INFORMATIK
BERICHTE

364 – 05/2012

Book Chapter:
Trajectory Databases

**Ralf Hartmut Güting, Thomas Behr,
Christian Düntgen**

Fakultät für Mathematik und Informatik
Postfach 940
D-58084 Hagen

3

Trajectory Databases

Ralf Hartmut Güting, Thomas Behr, Christian Düntgen
Fernuniversität Hagen, Germany
{*rhg, thomas.behr, christian.duentgen*}*@fernuni-hagen.de*

In this chapter, we consider the problem of modeling and representing trajectories in the context of database systems. It is certainly possible to store a trajectory somehow in a standard database, e.g. as a relational table containing tuples with observations of the form (id, x, y, t). However, we are interested in a more abstract model that views a trajectory as a conceptual entity and preferably includes operations for querying. As this is not available in standard databases, we now focus on extensions of database systems that on the one hand provide a simple, adequate model of trajectories together with powerful querying facilities, and on the other hand an efficient implementation of such a query language.

Section 3.1 introduces such a data model based on abstract data types. In Section 3.2 we describe a prototypical database system, SECONDO, implementing this model. Section 3.3 discusses alternative representations of sets of trajectories in the context of this model. Important topics in the implementation are indexing and join techniques; these are addressed in Sections 3.4 and 3.5, respectively. Sections 3.6 and 3.7 give brief introductons to special querying facilities such as nearest neighbor queries and spatiotemporal pattern queries, respectively. Section 3.8 is devoted to the BerlinMOD benchmark for trajectory databases. Section 3.9 briefly introduces Hermes, another system implermenting the model of Section 3.1. The chapter closes with bibliographic notes in Section 3.10.

3.1 An Abstract Data Type Model for Trajectories

The approach taken is to consider a trajectory as an abstract data type called *moving point*. The meaning of such a type, or its *semantics*, is

given by the domain of the type (the set of possible values). The values of type *moving point* are simply functions from time into point values which could be represented as

$$f : \underline{instant} \rightarrow \underline{point}$$

where type *instant* represents a continuous domain of time and type *point* represents (x, y) positions in the Euclidean plane.[1] These are partial functions, i.e., they are defined only for a part of the time domain and may have gaps in their definition time. Values of type moving point (denoted *mpoint*) may be visualized as continuous functions in a three-dimensional (x, y, t) space as shown in Figure 3.1.

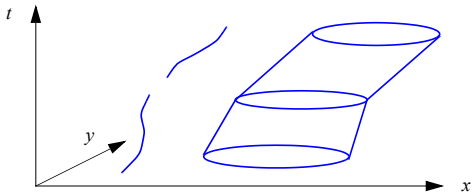

Figure 3.1 A value of type moving point (*mpoint*) and a value of type moving region (*mregion*)

We now have an abstract data type to represent time dependent locations in the plane. This is a simple and clean model to represent moving entities such as people, animals or vehicles. It is obvious that a physical entity can only be in one location at any given time; hence the model of a position as a function of time is adequate. Note that the model assumes we have complete knowledge about the movement (history) of an entity and there is no interest in representing how this knowledge was collected. In other words, this model does not care about *observations* of a moving object nor does it take uncertainty into account.

Geometrically, we have time dependent points in the plane, or a time dependent version of the spatial data type *point*. Clearly other time dependent geometries may be interesting as well, and the model also contains data types such as moving line or moving region. A moving region (*mregion*) value is a function from time into region values, also illustrated in Figure 3.1.

The data types can be embedded as attribute types into a DBMS data model, e.g. an object-relational model. The following example relation

[1] We denote data types in italics underlined font.

describes underground trains moving according to schedule on a certain day in the city of Berlin:[2]

```
Trains(Id: int, Line: int, Up: bool, Trip: mpoint)
```

Each tuple describes one train by its identifier, the number of the train line to which it belongs, in which direction along the route it was going, and the complete movement description in attribute *Trip*.

Together with the data types a suitable set of operations is provided, for example:

trajectory:	*mpoint*	→ *line*
deftime:	*mpoint*	→ *periods*
passes:	*mpoint* × *region*	→ *bool*
intersection:	*mpoint* × *mregion*	→ *mpoint*
distance:	*mpoint* × *point*	→ *mreal*
<:	*mreal* × *real*	→ *mbool*
sometimes:	*mbool*	→ *bool*
isempty:	*periods*	→ *bool*

These operations use further data types. The issue of systematically designing the type system and the operations is discussed below; here we briefly explain the example operations. **Trajectory** projects a moving point into the Euclidean plane, result is a value of the *line* spatial data type. **Deftime** projects on the time axis; result is a set of disjoint time intervals representable in type *periods*. **Passes** is a predicate checking whether the moving point ever goes through a given region. **Intersection** computes the time dependent result of the intersection between a point and a region which is a point whenever the moving point is inside the region; hence the result is an *mpoint* again. The **distance** between a moving point and a static point is a time dependent real number of a type *mreal*. The less than operator provides the time dependent comparison of two real numbers; result is a time dependent boolean value of type *mbool*. **Sometimes** returns true if the argument *mbool* ever assumes the value true, and **isempty** checks whether a *periods* value does not have any time interval.

