .StartPoint, .RoadCur,
                                    ifthenelse(.StartPoint = .PrevEndPoint, .RoadCur,
                                       set_startsmaller(.RoadCur,
                                              not(get_startsmaller(.RoadCur)))))]
consume;

let RoadLinks =
  (RoadLinksSimple feed filter [isdefined(.RoadCurve)])
  (RoadLinksComplex feed)
  concat
  filter [isdefined(.RoadCurve)]
consume;

# rest except oneways and cycles

let RoadRestH1 =
  RoadParts feed
  filter[not(iscycle(.WayCurve))]
  filter[.WayInfo afeed
    filter[.WayTagKey = "oneway"]
    count = 0]
  filter [.WayInfo afeed
    filter[.WayTagKey = "highway"]
    filter[.WayTagValue contains "link"]
    count = 0]
  unnest[WayInfo]
  filter[.WayTagKey = "highway"]
  project[WayId, WayCurve]
  sortby[WayId, WayCurve]
consume;

let RoadRestSimpleH1 =
  RoadRestH1 feed
  sortby[WayId, WayCurve]
  groupby[WayId; C: group feed count]
consume;

let RoadRestSimpleH2 =
  RoadRestSimpleH1 feed
  filter [.C = 1] {r1}
  RoadRestH1 feed {r2}
  hashjoin[WayId_r1, WayId_r2]
  filter [.WayId_r1 = .WayId_r2]
  projectextend[; WayId: .WayId_r1,
    RoadCurve: .WayCurve_r2]
consume;

let RoadRestSimpleH3 =
  RoadRestSimpleH1 feed
  filter [.C > 1] {r1}
  RoadRestH1 feed {r2}
  hashjoin[WayId_r1, WayId_r2]
  filter[.WayId_r1 = .WayId_r2]
  projectextend[; WayId: .WayId_r1,
    WayCurve: .WayCurve_r2,
    SegStart: getstartpoint(.WayCurve_r2)]
  sortby[WayId, SegStart, WayCurve]
  groupby[WayId; RoadC: group feed projecttransformstream[WayCurve] collect_sline[TRUE],
                StartRoad: group feed head[1] extract [SegStart]]
  extend[StartRoadC: getstartpoint(.RoadC)]
  projectextend[WayId; RoadCurve: ifthenelse(.StartRoad = .StartRoadC,
                                        .RoadC,
                                    set_startsmaller(.RoadC,
                                         not(get_startsmaller(.RoadC))))]
consume;

let RoadRestSimple =
  (RoadRestSimpleH2 feed)
  (RoadRestSimpleH3 feed)
  concat
  sortby[WayId, RoadCurve]
consume;

let RoadRestComplex =
  RoadRestH1 feed
  RoadRestSimple feed
    filter[not(isdefined(.RoadCurve))] {s}
  hashjoin[WayId, WayId_s]
  filter[.WayId = .WayId_s]
  projectextend[WayId, WayCurve; StartPoint: getstartpoint(.WayCurve)]
  sortby[WayId, WayCurve, StartPoint]
  groupby[WayId; RoadC: group feed projecttransformstream[WayCurve] collect_line[TRUE],
              StartPointCurve: group feed head[1] extract[StartPoint]]
  projectextendstream[WayId, StartPointCurve; RoadCur: .RoadC longlines]
  addcounter[PartNo,1]
  projectextend[WayId, PartNo, StartPointCurve,
    RoadCur; StartPoint: getstartpoint(.RoadCur),
      EndPoint: getendpoint(.RoadCur)]
  sortby[WayId, PartNo]
```

```
  extend_last[PrevEndPoint: ..EndPoint :: [const point value(0.0 0.0)]]
  sortby[WayId, PartNo]
  projectextend[WayId; RoadCurve: ifthenelse(.StartPointCurve = .StartPoint, .RoadCur,
                                  ifthenelse(.StartPoint = .PrevEndPoint, .RoadCur,
                                        set_startsmaller(.RoadCur,
                                                not(get_startsmaller(.RoadCur)))))]
consume;

let RoadRest =
  (RoadRestSimple feed filter [isdefined(.RoadCurve)])
  (RoadRestComplex feed)
  concat
  filter [isdefined(.RoadCurve)]
consume;

# build oneway routes

let RoadsByOneway =
  RoadParts feed
  filter [not(iscycle(.WayCurve))]
  filter [.WayInfo afeed
    filter [.WayTagKey = "oneway"]
    count > 0]
  projectextend[; Name: num2string(.WayId), Curve: .WayCurve]
consume;

# build roundabouts

let RoadsByCycle =
  RoadParts feed
  filter[iscycle(.WayCurve)]
  projectextend[;Name: num2string(.WayId),
    Curve: set_startsmaller(.WayCurve, TRUE)]
consume;

# build roads by relation if relation is a relation of way ids
# relations of relations can not be used for connecting roads yet

let RoadsByRelationH1 =
  NestedRelationRel feed
  filter [.RelInfo afeed
    filter [.RefInfo afeed
       filter[.RelTagKey = "route"]
       filter[.RelTagValue = "road"]
       count > 0]
     count > 0]
  unnest[RelInfo]
  project[RelId, RefCounter, MemberRef, MemberType, MemberRole, RefInfo]
consume;

let RoadsByRelationWaysH1 =
  RoadsByRelationH1 feed
  filter[.MemberType = "way"]
  NestedWayRel feed
  hashjoin[MemberRef, WayId]
  filter[.MemberRef = .WayId]
  project[RelId, MemberRole, RefCounter, WayCurve]
consume;

let RoadsByRelationWaysSimple =
  RoadsByRelationWaysH1 feed
  sortby[RelId, MemberRole, RefCounter, WayCurve]
  groupby[RelId, MemberRole; RoadCurve: group feed projecttransformstream[WayCurve] collect_sline[TRUE]]
consume;

let RoadsByRelationWaysComplex =
  RoadsByRelationWaysH1 feed
  RoadsByRelationWaysSimple feed
    filter[not(isdefined(.RoadCurve))] {t}
  hashjoin[RelId, RelId_t]
  filter[.RelId = .RelId_t]
  sortby[RelId, MemberRole, RefCounter, WayCurve]
  groupby[RelId, MemberRole; RoadC: group feed projecttransformstream[WayCurve] collect_line[TRUE]]
  projectextendstream[RelId, MemberRole; RoadCurve: .RoadC longlines]
consume;

let RoadsByRelationWaysH2 =
  (RoadsByRelationWaysSimple feed)
  (RoadsByRelationWaysComplex feed)
  concat
  NestedRelationRel feed {h}
  hashjoin[RelId, RelId_h]
  filter[.RelId = .RelId_h]
  projectextend[RelId, RoadCurve; RelInfo: .RelInfo_h]
  unnest[RelInfo]
  unnest[RefInfo_h]
  projectextend[RelId, RoadCurve; RelTagKey: .RelTagKey_h,
     RelTagValue: .RelTagValue_h]
  filter [(.RelTagKey = "ref") or (.RelTagKey = "name")]
consume;
```

