**REGULAR PAPER**

**Ralf Hartmut Güting · Victor Teixeira de Almeida ·
Zhiming Ding**

# Modeling and querying moving objects in networks

**Abstract** Moving objects databases have become an important research issue in recent years. For modeling and querying moving objects, there exists a comprehensive framework of abstract data types to describe objects moving freely in the 2D plane, providing data types such as *moving point* or *moving region*. However, in many applications people or vehicles move along transportation networks. It makes a lot of sense to model the network explicitly and to describe movements relative to the network rather than unconstrained space, because then it is much easier to formulate in queries relationships between moving objects and the network. Moreover, such models can be better supported in indexing and query processing. In this paper, we extend the ADT approach by modeling networks explicitly and providing data types for static and moving network positions and regions. In a highway network, example entities corresponding to these data types are motels, construction areas, cars, and traffic jams. The network model is not too simplistic; it allows one to distinguish simple roads and divided highways and to describe the possible traversals of junctions precisely. The new types and operations are integrated seamlessly into the ADT framework to achieve a relatively simple, consistent and powerful overall model and query language for constrained and unconstrained movement.

**Keywords** Moving object · Network · Spatio-temporal · Data type · ADT

## 1 Introduction

The field *of moving objects databases* has received a lot of research interest in recent years. This technology allows one to model in a database the movements of entities and to ask queries about such movements. In some cases only time-dependent locations need to be managed, leading to the

R. H. Güting (✉) · V. T. de Almeida · Z. Ding
LG Datenbanksysteme für neue Anwendungen, FernUniversität
Hagen, D-58084 Hagen, Germany
E-mail: rhg@fernuni-hagen.de

*moving point* abstraction, in other cases also the time-dependent shape or extent is of interest and we speak of *moving regions*. Examples of moving points are cars, aircraft, ships, mobile phone users, terrorists, or polar bears. Examples of moving regions are hurricanes, oil spills, forest fires, armies, tribes of people in history, or the spread of vegetation or of an illness.

Some of the interest is spurred by current trends in consumer electronics. Wireless networking enabled and position-aware (i.e. GPS equipped) devices such as PDAs, on-board units in vehicles, or even mobile phones have become relatively cheap and are predicted to be in widespread use in the near future. This will lead to many new kinds of applications such as location-based services. At the same time a huge volume of movement information (sometimes called trajectories) will become available and need to be managed and analyzed in database systems.

The field of moving objects databases came into being in the late nineties by two parallel developments which also had different perspectives of the field. These can be characterized as the *location management* and the *spatio-temporal database* perspective. First, the group around Wolfson in a series of papers [1–4] developed a model that allows one to keep track in a database of a set of time-dependent locations, e.g. of vehicles. A fundamental observation was that one should store in the database not the positions directly, leading to a very high volume of updates, but rather a motion vector. Only when the object's position as predicted by the motion vector deviates from the real position by more than a threshold, an update needs to be transmitted and executed on the database. They developed an interesting concept of *dynamic attributes* within a data model called MOST which are attributes of normal data types which change implicitly over time; hence the results of queries will change implicitly as time proceeds, leading to a notion of *continuous queries*. The related query language, FTL (future temporal logic), allows one to specify temporal relationships between objects in queries. This approach is restricted to moving point objects and it treats current and expected near future movement.

Second, in the European project CHOROCHRONOS [5] the spatio-temporal database perspective was explored, which means one tries to manage in a database time-dependent geometries. Earlier work in that area had generally admitted only discrete changes. This restriction was dropped and continuously moving points, lines, or regions were considered. A data model capturing complete histories of movement with a related query language was developed by Güting and colleagues [6–9]. They provided an algebra with data types such as *moving point, moving line*, and *moving region* together with a comprehensive set of operations. Such an algebra can be embedded into any DBMS data model and be implemented as a DBMS extension such as a data blade, cartridge, etc.

A second approach to data modeling was pursued in CHOROCHRONOS by Grumbach and colleagues [10, 11] who developed constraint-based models for handling geometries in general which includes spatio-temporal applications [12], since these can be viewed as existing in a 3D (2D + time) space. They also implemented a prototype system, called DEDALE. Some researchers outside CHOROCHRONOS have also contributed to constraint-based modeling of moving objects [13, 14]. Later Refs. [15, 16] have continued this approach to modeling focusing especially on moving points.

Since then, the field has flourished and a lot of work has been done especially on implementation issues, for example, on developing index structures [17–19], processing continuous queries of various types [20, 21], studying similarity of trajectories [22], developing test data generators [23], to name only some of the areas.

An important observation that has not been addressed in the research mentioned above is that in many cases objects do not move freely in the 2D space but rather within spatially embedded networks (roads, highways, even airlines have fixed routes). It would make a lot of sense to include spatial networks into a data model and query language for moving objects. One could then describe movement relative to the network rather than 2D space which would enable easier formulation of queries, more efficient representations and indexing of moving objects and so forth. This is discussed in more detail later.

It is interesting to observe that recent research has started to address the implementation-oriented aspects and developed specialized index structures for objects moving in networks [24, 25], or specialized query processing algorithms [26, 27]. There has also been work studying query processing for spatial networks, restricted to static objects [28]. A nice generator for test data, creating network-based moving objects, is available [29]. However, there is a big gap at the level of data modeling and querying. Only very few papers have touched modeling issues for moving objects in networks; these are discussed in Sect. 7. A comprehensive data model and query language for objects moving in networks does not yet exist. As long as this is so, it is not clear how the proposals for efficient indexing and query processing can be integrated and used in a DBMS. The purpose of this paper is to close that gap.

Hence the goal of this paper is to provide a comprehensive data model and query language for moving objects in networks, supporting the description and querying of complete histories of movement.[1] The design should fulfill the following requirements:

1. The model should contain an explicit concept of a network embedded in space, such as a transportation network. This is in contrast to modeling the network by standard facilities of a DBMS data model, e.g. an object-relational model. Fulfilling this requirement leads to easier and more powerful formulation of queries and to more efficient execution, as the system is able to create appropriate data structures to represent the network.

2. Positions of moving objects should be described relative to the network rather than the embedding space. In this way, the descriptions of moving objects become much more compact, since no geometric information needs to be stored. Geometry is stored once and for all with the network. Besides, discovering relationships between moving objects and parts of the network becomes much simpler and more efficient. For example, to find out whether an object is moving along some road one can check whether an identifier for the road occurs in the description of the movement. Otherwise a complicated and expensive geometric check would have to be used which might not even succeed due to numeric inaccuracies.

3. The information kept about the network must be extensible using the standard facilities of the DBMS data model rather than fixed. In a relational environment we should be able to create relations to add information relative to a network. For example, one could add or remove relations describing motels or speed limits along a highway network.

4. The model should allow us to describe static or moving objects relative to the network, such as static positions (motel, gas station, etc.), static regions (speed limit, construction area, etc.), moving positions (vehicles), and moving regions (traffic jam, part of network affected by snow storm, etc.).

5. Conceptually, there are now three different kinds of information present in the model, that is, the *network*, static or moving network positions or regions (*network space*), and spatial objects, e.g. the area of a suburb, a natural park, a river, or a fog area (*space*). The model should allow us to handle interactions or relationships between any two of them. For example:

   – *network–network space*: On which highway is this car? What is the next exit after a motel? Which part of the network can be reached within 50 km distance from a given network position? Find a shortest path between two network positions.
   – *network space–space*: At what times is the vehicle within the fog area? How many cars have left suburb

---

[1] Note that a "movement history" is not necessarily restricted to the past; it may include the current state and expected future behaviour.

*X* between 9 and 10 am? Are there any speed limits within natural parks?

– *network–space*: Which highways pass rivers? Return the part of the network that lies within forest *X*.

6. There exists a sophisticated framework to represent and query objects moving freely in space, developed in [7]. The framework offers many concepts and facilities that are also needed here. Hence the new model should be consistent with the one of [7] and be integrated as seamlessly as possible with it.

7. The modeling of networks should not be too simplistic. In a simple graph model we have just nodes and edges. However, paths over graphs are often the conceptual entities of interest. For example, in a road network nodes would be junctions of roads, edges correspond to pieces of roads between junctions, and roads are paths over this graph. It is fairly obvious that city maps are organized primarily by roads (e.g. they are the named entities), not by their junctions or edges. Hence such paths should play a prominent role also in the model. Besides, the model should accommodate the fact that sometimes we have divided roads (e.g. highways) and a position on one side of the road may be far away from the corresponding position on the other side of the road.

Our proposed model will fulfill all these requirements. We proceed as follows:

- We provide a formal definition of a spatially embedded network and of network locations and network regions. The model offers paths over the network graph as a basic concept, called *route*, it allows one to distinguish between simple and dual routes ("divided" roads), and to describe the possible transitions for moving objects, e.g. vehicles, at junctions.

- Networks are made available through a data type *network*.[2] An interface to a relational environment is offered that allows one to create a *network* value from relations of a certain form, and to export network information into relations.

- Two data types *gpoint* and *gline* are defined. A value of type *gpoint* is a position in a given network. A value of type *gline* is a region within a given network. These data types are integrated into the type system of [7] in such a way that the *moving* type constructor is applicable to them, so we also have the time-dependent types *moving*(*gpoint*) and *moving*(*gline*) These four data types form the core of our approach.

- Static operations are defined to treat the interaction of *gpoint* and *gline* values with (i) the network, and (ii) spatial data types. These new operations are included into the scope of lifting so that they are also available for *moving*(*gpoint*) and *moving*(*gline*). Some special operations for networks are defined, e.g. **shortest_path** and **trip**.

_____

[2] We write data types in italics underlined, and operations in bold face.

The paper is organized as follows. Section 2 develops a formal model of a transportation network and of network positions and regions. Section 3 formally defines the data types and their relational embedding, Sect. 4 presents the operations, and Sect. 5 shows some query examples. Section 6 addresses implementation concepts. Section 7 discusses related work, and finally Sect. 8 concludes the paper.

## 2 Modeling networks

A first idea, which may seem obvious at least to a computer scientist, is to model a transportation network as a directed graph $G = (V, E)$, with set of nodes $V$ and set of edges $E \subseteq V \times V$. The set of possible positions within the network could then be defined as

$$Pos(G) = V \cup (E \times (0, 1))$$

That is, an object is either in a node or on an edge; a value from the open interval $(0, 1)$ indicates a relative position on the edge. This view of a network is used for example, in [24, 29].

However, there are some indications that this model is not the best one. In the literature, Jensen et al. [30] emphasize that real world networks are quite complex and that a realistic modeling needs in fact multiple representations. One of the important views is the kilometer-post representation which describes positions in the network relative to distance markers on the roads. This is closely related to the concept of *linear referencing* widely used in the GIS-T literature (geographic information systems in transportation), see for example [31], where again positions are described relative to the length of a road. Linear referencing is also already available in commercial database products such as Oracle Spatial [32].

We will define networks in terms of routes (corresponding to roads or highways in real life and to paths over a graph) and positions on routes for the following reasons:

- Routes are the relevant conceptual entities in real life. We have names for roads, not for crossings or pieces of road between crossings. Also addresses are given relative to roads. The model should reflect this. It is then easy to relate network positions to these conceptual entities.

- Linear referencing is an important and widely used concept in GIS-T, which also indicates that positions should be described relative to routes rather than edges of a graph.

- The perhaps most practical reason is that the representation of a moving object becomes much smaller in this way. If positions are given relative to edges, then for example a car going along a highway at constant speed needs a change of description at every exit/junction because the edge identifier changes. If positions are given relative to routes, then the description needs to change only when the car changes the highway.

Hence a network will be defined as a set of routes and a set of junctions between these routes.

We wish to accomodate in the model the fact that routes can be bi-directional, i.e., admit movement in two directions, and that it may be necessary to distinguish positions on the two sides of a route, e.g. on a highway. On the other hand, there are also applications where one does not want to distinguish between positions on two sides of a road. One example are people moving around in a pedestrian zone.

We will offer two kinds of routes called *simple* and *dual* routes. There are also two concepts for positions on roads called *route measure* and *route location*. The *route measure* is independent from the kind of route (simple or dual), it is just a distance from the origin of the route. Junctions between two routes are positioned at two distinct route measures. The *route location* depends on the route type. For a simple route it is the same as the route measure; for a dual route it is a route measure plus a flag from the set {*up*, *down*}. Similarly, on a simple route, a *route interval* is given by two measures; on a dual route by two measures plus an *up-down* flag. The precise meaning of *up* and *down* is defined below.

Let the domain of routes be defined as

$$Route = \{(id, l, c, kind, start) | id \in \underline{int},$$
$$l \in \underline{real}, c \in \underline{line}, kind \in \{simple, dual\},$$
$$start \in \{smaller, larger\}\}$$

Hence a *route* description consists of an identifier *id*, a length *l*, a curve *c* describing its geometry in the plane, a route type *kind*, and a flag *start* indicating how route locations are to be embedded into space (explained below). The geometry *c* is a simple, non self-intersecting curve in the plane which may be open or closed (a cycle). It is represented by a polyline. In spatial databases and also in our framework a data type *line* is available that can represent such values [7].

Let *R* be a finite set of distinct routes (i.e. identifiers are pairwise distinct). A *route measure in R* consists of an identifier and a real number giving a position on that route.

$$RMeas(R) = \{(rid, d) | rid \in \underline{int}, d \in \underline{real},$$
$$\exists (rid, l, c, k, s) \in R \text{ such that}$$
$$\begin{cases} 0 \leq d \leq l & \text{if } c \text{ is not a closed curve} \\ 0 \leq d < l & \text{if } c \text{ is a closed curve} \end{cases}$$

On a closed curve, $d = 0$ and $d = l$ denote the same position. Since we are interested in a unique representation, we require $d < l$ for a closed curve.

There exists an obvious order on route measures for a given route, namely $(rid, d) < (rid, d') :\Leftrightarrow d < d'$.

A *junction in R* is a triple consisting of two route measures in *R* with distinct route identifiers and a *connectivity code*, an integer value encoding which movements through the junction are possible.

$$Junction(R) = \{(rm_1, rm_2, cc) | rm_1, rm_2 \in RMeas(R),$$
$$rm_1 = (r_1, d_1), rm_2 = (r_2, d_2), r_1 \neq r_2, cc \in \underline{int}\}$$

Representing the connectivity at junctions is essential for computing shortest paths which in turn are the basis for a concept of *directed distance* in the network. Connectivity codes work as follows. For each route we distiguish the *up* direction of movement where route measures are increasing, and the *down* direction where route measures are decreasing with movement, respectively. At a junction between routes *A* and *B* various transitions may be possible or not for a moving object, for example a transition $A_{up} \rightarrow B_{up}$. This is illustrated in Fig. 1b.