Given such operations, we can formulate queries. Let us assume that our example database contains the following objects of the respective types:

```
train7: mpoint
msnow: mregion
```

[2] Such a relation is available within the SECONDO system described in Section 3.2 in an example database called *berlintest*

```
mehringdamm: point
tiergarten: region
```

Query 1: How many trains passed through tiergarten ?

```
select count(*) from Trains where Trip passes tiergarten
```

Query 2: Find trains that got within 500 meters of location mehringdamm.

```
select * from Trains
where sometimes(distance(Trip, mehringdamm) < 500)
```

Query 3: For each train that ever was within the moving snow area msnow, determine the locations when it was inside.

```
select Id, trajectory(intersection(Trip, msnow)) as InSnow
from Trains
where not(isempty(deftime(intersection(Trip, msnow))))
```

3.1.1 Design of Types and Operations

The basic idea of the approach should now be clear. The question arises what exactly should be the types and operations. The goal is to set up a system of types and operations that is closed, simple, and powerful. To achieve this, the following design guidelines are observed.

- Closed: under application of type constructors, in particular:
  a. For all base types of interest, there are corresponding time dependent (moving) types.
  b. Definitions of static and moving types should be consistent.
  c. For each time dependent type, there are types to represent the projection to the domain and range of the respective function.
- Simple:
  d. The type system has many types - to avoid a proliferation of operations, one should use generic operations as much as possible.
  e. The space of possible operations should be explored systematically.
  f. Operations on static and moving types should be consistent.

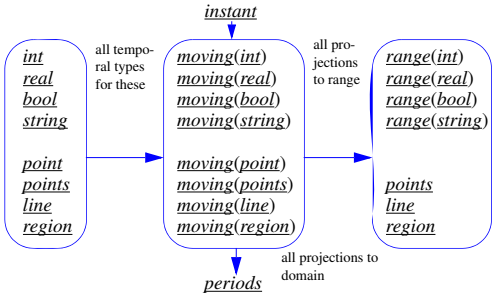

Figure 3.2 Type system

3.1.2 Type System

The type system used is shown in Figure 3.2.

It starts from the set of standard types *int*, *real*, *bool* and *string*, and spatial types *point*, *points*, *line* and *region*. A *point* value is a single point in the plane; *points* has a finite set of points. A *line* value is a finite set of continuous curves in the plane. A *region* is a closed subset of the plane; in general it consists of a finite set of faces (a face contains a connected subset of the plane) each of which may have 0 or more holes. The spatial types are illustrated in Figure 3.3.

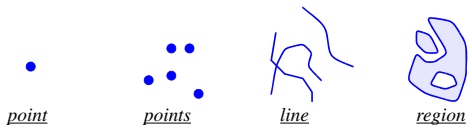

Figure 3.3 Spatial data types

The *moving* type constructor returns for a given static type α the type whose values are partial functions from the time domain into α. More formally, let A_α denote the domain of type α, i.e., the set of possible values of type α. Then the domain for type $moving(\alpha)$ is

$$A_{moving(\alpha)} := \{f | f : A_{instant} \to A_\alpha \text{ is a partial function}\}$$

One can observe that design rules (a) and (b) are fulfilled. The *range* type constructor provides for a given type α, which must have a total order, the type whose values are finite sets of intervals over the domain of α. So $range(int)$ is a set of integer intervals, $range(real)$ a set of real-valued intervals and so forth. Hence the projection into the range,

e.g. for a *moving*(*int*) can be represented by the *range*(*int*) type. This holds for all standard types. One can also show that the projections into the plane of a *moving*(*point*), *moving*(*line*), or *moving*(*region*) can be represented by the given types *points*, *line*, and *region*. The values of all types *moving*(α) (also called temporal types) can be projected on the time axis resulting in a *periods* value (*periods* is in fact another name for *range*(*instant*)). Hence design rule (c) is fulfilled.

3.1.3 Operations

The design of operations proceeds in three steps:

1. Define carefully a set of operations on the static types.
2. By a technique called lifting, make these operations time dependent.
3. Add some specific operations for the temporal types.

Step 1. For the static types, several classes of operations are introduced which are summarized in Table 3.1.

Class	Operations
Predicates	**isempty** $=, \neq$, **intersects**, **inside** $<, \leq, \geq, >$, **before**
Set Operations	**union**, **intersection**, **minus** **crossings**, **touch_points**, **common_border**
Aggregation	**min**, **max**, **avg**, **center**, **single**
Numeric	**no_components**, **size**, **perimeter**, **duration**, **length**, **area**
Distance and Direction	**distance**, **direction**
Base Type Specific	**and**, **or**, **not**

Table 3.1 *Operations on static types*

For lack of space we cannot explain all these operations here in detail, refer to the original literature (see Section 3.10). Operations are defined in a generic way ranging over all types for which they make sense.

Step 2. Lifting means to make a static operation time dependent by allowing any (combination) of its arguments to be time dependent. Obviously the result will be time dependent as well. Consider an operation **op** with signature

op: $\alpha_1 \times \alpha_2 \times ... \times \alpha_n \to \beta$

Lifting means that the signatures

op: $\underline{moving}(\alpha_1) \times \alpha_2 \times ... \times \alpha_n \to \underline{moving}(\beta)$
op: $\alpha_1 \times \underline{moving}(\alpha_2) \times ... \times \alpha_n \to \underline{moving}(\beta)$
...
op: $\underline{moving}(\alpha_1) \times \underline{moving}(\alpha_2) \times ... \times \alpha_n \to \underline{moving}(\beta)$
...
op: $\underline{moving}(\alpha_1) \times \underline{moving}(\alpha_2) \times ... \times \underline{moving}(\alpha_n) \to \underline{moving}(\beta)$

are available as well. The semantics of the static operation is transferred by requiring that the result of the lifted operation for any given instant of time is the same as that of the static operation evaluated on the arguments, evaluated at that instant. This ensures that design rule (f) is fulfilled.