```
let RoadsByRelationWaysH3 =
  RoadsByRelationWaysH2 feed
  filter[.RelTagKey = "ref"]
  projectextend[RelId, RoadCurve; Name: .RelTagValue]
consume;

let RoadsByRelationWaysH4 =
  RoadsByRelationWaysH2 feed
  RoadsByRelationWaysH3 feed
   projectextend[RelId, RoadCurve; RelTagKey: 'A', RelTagValue: 'A']
  mergediff
  filter[.RelTagKey = "name"]
  projectextend[RelId, RoadCurve; Name: .RelTagValue]
consume;

let RoadsByRelationWays =
  (RoadsByRelationWaysH3 feed)
  (RoadsByRelationWaysH4 feed)
  concat
consume;

# build roads relation from part relations build before

let Roads =
  ( ( (RoadsByRef feed
         projectextend[;Name: .Ref, Curve: .RoadCurve])
      (RoadsByName feed
         projectextend[;Name: .Name, Curve: .RoadCurve])
      concat)
    ( (RoadLinks feed
         projectextend[;Name: num2string(.WayId), Curve: .RoadCurve])
      (RoadRest feed
         projectextend[;Name: num2string(.WayId), Curve: .RoadCurve])
      concat)
    concat)
  ( ( (RoadsByCycle feed)
      (RoadsByOneway feed)
      concat)
    (RoadsByRelationWays feed
      projectextend[; Name: tostring(.Name), Curve: .RoadCurve])
    concat)
  concat
  filter [isdefined(.Curve)]
  extend[CurvLength: size(.Curve)]
  sortby [CurvLength desc, Name, Curve]
  rdup
  addcounter[Rid,1]
consume;

# Connect roads, junctions and sections

let ConnectRoadsAndJunctions =
  JunctionsAndWayCrossings feed
    project [Jid, Point, S1SectCurve, S1Side, S2Side]
  Roads feed
    project[Rid, Curve]
  itSpatialJoin[Point,Curve,4,8]
  filter[.Point inside .Curve]
  projectextend[Jid, Rid; PosOnRoad: atpoint(.Curve, .Point),
                          Side: ifthenelse(.S1SectCurve inside .Curve, .S1Side,
                                                                  .S2Side)]
consume;

let RoadsJunctionsLists =
  ConnectRoadsAndJunctions feed
  project[Rid, PosOnRoad, Jid]
  sortby [Rid, PosOnRoad, Jid]
  groupby[Rid; JuncList: group feed projecttransformstream[Jid] createlist]
consume;

let JunctionsPositionsOnRoads =
  ConnectRoadsAndJunctions feed
  projectextend[Jid; RouteLoc: createrloc(.Rid, .PosOnRoad, .Side)]
  sortby[Jid, RouteLoc]
  groupby[Jid; LocationList: group feed projecttransformstream[RouteLoc] createlist]
consume;

let ConnectSectionsAndRoads =
  RoadPartSectionsTmp1 feed
    project[Sid, SectCurve, StartPoint, EndPoint, Side]
  Roads feed
    project [Rid, Curve]
  itSpatialJoin[SectCurve, Curve, 4, 8]
  filter[.StartPoint inside .Curve]
  filter[.EndPoint inside .Curve]
  filter[.SectCurve inside .Curve]
  projectextend[Rid, Sid, Side; StartPos: atpoint(.Curve, .StartPoint),
    EndPos: atpoint(.Curve,.EndPoint)]
  projectextend[Rid, Sid; RoutePart: createrint(.Rid, .StartPos, .EndPos, .Side)]
consume;
```

```
let SectionsAtRoads =
  ConnectSectionsAndRoads feed
  sortby[Rid, RoutePart, Sid]
  groupby[Rid; ListSids: group feed projecttransformstream[Sid] createlist]
consume;

let SectionRouteIntervals =
  ConnectSectionsAndRoads feed
    project[Sid, RoutePart]
  sortby[Sid, RoutePart]
  groupby[Sid; IntervalList: group feed projecttransformstream[RoutePart] createlist]
consume;

let JunctionsAndWayCrossings2 =
  JunctionsAndWayCrossings feed
  JunctionsPositionsOnRoads feed {j1}
  hashjoin[Jid, Jid_j1]
  filter[.Jid = .Jid_j1]
  projectextend[Jid, Point, WayId1, WayId2, Sid1, S1SectCurve,
    S1StartPoint, S1EndPoint, S1StartJid, S1EndJid, S1Lenth,
    S1JuncAtStart, S1Side, Sid2, S2SectCurve, S2StartPoint, S2EndPoint, S2StartJid,
    S2EndJid, S2Lenth, S2JuncAtStart, S2Side;RoutePositions: .LocationList_j1]
consume;