In most cases junctions are built to allow for all 8 possible transitions from one route to the other. However, this is not always possible. Figure 1a shows an example of a physical highway junction where in fact only the transitions $A_{up} \rightarrow B_{up}$, $A_{up} \rightarrow B_{down}$, and $A_{down} \rightarrow B_{down}$ are possible. We can represent the possible transitions in a $4 \times 4$ matrix as shown in Fig. 1c. The 1's in the diagonal represent the fact that it is possible to stay on a route in the same direction. A transition such as for example $A_{up} \rightarrow A_{down}$ would be set to 1 if a U-turn were possible. In general, for the definition of the transition matrix, *A* and *B* are chosen such that the route identifier of *A* is smaller than that of *B*.
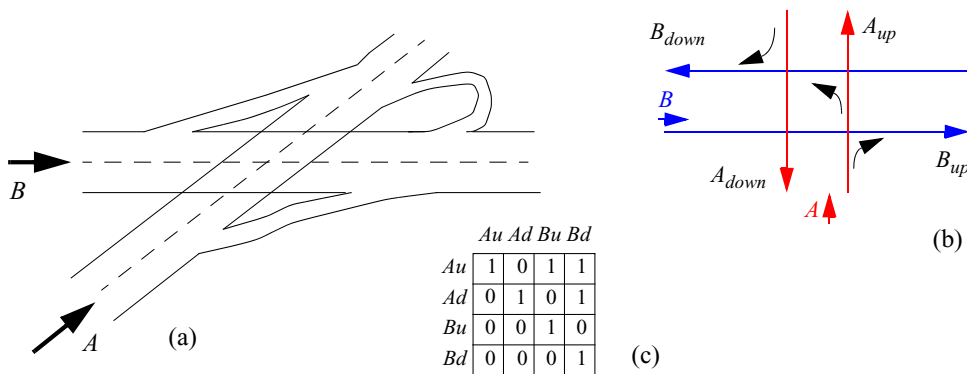


|    | Au | Ad | Bu | Bd |
|----|----|----|----|----|
| Au | 1  | 0  | 1  | 1  |
| Ad | 0  | 1  | 0  | 1  |
| Bu | 0  | 0  | 1  | 0  |
| Bd | 0  | 0  | 0  | 1  |

**Fig. 1** **a** A physical highway junction, **b** its diagramatical representation, and **c** the transition matrix
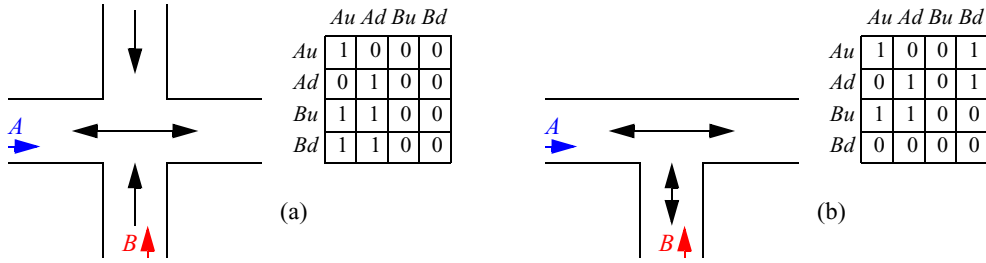
**Fig. 2** Transition matrices for **a** a junction involving two one-way parts of route B, **b** a T-junction

As a transition matrix has 16 bits, it can easily be represented in a (16 or more bit) integer value, and this is what the *connectivity code* means.

This concept is quite general. For example, it allows the representation of junctions involving one-way roads or of T-junctions, see Fig. 2. Here the arrows in the roads indicate the allowed direction of driving whereas arrows with route identifiers indicate up-directions of routes. Note that in the model oneway (parts of) roads are expressed by transition matrices.

If there is a junction between more than two routes in one location, we represent it as one junction for each pair of routes involved (e.g. $AB$, $BC$, and $AC$ for three routes $A$, $B$, and $C$).

We provide a predicate to evaluate the connectivity code. Let $r_1$ and $r_2$ be two routes participating in a junction with connectivity code $cc$, and let $s_1$ and $s_2$ be values from the set $Side = \{up, down, none\}$. Then the predicate

$$connects((r_1, s_1), (r_2, s_2), cc)$$

holds if a transition from route $r_1$, side $s_1$ to route $r_2$, side $s_2$ is possible according to the matrix stored in $cc$. The side value *none* is associated with simple roads.

A *network* is a pair $N = (R, J)$ where $R$ is a finite set of distinct routes and $J$ is a finite set of junctions in $R$. Let *Network* denote the set of all such pairs.

A *route location* in $R$ is either a route measure for a simple route, or a route measure augmented by a *side* value for a dual route. To have a uniform format we always add a *side* value which is *none* for simple routes.

$$RLoc(R) = \{(rid, d, side)|(rid, d)$$
$$\in RMeas(R), side \in Side,$$
$$\text{for } (rid, l, c, kind, start) \in R :$$
$$kind = simple \Leftrightarrow side = none\}$$

We then also speak of network locations. For a network $N = (R, J)$, the set *of network locations* is

$$Loc(N) = RLoc(R)$$

Equality on network locations is defined as follows. Let $(r_1, d_1, s_1)$ and $(r_2, d_2, s_2)$ be network locations in $N = (R, J)$.

$$(r_1, d_1, s_1) = (r_2, d_2, s_2) :\Leftrightarrow$$
$$(r_1 = r_2 \wedge d_1 = d_2 \wedge s_1 = s_2) \vee$$
$$(\exists((r_1, d_1), (r_2, d_2), cc) \in J \wedge connects((r_1, s_1),$$
$$(r_2, s_2), cc) \wedge connects((r_2, s_2), (r_1, s_1), cc))$$

Hence in addition to being equal in the usual sense, two network positions can be equal if they are both on a junction and it is possible to get from each of them to the other at this junction. This definition is the basis for operations such as forming the intersection of two network regions.

Using the underlying curve, a route location or a route measure can be mapped to a point in the 2D plane. Let $r = (rid, I, c, k, s)$ be a route with $l = length(c)$ and $rl = (rid, d, side)$ be a location on this route. The position of $rl$ denoted $pos(rl)$ is defined as shown in Fig. 3.

In Fig. 3, the definition of a "smaller" end point assumes the usual $xy$-lexicographic order of points in the 2D plane and the term *origin* refers to the smallest point on a closed curve (cycle) with respect to that order. Hence the flag $s$ allows one to define an "orientation" of the curve.

The position of a route measure $rm = (rid, d)$ is defined in the same way.

Not all networks according to this definition make sense, i.e., can exist as real networks in the 2D space. A network is called *consistent*, if

- the length of a route corresponds to the geometrical length of its curve, and
- the geometric position of a junction corresponds to the intersection of the two involved curves.

Observe that we do not require that each intersection of the curves is reflected in a junction, hence bridges, tunnels, etc. can be represented.

$$pos(rl) = \begin{cases} \text{the point on curve } c \text{ at distance } d & \text{if } c \text{ is not a closed curve} \\ \quad \text{from the smaller end point} & \text{and } s = smaller \\ \text{the point on curve } c \text{ at distance } d & \text{if } c \text{ is not a closed curve} \\ \quad \text{from the larger end point} & \text{and } s = larger \\ \text{the point on curve } c \text{ at distance } d & \text{if } c \text{ is a closed curve} \\ \quad \text{in clockwise direction from the origin} & \text{and } s = smaller \\ \text{the point on curve } c \text{ at distance } d & \text{if } c \text{ is a closed curve} \\ \quad \text{in counter-clockwise direction from the origin} & \text{and } s = larger \end{cases}$$
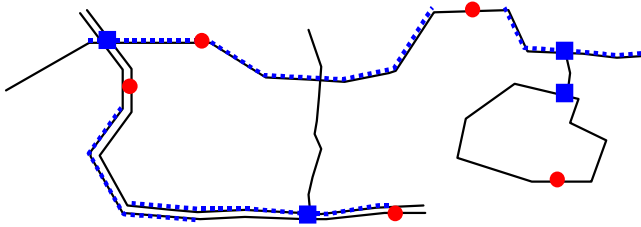
**Fig. 3** Definition of *pos*

**Fig. 4** A simple network example

Formally, we have: A network $N = (R, J)$ is called *consistent*, if

- for every route $(rid, l, c, k, s) \in R$, $l = length(c)$;
- for every junction $(rm, rm', cc)$, $pos(rm) = pos(rm')$

In the sequel we assume that all networks we deal with are consistent.

A *route interval* in the network $N$ is basically a pair of route locations on the same route. It can be represented as a quadruple $(rid, d_1, d_2, side)$ where $(rid, d_1, side)$ and $(rid, d_2, side)$ are route locations and $d_1 \leq d_2$. It is allowed that the interval degenerates into a single location. Semantically, a route interval $ri = (rid, d_1, d_2, side)$ comprises the set of all route locations $(rid, d, side)$ with $d_1 \leq d \leq d_2$; this set is denoted as $locs(ri)$.

Two route intervals $ri_1$, $ri_2$ are *quasi-disjoint* iff they are either on different routes, or they are on the same route and their sets of route locations are disjoint, i.e., $locs(ri_1) \cap locs(ri_2) = \emptyset$. Note that two route intervals on different routes covering the same junction are quasi-disjoint.

Let $N$ be a network. A *region* of $N$ is a finite set of quasi-disjoint route intervals in $N$. The set of all possible regions of network $N$ is denoted as $Reg(N)$.

The concepts of networks and their locations and regions are illustrated in Fig. 4. Dual routes are drawn doubled, junctions are represented by squares and network locations by filled circles. A network region, consisting of four route intervals is represented by dashed curves. One should note in this figure how locations and intervals on dual routes are associated with sides of the route.

Discussion of applications and limitations
of the network model

In this subsection, we discuss how the model can be used to represent various features of real road networks. First, one needs to keep in mind that any model is an abstraction and simplification of reality, designed to fit a particular purpose. For example, one abstraction is that roads are modeled as linear rather than areal features. Another is that on a highway we do not model different lanes, so we cannot describe how a car switches lanes.

The basic properties of our model are that there exist routes with an associated linear geometry and junctions corresponding to intersection points of such geometries. Furthermore for a route one can define whether it is simple or dual. The concept of routes is also intended to correspond

as much as possible to real life entities such as roads or highways. However, whether a highway or road can be modeled directly as a single route depends on various factors one of which is scale. In the following we discuss modelling options in the form of various questions coming up.

*How can we model roundabouts*? It is a matter of scale whether a roundabout is modeled explicitly. Basically it is a junction of several roads or highways. At a large scale, roads can be represented as routes and the roundabout as a collection of junctions, one for each pair of routes. The geometry of the roundabout is not represented (a point). At a small scale the roundabout may be modeled as a cyclic route and the roads crossing it have to be represented by several routes. In this case the geometry of the roundabout is represented in the model. Similar considerations hold for highway junctions which at a large scale are represented as points geometrically, and at a small scale by collections of routes in which case the geometry is visible.

*How do we describe distinct highways sharing the same piece of road?* Overlapping highways need to be decomposed into distinct routes, see Fig. 5.

*How do we model a road having different stretches of one-way or two-way traffic?* It is no problem to represent this in a single route, as such stretches are defined by transition matrices, see Fig. 2a.

*How do we model a road that is divided only in certain stretches?* Since a route can have only one status of single or dual, the road needs to be decomposed into several routes.

*The way a junction can be traversed may depend on the type of moving entity. For example, for trucks, buses, or pedestrians there may be different rules than for cars. Furthermore, rules for traversing a junction may also change during the day.* This cannot be expressed by the current model. However, an extension to define junctions with a set of transition matrices instead of a single one appears straightforward. In this case, distance and shortest path operations defined in the following sections would need an additional parameter identifying the transition matrix to be used. Similarly, an extension for time dependent transition matrices appears possible.

## 3 Data types and their relational embedding

In this section, we provide data types *network*, *gpoint*, and *gline* to represent the network, a position within the network, and a region within the network, respectively, based on the concepts defined in Sect. 2. We also provide an interface to exchange information between values of such types and a relational environment. The interface consists of a set of operations that allow one, for example, to create
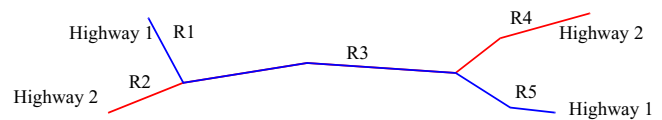


**Fig. 5** Decomposing overlapping highways into distinct routes

relations representing parts of the network, or to make available components of *gpoint* values as values of standard data types.

## 3.1 Data types

Defining a data type means to introduce a name for it and to define the set of possible values, i.e., the *domain* or the *carrier set*. We use the algebraic terminology and define carrier sets; for a type *t* its carrier set is denoted as $D_t$. For the type *network*, the carrier set is:

$$D_{network} = Network$$

The data types *gpoint* and *gline* obviously depend on existing networks. Let $N = \{N_1, \ldots, N_k\}$ be the set of networks present in the database. We define types *gpoint* and *gline* with carrier sets:

$$D_{gpoint} = \{(i, gp) | 1 \le i \le k \wedge gp \in (Loc(N_i) \cup \{\perp\})\}$$
$$D_{gline} = \{(i, gl) | 1 \le i \le k \wedge gl \in Reg(N_i)\}$$

So a value of such a type consists of a network number together with a position or region within that network. The network position may be undefined, which is represented by $\perp$ A network region is a set (of route intervals) which may be empty anyway.

## 3.2 Interface to relations and standard data types

We first provide a set of operations that allow us to convert between values of the abstract data types *network*, *gpoint*, and *gline* on the one hand, and relations and values of standard types on the other hand.

### 3.2.1 Relational views of a network

We can get routes, junctions (nodes) and sections (edges) of a network by the operations:

*network* $\rightarrow$ *rel* **routes**, **junctions**, **sections**

They return the information contained in a network according to the definition of Sect. 2 in relations with the respective schemas:

```
(route: int, length: real, curve: line,
  dual: bool, startsSmaller: bool)
(route1: int, meas1: real, route2: int,
  meas2: real, cc: int, pos: point)
(route: int, meas1: real, meas2: real,
  dual: bool, curve: line)
```

The last two components in the definition of *a route* are represented by booleans here. For a junction, the spatial position of the intersection of the two underlying curves is provided for convenience in the *pos* attribute. Also, for convenient querying each junction is returned twice so that each participating route occurs once in each of the attributes

*route1* and *route2*. A section is a junction-free piece of a route between two junctions; here the geometry is provided in the *curve* attribute.

Observe that the names of roads are not part of a network definition; like other information they can be attached externally in relations. For example, let us assume a *network* value called *Hagen* has been created for the city network of Hagen, and road names are stored in a relation

```
road(name: string, route: int,
  roadLevel: int)
```

The value of the *route* attribute corresponds to the route identifier in the network. The *roadLevel* is an example of further information; here it distinguishes major and minor roads.