For example, consider the static operations $<$ on reals and **intersection** between a point and a region. By lifting, the signatures

$<$:	$\underline{real} \times \underline{real} \to \underline{bool}$	**intersection** :	$\underline{point} \times \underline{region} \to \underline{point}$
	$\underline{real} \times \underline{mreal} \to \underline{mbool}$		$\underline{mpoint} \times \underline{region} \to \underline{mpoint}$
	$\underline{mreal} \times \underline{real} \to \underline{mbool}$		$\underline{point} \times \underline{mregion} \to \underline{mpoint}$
	$\underline{mreal} \times \underline{mreal} \to \underline{mbool}$		$\underline{mpoint} \times \underline{mregion} \to \underline{mpoint}$

are available[3] some of which occur in the introductory example. Lifting is applied to all operations of Table 3.1; hence they can all be used as time dependent operations.

Step 3. Finally, several classes of special operations on time dependent types are introduced, shown in Table 3.2.

In the first class, **deftime** and **trajectory** have already been explained. **Rangevalues** projects time dependent standard types to the respective range types, e.g. $\underline{moving}(\underline{int})$ to $\underline{range}(\underline{int})$. **Locations** provides the projection of a stepwise constant $\underline{moving}(\underline{point})$ as a $\underline{points}$ value; similarly **routes** gives the projection of a discretely changing $\underline{moving}(\underline{line})$ as a $\underline{line}$ value. In contrast, **traversed** yields the projection of a continuously changing $\underline{moving}(\underline{line})$ or $\underline{moving}(\underline{region})$ as a $\underline{region}$ value. **Inst** and **val** are explained below.

In the second class, the interaction of functions with values in their

[3] We generally abbreviate the formally defined notation $\underline{moving}(\alpha)$ by $\underline{m\alpha}$.

Class	Operations
Projection to Domain / Range	**deftime**, **rangevalues**, **locations**, **trajectory**, **routes**, **traversed**, **inst**, **val**
Interaction with Domain / Range	**atinstant**, **atperiods**, **initial**, **final**, **present** **at**, **atmin**, **atmax**, **passes**
Rate of Change	**derivative**, **speed**, **turn**, **velocity**

Table 3.2 *Operations on time dependent types*

domain (time) or the respective range is considered. **Atinstant** evaluates a *moving*(α) at a given *instant*, returning the result as a pair (i, v) with i an instant and $v \in A_\alpha$ (such pairs belong to a type *intime*(α) not yet mentioned). Operations **inst** and **val** can be used to access the components of such a pair. **Initial** and **final** return the first and last such pair for a given *moving*(α). **Atperiods** reduces a moving object to be defined only at the times of some *periods* value. **Present** is a predicate checking whether an object is (ever) defined at a given *instant* or *periods* value.

Operations **at**, **atmin**, **atmax**, **passes** handle interaction with values in the range of functions. **At** reduces a *moving*(α) to the times when its value is equal to or belongs to the value of some argument of type α.[4] **Atmin** and **atmax** reduce to the times with minimal or maximal value for a totally ordered type α. **Passes** is a predicate checking whether an argument value of a type related to α is ever assumed by a *moving*(α).

The last group addresses range of change. For example, one can get the derivative of a *moving*(*real*) or the speed of a *moving*(*point*) as a *moving*(*real*).

One can observe that operations are defined in a generic way and that the space of possible operations is structured in a somewhat systematic manner (design rules (d) and (e)).

The design of types and operations allows one to combine operations in very flexible ways so that a quite expressive query language results. More query examples are shown in later subsections.

[4] more precisely ...

3.1.4 Abstract and Discrete Model

In the model described so far, the semantics of time dependent types, i.e., of types $\underline{moving}(\alpha)$, have been simply defined as partial functions, disregarding completely the issue of how such functions can be represented. A function $f : A_{\underline{instant}} \rightarrow A_\alpha$ is simply an infinite set of pairs from $A_{\underline{instant}} \times A_\alpha$.

We call a model where it is allowed to define the semantics of types just in terms of infinite sets, an *abstract model*. An abstract model is conceptually simple and elegant, but it is not directly implementable, as computers cannot use infinite resources. To implement an abstract model, we have to provide a *discrete model* for it. In a discrete model, all the infinite sets of the abstract model have to be described in terms of finite representations.

3.1.5 The Discrete Model

The discrete model for the design above uses essentially the well known programming language representations for standard types such as *int*, *real*, *bool*, *string* and it uses linear representations for the spatial types (i.e., polylines, polygons, etc.). For the time dependent types, the so-called *sliced representation* is introduced. That means, to represent a function of time, the time domain is cut into disjoint time intervals (*slices*) such that within each slice the development can be represented by some simple function of time. "Simple" actually means finitely representable. In other words, the function for a slice can be described by a few parameters rather than an infinite set of pairs. Figure 3.4 illustrates the sliced representation for a *moving*(*real*) and a *moving*(*point*).