let JunctionsInAndOutComingSectionsH1 =
  JunctionsAndWayCrossings2 feed
  projectextend[Jid, Sid1,
    Sid2; S1InSect: ifthenelse(.S1Side = [const jdirection value(Both)],TRUE,
                      ifthenelse((.S1Side = [const jdirection value(Up)]) and
                                 not(.S1JuncAtStart),TRUE,
                        ifthenelse((.S1Side = [const jdirection value(Down)]) and
                                 .S1JuncAtStart,TRUE,FALSE))),
          S1OutSect: ifthenelse(.S1Side = [const jdirection value(Both)],TRUE,
                      ifthenelse((.S1Side = [const jdirection value(Up)]) and
                                 .S1JuncAtStart,TRUE,
                        ifthenelse((.S1Side = [const jdirection value(Down)]) and
                                 not(.S1JuncAtStart),TRUE,FALSE))),
          S2InSect: ifthenelse(.S2Side = [const jdirection value(Both)],TRUE,
                      ifthenelse((.S2Side = [const jdirection value(Up)]) and
                                 not(.S2JuncAtStart),TRUE,
                        ifthenelse((.S2Side = [const jdirection value(Down)]) and
                                 .S2JuncAtStart,TRUE,FALSE))),
          S2OutSect: ifthenelse(.S2Side = [const jdirection value(Both)],TRUE,
                      ifthenelse((.S2Side = [const jdirection value(Up)]) and
                                 .S2JuncAtStart,TRUE,
                        ifthenelse((.S2Side = [const jdirection value(Down)]) and
                                 not(.S2JuncAtStart),TRUE,FALSE)))]
consume;

let JunctionsInComingSectionsH1 =
  JunctionsInAndOutComingSectionsH1 feed
  filter [.S1InSect]
  projectextend[Jid; Sid: .Sid1]
consume;

let JunctionsInComingSectionsH2 =
  JunctionsInAndOutComingSectionsH1 feed
  filter [.S2InSect]
  projectextend[Jid; Sid: .Sid2]
consume;

let JunctionsInComingSectionsH3 =
  JunctionsInAndOutComingSectionsH1 feed
  filter[not(.S1InSect or .S2InSect)]
  projectextend[Jid; Sid: [const int value undef]]
consume;

let JunctionsInComingSections =
  ((JunctionsInComingSectionsH1 feed)
   (JunctionsInComingSectionsH2 feed)
   concat)
  (JunctionsInComingSectionsH3 feed)
  concat
  sortby [Jid, Sid]
  groupby[Jid; ListInSections: group feed projecttransformstream[Sid] createlist]
consume;


let JunctionsOutgoingSectionsH1 =
  JunctionsInAndOutComingSectionsH1 feed
  filter [.S1OutSect]
  projectextend[Jid; Sid: .Sid1]
consume;

let JunctionsOutgoingSectionsH2 =
  JunctionsInAndOutComingSectionsH1 feed
  filter [.S2OutSect]
  projectextend[Jid; Sid: .Sid2]
consume;
```

```
let JunctionsOutgoingSectionsH3 =
  JunctionsInAndOutComingSectionsH1 feed
  filter[not(.S1OutSect or .S2OutSect)]
  projectextend[Jid; Sid: [const int value undef]]
consume;

let JunctionsOutgoingSections =
  ((JunctionsOutgoingSectionsH1 feed)
   (JunctionsOutgoingSectionsH2 feed)
   concat)
  (JunctionsOutgoingSectionsH3 feed)
  concat
  sortby [Jid, Sid]
  groupby[Jid; ListOutSections: group feed projecttransformstream[Sid] createlist]
consume;

let JunctionsInAndOutComingSections =
  JunctionsInComingSections feed {i}
  JunctionsOutgoingSections feed {o}
  hashjoin[Jid_i, Jid_o]
  filter[.Jid_i = .Jid_o]
  projectextend[; Jid: .Jid_i,
                  ListInSections: .ListInSections_i,
                  ListOutSections: .ListOutSections_o]
consume;


# Create Inputfiles for Routes and Junctions for JNetwork

let InRoutes =
   Roads feed {r1}
   RoadsJunctionsLists feed {j1}
   hashjoin[Rid_r1, Rid_j1]
   filter[.Rid_r1 = .Rid_j1]
   projectextend[;Rid: .Rid_r1,
     Lenth: .CurvLength_r1,
     JuncList: .JuncList_j1]
   SectionsAtRoads feed {s1}
   hashjoin[Rid, Rid_s1]
   filter[.Rid = .Rid_s1]
   projectextend[Rid, JuncList, Lenth; SectList: .ListSids_s1]
   project[Rid, JuncList, SectList, Lenth]
   sortby [Rid]
   rdup
consume;

let InJunctions =
   JunctionsAndWayCrossings2 feed
   project[Jid, Point, RoutePositions]
   JunctionsInAndOutComingSections feed {s1}
   hashjoin[Jid, Jid_s1]
   filter[.Jid = .Jid_s1]
   projectextend[Jid, Point, RoutePositions; InSects: .ListInSections_s1,
                                             OutSects: .ListOutSections_s1]
   sortby[Jid]
   rdup
consume;

# Compute Adjacency lists for sections

let AdjacentSectionsUpHa =
  RoadPartSectionsTmp1 feed
  InJunctions feed
  hashjoin[EndJid, Jid]
  filter[.EndJid = .Jid]
  filter[.Side # [const jdirection value(Down)]]
  project[Sid, OutSects]
  sortby [Sid, OutSects]
  groupby [Sid; AdjSectUp: group feed projecttransformstream[OutSects] createlist]
consume;

let AddAdjacentSectionsUpMissing =
  RoadPartSectionsTmp1 feed project[Sid]
    sortby[Sid]
  AdjacentSectionsUpHa feed project[Sid]
    sortby[Sid]
  mergediff
  project[Sid] {s1}
  RoadPartSectionsTmp1 feed {s2}
  hashjoin[Sid_s1, Sid_s2]
  filter[.Sid_s1 = .Sid_s2]
  projectextend[;Sid: .Sid_s1,
    AdjSectUp: [const listint value(undefined)]]
consume;

let AdjacentSectionsUpH1 =
  (AdjacentSectionsUpHa feed)
  (AddAdjacentSectionsUpMissing feed)
  concat
consume;
```