*Examples*  We can connect the routes of the Hagen network with the street names by the join

```
SELECT *
FROM routes(Hagen) AS h, road AS r
  where h.route = r.route
```

We get the positions of all junctions on Bahnhofstrasse ordered by distance from the start by

```
SELECT j.meas1
FROM junctions(Hagen) as j, road as r
WHERE r.name = 'Bahnhofstrasse'
  AND j.route1 = r.route
ORDER BY j.meas1
```

"Find all sections of Bahnhofstrasse longer than 500 meters!" (assuming that distances are given in kilometers)

```
SELECT *
FROM sections(Hagen) AS s, road AS r
WHERE r.name = 'Bahnhofstrasse'
  AND s.route = r.route AND
  (s.meas2 - s.meas1) > 0.5
```

### 3.2.2 Constructing a network from relations

To construct a network, we need to supply two relations containing the relevant information for routes and junctions. The first relation must have a schema compatible to the one returned by the **routes** operation, i.e., with attribute types (int, real, line, bool, bool). The second must describe junctions [3] and have a schema like that returned by the **junctions** operation except for the last *point* attribute, that is (int, real, int, real, int). The operation for constructing a network is:

*rel* $\times$ *rel* $\rightarrow$ *network* **network**

Suppose we have relations

---

[3]  Here a junction may be represented by either one or two tuples.

```
HagenRoads (name: string, road: int,
  length: real, geometry: line,
  dual: bool, startsSmaller: bool,
  roadLevel: int)
HagenJunctions(road1: int, pos1: real,
  road2: int, pos2: real,
  junctionType: int)
```

The command

```
LET Hagen = network(
  SELECT road, length, geometry, dual,
    startsSmaller
  FROM HagenRoads, HagenJunctions)
```

will create the Hagen network as used in the examples above. If one desires to create a network containing just the major roads (say, of categories 1 and 2), this can be done by

```
LET HagenMajor = network(
  SELECT road, length, geometry, dual,
    startsSmaller
  FROM HagenRoads
  WHERE roadLevel <= 2,
  SELECT road1, pos1, road2, pos2,
    junctionType
  FROM HagenJunctions, HagenRoads
    AS h1, HagenRoads AS h2
  WHERE road1 = h1.road AND
    h1.roadLevel <= 2 AND
    road2 = h2.road AND
    h2.roadLevel <= 2)
```

Executing such a command creates an internal data structure for the network that enables efficient access and traversal.

### 3.2.3 Accessing *gpoint* and *gline* Values

A *gpoint* contains a position on a given route, possibly on one of its sides. We can access these components. *Side* values are represented by integers: $up = 1, down = -1, none = 0$.

| | | |
|---|---|---|
| *gpoint* | $\rightarrow$ *int* | **route** |
| *gpoint* | $\rightarrow$ *real* | **pos** |
| *gpoint* | $\rightarrow$ *int* | **side** |

A *gline* value describes a region of the network. Any computation to derive parts of the network, for example, to get the part of the network within a fog area, or for computing a shortest path, is done via *gline* values, whereas a *network* value, after creation, is static. Therefore the interface to export *gline* values into relations is quite important. Similarly as for networks, we can create relations by operations

*gline* $\rightarrow$ *rel* **routes**, **junctions**, **sections**

The operation **routes** returns a relation with the following schema:

```
(route: int, meas1: real, meas2: real,
  curve: line)
```

This relation will have one tuple for each route interval in the *gline* value; the curve is the piece of the route corresponding to that route interval. The operation **junctions** returns a relation with schema

```
(route1: int, meas1: real, route2: int,
  meas2: real, pos: point)
```

Operation **sections** returns a relation with schema:

```
(route: int, meas1: real, meas2: real,
  sectionMeas1: real,
  sectionMeas2: real, curve: line)
```

In this case each tuple represents a subinterval of a section with the associated piece of curve. The section itself is defined by the distances *sectionMeas1*, *sectionMeas2*. Hence it holds *sectionMeas1* $\leq$ *meas1* $\leq$ *meas2* $\leq$ *sectionMeas2*.

Finally, there is a special form of the **routes** operator called **path_routes** with signature

*gline* $\times$ *gpoint* $\rightarrow$ *rel* **path_routes**

which is suitable to transform *gline* values resulting from shortest path computations into relations. In this case, the *gline* value represents an ordered sequence of pieces of routes leading from a start point to a destination. It is returned as a relation with schema

```
(route: int, meas1: real, meas2: real,
  curve: line, no: int)
```

The second argument gives the start point; in the result relation pieces of routes are numbered in increasing distance from the start point within the *no* attribute.

### 3.2.4 Constructing *gpoint* and *gline* values

We offer the following operations:

*network* $\times$ *int*[*routeId*] $\times$ *real* $\times$ *int* $\rightarrow$ *gpoint* **gpoint**

The second argument is the route identifier,[4] the third the route measure, and the fourth the side value. The network argument is needed as otherwise it would not be clear which of the networks in the database is meant.

*network* $\times$ *int*[*routeId*] $\rightarrow$ *gline* **gline**
*network* $\times$ *int*[*routeId*]
$\times$ *real* $\times$ *real* $\times$ *int* $\rightarrow$ *gline* **gline**

These operations allows us to get *gline* values for a whole route or a route interval. Furthermore, we can construct a *gline* value corresponding to a set of routes and set of route intervals by supplying a relation for the remaining arguments after the first one. Hence for relations with schemas

```
(int)
(int, real, real, int)
```

we can apply the operation

*network* $\times$ *rel* $\rightarrow$ *gline* **gline**

We can also get the whole network as a *gline* value:

*network* $\rightarrow$ *gline* **gline**

---

[4] It would be nice to have a special data type *routeId* for route identifiers as this would make signatures more readable. On the other hand, this would make the type system more complex. As a compromise, we use the notation shown to indicate that semantically the second argument is a route identifier rather than an integer.

## Examples

The part of the Hagen network for the road "Bahnhof-strasse."

```
gline(Hagen,
  SELECT route
  FROM road
  WHERE name = 'Bahnhofstrasse')
```

The last kilometer of "Bahnhofstrasse." (The length operation is defined below).

```
gline(Hagen,
  SELECT route, length(Hagen, route) - 1,
    length(Hagen, route), 0
  FROM road WHERE name ='Bahnhofstrasse')
```

### 3.2.5 Accessing route information

Given a route identifier, we can access the information belonging to that route:

$network \times int[routeId] \rightarrow real$ **length**
$network \times int[routeId] \rightarrow line$ **curve**
$network \times int[routeId] \rightarrow bool$ **dual**
$network \times int[routeId] \rightarrow bool$ **startsSmaller**

### 3.2.6 Comparing *gpoint* and *gline* with a route identifier

Here we have operations:

$gpoint \times int[routeId] \rightarrow bool$ **inside**
$gline \times int[routeId] \rightarrow bool$ **intersects**
$gline \times int[routeId] \rightarrow bool$ **inside**
$int[routeId] \times gline \rightarrow bool$ **inside**

A *gpoint* can belong to a route. A *gline* value can overlap a route (**intersects**), be contained in it, or contain the route. Here we don't need to provide the network as an explicit argument as it is contained in the *gpoint* or *gline* value.

## 4 Operations

In this section we provide the bulk of operations on *gpoint* and *gline* as well as on the derived types *moving(gpoint)* and *moving(gline)* in a way that is consistent with the framework of [7].

### 4.1 Integrating *gpoint* and *gline* into the type system

The framework of [7] provides a collection of data types, or a *type system,* specified as a signature as shown in Table 1. This specification technique is taken from [33].

A signature in general consists of sorts and operations; in this signature the sorts are *kinds* and represent sets of data types, and the operations are *type constructors*. Type constructors may take arguments or not, in the latter case they are *constant types*. The available types of the type system

are the terms generated by this signature. For example, the kind *BASE* contains the types *int*, *real*, *string*, and *bool*; the *range* type constructor is applicable to those types as well as to the type *instant* in kind *TIME*; therefore the types in the kind *RANGE* are *range(int)*, ..., *range(instant)*.

The meanings of the spatial types are as follows: A *points* value is a finite set of points in the plane; a *line* value is basically a finite set of curves, and a *region* value a finite set of disjoint *faces* (closed subsets of the plane) which may have holes. Note that a position in a network corresponds spatially to a *point* value, and a part of a network to a *line* value; this is why the types are called *gpoint* and *gline*.

The *range* type constructor represents finite sets of disjoint intervals over the base types and *instant*: *range(int)*, ..., *range(instant)*. The type *range(instant)* representing sets of time intervals is also called *periods*.

The *moving* constructor, applied to a type $\alpha$, yields a type *moving($\alpha$)* whose values are partial functions from time into values of type $\alpha$, e.g. *moving(int)*, *moving(real)*, and *moving(region)*. The *intime* type constructor is very simple: for an argument type $\alpha$ a value of *intime($\alpha$)* is a pair consisting of an *instant* and an $\alpha$ value.

The type system is now extended by the types *gpoint* and *gline* as shown in Table 1. This means that the new types *gpoint* and *gline* and also the types *moving(gpoint)*, *moving(gline)*, *intime(gpoint)*, and *intime(gline)* are available.

The reader may wonder how such values are represented. The model in [7] is an *abstract model* as discussed in [6, 7] which means that it defines the domains or carrier sets of its data types in general in terms of infinite sets. To be able to implement an abstract model, one must provide a corresponding *discrete model*, that is, define finite representations for all the data types of the abstract model. For [7] this has been done in [8]. For all the *moving* types, the so-called *sliced representation* is proposed which represents a time dependent value as a sequence of *slices* such that within each slice the development of the value can be represented by some "simple" function. Figure 6 illustrates this for a *moving(real)* and a *moving(points)* value. The simple functions within a slice are for a *moving(real)* quadratic polynomials or square roots of such, and for *moving(points)* just linear movement of each component point. For more details and a justification see [8]. It is not difficult to extend this for *moving(gpoint)* values to a sliced representation providing linear functions for the time-dependent location, and for *moving(gline)* providing a linear

**Table 1** The type system of [7] (without bold parts) and the extended type system

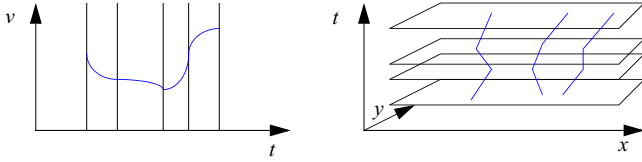|  | | |
|---|---|---|
| | → *BASE* | *int*, *real*, *string*, *bool* |
| | → *SPATIAL* | *point*, *points*, *line*, *region* |
| | → **GRAPH** | **gpoint**, **gline** |
| | → *TIME* | *instant* |
| *BASE* ∪ *SPATIAL* ∪ **GRAPH** | → *TEMPORAL* | *moving*, *intime* |
| *BASE* ∪ *TIME* | → *RANGE* | *range* |

**Fig. 6** Sliced representation of a *moving*(*real*) and a *moving*(*points*) value

function for each route interval boundary. This is shown in Sect. 6.

### 4.2 Defining operations

For defining operations, [7] proceeds in three steps:

1. A comprehensive set of operations on non-temporal types is defined.
2. By a process called lifting, all operations in that set are made uniformly available to temporal types as well. This ensures consistency between operations on non-temporal and temporal types.
3. Specific operations are added to deal with temporal types.

Most of the operations are defined in a generic way and range over as many types as possible. To define generic operations, [7] introduces a concept of so-called *spaces* and distinguishes data types representing single values and sets of values in a given space. These are called *point types* and *point set types,* respectively. For example, there is a space *Integer* having a point type *int.* representing a single value in that space, and a point set type *range*(*int*). representing sets of integers. Another space is *2D* having a point type *point*, representing a single value from the 2D plane, and three point set types *points*, *line*, and *region*, representing finite or infinite sets of points in the 2D plane. The classification of types into spaces is shown in Table 2.

Generic operations are defined in [7] based on these notions using type variables $\pi$ and $\sigma$, ranging over the point and point set types in the respective space. For example, the predicates **inside** and **intersects** are defined as shown in Table 3. Hence, the operation **inside** is defined e.g. for the type combinations $int \times range(int) \rightarrow bool$ or $point \times region \rightarrow bool.$

**Table 2** Classification of Non-Temporal Types

| | | | Point type | Point set types |
|---|---|---|---|---|
| ID Spaces | Discrete | Integer | *int* | *range*(*int*) |
| | | Boolean | *bool* | *range*(*bool*) |
| | | String | *string* | *range*(*string*) |
| | Continuous | Real | *real* | *range*(*real*) |
| | | Time | *instant* | *periods* |
| 2D Space | | 2D | *point* | *points*, *line*, *region* |
| | | Graph | *gpoint* | *gline* |

**Table 3** Definition of generic operations

| | | |
|---|---|---|
| $\pi \times \sigma$ | $\rightarrow bool$ | **inside** |
| $\sigma_1 \times \sigma_2$ | $\rightarrow bool$ | **intersects** |

A second, more simple technique for defining generic operations uses type variables as arguments to type constructors. For example, the operation

$$moving(\alpha) \quad \rightarrow periods \quad \textbf{deftime}$$

is defined for all types to which the *moving* constructor is applicable and returns the set of time intervals when the function is defined.

In Table 2, we have already added a new space *Graph* with the point type *gpoint* and the point set type *gline*. By integrating the types *gpoint* and *gline* into the type system of [7], a large number of generic operations are already defined for them. For example, **inside** and **deftime** are defined with signatures:

| | | |
|---|---|---|
| $gpoint \times gline$ | $\rightarrow bool$ | **inside** |
| $moving(gpoint\}$ | $\rightarrow periods$ | **deftime** |
| $moving(gline)$ | $\rightarrow periods$ | **deftime** |

### 4.3 Operations on non-temporal types

In this section we first check which of the non-temporal operations from [7] apply to the new data types *gpoint* and *gline*. In the second subsection we consider the interaction between network space and space, that is between types *gpoint* and *gline* on the one hand, and *point*, *points*, *line*, and *region* on the other hand. All operations defined in this section will later be lifted to temporal types.

#### 4.3.1 Generic operations

The generic operations applicable to types *gpoint* and *gline* are collected in Table 4. In some cases the semantics has to be slightly adapted. Some brief comments:

For the predicates, the meaning should be obvious. For the set operations, note that operations on single values are included, as this is later interesting for the lifted case. For example, the **intersection** between two *gpoint* values is the *gpoint* if they are equal, or undefined otherwise. This is not very interesting. However, in the lifted version we have, for example, the signature $moving(gpoint) \times gpoint \rightarrow moving(gpoint)$ which allows us to find the part of the *moving*(*gpoint*) when it was at a particular network position, a quite useful operation.

The elements of *gline* values are always closed intervals. This means that if we subtract a *gpoint* from a *gline*. the result is in any case the original *gline* value, as closure is applied after the operation (similar as for spatial values in [7]).