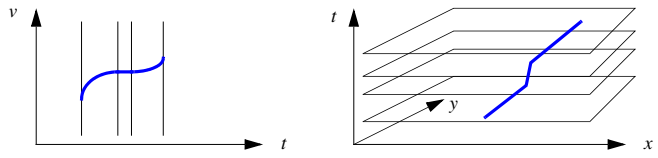

Figure 3.4 Sliced representations for *moving*(*real*) and *moving*(*point*)

The representation of a single slice, consisting of the time interval and the function description, is called a *unit*. In the discrete model it makes sense to introduce explicit data types for units, e.g. *upoint*, *ureal*, *ubool*. Such types are available in the SECONDO system described below.

The representations of functions within a slice (called *unit functions*) are chosen to support as many operations of the abstract model as possible in a consistent way. For a *moving*(*point*) a linear function of time is used, similarly for *moving*(*line*), *moving*(*region*) linearly moving boundary lines. The motivation is to have simple geometries in the 3D (x, y, t) space that can be easily handled in algorithms for the operations. For *moving*(*real*), unit functions are quadratic polynomials of time or square roots thereof. This allows one to represent the time dependent distances between moving objects, or the development of the perimeters or sizes of moving regions correctly.

3.2 Secondo: An Extensible DBMS

The types and operations described can be implemented within an extension module (data blade, cartridge, extender, etc.) to a DBMS. The model has been (partially) implemented in at least two DBMS prototypes, Hermes and Secondo. In this section we describe the Secondo environment.

3.2.1 Overview

Secondo is a DBMS prototype developed since about 1995 at University of Hagen. The focus in this project has been on an extensible and modular architecture, and support for spatial and later spatio-temporal applications. Secondo was intended to be a platform for the implementation of different DBMS data models, for example, relational, object-relational, sequence-oriented, graph- or network-oriented data models, and has the following features:

- It does not have a fixed data model. Instead, it provides a *system frame* that can be filled with implementations of different data models. A formalism called *second-order signature* provides an interface between system frame and contents.
- The system frame contains the data model independent parts of a DBMS, for example, a storage manager, a generic query processor, a generic command interface, etc.
- The data model dependent parts are implemented within so-called *algebra modules*. Each algebra module provides a collection of data types (type constructors, to be precise) and operations. Note that

algebra modules encompass all parts of a data model implementation. Hence there are algebra modules with types for relations and tuples and query processing operations such as join methods, and there are algebra modules with types for indexes such as a B-tree or R-tree with the respective search operations.

- SECONDO provides query languages at two levels, called the *descriptive* and the *executable language*.
- The descriptive language is an SQL dialect. The query optimizer translates queries in this language to the executable language.
- The executable language consists of some generic commands and the operations of all algebras available in the system. A query in the executable language is an algebra term built from database objects, constants, and operators. It is in fact a precise query plan where each operator has associated a fixed algorithm. Hence, the executable language does not have a join operator, but instead has operators like hashjoin, sortmergejoin, loopjoin and so forth.

To our knowledge it is a unique feature of Secondo that the user or application can formulate queries at the query plan level directly. Any plan that the optimizer can generate, the user can also type in. Queries at the executable level are completely type-checked before execution. This capability is crucial in providing quick extensibility, because a new query processing operator can be implemented and used directly, without a need to integrate it into SQL and query optimization. It is also highly useful for experimenting with query processing strategies and algorithms, as one has full control of composing a query plan.

Secondo consist of three major components, namely, the *kernel*, the *optimizer*, and the *graphical user interface (GUI)*. These are written in different programming languages and can run as cooperating processes. They are described in the following subsections.

3.2.2 Second-Order Signature

Second-order signature (SOS) is a framework that allows one to define a DBMS data model and query language (e.g. a relational model with relation algebra) at the descriptive level, or a representation model with query processing operations at the executable level. Currently it is used within Secondo to describe the executable level.

The basic idea is to use two coupled signatures. A signature in general has sorts and operators and defines a set of terms. Here the first signature

is used to define a type system; its sorts are called kinds and its operators are type constructors. The following signature defines a relational model.

kinds IDENT, DATA, TUPLE, REL

type constructors

	→ DATA	*int*, *real*, *bool*, *string*
(IDENT ×DATA)+	→ TUPLE	*tuple*
TUPLE	→ REL	*rel*

A kind is essentially a collection of types. Here *int*, *real*, *bool*, *string* are constant type constructors (i.e., type constructors without arguments) with result types in the kind DATA. The *tuple* constructor takes a non-empty list of pairs where the first element is an identifier (an attribute name) and the second a type in DATA; it returns a tuple type. The *rel* constructor takes a tuple type (from kind TUPLE) and returns a corresponding relation type. The terms of this signature are precisely the available types of the type system. The term

rel(*tuple*([Name: *string*, Age: *int*]))

belongs to kind REL and denotes a relation type, equivalent to a relation schema in standard database terminology.

The second signature introduces operations over the types defined by the first signature. At the executable level, these are query processing operations. The following are a few example operations.

operators

∀ *data* in DATA.			
data ×*data*	→ *bool*	=, <, >	`_ # _`
∀ *tuple*: *tuple*(*list*) in TUPLE, *attrname* in IDENT, *member*((*attrname*, *attrtype*), *list*).			
tuple ×*attrname*	→ *attrtype*	**attr**	`# ( _ , _ )`
∀ *tuple*: *tuple*(*list*) in TUPLE.			
rel(*tuple*)	→ *stream*(*tuple*)	**feed**	`_ #`
stream(*tuple*) ×(*tuple* → *bool*)	→*stream*(*tuple*)	**filter**	`_ # [ _ ]`
stream(*tuple*)	→*rel*(*tuple*)	**consume**	`_ #`

Here we use quantification over kinds to define generic operator signatures. Hence *data* is a type variable ranging over types in kind DATA, and three comparison operators are defined to be applicable to equal types, e.g. <: *int* × *int* → *bool*. **Attr** is an operator to access an attribute value within a tuple. At the executable level, query processing operators usually work in a pipelined fashion; this is provided by a special type constructor *stream*. Operator **feed** scans a relation and provides a stream of tuples of the given tuple type of the relation. Operator **filter**

implements a selection on a stream of tuples, checking each tuple with the predicate given as a second argument. Operator **consume** collects a stream of tuples back into a relation. The notations on the right define the operator syntax, where `#` represents the operator and `_` an argument.