```
let AdjacentSectionsDownHa =
  RoadPartSectionsTmp1 feed
  InJunctions feed
  hashjoin[StartJid, Jid]
  filter[.StartJid = .Jid]
  filter[.Side # [const jdirection value(Up)]]
  project[Sid, OutSects]
  sortby [Sid, OutSects]
  groupby [Sid; AdjSectDown: group feed projecttransformstream[OutSects] createlist]
consume;

let AddAdjacentSectionsDownMissing =
  RoadPartSectionsTmp1 feed project[Sid]
    sortby[Sid]
  AdjacentSectionsDownHa feed project[Sid]
    sortby[Sid]
  mergediff
  project[Sid] {s1}
  RoadPartSectionsTmp1 feed {s2}
  hashjoin[Sid_s1, Sid_s2]
  filter[.Sid_s1 = .Sid_s2]
  projectextend[;Sid: .Sid_s1,
    AdjSectDown: [const listint value(undefined)]]
consume;

let AdjacentSectionsDownH1 =
  (AdjacentSectionsDownHa feed)
  (AddAdjacentSectionsDownMissing feed)
  concat
  sortby[Sid]
consume;

let ReverseAdjacentSectionsUpHa =
  RoadPartSectionsTmp1 feed
  InJunctions feed
  hashjoin[StartJid, Jid]
  filter[.StartJid = .Jid]
  filter[.Side # [const jdirection value(Down)]]
  project[Sid, InSects]
  sortby [Sid, InSects]
  groupby [Sid; RevAdjSectUp: group feed projecttransformstream[InSects] createlist]
consume;

let AddReverseAdjacentSectionsUpMissing =
  RoadPartSectionsTmp1 feed project[Sid]
    sortby[Sid]
  ReverseAdjacentSectionsUpHa feed project[Sid]
    sortby[Sid]
  mergediff
  project[Sid] {s1}
  RoadPartSectionsTmp1 feed {s2}
  hashjoin[Sid_s1, Sid_s2]
  filter[.Sid_s1 = .Sid_s2]
  projectextend[;Sid: .Sid_s1,
    RevAdjSectUp: [const listint value(undefined)]]
consume;

let ReverseAdjacentSectionsUpH1 =
  (ReverseAdjacentSectionsUpHa feed)
  (AddReverseAdjacentSectionsUpMissing feed)
  concat
  sortby[Sid]
consume;

let ReverseAdjacentSectionsDownHa =
  RoadPartSectionsTmp1 feed
  InJunctions feed
  hashjoin[EndJid, Jid]
  filter [.EndJid = .Jid]
  filter [.Side # [const jdirection value(Up)]]
  project[Sid, InSects]
  sortby [Sid, InSects]
  groupby [Sid; RevAdjSectDown: group feed projecttransformstream[InSects] createlist]
consume;

let AddReverseAdjacentSectionsDownMissing =
  RoadPartSectionsTmp1 feed project[Sid]
    sortby[Sid]
  ReverseAdjacentSectionsDownHa feed project[Sid]
    sortby[Sid]
  mergediff
  project[Sid] {s1}
  RoadPartSectionsTmp1 feed {s2}
  hashjoin[Sid_s1, Sid_s2]
  filter[.Sid_s1 = .Sid_s2]
  projectextend[;Sid: .Sid_s1,
    RevAdjSectDown: [const listint value(undefined)]]
consume;

let ReverseAdjacentSectionsDownH1 =
  (ReverseAdjacentSectionsDownHa feed)
```

```
  (AddReverseAdjacentSectionsDownMissing feed)
  concat
  sortby [Sid]
consume;

# Restrictions for section connections by nodes

let ViaNodesRel =
  NestedRelationRel feed
  filter [.RelInfo afeed
    filter [.RefInfo afeed
      filter [.RelTagKey contains "restriction"]
      count > 0]
    count > 0]
  unnest [RelInfo]
  filter[.MemberRole = "via"]
  filter [.MemberType = "node"]
  SpatialPosOfNodes feed
  hashjoin[MemberRef,NodeId]
  filter[.MemberRef = .NodeId]
  projectextend[RelId, NodeId,
    NodePos; RelTagVal: .RefInfo afeed extract [RelTagValue]]
consume;

let FromWaysRel =
  NestedRelationRel feed
  filter [.RelInfo afeed
    filter [.RefInfo afeed
      filter [.RelTagKey contains "restriction"]
      count > 0]
    count > 0]
  unnest [RelInfo]
  filter [.MemberRole = "from"]
  SpatialWayCurve feed
  hashjoin[MemberRef, WayId]
  filter[.MemberRef = .WayId]
  project[RelId, WayId, WayCurve]
consume;

let ToWaysRel =
  NestedRelationRel feed
  filter [.RelInfo afeed
    filter [.RefInfo afeed
      filter [.RelTagKey contains "restriction"]
      count > 0]
    count > 0]
  unnest [RelInfo]
  filter [.MemberRole = "to"]
  SpatialWayCurve feed
  hashjoin[MemberRef, WayId]
  filter[.MemberRef = .WayId]
  project[RelId, WayId, WayCurve]
consume;

let NodeRestrictions =
  ViaNodesRel feed {v}
  FromWaysRel feed {f}
  hashjoin[RelId_v, RelId_f]
  filter[.RelId_v = .RelId_f]
  project [RelId_v, NodeId_v, NodePos_v, RelTagVal_v, WayId_f]
  ToWaysRel feed {t}
  hashjoin[RelId_v, RelId_t]
  filter[.RelId_v = .RelId_t]
  project[NodeId_v, NodePos_v, RelTagVal_v, WayId_f, WayId_t]
  sortby [NodeId_v, NodePos_v, WayId_f, WayId_t, RelTagVal_v]
  rdup
consume;

let ConnectRestrictionsWithJunctions =
  JunctionsAndWayCrossings feed
  NodeRestrictions feed
  itSpatialJoin[Point, NodePos_v,4,8]
  filter[.Point = .NodePos_v]
  filter[((.WayId_f = .WayId1) and (.WayId_t = .WayId2)) or
         ((.WayId_f = .WayId2) and (.WayId_t = .WayId1))]
  projectextend[RelTagVal_v; Sid_f: ifthenelse(.WayId_f = .WayId1, .Sid1, .Sid2),
                             Sid_t: ifthenelse(.WayId_t = .WayId1, .Sid1, .Sid2)]
consume;

# remove not connected sections from adjacency lists

let NoRestrictions =
  ConnectRestrictionsWithJunctions feed
  filter [.RelTagVal_v contains "no"]
consume;

let AdjacentSectionsUpH2 =
  AdjacentSectionsUpH1 feed
  NoRestrictions feed
  hashjoin[Sid, Sid_f]
  filter[.Sid = .Sid_f]
```