**Table 4** Non-temporal operations (implied by [7]); subject to lifting

| | | | |
|---|---|---|---|
| Predicates, unary | *gpoint* | $\rightarrow$ *bool* | **isempty[undefined]** |
| | *gline* | $\rightarrow$ *bool* | **isempty[undefined]** |
| Predicates, binary | *gpoint* × *gpoint* | $\rightarrow$ *bool* | =, ≠ |
| | *gpoint* × *gline* | $\rightarrow$ *bool* | **inside** |
| | *gline* × *gline* | $\rightarrow$ *bool* | **inside**, **intersects** |
| Set operations, point/point | *gpoint* × *gpoint* | $\rightarrow$ *gpoint* | **intersection, minus** |
| Set operations, point/point set | *gpoint* × *gline* | $\rightarrow$ *gpoint* | **intersection** |
| | *gpoint* × *gline* | $\rightarrow$ *gline* | **union** |
| | *gpoint* × *gline* | $\rightarrow$ *gpoint* | **minus** |
| | *gline* × *gpoint* | $\rightarrow$ *gline* | **minus** |
| Set operations, point set/point set | *gline* × *gline* | $\rightarrow$ *gline* | **union, intersection, minus** |
| Aggregation | *gline* | $\rightarrow$ *gpoint* | **min, max** |
| | *gline* | $\rightarrow$ *gpoint* | **avg[center]** |
| | *gline* | $\rightarrow$ *gpoint* | **single** |
| Numeric | *gline* | $\rightarrow$ *int* | **no_components** |
| | *gline* | $\rightarrow$ *real* | **size[length]** |
| Distance and direction | *gpoint* × *gpoint* | $\rightarrow$ *real* | **distance** |
| | *gpoint* × *gline* | $\rightarrow$ *real* | **distance** |
| | *gline* × *gline* | $\rightarrow$ *real* | **distance** |
| | *gpoint* × *gpoint* | $\rightarrow$ *real* | **direction** |

The **min**, and **max** operations are defined in [7] only for ID spaces as they need a total order. The **avg** operation (with alias name **center**) is defined in 2D, but it does not make sense for a general *gline* value as the resulting point according to the semantics defined there does not need to lie on the network. However, **center**, **min** and **max** would be quite useful to get the center, start or end of a traffic jam, respectively, on a single route. We define these three operations to yield the "natural" result if the *gline* value is restricted to a single route, and otherwise to be undefined (i.e., return ⊥).

The **single** operation, as defined in [7], returns a (proper) *gpoint* value if the argument *gline* does in fact consist only of a single network location, otherwise ⊥.

Components of a *gline* value are the connected components in the usual sense. This is used in the **no_components** and later the **decompose** operation.

The **distance** is now the directed distance along the shortest path through the network. It is possible to also get the Euclidean distance by first converting to spatial values, explained below. The **direction** operation is defined in [7] to return the angle (in degrees) between the *x*-axis and a line from the first to the second point. It has the same meaning here (for the spatial positions of the *gpoint* values).

### 4.3.2 Interaction between network space and space

We now consider operations like forming the intersection between a *gline* value and a *region* value, for example, to find a part of the network lying within a fog area. Obviously, such operations do not yet exist in [7]. First of all, it should be possible to convert between network and spatial values.

We provide operations:

| | | |
|---|---|---|
| *gpoint* | $\rightarrow$ *point* | **in_space** |
| *gline* | $\rightarrow$ *line* | **in_space** |
| *network* × *point* | $\rightarrow$ *gpoint* | **in_network** |
| *network* × *points* | $\rightarrow$ *gline* | **in_network** |
| *network* × *line* | $\rightarrow$ *gline* | **in_network** |
| *network* × *region* | $\rightarrow$ *gline* | **in_network** |

The first direction of conversion, by **in_space**, always works. In the other direction, a *point* is converted to a *gpoint* if it lies on the network; otherwise ⊥ is returned. For *points*, *line*, and *region*, the intersection of their underlying point sets in the plane with (the underlying point set of the *line* value of) the network is formed and returned as a *gline* value.

Second, the type *gpoint* is defined to be a subtype of *point*, and *gline* a subtype of *line*. This induces a set of signatures including, for example, the following:

| | | |
|---|---|---|
| *gpoint* × *region* | $\rightarrow$ *bool* | **inside** |
| *gline* × *line* | $\rightarrow$ *bool* | **intersects** |
| *gpoint* × *region* | $\rightarrow$ *real* | **distance** |

The semantics is the one obtained by substituting for the *gpoint* or *gline* argument its associated *point* or *line* value, respectively. This is for convenience mainly as we could achieve the same effect by explicitly converting first, i.e., using **in_space**(*x*) instead of argument *x*. In addition, it allows for a more efficient implementation.

Third, we would like to be able to get the results of set operations in network space rather than in space. By the subtyping rule just introduced, the result type of **intersection**(*gl*, *r*) of a *gline* value *gl* and a *region* value *r* is *line*. We

**Table 5** Operations from Sect. 3.2 subject to lifting

| Accessing a *gpoint* | *gpoint* | $\rightarrow$ *int* | **route** |
|---|---|---|---|
| | *gpoint* | $\rightarrow$ *real* | **pos** |
| | *gpoint* | $\rightarrow$ *int* | **side** |
| Comparing *gpoint* and *gline* with a route identifier | *gpoint* $\times$ *int* [*routeId*] | $\rightarrow$ *bool* | **inside** |
| | *gline* $\times$ *int* [*routeId*] | $\rightarrow$ *bool* | **intersects** |
| | *gline* $\times$ *int* [*routeId*] | $\rightarrow$ *bool* | **inside** |
| | *int* [*routeId*] $\times$ *gline* | $\rightarrow$ *bool* | **inside** |

**Table 6** Temporal Operations (implied by [7])

| Projection to domain and range | For $\alpha$ in {*gpoint*, *gline*} | | |
|---|---|---|---|
| | *moving*($\alpha$) | $\rightarrow$ *periods* | **deftime** |
| | *moving*(*gpoint*) | $\rightarrow$ *gline* | **trajectory** |
| | *moving*(*gline*) | $\rightarrow$ *gline* | **traversed** |
| | *intime*($\alpha$) | $\rightarrow$ *instant* | **inst** |
| | *intime*($\alpha$) | $\rightarrow$ $\alpha$ | **val** |
| Interaction with Domain and Range | For $\alpha$ in {*gpoint*, *gline*} | | |
| | *moving*($\alpha$) $\times$ *instant* | $\rightarrow$ *intime*($\alpha$) | **atinstant** |
| | *moving*($\alpha$) $\times$ *periods* | $\rightarrow$ *moving*($\alpha$) | **atperiods** |
| | *moving*($\alpha$) | $\rightarrow$ *intime*($\alpha$) | **initial**, **final** |
| | *moving*($\alpha$) $\times$ *instant* | $\rightarrow$ *bool* | **present** |
| | *moving*($\alpha$) $\times$ *periods* | $\rightarrow$ *bool* | |
| | For $\alpha, \beta$ in {*gpoint*, *gline*} | | |
| | *moving*($\alpha$) $\times$ $\beta$ | $\rightarrow$ *moving*(*min*($\alpha, \beta$)) | **at** |
| | *moving*($\alpha$) $\times$ $\beta$ | $\rightarrow$ *bool* | **passes** |
| When operation | For $\alpha$ in {*gpoint*, *gline*} | | |
| | *moving*($\alpha$) $\times$ ($\alpha \rightarrow$ *bool*) | $\rightarrow$ *moving*($\alpha$) | **when** |
| Lifting | Lifting | | |
| Rate of change | *moving*(*gpoint*) | $\rightarrow$ *moving*(*real*) | **speed**, **mdirection** |

introduce variants **g_intersection**, **g_union**, and **g_minus** to get the result in network space. Hence the result type of **g_intersection**(*gl*, *r*) is *gline*. Again, we might explicitly convert using **in_network**(*n*, **intersection**(*gl*, *r*)) instead of **g_intersection**(*gl*, *r*), but this is less convenient to use and much less efficient to implement.

### 4.4 Operations on temporal types

#### 4.4.1 Lifting

All operations defined in Sect. 4.3 are now subject to *temporal lifting*. Lifting means that if we have an operation with signature $\alpha_1 \times \alpha_2 \times \cdots \times \alpha_n \rightarrow \beta$, then the operation is also available for signatures

$$\alpha_1' \times \alpha_2' \times \cdots \times \alpha_n' \rightarrow \underline{moving}(\beta)$$

where $\alpha_i' \in \{\alpha_i, \underline{moving}(\alpha_i)\}$. Hence, each of the argument types may become time-dependent which makes the result type time-dependent as well. In addition to the operations of Sect. 4.3 we include some operations from Sect. 3.2 into the scope of lifting, namely the operations shown in Table 5.

Some example signatures that we get by lifting are the following:

$$\begin{aligned} moving(gpoint) & \\ \times\, gline & \rightarrow moving(bool) \quad \textbf{inside} \\ moving(gline) & \rightarrow moving(real) \quad \textbf{size[length]} \\ moving(gpoint) & \\ \times\, gpoint & \rightarrow moving(real) \quad \textbf{distance} \\ moving(gpoint) & \rightarrow moving(point) \quad \textbf{in\_space} \end{aligned}$$

#### 4.4.2 Generic operations

Values of temporal types are functions from time into some domain, and there is a set of generic operations defined in [7] to deal with such functions. In Table 6 it is shown how these operations apply to the new data types *moving*(*gpoint*) and *moving*(*gline*). The semantics of these operations is defined in [7].[5] We briefly recall the meaning or adapt it to the network environment: **Deftime** yields the time intervals when the function is defined. **Trajectory** and **traversed** project a moving *gpoint* or *gline* into the network space, i.e., yield the traversed part of the network. **Inst** and **val** just return the

---

[5] The **mdirection** operation was added later in [9].

two components of an *intime* value. A function can be restricted to an instant or to a set of time intervals by **atinstant** and **atperiods**, respectively. **Initial** and **final** yield the first and last (*instant*, $\alpha$) pair of the function. The **present** predicate allows one to check whether the function is defined at an instant or at some time during a given set of time intervals. **At** restricts a function to the times when its value lies within the second argument. In this case the result type is the minimum in an assumed order *gpoint* < *gline*. For example, if we restrict a *moving*(*gline*) to a *gpoint* value, the result is a *moving*(*gpoint*). The **passes** predicate, analogous to **present**, allows one to check whether the function ever assumes one of the values of the second argument. Some of these operations can be derived, i.e., expressed in terms of others as discussed in [7].

The **when** operation allows one to restrict a function to the times when its function value fulfills the predicate given as a second argument. For example, we might restrict a vehicle $v$ (a *moving*(*gpoint*)) to the times when it was inside a park *Park* by

```
v when[fun (g: gpoint) g inside Park]
```

In this paper we introduce an additional abbreviation for the parameter function: The type of the argument is clear anyway, and we allow to refer to the argument by a "." symbol. Hence this can be written as

```
v when[. inside Park]
```

One might think that it is impossible to implement the **when** operation, since the first argument is a continuous function. But it is in fact implementable by a rewriting technique as long as all operators used within the predicate are within the scope of lifting [7].

The meaning of **speed** is clear; **mdirection** returns the time-dependent direction of movement (a number as described above for **direction**).

## 4.5 Operations on sets of objects

There is only one such operation in the design of [7] which, however, plays an important role. The **decompose** operation returns for a multi-component value its components as separate values. For example, for a set of intervals (e.g. *periods*) each interval is a component; for a *region* a face, for a *line* or *gline* value a connected component of the underlying graph. Furthermore, for functions, i.e. values of type *moving*($\alpha$) components are defined to be *maximal continuous components*. Hence such a value is split at discontinuities. In this paper it is necessary to define continuity for *moving*(*gpoint*) and *moving*(*gline*) values. This is done in Appendix A.

## 4.6 Network-specific operations

In the previous subsections of Sect. 4 we have studied how the new data types *gpoint* and *gline* can be integrated into the type system and the framework of operations of [7]. In this section we consider some further operations specific for networks. Admittedly the design in this section is a bit less systematic and complete than that of the previous sections.

We would like to be able to get the part of the network within a given distance from a network position, to determine the connected component containing a given position, to compute shortest paths, and to compute trips. This is provided by the operations shown.

| | | |
|---|---|---|
| *gpoint* × *real* | → *gline* | **circle**, **out_circle**, **in_circle** |
| *gpoint* | → *gline* | **component** |
| *gpoint* × *gpoint* | → *gline* | **shortest_path** |
| *gpoint* × *gpoint* × *instant* | → *moving*(*gpoint*) | **trip** |

**Circle** returns the part of the network within distance $v$ of $u$ ($u$ being the first and $v$ the second argument) using an undirected distance measure along routes and ignoring connectivity in junctions. Directed versions are the two other operations; **out_circle** returns only the part reachable within distance $v$ when starting from $u$; **in_circle** returns the part of the network from which $u$ can be reached within distance $v$. **Component** returns the connected component of the network containing the argument position $u$. **Shortest_path** returns the shortest path from $u$ to $v$.

The operation **trip** allows one to construct a time-dependent (traversal of a) shortest parth, as follows. Suppose we have a relation with schema of type (gline, mreal), for example

```
speeds (route_interval: gline, speed:
  mreal)
```

which associates with each route interval the time dependent speed one can drive here. This information can be fed into the internal data structures representing the network by a command

```
update(Highways, speeds)
```

After that, **trip** computes a description of a trip in the form of a *moving*(*point*) that moves from $u$ to $v$ starting at instant $w$. In the computation it is assumed that the vehicle traverses each route interval at the admissible speed of the respective time. Algorithms related to such an operator are discussed in [34].

## 4.7 Auxiliary operations

To complete our query language, we introduce a few auxiliary operations, mainly for dealing with time (Table 7). The first group of operations constructs time intervals, for example, **year**(2003), or **day**(2003, 8, 11). The bounding time interval for two given intervals can be formed by the **period** operator; e.g. **period**(**month**(2003, 7), **month**(2003, 12)) returns the second half of the year 2003. Operations **start** and **end** are defined as aliases for **min** and **max** on

**Table 7** Auxiliary operations

| | | | |
|---|---|---|---|
| Constructing time intervals | $int$ | $\rightarrow periods$ | **year** |
| | $int \times int$ | $\rightarrow periods$ | **month** |
| | $int \times int \times int$ | $\rightarrow periods$ | **day** |
| | $int \times int \times int \times int$ | $\rightarrow periods$ | **hour** |
| | $int \times int \times int \times int \times int$ | $\rightarrow periods$ | **minute** |
| | $int \times int \times int \times int \times int \times int$ | $\rightarrow periods$ | **second** |
| Bounding time interval | $periods \times periods$ | $\rightarrow periods$ | **period** |
| Aliases | $periods$ | $\rightarrow instant$ | **min[start]**, **max[end]** |
| Current movement | $moving(\alpha)$ | $\rightarrow \alpha$ | **current** |
| | | $\rightarrow instant$ | **now** |
| Accessing instants | $instant$ | $\rightarrow int$ | **year**, **month**, **day**, **hour**, **minute**, **second** |
| Manipulating instants | $instant \times real$ | $\rightarrow instant$ | **+**, **−** |
| Generic intervals | $\alpha \times \alpha$ | $\rightarrow range(\alpha)$ | **range** |
| | $range(\alpha)$ | $\rightarrow range(\alpha)$ | **open**, **closed**, **leftclosed**, **rightclosed** |
| | | $\rightarrow int$ | **minint**, **maxint** |
| | | $\rightarrow real$ | **minreal**, **maxreal** |
| | | $\rightarrow instant$ | **mininstant**, **maxinstant** |
| Geocoding | $string \times string \times int$ | $\rightarrow point$ | **address2point** |

*period* values. **Now** returns the current instant of time and **current** the value of a moving object at the current instant. Given an *instant*, one can extract its year, month, etc. by the respective operations. One can add and subtract a real number that is the duration of a time interval, to an *instant* to obtain the corresponding earlier or later instant. The next group has operations for constructing generic intervals, e.g. **open**(**range**(0, **maxreal**)) constructs the interval of positive real numbers. Finally, **address2point** is an interface to an external geocoding service which for a given street address returns a *point* value. The first two arguments are the name of a city and a street, and the third is a house number. Using the **in_network** operation we can map the resulting *point* into a *gpoint*.