With the given operators, one can formulate the following simple query plan. Let us assume a relation *People* with the type/schema shown above exists in the database, and *Person* is the type of tuples of *People*.

```
People feed filter[fun(t: Person) attr(t, Age) > 20] consume
```

All the mentioned operators are in fact implemented in SECONDO and one can type this plan exactly in the form shown. Moreover, some abbreviations are provided so that one can write:

```
People feed filter[.Age > 20] consume
```

The complete query with the function argument shown above is derived from this user query.

The SOS framework includes a small set of generic (i.e., data model independent) commands for data definition and data manipulation. A database is a pair (T, O) where T is a set of named types and O a set of named objects. The seven basic commands shown in Table 3.4 are available. Here *type expression* denotes any term built from database types and type constructors of the available algebras, and *value expression* any term built from database objects (or constants) and operators of the available algebras.

3.2.3 Secondo Kernel

The kernel implements specific data models and is extensible by algebra modules. It provides query processing over the implemented algebras. It uses an underlying storage manager (BerkeleyDB) to provide stable storage at the level of files and records, including transaction management, locking and recovery. The kernel is written in C++.

Secondo implements the generic commands of the SOS framework and a few more, shown in Table 3.3.

Note that any implemented type constructor must provide methods to convert between the internal representation of a value of the type and an external nested list format (similar to lists in Lisp or Prolog). This makes it possible to have generic commands to save any object, or an entire database, into an editable text file format. This format is also used for exchanging data between Secondo components, e.g. the kernel and the GUI.

Basic Commands	Inquiries
`type <ident> = <type expression>`	`list type constructors`
`delete type <ident>`	`list operators`
`create <ident>: <type expression>`	`list algebras`
`update <ident> := <value expression>`	`list algebra <ident>`
`let <ident> = <value expression>`	`list databases`
`delete <ident>`	`list types`
`query <value expression>`	`list objects`
Databases	**Transactions**
`create database <ident>`	`begin transaction`
`delete database <ident>`	`commit transaction`
`open database <ident>`	`abort transaction`
`close database`	
Import and Export	
`save database to <file>`	
`restore database <ident> from <file>`	
`save <ident> to <file>`	
`restore <ident> from <file>`	

Table 3.3 *Secondo kernel generic commands*

3.2.4 Query Optimizer

The query optimizer is a large and important component of the system, but we do not have space in this chapter to present it in any detail. We can just mention a few properties:

- The optimizer performs conjunctive query optimization, i.e., takes a set of relations and a set of selection or join predicates, and computes an execution plan for them in the Secondo executable language. This initial plan yields a stream of tuples to which further query processing operations are applied according to the SQL query (e.g. for grouping, aggregation, projection).
- The optimizer considers all possible plans according to the given rules for translating selections and joins. The complexity for this is exponential, but it works fine (i.e., within a second) for up to about 9 or 10 predicates. The resulting plan is optimal if cost functions and selectivity estimates are precise, and predicates are not correlated.
- The operations of the implemented algebras on attribute types (atomic operations from the point of view of the relational model) can be used

directly in the WHERE-clause and the SELECT-clause. All operations of the model of Section 3.1 are atomic in this sense.

- The optimizer determines selectivities of predicates by evaluating each predicate individually on a materialized sample, or two samples for join predicates. Note that this is the only technique that works in a complex application context as this one, where histogram support for all operations is impossible to obtain.
- The optimizer is extensible by translation rules (specifying how selection or join predicates can be implemented by executable operations, including index accesses) and by cost functions for query processing operations.

3.2.5 Executable Language vs. SQL

There are two language levels available for querying in Secondo; hence a discussion is appropriate what are the advantages or disadvantages of using either of these levels.

Using SQL and the optimizer has the following advantages:

- The structure of SQL is well known and the query is easy to formulate if just atomic operations of the data model are needed.
- The user does not need to learn or remember individual query processing operations such as specialized join methods or index access methods.
- For a complex query, there are many alternatives and the optimizer can make a good choice based on cost estimation. In contrast, the user can only guess or rely on experience to construct a good plan.

However, using the executable level directly also has some advantages:

- The SQL level is restricted to the relational model and the structure of an SQL query is quite rigid. Any extension to the SQL language has to happen in an ad-hoc manner (e.g. adding new keywords, clauses). In contrast, the executable level is entirely free in choosing the data model. Currently there are, for example, nested relations and network structures that go beyond the relational model, also streams of values, arrays of relations (suitable for parallel processing) and so forth.
- Similarly, query processing operations can be chosen and added freely. For example, there are Secondo operations to convert a stream of observations into a moving(point), to compute derived attributes from an expression over attributes in two adjacent tuples in a tuple stream,

to import data from Shape files, to send a stream of tuples to another computer via TCP/IP, to read/write tuple streams from/to files, etc. Some example queries involving special operations for continuous nearest neighbor queries are shown in Section 3.6. It is completely unclear how such queries could be expressed in SQL.