```
  projectextend[Sid; AdjSectU: .AdjSectUp - .Sid_t]
consume;

let AdjacentSectionsUpH3 =
  AdjacentSectionsUpH1 feed project[Sid]
    sortby[Sid]
  AdjacentSectionsUpH2 feed project[Sid]
    sortby[Sid]
  mergediff
  AdjacentSectionsUpH1 feed {s1}
  hashjoin[Sid, Sid_s1]
  filter[.Sid = .Sid_s1]
  projectextend[;Sid: .Sid_s1,
    AdjSectU: .AdjSectUp_s1]
consume;

let AdjacentSectionsUpH4 =
  (AdjacentSectionsUpH2 feed)
  (AdjacentSectionsUpH3 feed)
  concat
  projectextend[Sid; AdjSectUp: .AdjSectU]
consume;

let AdjacentSectionsDownH2 =
  AdjacentSectionsDownH1 feed
  NoRestrictions feed
  hashjoin[Sid, Sid_f]
  filter[.Sid = .Sid_f]
  projectextend[Sid; AdjSectD: .AdjSectDown - .Sid_t]
consume;

let AdjacentSectionsDownH3 =
  AdjacentSectionsDownH1 feed project[Sid]
    sortby[Sid]
  AdjacentSectionsDownH2 feed project[Sid]
    sortby[Sid]
  mergediff
  AdjacentSectionsDownH1 feed {s1}
  hashjoin[Sid, Sid_s1]
  filter[.Sid = .Sid_s1]
  projectextend[;Sid: .Sid_s1,
    AdjSectD: .AdjSectDown_s1]
consume;

let AdjacentSectionsDownH4 =
  (AdjacentSectionsDownH2 feed)
  (AdjacentSectionsDownH3 feed)
  concat
  projectextend[Sid; AdjSectDown: .AdjSectD]
consume;

let ReverseAdjacentSectionsUpH2 =
  ReverseAdjacentSectionsUpH1 feed
  NoRestrictions feed
  hashjoin[Sid, Sid_t]
  filter [.Sid = .Sid_t]
  projectextend[Sid; RevAdjSectU: .RevAdjSectUp - .Sid_f]
consume;

let ReverseAdjacentSectionsUpH3 =
  ReverseAdjacentSectionsUpH1 feed project[Sid]
    sortby[Sid]
  ReverseAdjacentSectionsUpH2 feed project[Sid]
    sortby[Sid]
  mergediff
  ReverseAdjacentSectionsUpH1 feed {s1}
  hashjoin[Sid, Sid_s1]
  filter[.Sid = .Sid_s1]
  projectextend[;Sid: .Sid_s1,
    RevAdjSectU: .RevAdjSectUp_s1]
consume;

let ReverseAdjacentSectionsUpH4 =
  (ReverseAdjacentSectionsUpH2 feed)
  (ReverseAdjacentSectionsUpH3 feed)
  concat
  projectextend[Sid; RevAdjSectUp: .RevAdjSectU]
consume;

let ReverseAdjacentSectionsDownH2 =
  ReverseAdjacentSectionsDownH1 feed
  NoRestrictions feed
  hashjoin[Sid, Sid_t]
  filter[.Sid = .Sid_t]
  projectextend[Sid; RevAdjSectD: .RevAdjSectDown - .Sid_f]
consume;

let ReverseAdjacentSectionsDownH3 =
  ReverseAdjacentSectionsDownH1 feed project[Sid]
    sortby[Sid]
  ReverseAdjacentSectionsDownH2 feed project[Sid]
```

```
    sortby[Sid]
  mergediff
  ReverseAdjacentSectionsDownH1 feed {s1}
  hashjoin[Sid, Sid_s1]
  filter[.Sid = .Sid_s1]
  projectextend[;Sid: .Sid_s1,
    RevAdjSectD: .RevAdjSectDown_s1]
consume;

let ReverseAdjacentSectionsDownH4 =
  (ReverseAdjacentSectionsDownH2 feed)
  (ReverseAdjacentSectionsDownH3 feed)
  concat
  projectextend[Sid; RevAdjSectDown: .RevAdjSectD]
consume;

# leave only existing connections in adjacency lists

let OnlyRestrictions =
  ConnectRestrictionsWithJunctions feed
  filter [.RelTagVal_v contains "only"]
consume;

let AdjacentSectionsUpH5 =
  AdjacentSectionsUpH4 feed
  OnlyRestrictions feed
  hashjoin[Sid, Sid_f]
  filter[.Sid = .Sid_f]
  projectextend[Sid; AdjSectU: restrict(.AdjSectUp, .Sid_t)]
consume;

let AdjacentSectionsUpH6 =
  AdjacentSectionsUpH4 feed project[Sid]
    sortby[Sid]
  AdjacentSectionsUpH5 feed project[Sid]
    sortby[Sid]
  mergediff
  AdjacentSectionsUpH4 feed {s1}
  hashjoin[Sid, Sid_s1]
  filter[.Sid = .Sid_s1]
  projectextend[;Sid: .Sid_s1,
    AdjSectU: .AdjSectUp_s1]
consume;

let AdjacentSectionsUp =
  (AdjacentSectionsUpH5 feed)
  (AdjacentSectionsUpH6 feed)
  concat
  projectextend[Sid; AdjSectUp: .AdjSectU]
consume;

let AdjacentSectionsDownH5 =
  AdjacentSectionsDownH4 feed
  OnlyRestrictions feed
  hashjoin[Sid, Sid_f]
  filter[.Sid = .Sid_f]
  projectextend[Sid; AdjSectD: restrict (.AdjSectDown, .Sid_t)]
consume;

let AdjacentSectionsDownH6 =
  AdjacentSectionsDownH4 feed project[Sid]
    sortby[Sid]
  AdjacentSectionsDownH5 feed project[Sid]
    sortby[Sid]
  mergediff
  AdjacentSectionsDownH4 feed {s1}
  hashjoin[Sid, Sid_s1]
  filter[.Sid = .Sid_s1]
  projectextend[;Sid: .Sid_s1,
    AdjSectD: .AdjSectDown_s1]
consume;

let AdjacentSectionsDown =
  (AdjacentSectionsDownH5 feed)
  (AdjacentSectionsDownH6 feed)
  concat
  projectextend[Sid; AdjSectDown: .AdjSectD]
consume;

let NotLongerAdjacentSections =
   (AdjacentSectionsUpH5 feed projectextend[Sid; AdjSect: .AdjSectU])
   (AdjacentSectionsDownH5 feed projectextend[Sid; AdjSect: .AdjSectD])
   concat
   OnlyRestrictions feed
   hashjoin[Sid, Sid_f]
   filter[.Sid = .Sid_f]
   projectextend[Sid; NotLongerAdj: .AdjSect - .Sid_t]
   projectextendstream[Sid; AdjSect: createstream(.NotLongerAdj)]
   sortby [AdjSect, Sid]
   rdup
   groupby[AdjSect; Sids: group feed projecttransformstream [Sid] createlist]
```