# 5 Example applications and queries

In this section and Appendix B, we perform an "experimental evaluation" of our design by trying to formulate a large number of queries that had been written down in natural language before. Queries were written just with the application in mind without consideration how they could be expressed in the new query language. One of the queries in fact, we were not able to formulate. We consider three different applications, an express parcel delivery service, vehicles in a highway network, and an application analyzing traffic jams.

Some of these queries are easy to formulate, others are complex. Showing query formulations for all queries gets a bit too long. We might select some queries from each of the three applications, but such a selection might be biased. To provide an unbiased impression of how easy or difficult it is to formulate the queries, in the sequel we show the complete set of queries for the first application. Query formulations for the other two applications are given in Appendix B.

## 5.1 Express package delivery application

This application models a company offering express delivery of packages as they exist in many big cities. We consider analysis of past movements as well as questions coming up in real time. The people moving around are called "postmen" even if they are female. For concreteness, this company is assumed to deliver within the city network of Hagen, Germany.

We assume that we have the road network of the city of Hagen called *Hagen*, a relation *road* giving road names, a relation *postman* describing post workers' trips, and a relation *city_area* describing regions of Hagen as shown as follows:

```
road(name: string, route: int)
postman(name: string, trip: mgpoint)
city_area(name: string, reg: region)
```

### 5.1.1 Queries on historical information

**Query P1**: Where was postman Bob at 10 am of last Saturday? There are two possible interpretations:

(i) return a network location (i.e., a *gpoint*)

```
LET Saturday10am = start
  (hour(2003,8,9,10));
SELECT val(atinstant(trip,
  Saturday10am))
FROM postman
WHERE name = 'Bob'
```

(ii) Return a road name.

```
SELECT r.name
FROM postman, road AS r
WHERE name = 'Bob' AND
  route(val(atinstant(trip,
  Saturday10am))) = r.route
```

**Query P2**: Which places did Bob visit between 9:00 am and 11:00 am of last Saturday?

(i) as part of the network (result is a *gline* value)

```
LET Saturday9to11 = period(hour(2003,
  8,9,9), hour(2003, 8, 9, 10));
SELECT trajectory(atperiods(trip,
  Saturday9to11))
FROM postman
WHERE name = 'Bob'
```

(ii) as a set of road names

```
LET Bob_roads = routes(
  ELEMENT(
    SELECT trajectory(atperiods(trip,
      Saturday9to11))
    FROM postman
    WHERE name = 'Bob'));
SELECT name
FROM Bob_roads AS b, road AS r
WHERE b.route = r.route
```

The **routes** operation used here transforms the *gline* value returned by **trajectory** into a relation.

**Query P3**: Find all post workers who visited Rathausstrasse in the afternoon (2:00 pm through 6:00 pm) of last Friday.

```
LET FridayAfternoon =
  period(hour(2003, 8, 8, 14),
  hour(2003, 8, 8, 17));
LET Rathausstrasse = ELEMENT(
  SELECT route FROM road WHERE name =
    'Rathausstrasse');
SELECT name
FROM postman
WHERE Rathausstrasse inside
  trajectory(atperiods(trip,
    FridayAfternoon))
```

The **inside** operation used here has signature *int*[*routeId*] × *gline* → *bool*.

**Query P4**: Find all post workers who stayed in Hagener Strasse for more than one hour yesterday.

```
LET yesterday = day(2003,8,11);
LET one_hour = duration
  (hour(2003,1,1,0));
LET HagenerStrasse =
  ELEMENT(SELECT route FROM road
  WHERE name = 'Hagener Strasse');
SELECT name
FROM postman
WHERE duration(deftime(at(atperiods
  (trip, yesterday), HagenerStrasse)))
    > one_hour
```

**Query P5**: Find all post workers who stayed within 1 km to each other for more than two hours last Saturday.

```
LET Saturday = day(2003, 8, 9);
SELECT p1.name, p2.name
FROM postman AS p1, postman AS p2
WHERE EXISTS
  SELECT *
  FROM SET(distfunc,
    atperiods(distance(p1.trip,
      p2.trip), Saturday)
    when[fun(d: real) d < 1])
  decompose[distfunc,
    distfunc_component]
  WHERE duration(deftime
    (distfunc_component)) > 2 * one_hour
```

This is a fairly complicated query. For each pair of postmen we compute the real function describing their distance (by the lifted **distance** operator) and reduce it to Saturday and also (by **when**) to the parts when the distance is less than 1 km. By the SET construct (notation from [7]) we transform this into a relation with a single tuple and attribute called *dist-func*. Then the **decompose** operator transforms this into a relation with several distinct tuples, one for each continuous component of the distance function (see [7] for a more detailed explanation of **decompose**). We select such tuples for which the duration of the function is more than two hours. If the result is not empty, then the pair of postmen's names is returned.

Observe that for this query it is crucial that we have the advanced operations of [7] like **when** or **decompose** at our disposal.

**Query P6**: For the last week, who visited the Marktring area most often?

```
LET LastWeek = period(day(2003, 8, 4),
  day(2003, 8, 9));
LET Marktring = ELEMENT(
  SELECT reg FROM city_area
  WHERE name = 'Marktring');
SELECT name, no_components(
  at(atperiods(trip inside Marktring,
  LastWeek), true)) AS no_times
FROM postman
ORDER BY no_times
FIRST 1
```

Here for each postman we reduce its trip to last week, then check whether it was inside Marktring which yields a moving boolean. We reduce this to the times when it was true and count the number of continuous components. This is the number of times the postman was inside the Marktring area last week. We then sort the resulting relation by this number and return the top-ranked postman.[6]

This query demonstrates interaction between network and spatial values, since Marktring is purely geometric information.

**Query P7**: In the last week, which streets were visited most often?

```
LET frequencies =
  SELECT r.name AS name,
    no_components(at(atperiods(p.trip,
    LastWeek) inside r.route, true))
    AS no_times
  FROM postman as p, road as r
  WHERE trajectory(atperiods(p.trip,
    LastWeek)) intersects r.route;
SELECT name, SUM(no_times) AS visited
FROM frequencies
GROUP BY name
ORDER BY visited
```

Here the first query computes for each street and each postman how often he/she visited that street last week. The second query groups by street and returns the sum of visits for each street.

---

[6] The construct FIRST *n* is a small extension of SQL proposed in [35] that allows one to retrieve only some specified number of resulting tuples.

## 5.1.2 Queries on current information

For this section we assume that postmen are currently moving and the relation *postman2* stores their trips up to now.

```
postman2(name: string, trip: mgpoint,
  dest: gpoint)
```

Trip descriptions are "continuously" updated (extended) according to some location update policy. In addition to the trips, the relation stores the current destination (i.e. the location where the current parcel is to be delivered).

**Query P8**: Who is currently closest to Boeler Strasse 122 and moving towards that direction?

When a postman is moving in the direction of a given network position, it means, that his/her distance to that position is decreasing. We can determine that via the derivative of the distance.

```
LET Boeler122 =
  in_network(address2point('Hagen',
  'Boeler Strasse', 122));
SELECT name, distance(current(trip),
  Boeler122) AS dist
FROM postman
WHERE current(derivative(distance(trip,
  Boeler122))) < 0
ORDER BY dist
FIRST 1
```

In this query we use operator **address2point** (Sect. 4.7). We take the derivative of the *mreal* returned by the lifted **distance** operation.

**Query P9**: Who is closest to Bob and the desired destinations of their current packages are within 2 km? (Suppose that Bob has a problem with his motorcycle and needs another postman to help him with his package.)

This means: Among those postmen for which the destination is within 2 km of Bob's destination, who is closest to Bob? For this query, it is not clear whether the distance from Bob's destination to the other one or vice-versa is meant. We just take one of the cases.

```
SELECT p.name, distance(current(p.trip),
  current(bob.trip)) AS dist
FROM postman2 AS bob, postman2 AS p
WHERE bob.name = 'Bob' AND distance
  (bob.dest, p.dest) < 2
ORDER BY dist
FIRST 1
```

**Query P10**: Who is now moving eastwards in the Hagener Strasse?

(i) Let us assume that the Hagener Strasse is a dual road and we know that "eastwards" is the *up* direction of the Hagener Strasse.

```
SELECT p.name
FROM postman2 as p, road as r
WHERE r.name = 'Hagener Strasse'
  AND current(p.trip) inside r.route
    AND side(current(p.trip)) = 1
```

(ii) The second method uses the current direction of movement in space. It is determined to be "eastward" if it is within an angular range from 300 degrees through 60 degrees (chosen somewhat arbitrarily). This method is only valid if the road does not have parts going in another direction.

```
SELECT p.name
FROM postman2 as p, road as r
WHERE r.name = 'Hagener Strasse' AND
  current(p.trip) inside r.route
  AND NOT(current
    (mdirection(p.trip)) > 60
  AND current
    (mdirection(p.trip)) < 300)
```

(iii) A third method determines the relative position of the origin and the end point of Hagener Strasse to decide whether "eastward" is the direction of increasing or decreasing positions on the route.

```
SELECT p.name
FROM postman2 as p, road as r
WHERE r.name = 'Hagener Strasse'
  AND current(p.trip) inside r.route
  AND (
    (startsSmaller(Hagen, r.route)
  AND current (derivative(pos(p.trip))) > 0)
    OR
  (NOT startsSmaller(Hagen, r.route) AND
    current(derivative(pos(p.trip))) < 0)
  )
```

**Query P11**: Which street has more than ten postmen currently?

```
SELECT r.name
FROM road AS r, postman2 AS p
WHERE current(p.trip) inside r.route
GROUP BY r.name, r.route
HAVING COUNT(*) > 10
```

**Query P12**: Find all postmen who have stayed in the same street for more than 1 h up to now.

```
LET one_hour = ...//defined in query P4
SELECT p.name
FROM postman2 AS p
WHERE EXISTS
  SELECT *
  FROM SET(rno, route(p.trip)) decompose
    [rno, rno_intvl]
  WHERE now inside deftime(rno_intvl)
    AND duration(range(start(deftime
      (rno_intvl)), now) > one_hour
```

This is a bit complicated. For each postman, we apply the route operation to its *trip* attribute which yields a *moving*(*int*) value. We transform this into a relation to be able to apply the **decompose** operator. So SET(rno, route(p.trip)) is a relation with one tuple and one attribute *rno* of type *moving*(*int*). **Decompose** decomposes the *moving*(*int*) into continuous components, that is, one *moving*(*int*) value for each integer value assumed; these are stored in the new *rno_intvl* attribute. We then select from that relation intervals overlapping the current time and starting more than an hour ago. If the result relation is not empty, then this postman qualifies for the result.

**Query P13**: Who will pass Hagener Strasse before he/she can deliver his/her current package?

We need to compute a shortest path from the postman's current position to the destination and see whether it intersects Hagener Strasse.

```
LET HagenerStrasse = ...//defined in
  query P4
SELECT p.name
FROM postman2 AS p
WHERE shortest_path(current(p.trip),
  p.dest) intersects HagenerStrasse
```

**Query P14**: Find all postworkers currently in the Hagener Strasse and report their location.

```
SELECT current(p.trip)
FROM postman2 AS p
WHERE current (p.trip) inside
  HagenerStrasse
```

# 6 Implementation

In this section we address some of the implementation issues of the proposed framework. The data types and operations are currently being implemented as two new algebras in the SECONDO extensible DBMS [36, 37]. SECONDO does not have a fixed data model but allows one to implement DBMS data models as a set of algebra modules; each providing some types (type constructors, to be precise) and operations. For example, there exist algebras to represent standard data types, spatial data types, relations, B-tree indexes, R-tree indexes, or midi files, each with appropriate operations. We will add one algebra to represent the network with operations, and another algebra for the types *gpoint*, *gline*, *mgpoint*, and *mgline*. For an up-to-date overview of SECONDO see [37].

SECONDO is implemented on top of BerkeleyDB as a storage manager and provides an interface offering concepts for persistent storage such as files and records, with services such as concurrency control, recovery, and transaction management.

SECONDO offers a specific concept for the implementation of attribute data types. Such a type has to be represented by a record, called *root record*, which may have one or more components that are (references to) so-called *database arrays*. Database arrays are essentially arrays with any desired field size and number of fields; additionally they are automatically either represented inline in a tuple representation, or outside in a separate list of pages, depending on their size [38]. The root record is always represented within the tuple. Hence, in the following, our data types (except *network*) are represented by a root record and possibly some database arrays.

## 6.1 Data structures

In this section we present data structures for the data types *network*, *gpoint*, *gline*, *mgpoint* and *mgline*.

### 6.1.1 Type *network*

Type *network* is an abstract data type and we are free to represent it in any way we see fit. The main requirements are:

- The representation should support the export of relations from the network through operations **routes**, **junctions**, **sections**.
- Direct access to routes, given a route identifier, should be possible, e.g. to support construction or access operations of Sect. 3.2.4 or 3.2.5.
- The graph structure of the network should be represented explicitly to support network-specific operations like **shortest_path**.

We decided to represent a network by

(i) three relations called *routes, junctions,* and *sections,* and
(ii) a persistent adjacency list data structure

Note that in SECONDO it is possible that a value of a data type is represented in terms of other types of SECONDO, hence we can build a data structure having relations as components, which are not visible or directly accessible to the user. So the three relations representing the network are hidden from the user. Furthermore, it is possible to implement an operator using SECONDO queries. Therefore an operator like **routes** can be implemented by a query on an internal relation which may perform some projections, selections, even joins, if needed.

As stated in Sect. 2, a network $N$ is a pair $N = (R, J)$ where $R$ is a finite set of distinct routes and $J$ is a finite set of junctions in $R$. Sets $R$ and $J$ are represented by relations with the following schemas:

```
routes(id: int; length: real; curve:
  line; kind: bool; start: bool)
junctions(rlid: int; rlrc: int; posl: real;
  r2id: int, r2rc: int; pos2: real; cc:
    int; pos: point)
```

The tuple of the *routes* relation is equivalent to the domain of routes *Route* (see Sect. 2). The tuple of the *junctions* relation is somewhat different from the domain of junctions *Junction(R)*. Together with the route identifier, we keep another kind of pointer for direct access in the *routes* relation. It is the record identifier allowing for direct access to the stored tuple. With this pointer we can avoid a search in the *routes* relation if we want to find the routes pointed to by the route identification *r1id* and/or *r2id*. We then have the two new attributes *r1rc* and *r2rc* for this purpose.