- In many application cases, a query for the analysis of moving objects involves some complex processing steps, but it is not complex in terms of a large number of joins. In such cases, the advantages of using the query optimizer disappear.
- System development generally proceeds from the executable level to the level of SQL and query optimization. One can quickly add a query processing operator to the system and use it, much faster than figuring out how SQL extensions, optimization rules, and cost estimates can be done. As soon as the executable operator is available, one can already run the query and solve the application problem.

In summary, using the executable level is harder in formulating the query, but at the same time much more flexible and expressive than using the SQL level. In fact, the executable level is a kind of specialized programming language, intermediate in complexity between a genuine programming language such as C++, and SQL.

3.2.6 The Graphical User Interface

SECONDO comes with a graphical user interface. It is written in Java and communicates with the SECONDO-Kernel via TCP/IP. Thereby, the SECONDO server and the user interface can be installed on different computers.

In Figure 3.5, a screenshot of the GUI is shown. It consists of three main components, the command panel (top left), the object manager (top right), and the currently active viewer (bottom; here, the HoeseViewer). The command panel gets commands from the user and executes them. The commands can be divided into three groups: SECONDO commands, GUI commands, and SQL commands. A GUI command controls the GUI. For example, a new viewer can be added at runtime using the command "gui addViewer <Name>". An SQL command is sent to the optimizer server. The server returns the command translated to executable level. This command is treated as if the user entered the command directly as a SECONDO command. A SECONDO command is sent to the SECONDO server. The server returns the result as a nested

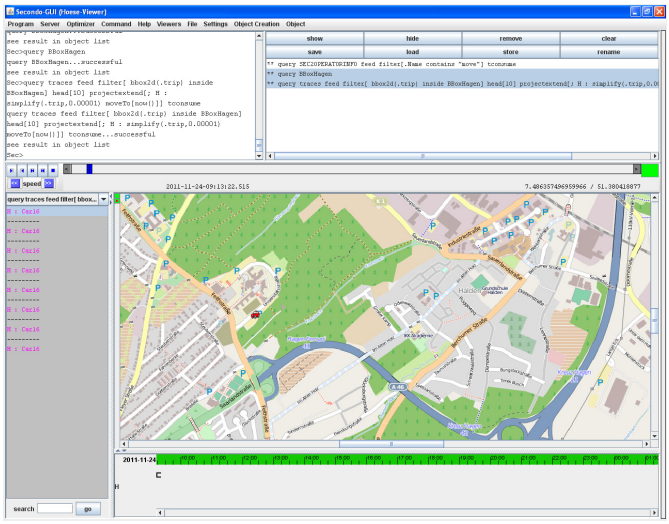

Figure 3.5 GUI screenshot

list. The list is wrapped into a so-called SecondoObject, consisting of an ID, a name (currently the command itself), and a nested list describing the value of the object. This object is given to the object manager for further processing. The object manager in turn, first adds the object to the list of available objects. After that, a viewer is selected following certain rules and the object is passed on to this viewer.

SECONDO can be extended with new data types. Because it is unpredictable which extensions are implemented in the future, the GUI must also be extensible. This is realized by so-called viewers. Each viewer is a component which can handle a certain set of data types. A type may be handled by more than one viewer. Some of the currently available viewers are themselves extensible.

The HoeseViewer

The currently most powerful viewer is the *HoeseViewer*, named by its first author. This viewer is able to display simple objects (*int*, *string*, ...), spatial objects (*point*, *line*, *region*), and spatio-temporal objects. All objects can also be embedded in a relation. Spatio-temporal objects are displayed as an animation. If an animation is started, a ticker is activated. At each tick, each object's shape and position at the instant

to display are called and then plotted. Other temporal objects (*mbool*, *mreal*) are displayed as 2D function graphs within a separate frame.

The appearance of objects is controlled by so-called categories, for example, the color, line width and label can be configured. Some of the parameters can be set to be dependent on an attribute value. Thus, for example, the color of a moving point can represent its speed stored in another attribute.

If spatial objects are given in geographic coordinates (longitude/latitude), maps can be used as a background. The HoeseViewer provides a set of preconfigured map servers but can also use other servers.

3.3 Representations for Sets of Trajectories

Storing and analyzing trajectories relies on methods to represent trajectory data within a database. In this section, we show how trajectory data can be loaded and represented in SECONDO. The DB commands are presented in the SECONDO executable language.

In Section 3.1 the *Trains* example relation was introduced with schema:

```
Trains(Id: int, Line: int, Up: bool, Trip: mpoint)
```

Let us assume we do not rely on time tables anymore and now capture the real positions of metro trains in Berlin. The vehicles are equipped with sensors and periodically send messages

```
(Id: int, Line: int, Up: bool, Time: instant,
  PosX: real, PosY: real)
```

to a DB server. Besides the attributes explained above, *Time* is the message's timestamp, and the geographic position at the associated timestamp is given as Gauss-Krueger projected geocoordinates: *PosX* is the easting, *PosY* the northing coordinate. Each incoming message is inserted into a CSV table

```
TrajCSV(Id: int, Line: int, Up: bool, Time: instant,
  PosX: real, PosY: real)
```

Due to the natural injectivity of the mapping of time to a *Line* number, an *Up* flag setting and a *Position* for each physical object, $\{Id, Time\}$ is a key for $TrajCSV$.