```
consume;

let ReverseAdjacentSectionsUpH5 =
   ReverseAdjacentSectionsUpH4 feed
   NotLongerAdjacentSections feed {s1}
   hashjoin[Sid, AdjSect_s1]
   filter[.Sid = .AdjSect_s1]
   projectextend[Sid; RevAdjSectU: .RevAdjSectUp - .Sids_s1]
consume;

let ReverseAdjacentSectionsUpH6 =
  ReverseAdjacentSectionsUpH4 feed project[Sid]
    sortby[Sid]
  ReverseAdjacentSectionsUpH5 feed project[Sid]
    sortby[Sid]
  mergediff
  ReverseAdjacentSectionsUpH4 feed {s1}
  hashjoin[Sid, Sid_s1]
  filter[.Sid = .Sid_s1]
  projectextend[;Sid: .Sid_s1,
    RevAdjSectU: .RevAdjSectUp_s1]
consume;

let ReverseAdjacentSectionsUp =
  (ReverseAdjacentSectionsUpH5 feed)
  (ReverseAdjacentSectionsUpH6 feed)
  concat
  projectextend[Sid; RevAdjSectUp: .RevAdjSectU]
consume;

let ReverseAdjacentSectionsDownH5 =
   ReverseAdjacentSectionsDownH4 feed
   NotLongerAdjacentSections feed {s1}
   hashjoin[Sid, AdjSect_s1]
   filter[.Sid = .AdjSect_s1]
   projectextend[Sid; RevAdjSectD: .RevAdjSectDown - .Sids_s1]
consume;

let ReverseAdjacentSectionsDownH6 =
  ReverseAdjacentSectionsDownH4 feed project[Sid]
    sortby[Sid]
  ReverseAdjacentSectionsDownH5 feed project[Sid]
    sortby[Sid]
  mergediff
  ReverseAdjacentSectionsDownH4 feed {s1}
  hashjoin[Sid, Sid_s1]
  filter[.Sid = .Sid_s1]
  projectextend[;Sid: .Sid_s1,
    RevAdjSectD: .RevAdjSectDown_s1]
consume;

let ReverseAdjacentSectionsDown =
  (ReverseAdjacentSectionsDownH5 feed)
  (ReverseAdjacentSectionsDownH6 feed)
  concat
  projectextend[Sid; RevAdjSectDown: .RevAdjSectD]
consume;

let AllAdjacentSectionLists =
  AdjacentSectionsUp feed {u}
  AdjacentSectionsDown feed {d}
  hashjoin[Sid_u, Sid_d]
  filter [.Sid_u = .Sid_d]
  projectextend [;Sid: .Sid_u,
                 AdjacentSectUp: .AdjSectUp_u,
                 AdjacentSectDown: .AdjSectDown_d] {a}
  ReverseAdjacentSectionsUp feed {ru}
  ReverseAdjacentSectionsDown feed {rd}
  hashjoin[Sid_ru, Sid_rd]
  filter [.Sid_ru = .Sid_rd]
  projectextend [;Sid: .Sid_ru,
                 ReverseAdjacentSectUp: .RevAdjSectUp_ru,
                 ReverseAdjacentSectDown: .RevAdjSectDown_rd] {r}
  hashjoin[Sid_a, Sid_r]
  filter[.Sid_a = .Sid_r]
  projectextend[;Sid: .Sid_a,
                 AdjSectUp: .AdjacentSectUp_a,
                 AdjSectDown: .AdjacentSectDown_a,
                 RevAdjSectUp: .ReverseAdjacentSectUp_r,
                 RevAdjSectDown: .ReverseAdjacentSectDown_r]
  sortby [Sid]
consume;

# Sections relation for jnet creation

let InSections =
  RoadPartSectionsTmp1 feed
  project[Sid, SectCurve, StartJid, EndJid, Side, VMax, Lenth]
  SectionRouteIntervals feed {r1}
  hashjoin[Sid, Sid_r1]
  filter[.Sid = .Sid_r1]
```

```
  projectextend[Sid, SectCurve, StartJid, EndJid, Side, VMax,
    Lenth; RouteInter: .IntervalList_r1]
  AllAdjacentSectionLists feed {l}
  hashjoin[Sid, Sid_l]
  filter[.Sid = .Sid_l]
  projectextend[Sid, SectCurve, StartJid, EndJid, Side, VMax, Lenth,
    RouteInter; AdjSectUp: .AdjSectUp_l,
                AdjSectDown: .AdjSectDown_l,
                RevAdjSectUp: .RevAdjSectUp_l,
                RevAdjSectDown: .RevAdjSectDown_l]
  sortby[Sid]
  rdup
consume;

# build network

query createjnet("MHTTestJNetwork", 0.000001, InJunctions, InSections, InRoutes);

# script finished close database

close database;

quit;
```

5.4 Match GPS-Tracks to Networks Generated from Open Street Map Data

The `MapMatchingAlgebra` algebra module provides operators creating single moving network positions from data collected by GPS-devices related to networks imported from open street map as described in Section 5.3 using Multiple Hypothesis Technique [15].

The operator **mapmatchmht** was implemented by one of our students as part of his final thesis [13]. It enables us to create *mgpoint* from GPS data files. The operator **mapmatchmht** supports the following signatures:
network × *mpoint* [× *real*] → *mgpoint*
network × *ftext* [× *real*] → *mgpoint*
network × *stream*(*inputtuple*) [× *real*] → *mgpoint*

The optional *real* parameter can be used to overwrite the tolerance value stored in the *network* object for the current map matching operation. The *ftext* identifies the file with the GPS-data. The *inputtuple* in the third signature consists of a *tuple* with the attributes: *Lat*: *real*, *Lon*: *real*, *Time*: *instant*, *Fix*: *int*, *Sat*: *int*, *Hdop*: *real*, *Vdop*: *real*, *Pdop*: *real*, *Course*: *real*, *Speed*: *real*. Each *inputtuple* describes a line of the data set of an GPS track.

Assumed we have a *network* object created by the script `NetworkFromFullOSMImport.SEC` and an GPS-Track stored in the file *gps.data* we can create an *mgpoint* from this data sources in the same database the network object is allocated by using the SECONDO command:

```
let testMGP1 = mapmatchmht(netobj, 'gps.data');
```

Analogous we can convert an existing *mpoint* to his network representation:

```
let testMGP2 = mapmatchmht(netobj, mpointdata);
```

Or use a stream of tuples with GPS data created by the operator **gpximport** as data source by typing:

```
let testMGP3 = mapmatchmht(netobj, gpximport('gpx.data'));
```

In all cases we could use a *real* value as third parameter to overwrite the tolerance factor given in the network object, for example:

```
let testMGP1Tol = mapmatchmht(netobj, 'gps.data', 0.0001);
```

The operator **jmapmatchmht** works almost analogous to **mapmatchmht** for the second network implementation. The supported signatures are:
jnet × *mpoint* → *mjpoint*
jnet × *ftext* → *mjpoint*
jnet × *stream*(*inputtuple*)→ *mjpoint*

Assumed we have a *jnet* object created by the script `NetworkFromFullOSMImport.SEC` and a GPS-Track stored in the file *gps.data* we can create an *mjpoint* from this data sources in the same database the jnetwork object is allocated by using the SECONDO command:

```
let testMJP1 = jmapmatchmht(jnetobj, 'gps.data');
```

Analogous we can convert an existing *mpointdata* object of data type *mpoint* to his jnetwork representation:

```
let testMJP2 = jmapmatchmht(jnetobj, mpointdata);
```

Or use a stream of tuples with GPS data created by the operator *gpximport* as data source by typing:

```
let testMJP3 = jmapmatchmht(jnetobj, gpximport('gpx.data'));
```

Chapter 6

Traffic Estimation

6.1 Introduction

In the context of the first network implementation it was planned to support traffic estimation for historic moving information in the network data model[1]. Therefore, we implemented in the `TemporalNetAlgebra` an additional data type (see Section 6.2) and some operations (see Section 6.3.1) collecting and transforming the information of the individual movement into an intermediate data format which can be used for traffic estimation. Further we implemented a additional SECONDO algebra module `TrafficAlgebra` providing operations for traffic and traffic flow analysis on the data types of the `TemporalNetAlgebra` (see Section 6.3.2).

In the last section of this chapter we present some examples showing the usage of the presented operations.

6.2 Data Type for Traffic Information Estimation

The data type *mgpsecunit* (see Table 6.2) was introduced to support traffic estimation and indexing of *mgpoint* values. The data type *mgpsecunit* reduces the complex information given in a *mgpoint* value to the values which are useful for traffic estimation.

secId	*int*	identifier of a network section
partNo	*int*	part number on this section[2].
direct	*int*	moving direction of the source *mgpoint* value within this section part
avgSpeed	*real*	average speed of the source *mgpoint* value within this section (part)
time	*periods*	time interval the the source *mgpoint* value moved within this section (part)

Table 6.1: Attributes of Traffic Data Type

6.3 Operations for Traffic Estimation

The first set of operators presented in Section 6.3.1 transforms the historical movement information of a set of *mgpoint* into a stream of *mgpsecunit* values (see Section 6.3.1). This stream respectively the information provided by this stream can be used by a set of operators for the analysis of the traffic flow or estimation of section (parts) affected by heavy traffic, like described in Section 6.3.2

6.3.1 Compress Data for Traffic Estimation

The operations of Table 6.2 transform the input values into streams of corresponding *mgpsecunits* values. The algorithm is almost the same for all three operations. The information of the units of the incoming *mgpoint*

[1] In fact the missing *side* value within the implementation of the data type *route interval* prevents us from the complete implementation of this idea. Without *side* values within *route intervals* we are not able to estimate if the heavy traffic is on the up or the down side of the Highway.

[2] For traffic estimation it is useful to divide long sections into smaller parts. For example a section belonging to a motorway may have a total length of 16 km, but there is only a traffic jam of 2 km inside this section. To solve this problem we have the possibility to split the network sections longer than a given length value into parts of this user defined maximum length. The splitting starts at the smaller point of the section and the first part has the number 1. The length of the last part might be shorter than the given maximum length value.

values are extracted and merged to a set of *mgpsecunita* for each *mgpoint* value. Merging means that as long as the *mgpoint* moves on the same section (part) in the same direction the information provided by the different units is used to write a single *mgpsecunit* value with defining the section part passed by the *mgpoint* and the average speed of the *mgpoint* at this section (part). At last the result is returned as *stream* of *mgpsecunits*. The time complexity is $O(m)$ for each incoming *mgpoint*. For a set of x *mgpoint* values we get a total time complexity of $O(\sum_{i=0}^{x} m_x)$.

Operator	Signature
mgpsecunits	*rel*(*tuple*((*a*$_1$ *x*$_1$)(*a*$_2$ *x*$_2$)...(*a*$_2$ *x*$_2$))) ×*a*$_i$× *network* × *real* → *stream*(*mgpsecunit*)
mgpsecunits2	*mgpoint* × *real* → *stream*(*mgpsecunit*)
mgpsecunits3	*stream*(*mgpoint*) × *real* → *stream*(*mgpsecunit*)

Table 6.2: Operators Merging Moving Information for Traffic Estimation

6.3.2 Traffic Estimation

The operation **mgpsu2tuple** converts a stream of *mgpsecunit* values into a stream of tuples with the attributes of the data type *mgpsecunit*[3]. The time complexity is given by the number of stream elements. The transformation into a relation enables the user to resort the tuples by attribute values using the standard **sortby** operation of SECONDO. The sorted stream of tuples can be used as input for the traffic estimation operations in Table 6.3.

Operator	Signature
trafficflow	*rel*(*tuple*(*mgpsecunit*)) → *rel*(*tuple*(*int*, *int*, *int*, *mint*))
trafficflow2	*stream*(*mgpsecunit*) → *rel*(*tuple*(*int*, *int*, *int*, *mint*))
traffic	*stream*(*mgpsecunit*) → *rel*(*tuple*(*int*, *int*, *int*, *mreal*, *mint*))
traffic2	*stream*(*mgpoint*) → *rel*(*tuple*(*int*, *int*, *int*, *mreal*, *mint*))
heavytraffic	*rel*(*tuple*(*int*, *int*, *int*, *mreal*, *mint*)) × *real* × *int* → *rel*(*tuple*(*int*, *int*, *int*, *mreal*, *mint*))

Table 6.3: Traffic Estimating Operators

The operators **trafficflow** and **trafficflow2** compute the number of cars in the defined section part and direction as *mint* value, whereas the operators **traffic** and **traffic2** additionally return the average speed of the cars as *mreal* value.

The operator **heavytraffic** shrinks the traffic relation produced by **traffic** and **traffic2** to the times and places where the average speed is slower than the query parameter *real* or the number of cars is higher than the query parameter *int*.

6.4 Examples for Traffic Estimation

We can manipulate the data generation script of the BerlinMOD Benchmark to generate data for traffic estimation. Different from the BerlinMOD Benchmark approach we need many cars on a single day. Therefore, we manipulate the parameters for *SCALEFCARS* and *SCALEFDAYS* in lines 143 and 144 of the script. The number of generated cars will be $2000 * SCALEFCARS$ and the number of observation days $28 * SCALEFDAYS$.

The generated data can be translated into network representation by:

```
#open database

open database berlinmod;

# Build a network from street data.

let B_ROUTES =
  streets feed
  projectextendstream[; geoData: .geoData polylines[TRUE]]
  addcounter[id,1]
  projectextend [id; lengt : size(.geoData),
                     geometry: fromline(.geoData),
                     dual: TRUE,
                     startSmaller: TRUE]
```

[3] **mgpsu2tuple**: *stream*(*mgpsecunit*) → *stream*(*tuple*(sid *int*, part *int*, direction *int*, speed *real*, starttime *instant*, endtime *instant*, leftclosed *bool*, rightclose *bool*))

```
consume;

let B_JUNCTIONS =
  B_ROUTES feed {r1}
  B_ROUTES feed {r2}
  symmjoin [(.id_r1 < ..id_r2) and (.geometry_r1 intersects ..geometry_r2)]
  projectextendstream[id_r1, geometry_r1, id_r2,
    geometry_r2; CROSSING_POINT: components(crossings(.geometry_r1,
                                                      .geometry_r2))]
  projectextend[; r1id: .id_r1,
                  r1meas: atpoint(.geometry_r1, .CROSSING_POINT, TRUE),
                  r2id: .id_r2,
                  r2meas: atpoint(.geometry_r2, .CROSSING_POINT, TRUE),
                  cc: 65535]
consume;

let B_NETWORK = thenetwork(1, 1.0, B_ROUTES, B_JUNCTIONS);

# Translate Moving Objects in Network Representation

let dataSNcar =
  dataScar feed
  projectextend[Licence, Model, Type; Trip: mpoint2mgpoint(B_NETWORK, .Trip)]
consume;
```

On this database we can perform traffic estimating queries like:

```
let q1 =
  dataSNcar mgp2mgpsecunits[Trip,B_NETWORK,1000.0]
trafficflow2;

let q2 =
  dataSNcar mgp2mgpsecunits[Trip,B_NETWORK,1000.0]
  transformstream
  project[Elem]
  sortby[Elem asc]
trafficflow;

let q3 =
  dataSNcar mgp2mgpsecunits[Trip,B_NETWORK,1000.0]
  namedtransformstream[MGPSec]
  project[MGPSec]
  sortby[MGPSec asc]
trafficflow;

let q4 =
  dataSNcar feed
  projecttransformstream[Trip]
  mgp2mgpsecunits3[1000.0]
trafficflow2;

let q5 =
  dataSNcar feed
  projecttransformstream[Trip]
  mgp2mgpsecunits3[1000.0]
  transformstream project[Elem]
  sortby[Elem asc]
trafficflow;

let q6 =
  dataSNcar feed
  projecttransformstream[Trip]
  mgp2mgpsecunits3[1000.0]
  namedtransformstream[MGPSec]
  project[MGPSec]
  sortby[MGPSec asc]
trafficflow;

let q7 =
  dataSNcar mgp2mgpsecunits[Trip,B_NETWORK,1000.0]
traffic;

let q8 =
  dataSNcar feed
  projecttransformstream[Trip]
  mgp2mgpsecunits3[1000.0]
traffic;

let q9 =
  dataSNcar feed
  projecttransformstream[Trip]
traffic2[1000.0];

let ht1 = q9 heavytraffic [8.777,2];

let ht2 = q9 heavytraffic [2.333,2];
```

Bibliography

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

[2] S. Dieker and R. 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.

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

[4] 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.

[5] M. Erwig, R. 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.

[6] L. Forlizzi, R. 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.

[7] Open Street Map Foundation. Open Street Map Web Site. `http://www.openstreetmap.org`, last visited 2013, January.

[8] R. 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.

[9] R. Güting, M. Böhlen, M. Erwig, C. Jensen, N. Lorentzos, M. Schneider, and M. Vazirgiannis. A Foundation for Representing and Querying Moving Objects. *ACM Trans. Database Syst.*, 25(1):1–42, 2000.

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

[11] Fernuniversität Hagen. Secondo Web Site. `http://dna.fernuni-hagen.de/secondo/index.html`, last visited 2013, February.

[12] N.J. Nilsson P.E. Hart and B.Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on System Science and Cybernetics*, 4(2):100–107, 1968.

[13] M. Roth. Map matching von gps-gestützten positionsdaten im erweiterbaren datenbanksystem SECONDO. Bachelorarbeit, Fernuniversität in Hagen, 2012.

[14] M. Scheppokat. Datenbanken für Bewegliche Objekte in Netzen Prototypische Implementierung und experimentelle Auswertung. Diplomarbeit, Fernuniversität in Hagen, 2007.

[15] N. Schuessler and K. Axhausen. Map-matching of gps traces on high-resolution navigation networks using the multiple hypothesis technique (mht).

[16] 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.

[17] 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.