Road sections corresponding to edges of the network graph can be derived from routes and junctions. They are needed for export by the **sections** operation, but also internally to support operations like **shortest_path**. We store all sections in a third relation called *sections* with the following schema:

```
sections(rid: int; rrc: int; posl: real;
  pos2: real; dual: bool; length: real;
  curve: line)
```
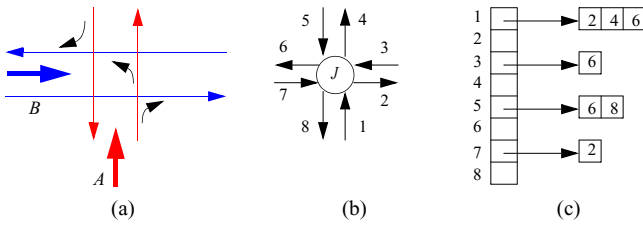
**Fig. 7** Adjacency list structure for a junction: **a** the junction, **b** the node with incident edges, **c** the adjacency list structure

The schema is similar to the one exported by **sections** (Sect. 3.2.1).

To support **shortest_path** and similar operations we also need to be able to find outgoing edges from a node efficiently. This is the purpose of the adjacency list data structure. Recall that the standard adjacency list structure (in main memory) is an array indexed by node numbers; each array entry contains a pointer to a list of successors of this node. This standard structure does not allow us to express the information in a transition matrix, since, when a node has been reached, it is not known, how it has been reached, and a restriction among its successors based on that is not possible. We here invent a slightly modified version of the adjacency list structure which consists of an array *indexed by edge numbers*; each array entry contains a list of successor edge numbers at the target junction of the edge, to which a transition is possible. The data structure is illustrated in Fig. 7 for the junction of Fig. 1. Note that the space requirements are proportional to the number of junctions.

The array for the edges (sections) is represented by a database array; each entry is a pointer (identifier) to another DBArray containing outgoing edges for the corresponding end node of the section. Observe that several edges with different associated curves may connect the same nodes. Because a DBArray is stored in one record of the underlying storage system, each box in Fig. 7c corresponds to one record. The array for edges will be kept entirely in memory whenever possible.[7]

Hence finding successors of an edge will cost one disk read operation (for the record with successor edge numbers) plus one read operation for each successor edge (to access the record of the section relation). Further improvements like clustering of successor records with section records, or moving some information from section records into successor records (so that one can avoid reading section records) are possible but beyond the scope of this paper.

Algorithms for computing shortest paths like Dijkstra's algorithm or A* [39, 40] can be adapted in a straightforward way to use the modified adjacency list structure; they need to keep directed edges ordered by distance of their target nodes in the priority queue instead of nodes.

### 6.1.2 Types *gpoint* and *gline*

For representing the *gpoint* type constructor, a simple data type is required. It is represented as a root record with the following format:[8]

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

The first value *nid* corresponds to a network identifier, *rid* and *pos* represent a route measure (*RMeas*), and *side* contains the side information.

The *gline* data structure is slightly more complex, since it contains a finite set of quasi-disjoint route intervals (see Sect. 2). We represent this set of route intervals as an (ordered) database array:

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

As for the *gpoint* data type, the *gline* needs a network identifier *nid*. The set of route intervals is represented by *rints*. Every interval contains the route identification *rid*, the interval position (*pos1*, *pos2*), and the side of the route represented by *side*.

### 6.1.3 Types *mgpoint* and *mgline*

For representing data types *moving*(*gpoint*) and *moving*(*gline*) (*mgpoint* and *mgline* for short) we use the same strategy as in [8, 9] outlined already in Sect. 4.1, namely to use the sliced representation. Each slice is represented by a so-called *temporal unit* consisting of a time interval and a description of the temporal development during this time interval via some *unit function*. The data type for a unit is called the *unit type*. The overall data structure for a value of type *moving*($\alpha$) is then a database array containing the corresponding $\alpha$-units ordered by time interval. A total order exists since all unit time intervals are disjoint. Hence in the sequel we only need to describe the unit types for *gpoint* and *gline* called *ugpoint* and *ugline*, respectively.

For the *gpoint* temporal unit we need to store the network identification *nid*, the route identification *rid*, a time interval (*t1*, *t2*), two graph positions *pos1* and *pos2* representing the positions at the beginning and the end of the time interval, and the side of the route *side*.

---

[7] For extremely large network databases one might have a problem here and need to invent further strategies; however, with today's memory sizes this will normally not be a problem.

[8] The structures are described in pseudo-code; the actual implementation is in C++.

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

The unit function evaluates a position inside the unit time interval assuming a linear movement from *pos1* to *pos2* in the time interval (*t1*, *t2*). It is important to note that we store the route identification inside the data structure because the movement in the time interval must be on the same route. If the point changes the route, then a new temporal unit must be created (as well as for any change of speed).

Using the same approach, the temporal unit of the *gline* data type contains a set of moving intervals. The data type is then

```
ugline: record {
  nid: int;
  rtints: DBArray of record
              {
                rid: int;
                side: {up, down, none}
                t1: Instant;
                t2: Instant;
                pos11: real;
                pos12: real;
                pos21: real;
                pos22: real; }
}
```

where $pos_{ij}$ represents $pos_i$ at time instant $t_j$. The positions inside the temporal unit are also calculated assuming linear movements.

It is important to note that for every time $t$ inside the temporal unit time interval, the resulting *gline* value from the temporal function should be valid, i.e., its route intervals must be quasi-disjoint (see Sect. 2). When this constraint does not apply, then a new *gline* temporal unit must be created.

We have required at the start of Sect. 6 that every attribute type must be represented by a root record plus possibly some database arrays. This is violated by the definition of *ugline* as this record would be an element of a DBArray for *mgline* so that we get a two-level tree of database arrays. This problem is solved by a technique described in [8] to actually represent the DBArray of each unit not separately, but as a subarray of a global array for all units of this *mgline* value.

## 6.2 Query execution

In this section we try to give an idea of query processing in the proposed framework. We discuss two example queries from Sect. 5.1.

**Query P2**. Which places did Bob visit between 9:00 am and 11:00 am of last Saturday?

```
LET Saturday9to11 = period
  (hour(2003,8,9,9), hour(2003,8,9,10));
```

This part of the query requires O(l) time, only operators to construct a time interval are called.

```
SELECT trajectory(atperiods(trip,
  Saturday9to11))
FROM postman
WHERE name = 'Bob'
```

An index on the *name* attribute of *the postman* relation can permit (almost) direct access to the postman whose name is Bob. The database array representing Bob's *trip* (*mgpoint*) is searched with the **atperiods** operator to return the parts of the movement that belong to last Saturday from 9:00 to 11:00 am. This is done using a binary search for the first unit in the trip overlapping the time interval and then a sequential scan reading the units belonging to the search interval. Assuming that the $r$ units contained in the time interval fit into $R$ disk pages, this algorithm will take $O(\log n + R)$ disk accesses, where $n$ is the number of components of the *mgpoint* for Bob's trip. The result is processed further in memory by the **trajectory** operator.

**Query P4**. Find all post workers who stayed in Hagener Strasse for more than one hour yesterday.

```
LET yesterday = day(2003,8,11);
LET one_hour = duration(hour(2003,1,1,0));
```

These parts of the query are instantaneous again, only operators to construct time intervals are called.

```
LET HagenerStrasse =
  ELEMENT(SELECT route FROM road
    WHERE name = 'Hagener Strasse');
```

An index on the *name* attribute of the *road* relation can permit direct access to the Hagener Strasse.

```
SELECT name
FROM postman
WHERE duration(deftime(at(atperiods(
  trip, yesterday), HagenerStrasse)))
  > one_hour
```

This query is more complex: for each tuple in the *postman* relation, first the operation **atperiods** reduces the *trip* to *yesterday*; then the **at** operation returns the part of yesterday's trip that were in Hagener Strasse; then the **deftime** operation takes only the temporal dimension of the trip, and finally the **duration** operation is called to sum the intervals of the trip and this value is compared to see if it is greater than one hour *(one_hour* variable).

A first, straightforward way to execute the query is the following: All tuples of *the postman* relation are read. For each tuple, the **atperiods** operator selects only the parts of the trip that were realized yesterday. This is done using a binary search and then a sequential scan of units, as for the previous query. After this, the **at** operator will run through all the units that are returned by the **atperiods** operator and reduce to the units with the route *HagenerStrasse*. Then **deftime** and **duration** are computed as in [9].

Of course, it may be costly to read all tuples of *the postman* relation. An index could support the processing of this most expensive part of the query:

```
at(atperiods(trip, yesterday),
  HagenerStrasse)
```

A spatio-temporal index *postman_trip* with key *<time_interval, route_id>* with a pointer to the tuple record could support this part of the query. An example of such an index is [41]. Every unit of every trip in *the postman* relation produces an entry in the index. Furthermore, we hope that the query optimizer can be made smart enough to add further conditions to the query as in

```
SELECT name
FROM postman
WHERE duration(deftime(at(atperiods
  (trip, yesterday), HagenerStrasse)))
    > one_hour
  AND present (trip, yesterday)
  AND passes(trip, HagenerStrasse)
```

and to map the latter two conditions to a search on the index *postman_trip*. This is a kind of window query with time interval equal to *yesterday* and route identifier taken from *HagenerStrasse*. Using this index, we can avoid reading tuples in *the postman* relation that do not have trips that *yesterday* passed through *HagenerStrasse*.

The purpose of this section was only to give some idea about query processing issues and strategies in this framework. A detailed study of query processing, implementation of all operators, and optimization rules needed is left to future work.

## 7 Related Work

Related work in general was described in the introduction. In this section, we first give some background on earlier approaches dealing with graphs in databases and spatially embedded networks. We then discuss the most closely related work, especially addressing issues of modeling and querying moving objects in networks, in some detail.

Graph models in databases were studied around 1990 by several groups, in particular by Cruz, Mendelzon, and Wood (e.g. [42]) and by Gyssens, Paredaens, and van Gucht (e.g. [43]), see also [44]. These works did not consider spatially embedded networks, one of their main goals was a new approach to recursive query processing.

The modeling of spatially embedded networks was addressed by Erwig and Güting [45] and Güting [46]. Reference [45] integrates object-oriented modeling, spatial data types, graphs and sequences in a rich data model that also employs concepts from functional programming. The approach in [46] is perhaps a bit simpler and more streamlined. Here an emphasis is on integrating graph modeling into an object-oriented modeling environment seamlessly. The model offers object classes of three kinds called simple classes, link classes, and path classes, which can be used for general data modeling, but also to model nodes, edges, and path objects, respectively, of a graph or network. To obtain spatial networks, one associates a node with a point in the plane and an edge with a polyline. Both [45, 46] already emphasize the importance of path objects, which correspond to routes in our current model. Güting [46] further develops an extension of SQL to deal with general querying and graph and spatial network querying in a smoothly integrated way. Compared to this paper, in these works the network modeling is a bit less precise (e.g. there are no dual roads or transition matrices), and of course, there is no modeling or querying of moving objects.

Network query processing, in particular shortest path computation, has been considered by the groups around Shekhar (e.g. [47, 48]) and Rundensteiner (e.g. [49, 50]). In [48] an adjacency list data structure clustered into pages is developed. Clustering is based on *z*-order of the node positions.

A recent paper [28] has studied data structures for spatial networks and evaluation of several types of queries (nearest neighbors, range query, closest pairs, distance join). They compare Euclidean and network-based evaluation of such queries. The data structure employed is similar to the one we use in Sect. 6 if we add clustering of nodes and a spatial index on geometries in the *sections* relation. Their representation of junctions is different, however: To represent possible transitions at a junction a graph with 8 nodes is needed (see [28], Fig. 3.2). Our adjacency list scheme appears simpler and more efficient. This work deals with static networks only; there are no moving objects considered.

We now pass on to moving objects.

The work of the Wolfson group cited in the introduction does, in fact, assume that objects move in networks. However, the network is not modeled in any way. A dynamic location attribute is given by a polyline (in 2D space) plus some additional information like start time, start position, and speed. It is assumed that the polyline (trajectory) has been derived from the network initially; thereafter the movement is described in purely geometric terms. There is no (explicit) relationship to the network any more. The problem of discovering relationships between moving objects and the network later in queries is not addressed anywhere. Since trajectories are represented geometrically, not in terms of network elements, this would require expensive geometric computations.

A paper by Vazirgiannis and Wolfson [51] considers modeling and querying moving objects in road networks. The network model is a relation representing "blocks" which means edges of the network graph. Each tuple contains a polyline describing the geometry of the edge. It also contains very application-specific information such as the ranges of street numbers on the left and right side of the road, or the left side and right side Zip codes. The model is in fact taken directly from a company providing geographic data.

The network model basically corresponds to an undirected graph, where nodes are street crossings and edges are city road blocks. The model is not defined formally, and it is not a generic but rather an application-specific model.

Moving objects are described by geometric polylines, as in the earlier work mentioned above. The approach is to compute a shortest path on the network, then to assign

traveling times (using length of the block and speed limit) and so to arrive at a trajectory. After this process, again the trajectory does not contain any references to network components.

Comparing to our work, it is the usual graph model, in contrast to our route-oriented model. It does not model objects moving in networks, only derives 2D moving objects initially. This fact leads them also to use 3D indices (2D + time) instead of network-space indices.

Regarding the query language, a few ad-hoc extensions of SQL are proposed, using modifiers in the WHERE clause

```
WITHIN (DISTANCE s | TRAVELTIME t) FROM R
[ALONG EXISTING PATH | ALONG SHORTEST PATH]
[(ALWAYS BETWEEN) | (SOMETIMES BETWEEN)
  starttime AND endtime]
```

What can be expressed in this language is extremely limited when compared to our proposal.

Two papers by Jensen et al. [30, 52] have looked at data modeling issues for spatial networks with respect to possible uses for location-based services. They describe as a case study the data model used by the Danish road directory and a Danish company. The emphasis here is to explain that real road networks are quite complex, and that just simple directed graph models are not sufficient. The case study suggests a model that uses several interrelated representations, called (i) the kilometer post representation, (ii) the link-node representation, (iii) the geographical representation, and (iv) the segment representation. The kilometer post representation corresponds roughly to our route-oriented model, the link-node representation is related to directed and undirected graphs. The geographical representation is given by geometric position of certain points on a road. The segment representation is hidden from a user and is basically a higher-level graph than the link-node representation. All these representations are expressed in terms of relational tables.

This is an interesting application study, and we have drawn some of our motivation to use a route-oriented model from the first of these papers [30]. This is not a formalized database model, and it is too complex to be a good basis for a query language. Moving objects are not modeled.

The same group has described a more formalized model incorporating some of these ideas [53]. They propose to use two different models of a network together, called (i) the *2D representation* and (ii) the *graph representation*. The first is geared to describing a network at very high detail, the second should support efficient computations. The 2D model is essentially a graph whose edges (called *road segment*) are pieces of road that can be represented by a straight line segment, and whose nodes are connections between such segments. A road segment has some associated information, beyond the line segment in particular a code for an allowed movement direction. A node (called *connection*) is a 2D point with an associated set of incident road segments. Interestingly, they have independently from us also discovered the idea of a transition matrix called a connection matrix here. Hence a connection (node) also has a connection ma-

trix; if there are $m$ incident segments the connection matrix is an $m \times m$ matrix.

Beyond the network, the 2D representation has *data points* and *query points*. A *data point* is a set of triples (*point*, *segment*, *accessibility code*) plus a list of properties (which are just keywords). It models a facility and how it is accessible from one or more road segments. The accessibility code tells whether one can enter from one or both directions of the road segment. A query point is given by a triple (*point*, *segment*, *direction*) and models a moving object or vehicle at one instant of time when a query is issued. *Direction* is the current direction of movement.