3.3.1 Loading Data

First, we show how the "raw" trajectory data from a given CSV text file *Traj.csv* can be imported to a SECONDO database.[5] The text file provides the five column table *TrajCSV* as described before.

```
let TrainsRaw = [const rel(tuple([Id: int, Line: int, Up: bool,
    Time: instant, PosX: real, PosY: real])) value ()]
  csvimport['Traj.csv', 0, "", ","]
  projectextend[Id, Line, Up, Time; Pos: makepoint(.PosX, .PosY)]
  consume;
```

This creates a relation

```
TrainsRaw(Id: int, Line: int, Up: bool, Time: instant, Pos: point)
```

where attribute *Pos* contains the position data as a *point* (easting, northing). In the following, we briefly investigate two different ways to represent trajectories more effectively according to the data model of Section 3.1 within SECONDO: the *compact representation* and the *unit representation*.

3.3.2 Compact Representation

In *TrainsRaw* the information on a vehicle is distributed among many tuples. Using the model of spatio-temporal data types, we now express the same data in a relation with only a single tuple per vehicle. The data type *mpoint* is used to capture the temporal development of attribute *Pos*. We achieve this by grouping *TrainsRaw* by *Id* and applying the **approximate** operator to each group. Using *Time* as the least significant sorting criterion prior to grouping guarantees that the messages for each train enter the **approximate** operator in increasing temporal order:

```
let Trains = TrainsRaw feed
  sortby[Id, Line, Up, Time]
  groupby[Id, Line, Up; Trip: group feed approximate[Time, Pos] ]
  consume;
```

The result is the relation *Trains* with the schema shown earlier. This is what we call the *compact representation* of moving object data. It is easy to apply many different kinds of temporal and spatio-temporal operations as introduced in Section 3.1 to the temporal attribute.

[5] It is also possible to import NMEA recordings using an operator **mneaimport**.

3.3.3 Unit Representation

A second way to represent moving object data is to employ the respective unit types. Several operations allow one to transform a moving type object to a stream of unit type objects and vice versa, so it is easy to translate between say an *mpoint* and a set of *upoint*s and to use both kinds of data types together. A *upoint* represents a single time interval and a linear movement of a single object during this time. Let us create the unit representation for relation $Trains$:

```
let UnitTrains = Trains feed
  projectextendstream[Id, Line, Up; UTrip: units(.Trip)]
  addcounter[No, 0] consume;
```

The result is a relation

```
UnitTrains(Id: int, Line: int, Up: bool, UTrip: upoint, No: int)
```

For each vehicle identifier, $UnitTrains$ contains a set of tuples, each of which contains one of the temporally disjoint units whose union forms the train's complete trajectory:

The **units** operator converts each *mpoint* to a stream of *upoint*s, and **projectextendstream** performs a specialized loopjoin. The **addcounter** operator extends the tuples with a counter attribute called No (an *int* key, starting with value 0 for the first tuple, then being incremented for each following tuple). Because this *unit representation*, as we call it, replicates attributes Id, $Line$ and Up, it is less space efficient. However, it has a higher degree of organisation than $TrainsRaw$, and will show quite useful when creating indexes supporting certain query types (like temporal/spatial/spatio-temporal window queries and nearest neighbor queries, see Sections 3.4 and 3.6).

3.3.4 Object-Based vs. Trip-Based Representation

If the complete trajectory data on an object (using any one of the three representations) can be referenced within a representation, we call this *object-based*, or a *raw trajectory*. Often, such a raw trajectory is decomposed into several "meaningful" parts — *semantic trajectories* — like a series of trips. In Secondo, the **sim_trips** operator identifies the boundaries of such parts — other suitable methods (not shown here) may divide raw trajectories into other forms of *semantic trajectories*. Given a full trajectory (an *mpoint*) and a minimum pause duration, it splits the *mpoint* into a series of trips. As an example, we decompose

the trajectories *Trip* within relation *Trains*. Whenever a train stops for at least 10 seconds (10,000 milliseconds), its trajectory is split:

```
let TrainTrips = Trains feed
  projectextendstream[Id, Line, Up; PartialTrip: .Trip
    sim_trips[create_duration(0, 10000)]]
  addcounter[PartialTripId, 0]
  consume;
```

The result is a relation

```
TrainTrips(Id: int, Line: int, Up: bool, PartialTrip: mpoint,
  PartialTripId: int).
```

We call it a *trip-based representation*, because it allows one to refer to single trips of vehicles (using the added key attribute *PartialTripId*) rather than to a vehicle's complete trajectory.

3.4 Indexing

SECONDO contains several algebra modules providing indexes for different purposes. For indexing simple data types, B-trees or hash tables are used while for spatial and spatio-temporal objects, R-trees are applied. A variant, the TB-tree, is also available in SECONDO. For indexing current positions of moving points, the SETI-structure has been implemented. For indexing moving objects in networks, the MON-Tree-Algebra provides appropriate data structures. Further implemented index structures are the X-tree and the M-tree. Whereas a B-tree-search yields the final result directly, spatial or spatio-temporal indexes (like the R-tree) return a candidate set and then the real intersection must be checked in a further step[6]. The candidate set contains all results fulfilling the condition (true hits) but may contain also elements where the condition fails (false hits).