The graph representation (ii) is a directed graph with an additional relationship on edges called *co-edges*. Two edges are co-edges if they belong to the same road segment and one can switch from one to the other by a U-turn. Data points are now represented relative to edges. Hence a facility having $n$ access points must now be represented $n$ times. A query point is given by an edge plus position[9] plus speed and time of last update. Hence one can compute a movement function as long as the object is on this edge. When it would leave the edge, it stays at its end. This is like the dynamic attributes of Wolfson. The paper further describes how the graph representation can be derived from the 2D representation.

This model is the closest there is to our network model. Nevertheless there are still big differences.

- Both the 2D and the graph representation are graph-oriented, i.e., they do not offer a route-oriented model as we do. For the reasons described in the introduction it is important to use a route-oriented model.
- They do not offer a model for moving objects in networks. Trajectories of moving points relative to the network are not available. Obviously, there is also no modeling of either static or moving network regions.
- There is no associated query language for this model.

We feel the dual model described here is too complicated to be a suitable basis for representing moving objects in networks and for defining queries relative to the model. The model is geared towards location management applications; this is why only query points, instead of moving objects, are modeled. The very detailed modeling of facility access in the 2D representation is interesting. However, facility access can be modeled well enough by our concept of graph positions (*gpoint*). On a dual road, if it needs to be accessible from both directions, we have to store two entities, one for each side of the road, as they do for the graph model, too.

It is interesting that the concept of a transition or connectivity matrix also occurs in their work. Using two-way connectivity that can be encoded into an integer may be more practical than storing matrices of varying size.

Recent related work by our own group is [54–56]. Reference [54] develops a model for dynamic networks. It describes how availability of nodes or edges can change over

---

9 In fact, the model generally uses two positions, one in terms of length of the edge, another in terms of weight which roughly corresponds to estimated travel time.

time. In particular, edges may be partially blocked and later reopened. A query language is defined that allows one to formulate questions about past states of the network. The underlying network model is still graph-oriented (rather than route-oriented), and this paper does not consider moving objects.

The work in [55] builds upon the model defined in this paper, providing several extensions. It integrates the model with the concepts developed in [54] for dynamic networks. It further introduces transitions between several such networks and considers location update strategies, querying with uncertainty, and prediction of future locations. Finally, paper [56] defines the MON-tree, a route-oriented index for network-constrained moving objects.

## 8 Conclusions

The contribution of this paper is a precise and comprehensive data model and query language for moving objects in networks. Whereas there exists some work on modeling networks, and some work on querying mobile objects is network-related, to our knowledge there is nowhere in the literature an integrated approach to modeling and querying with comparable expressive power to the model and language of this paper. In more detail, the contributions are the following:

- We provide a precise formal model of a spatially embedded network in terms of routes and junctions, which is a better basis for representing moving objects than nodes and edges. The model is detailed enough to distinguish simple and divided roads and describe connectivity at junctions.
- We offer abstract data types for a network and for static and moving network positions and regions. We are not aware of any other model providing moving object trajectories relative to networks, neither of any work defining static or moving network regions. The new data types are integrated into a relational environment with suitable interface functions.
- We offer a rich and systematically designed algebra to work with the new data types. The expressive power of the resulting query language is demonstrated by a large number of queries for three example applications.
- An implementation strategy is outlined in the paper, and a prototype implementation is underway.

The part of the model dealing with moving entities is designed as a careful extension of the earlier model of [7]. Some may feel that this limits the novelty of this approach. Indeed, it is more difficult to describe an extension to an existing framework (we have done our best in Sect. 4) than to invent everything from scratch. Possibly it is also less fun to read. Nevertheless, we are deeply convinced that this is the right approach, for the following reasons.

- The model of [7] is a comprehensive model of abstract data types for moving entities in an unconstrained environment. It is obvious that constrained movement (in

networks) is a special case of that. Therefore it is clear that lots of concepts from the unconstrained environment apply and can and should be reused. If one looks carefully at the example queries, then it is obvious that the expressive power of the overall approach is achieved by the combination of new types and operations of this paper with the generic operations and facilities (e.g. lifting) of [7].
- It is possible to use data types for unconstrained and constrained movement together in a single seamless, consistent framework.
- Beyond just the model of [7], a lot of further work has been invested into implementation issues [8, 9], parts of which can be reused (see Sect. 6).
- Regardless of how it was achieved, this is simply the most comprehensive model for moving objects in networks and the most expressive query language that has been proposed so far, which proves the success of this approach.

Future work will address implementation issues such as algorithms for the operations and query processing and optimization strategies, including indexing. Furthermore, extensions for traffic jam discovery and prediction are an interesting research issue. Note that it would be easy in our framework to add an operator that computes a set of moving traffic jams from a set of moving objects, as all needed data types are available. The question is what exactly defines a traffic jam, what parameters such an operator should have, and by which algorithm it could be evaluated.

## Appendix A: Definition of continuity for *moving*(*gpoint*) and *moving*(*gline*)

Whereas it is obvious what continuity means for a *moving*(*real*), this is not so clear for other types *moving*($\alpha$), e.g. *moving*(*region*). [7] provides an extended definition of continuity based on a function $\psi$ which defines a measure of dissimilarity between two $\alpha$ values, hence is defined as $\psi : A_\alpha \times A_\alpha \to \mathbb{R}$. The $\psi$ function is then defined for each data type $\alpha$ to be 0 when the two $\alpha$ values are equal, and to approach 0 when the values get more and more similar.

What we need to do here is to define continuity for *moving*(*gpoint*) and *moving*(*gline*) values. We decide that a discontinuity occurs when a *moving*(*gpoint*) changes the route, and of course also, when it changes the location on a route without traversing the intermediate locations. Hence we define for two *gpoint* values:

$$\psi((i_1, (r_1, d_1, s_1)), (i_2, (r_2, d_2, s_2)))$$
$$= \begin{cases} |d_1 - d_2| & \text{if } i_1 = i_2 \wedge r_1 = r_2 \wedge s_1 = s_2 \\ 1 & \text{otherwise} \end{cases}$$

Recall that a *gpoint* value consists of a network number and a triple which in turn consists of a route identifier, a route measure, and a side

value. For a *moving(gline)* we employ the symmetric difference (as for region values in [7]) and define for two *gline* values

$$\psi(GL_1, GL_2) = \mathbf{size}(\mathbf{g\_minus}(GL_1, GL_2))$$
$$+ \mathbf{size}(\mathbf{g\_minus}(GL_2, GL_1))$$

using the operations defined above.

## Appendix B: Further query formulations

### Vehicles on Highway Networks

For this application we will assume that we have a network called GermanHighways and the following relations:

```
highway(no: int, route: int)
vehicle(licence: string, trip: mgpoint)
gas_station(company: string, id: int,
  loc: gpoint)
motel(name: string, chain: string, min_rate:
  int, max_rate: int, pool: bool, loc: gpoint)
speed_limit(limit: int, stretch: gline)
```

For the vehicles we assume that their locations are updated regularly according to a location update policy. Regarding their future trajectories, there are some vehicles that have informed the server about their destination, so that their trip attribute includes the estimated future movement up to the destination. For the other vehicles the trip is maintained under updates assuming that the car will continue on this highway at the speed of the speed limit +15 km/h or at 160 km/h if there is no speed limit, up to the end of this highway.[10] The estimated trip ends there.

### Queries on past and static information

The first two queries just demonstrate that we can get information about facilities on the network.

**Query H1**: Order highways by their average distance between gas stations.

```
SELECT h.no, length(h.route)/(COUNT(*) + 1)
  AS dist
FROM highway AS h, gas_station AS g
WHERE g.loc inside h.route
GROUP BY h.no
ORDER BY dist
```

We assume a highway with *n* gas stations is divided by them into *n* + 1 parts. Highways are dual routes, but a gas station usually occurs on both sides of the highway, therefore we can ignore this.

**Query H2**: Which percentage of the German highway network does have a speed limit?

```
ELEMENT(SELECT SUM(length(route)) FROM highway)/
ELEMENT(SELECT SUM(length(stretch))
  FROM speed_limit)
```

**Query H3**: How many cars passed gas station *X* today?

```
SELECT COUNT(*)
FROM vehicle AS v, gas_station AS g
WHERE g.id = X AND v.trip passes g.loc
```

---

[10] These assumptions are not unrealistic for Germany.

**Query H4**: How does traffic density at km 140 of highway 45 of the network change through the day?

One needs to decide for one side of the highway; let us assume the *up* direction is asked for. We assume constants *up* = 1, *down* = −1 have been defined in the database.

We will evaluate the query on a specific day (yesterday) and produce a table that lists for each hour of the day the number of cars that passed. To keep it simple we assume each car passes this location only once on the day and use for evaluation only the first instant when it passes.

```
LET yesterday = ...;
LET highway45 = ELEMENT(SELECT route
  FROM highway WHERE no = 45);
LET location = gpoint (GermanHighways,
  highway45, 140, up);
LET passing_times =
  SELECT hour(inst(initial(at(atperiods
    (v.trip, yesterday), location))))
  AS hour
  FROM vehicle AS v
  WHERE atperiods(v.trip, yesterday)
    passes location;
SELECT hour, COUNT(*) AS no_vehicles
FROM passing_times
GROUP_BY hour
```

**Query H5**: Find vehicles that exceeded the speed limit by more than 20%; when and where did that occur?

```
SELECT v.licence,
  deftime(speed(at(v.trip, s.stretch)) when[.
    > s.limit * 1.2])
  AS time
  trajectory(atperiods(v.trip,
    deftime(speed(at(v.trip, s.stretch))when
    [. > s.limit * 1.2]))
  AS place
FROM vehicle AS v, speed_limit AS s
WHERE v. trip passes s.stretch
  AND max(rangevalues(speed(at(v.trip, s.
    stretch)))) > s.limit * 1.2
```

**Query H6**: Which fraction of vehicles passes from highway 45 to highway 1 at their junction?

We retrieve vehicles that passed the junction between highways 45 and 1 and that one minute earlier have been on highway 45. We then compare among such vehicles how many are 1 min later still on highway 45 or on highway 1, respectively.

```
LET one_minute = duration
  (minute(2003, 1, 1, 0, 0));
LET Junction = ELEMENT(
  SELECT single(intersection(gline(h1.route),
    gline(h2.route)))
  FROM highway AS h1, highway AS h2
  WHERE hl.no = 45 AND h2.no = 1);
LET cars =
  SELECT inst(initial(at(v.trip, Junction)) AS
    junction_time, route(val(atinstant(v.trip,
      junction_time - oneminute)))
    AS route1,
    route(val(atinstant(v.trip, junction_time +
      one_minute)))
    AS route2,
  FROM vehicle AS v
  WHERE v.trip passes Junction;
LET total = ELEMENT(
  SELECT COUNT(*)
  FROM cars AS c, highway AS h1
  WHERE hl.no = 45
    AND c.routel = h1.route);
```

```
LET switch = ELEMENT( SELECT COUNT(*)
  FROM cars AS c, highway AS h1, highway AS h2
  WHERE h1.no = 45 AND h2.no = 1
    AND c.routel = h1.route AND c.route2 =
      h2.s route);
switch / total
```

This final expression is the result of the query.


## Queries on future information

**Query H7**: Which vehicles will reach gas station X within the next 30 min?

```
LET one_minute = ...//Query H6
SELECT v.licence
FROM vehicle AS v, gas_station AS g
WHERE g.id = X AND
  atperiods(v.trip, range(now, now + 30 *
  one_minute)) passes g.loc
```

**Query H8**: Keep me informed about the 5 closest motels along the highway.

We see no way to formulate this query.

**Query H9**: Keep me informed about motels within 5 kms distance along the highway.

We wish to see the result as a list of motel names together with time periods indicating when the respective motel belongs to the result set. Furthermore, we wish to get a relation containing those time instants when the result set changes. A similar result is obtained in evaluating such continuous queries in Wolfson's approach [1].

```
LET mytrip = ...// an mgpoint describing my
  estimated trip
LET CloseMotels =
  SELECT m.name AS name,
    deftime(distance(atperiods(mytrip, range
    (now, maxinstant)), m.loc)
    when[. < 5]) AS time_period
  FROM motel AS m
  WHERE atperiods(mytrip, range(now,
    maxinstant)) passes m.loc;
(SELECT start(time_period) AS time FROM
  CloseMotels)
  UNION
(SELECT end(time_period) AS time FROM
  CloseMotels)
ORDER BY time
```


## Traffic jams

For this application we will use the German highways network and a relation containing the traffic jams as a *mgline* attribute. We will assume that the traffic jams were calculated and stored in this relation. We further assume that traffic jams are restricted to single routes, as is usually the case. A traffic jam spilling over into another route would be represented by two distinct traffic jams. Finally, a traffic jam consists of a single continuous piece of route, hence a *gline* value with a single component.

```
highway(no: int, route: int)
vehicle(licence: string, trip: mgpoint)
traffic_jam(no: int, area: mgline)
```

**Query T1**: Which traffic jams exist now?

Current traffic jams might be returned in different ways. We first construct a table that has rows with the highway number and the start and end position of each traffic jam.

```
SELECT h.no, pos(min(current(j.area)))
  AS startpos, pos(max(current(j.area)))
  AS endpos
FROM traffic_jam AS j, highway AS h
WHERE current(j.area) intersects h.route
```

Second, we can assume that a graphical user interface can display *gline* values (say, against the background of the entire highway network). In this case, the query can simply be:

```
SELECT current(area) AS current_jam
FROM traffic_jam
```

**Query T2**: When and where did traffic jam *X* appear and disappear, respectively?

```
SELECT inst(initial(j.area)) AS starttime,
  inst(final(j.area)) AS endtime,
  val(initial(j.area)) AS startarea,
  val(final(j.area)) AS endarea
FROM traffic_jam AS j
WHERE j.id = X
```

We assume the user interface allows one to display *instant* and *gline* values.

**Query T3**: During which times did *X* grow and shrink, respectively?

If a traffic jam grows (shrinks), the length of its associated *mgline* also grows (shrinks). We can then use the derivative of this length to see whether it is growing or shrinking. We will construct a table that has one column *change* with entries either "grow" or "shrink", and as a second column *periods* a set of time intervals (a *periods* value) when this was happening.

```
LET changes =
  SELECT ''grow'' AS change,
    deftime(derivative(length(j.area))
    when[. > 0]) AS periods
  FROM traffic_jam AS j
  WHERE j.id = X
  UNION
  SELECT ''shrink'' AS movement,
    deftime(derivative(length(j.area))
    when[. < 0]) AS periods
  FROM traffic_jam AS j
  WHERE j.id = X
```

If the user interface can display *periods* values (as it already can in our SECONDO implementation), we can display the *changes* relation directly and are done. Otherwise, let us assume we wish to construct a table that has one row for each time interval. This can be done as follows:

```
LET changes2 = changes decompose[periods,
  period];
SELECT change, start(period) AS starttime,
  end(period) as endtime
FROM changes2
```

Here we assume that **start** and **end** are available as alias names [7] for the operations min and max, when applied to *periods* values.

**Query T4**: At what time did it grow most?

The query may in general return a set of time intervals, rather than a single instant.

```
SELECT deftime(atmax(derivative(length
(j.area)) when[. > 0])) AS periods
FROM traffic_jam AS j
WHERE j.id = X
```

Again we assume the GUI can display *periods* values; otherwise the query needs to be extended like the previous one, using **decompose**.

**Query T5**: What time did vehicle *X* spend within traffic jam *Y*?

This query can also return more than one time interval, i.e., vehicle *X* can enter and leave traffic jam *Y* more than once.

```
SELECT deftime(at((v.trip inside j.area),
  TRUE)) AS periods
FROM traffic_jam AS j, vehicle AS v
WHERE v.license = X AND j.no = Y
```

First, we take the vehicle *X* and traffic jam *Y*, and get a moving boolean describing when the first was inside the second. Then, we take the periods when this was true.

**Query T6**: At what speed did traffic jam *X* move?

```
SELECT speed(center(area))
FROM traffic_jam
WHERE no = X
```

Result is a moving real which must be shown at the user interface.

**Query T7**: Show the part of the highway network that was affected by traffic jams yesterday.

Obviously, the result should be a *gline* value. We take each traffic jam that existed yesterday, reduce it to yesterday, compute its projection, and then form the union of all such projections.

We need an aggregate function that forms the union of a set of *gline* values. A general technique to define such functions was introduced in [7]. We can write:

```
LET gl_union = AGGREGATE(union, TheEmptyGLine);
```

To define an aggregate function, one needs to specify a binary operator and a neutral element. The latter is returned if the aggregation is applied to an empty relation. Here we assume that *TheEmptyGLine* is a constant in the database of type *gline* containing an empty value. Then the query is:

```
LET yesterday = day(2003, 8, 11);
SELECT gl_union(traversed(atperiods(j.area,
  yesterday))) AS yesterday_jams
FROM traffic_jam AS j
WHERE present(j.area, yesterday)
```

**Query T8**: Did any two traffic jams merge into one?

This raises the question how identities of traffic jams are managed in data capture. Let us assume that a merge can happen in either of three ways: (i) the first traffic jam continues to exist and absorbs the second, (ii) the second absorbs the first, and (iii) both former traffic jams cease to exist and a new one is established.

We can discover all these cases by the fact that a discontinuity in the movement occurs (see the definition in Sect. 4.5). Hence in a first step, we decompose traffic jams into continuous pieces. Then, we search for three instances *j1*, *j2*, and *j3* of (continuous pieces of) traffic jams with these two properties:

- The end times of *j1* and *j2* are equal and the same as the starting time of *j3*.
- At that time, the union of the areas of *j1* and *j2* must be equal to the area of *j3*.

As a result, we return the time that the merge occurred, the identifiers and areas of the two merged traffic jams, and the identifier and area of the traffic jam after the merge. The query can then be formulated as follows:

```
LET cont_jam = traffic_jam decompose[area,
  cont_area];
SELECT inst(initial(j3.cont_area)) AS time,
  jl.no AS tjam1, val(final(j1.cont_area))
    AS tjamarea1,
  j2.no AS tjam2, val(final(j2.cont_area))
    AS tjamarea2,
  j3.no AS newtjam, val(initial(j3.cont_area))
    AS newtjamarea
FROM cont_jam AS j1, cont_jam AS j2,
  cont_jam AS j3,
WHERE inst(final(j1.cont_area)) = inst(initial
  (j3.cont_area))
```

```
AND inst(final(j2.cont_area)) = inst(initial
  (j3.cont_area))
AND val(initial(j3.cont_area)) = union(final
  (j1.cont_area), final(j 2.cont_area))
```

## References

1. Sistla, A.P., Wolfson, O., Chamberlain, S., Dao, S.: Modeling and querying moving objects. In: Proceedings of the 13th International Conference on Data Engineering (ICDE), pp. 422–432 (1997)
2. Wolfson, O., Chamberlain, S., Dao, S., Jiang, L., Mendez, G.: Cost and imprecision in modeling the position of moving objects. In: Proceedings of the 14th International Conference on Data Engineering (ICDE), pp. 588–596 (1998)
3. Wolfson, O., Xu, B., Chamberlain, S., Jiang, L.: Moving object databases: Issues and solutions. In: Proceedings of the 10th International Conference on Scientific and Statistical Database Management (SSDBM), pp. 111–122 (1998)
4. Wolfson, O., Sistla, A.P., Chamberlain, S., Yesha, Y.: Updating and querying databases that track mobile units. Distributed and Parallel Databases **7**(3), 257–387 (1999)
5. Koubarakis, M., Pernici, B., Schek, H.J., Scholl, M., Theodoulidis, B., Tryfona, N., Sellis, T., Frank, A.U., Grumbach, S., Güting, R.H., Jensen, C.S., Lorentzos, N., Manolopoulos, Y., Nardelli, E. (eds.): Spatio-temporal databases: The CHOROCHRONOS approach. Springer-Verlag, Lecture Notes in Computer Science 2520 (2003)
6. Erwig, M., Güting, R.H., Schneider, M., Varzigiannis, M.: Spatio-temporal data types: An approach to modeling and querying moving objects in databases. GeoInformatica **3**(3), 265–291 (1999)
7. Güting, R.H., Bohlen, M.H., Erwig, M., Jensen, C.S., Lorentzos, N.A., Schneider, M., Vazirgiannis, M.: A foundation for representing and querying moving objects. ACM Transactions on Database Systems **25**(1), 1–42 (2000)
8. Forlizzi, L., Güting, R.H., Nardelli, E., Schneider, M.: A data model and data structures for moving objects databases. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 319–330 (2000)
9. Cotelo Lema, J.A., Forlizzi, L., Güting, R.H., Nardelli, E., Schneider, M.: Algorithms for moving objects databases. The Computer Journal **46**(6), 680–712 (2003)
10. Grumbach, S., Rigaux, P., Segoufin, L.: The DEDALE system for complex spatial queries. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 213–224 (1998)
11. Rigaux, P., Scholl, M., Segoufin, L., Grumbach, S.: Building a constraint-based spatial database system: Model, languages, and implementation. Information Systems **28**(6), 563–595 (2003)
12. Grumbach, S., Rigaux, P., Segoufin, L.: Spatio-temporal data handling with constraints. GeoInformatica **5**(1), 95–115 (2001)
13. Chomicki, J., Revesz, P.: Constraint-based interoperability of spatio-temporal databases. In: Proceedings of the 5th International Symposium on Large Spatial Databases (SSD), pp. 142–161 (1997)
14. Chomicki, J., Revesz, P.: A geometric framework for specifying spatiotemporal objects. In: Proceedings of the 6th International Workshop on Temporal Representation and Reasoning (TIME), pp. 41–46 (1999)
15. Su, J., Xu, H., Ibarra, O.H.: Moving objects: Logical relationships and queries. In: Proceedings of the 7th International Symposium on Spatial and Temporal Databases (SSTD), pp. 3–19 (2001)
16. Mokhtar, H., Su, J., Ibarra, O.H.: On moving object queries. In: Proceedings of the ACM Symposium on Principles of Database Systems (PODS), pp. 188–198 (2002)
17. Agarwal, P.K., Arge, L., Erickson, J.: Indexing moving points. In: Proceedings of the 19th Symposium on Principles of Database Systems, Dallas, Texas, pp. 175–186 (2000)

18. Pfoser, D., Jensen, C.S., Theodoridis, Y.: Novel approaches in query processing for moving object trajectories. In: Proceedings of the 26th Int. Conference on Very Large Data Bases (Cairo, Egypt), pp. 395–406 (2000)
19. Hadjieleftheriou, M., Kollios, G., Tsotras, V.J., Gunopulos, D.: Efficient indexing of spatiotemporal objects. In: Proceedings of the 8th International Conference on Extending Database Technology (EDBT, Prague, Czech Republic), pp. 251–268 (2002)
20. Song, Z., Roussopoulos, N.: K-nearest neighbor search for moving query point. In: Proceedings of the 7th International Symposium on Spatial and Temporal Databases (SSTD), pp. 79–96 (2001)
21. Tao, Y., Papadias, D.: Spatial queries in dynamic environments. ACM Transactions on Database Systems **28**(2), 101–139 (2003)
22. Yanagisawa, Y., Akahani, J., Satoh, T.: Shape-based similarity query for trajectory of mobile objects. In: Proceedings of the 4th International Conference on Mobile Data Management (MDM, Melbourne, Australia), pp. 63–77 (2003)
23. Theodoridis, Y., Silva, J.R.O., Nascimento, M.A.: On the generation of spatiotemporal datasets. In: Proceedings of the 6th Int. Symposium on Spatial Databases (Hong Kong, China), pp. 147–164 (1999)
24. Frentzos, R.: Indexing moving objects on fixed networks. In: Proceedings of the 8th International Symposium on Spatial and Temporal Databases (SSTD), pp. 289–305 (2003)
25. Pfoser, D., Jensen, C.S.: Indexing of network constrained moving objects. In: Proceedings of the 11th International Symposium on Advances in Geographic Information Systems (ACM-GIS), pp. 25–32 (2003)
26. Shababi, C., Kolahdouzan, M.R., Sharifzadeh, M.: A road network embedding technique for K-nearest neighbor search in moving objects databases. GeoInformatica **7**(3), 255–273 (2003)
27. Jensen, C.S., Kolavr, J., Pedersen, T.B., Timko, I.: Nearest neighbor queries in road networks. In: Proceedings of the llth ACM Symposium on Advances in Geographic Information Systems (ACM-GIS, New Orleans, Louisiana), pp. 1–8 (2003)
28. Papadias, D., Zhang, J., Mamoulis, N., Tao, Y.: Query processing in spatial network databases. In: Proceedings of the 29th Conference on Very Large Databases (VLDB), pp. 790–801 (2003)
29. Brinkhoff, T.: A framework for generating network-based moving objects. GeoInformatica **6**(2), 153–180 (2002)
30. Jensen, C.S., Pedersen, T.B., Speicys, L., Timko, I.: Data modeling for mobile services in the real world. In: Proceedings of the 8th International Symposium on Spatial and Temporal Databases (SSTD), pp. 1–9 (2003)
31. Scarponcini, P.: Generalized model for linear referencing in transportation. GeoInformatica **6**(1), 35–55 (2002)
32. Oracle Spatial Linear Referencing System User's Guide: Release 8.1.6., Oracle Press (2000)
33. Güting, R.H.: Second-order signature: A tool for specifying data models, query processing, and optimization. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 277–286 (1993)
34. Chon, H.D., Agrawal, D., El Abbadi, A.: FATES: Finding a time dependent shortest path. In: Proceedings of the 4th International Conference on Mobile Data Management (MDM), pp. 165–180 (2003)
35. Carey, M.J., Kossmann, D.: On saying "enough already!" in SQL. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 219–230 (1997)
36. Dieker, S., Güting, R.H.: Plug and play with query algebras: Secondo. A generic DBMS development environment. In: Proceedings of the International Database Engineering and Applications Symposium (IDEAS), pp. 380–390 (2000)
37. Güting, R.H., Behr, T., de Almeida, V.T., Ding, Z., Hoffmann, F., Spiekermann, M.: SECONDO: An extensible DBMS architecture and prototype. Fernuniversität Hagen, Informatik-Report 313, (2004)
38. Dieker, S., Güting, R.H.: Efficient handling of tuples with embedded large objects. Data & Knowledge Engineering **32**(3), 247–269 (2000)
39. Hart, P.E., Nilsson, N.J., Raphael, B.: A formal basis for the determination of minimum cost paths. IEEE Transactions on System Science and Cybernetics, SSC-**4**(2), 100–107 (1968)
40. Nilsson, N.J.: Principles of Artificial Intelligence. Springer-Verlag (1982)
41. Becker, B., Gschwind, S., Ohler, T., Seeger, B., Widmayer, P.: An asymptotically optimal multiversion B-Tree. VLDB Journal **5**(4), 264–275 (1996)
42. Cruz, I.F., Mendelzon, A.O., Wood, P.T.: A graphical query language supporting recursion. In: Proceedings ACM SIGMOD, pp. 323–330 (1987)
43. Gyssens, M., Paredaens, J., van Gucht, D.: A graph-oriented object database model. In: Proceedings ACM Conference on Principles of Database Systems (PODS), pp. 417–424 (1990)
44. Mannino, M., Shapiro, L.: Extensions to query languages for graph traversal problems. IEEE Transactions on Knowledge and Data Engineering **2**, 353–363 (1990)
45. Erwig, M., Güting, R.H.: Explicit graphs in a functional model for spatial databases. IEEE Transactions on Knowledge and Data Engineering **6**(5), 787–804 (1994)
46. Güting, R.H.: Modeling and querying graphs in databases. In: Proceedings of the 20th International Conference on Very Large Databases, pp. 297–308 (1994)
47. Shekhar, S., Kohli, A., Coyle, M.: Path computation algorithms for advanced traveller information systems. In: Proceedings of the 9th International Conference on Data Engineering (ICDE, Vienna, Austria), pp. 31–39 (1993)
48. Shekhar, S., Liu, D.R.: CCAM: A connectivity-clustered access method for networks and network computations. IEEE Transactions on Knowledge and Data Engineering **9**(1), 102–119 (1997)
49. Huang, Y.W., Jing, N., Rundensteiner, E.A.: Path queries for transportation networks: Dynamic reordering and sliding window paging techniques. In: Proceedings of the 4th ACM-GIS Conference, pp. 9–16 (1996)
50. Huang, Y.W., Jing, N., Rundensteiner, E.A.: Integrated query processing strategies for spatial path queries. In: Proceedings of the 13th International Conference on Data Engineering (ICDE, Birmingham, U.K.), pp. 477–486 (1997)
51. Vazirgiannis, M., Wolfson, O.: A spatiotemporal query language for moving objects on road networks. In: Proceedings of the 7th International Symposium on Spatial and Temporal Databases (SSTD), pp. 20–35 (2001)
52. Hage, C., Jensen, C.S., Pedersen, T.B., Speicys, L., Timko, I.: Integrated data management for mobile services in the real world. In: Proceedings of the 29th International Conference on Very Large Databases (VLDB, Berlin, Germany), pp. 1019–1030 (2003)
53. Speicys, L., Jensen, C.S., Kligys, A.: Computational data modeling for network-constrained moving objects. In: Proceedings of the llth ACM Symposium on Advances in Geographic Information Systems (ACM-GIS, New Orleans, Louisiana), pp. 118–125 (2003)
54. Ding, Z., Güting, R.H.: Modeling temporally variable transportation networks. In: Proceedings of the 9th Int. Conference on Database Systems for Advanced Applications (DASFAA, Jeju Island, Korea), pp. 154–168 (2004)
55. Ding, Z., Güting, R.H.: Managing moving objects on dynamic transportation networks. In: Proceedings of the 16th International Conference on Scientific and Statistical Database Management (SSDBM, Santorini Island, Greece), pp. 287–296 (2004)
56. Almeida, V., Güting, R.H.: Indexing the trajectories of moving objects in networks. GeoInformatica **9**(1), 33–60 (2005)