When indexing moving points by R-trees, different granularities can be chosen. The roughest one is to index the *mpoint* as a whole. If an *mpoint* was observed over a long period, its bounding box may be very large, leading to a lot of dead space within the index. The index will contain only a few entries, but its selectivity is bad; this means the resulting candidate set will contain a lot of false hits. The other extreme is to index single units of the *mpoint*. Here, compared with indexing of the whole *mpoint*, less dead space is produced. But the complete *mpoint*

[6] This is the so-called filter-and-refine strategy known from the literature

is distributed over many index entries. Thus, depending on the query, duplicates must be removed or the result entries must be merged. A third way is indexing groups of connected units. All three possibilities are available in SECONDO.

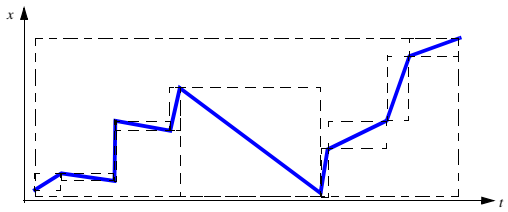

Figure 3.6 Indexing whole *mpoints* and *upoints*

Another differentiation can be made in the indexed dimensions. It is possible to use separate indexes for the spatial and for the time dimension or to use a 3D-R-tree to index the three dimensions together. The former is suitable for pure spatial and temporal searches, respectively. For example, the spatial index is the best choice if asking which moving points have passed a certain area. If another condition restricts the time when the *mpoint* passed the area, an index over all dimensions is useful.

In SECONDO, an R-tree can be created using the operators **creatertree** and **bulkloadrtree**, respectively. Both variants get a tuple stream extended by the tuple's id and the name of the attribute to index. For the *bulkloadrtree* operator, the tuples must first be sorted by the bounding box according to the z-order, but this operator is much faster than *creatertree*. For example,

```
let UnitTrains_UTrip = UnitTrains feed addid
  extend[B: bbox2d(.UTrip)] sortby[B] bulkloadrtree[B]
```

creates an index over the spatial dimension of the units of underground trains stored in the UTrip attribute of the relation UnitTrains. By typing the query:

```
query UnitTrains_UTrip UnitTrains
  windowintersects[bbox(thecenter)]
  filter[trajectory(.UTrip) intersects thecenter] consume
```

we can determine which units of the relation UnitsTrains intersect the region named thecenter.

3.5 Spatial Join

The operator **spatialjoin** takes two tuple streams each having a spatial or spatio-temporal attribute a_1 and a_2, respectively. It joins such tuples of the streams where the bounding boxes of a_1 and a_2 intersect.

The operator works in two main steps. In the first step, the tuples are distributed over a regular grid according to the attribute. More precisely, each tuple is inserted into all grid cells intersected by the attribute. The grid is chosen in such way that within each cell, the number of objects is small enough to handle all objects in main memory. After that, an R-tree based spatial join is performed for each cell. Here, first a main memory R-tree for the elements a_1 of the first tuple stream is built. Then, the second tuple stream is scanned and, for each tuple, a search on the R-tree is performed. Matching tuples are concatenated and returned.

Some tuples may have intersecting boxes in different cells. To avoid creating duplicate answers, a test is made whether the smallest intersection point of the two overlapping boxes actually lies in the current cell. In this way duplicates can be eliminated without an expensive sorting step.

The operator obviously needs a definition of the grid, i.e., the total space covered by the grid, and the cell dimensions. To determine this, before performing the two steps mentioned above, the operator in a first pass reads both streams completely, buffering them in memory if possible and otherwise writing them to disk. The cell size is then determined based on the total number of objects and the average object sizes.

A usage example of the spatialjoin operator is:

```
query UnitTrains feed extend[B: bbox(.UTrip)] a
  UnitTrains feed extend[B: bbox(.UTrip)] b
  spatialjoin[B_a, B_b]
  filter[.Id_a < .Id_b]
  filter[intersection(.UTrip_a, .UTrip_b) count >0]
  consume
```

With this query, we find all pairs of units from $UnitsTrains$ meeting each other. In a first step, we feed the relation $UnitTrains$ into two tuple streams. We extend both streams by the 3D-bounding box of the unit. To avoid name conflicts, attributes are renamed by appending $\_a$ and $\_b$, respectively, to the name of each attribute. Then, the **spatialjoin** operator concatenates the pairs of tuples having intersecting bounding boxes as the candidate set. From this stream, these tuples are selected

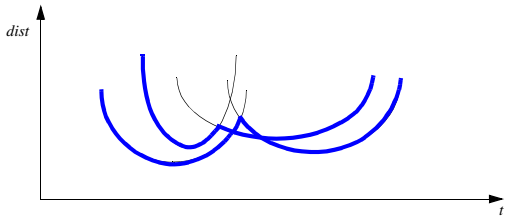

Figure 3.7 Two nearest neighbors

whose *upoint* values actually meet. The resulting stream is collected into a relation.

Because each cell can be handled separately, this operation can be distributed to several computers for parallel processing.

3.6 Nearest Neighbor Queries

Finding the k nearest neighbors to a query object is an important function. For example, a car driver may request the next three gas stations from his current position. As the car is moving, the result set will change in time.

Nearest neighbor queries are well studied for (static) spatial objects. Usually, a distance scan on an R-tree is performed to compute the result.

Here, we solve the most complex case, which is finding the k nearest neighbors to an *mpoint* q within a set of *mpoints*. It is obvious that the result set also may change in time. The complete algorithm is wrapped in an operator **knearest**, which gets an *mpoint* and receives a stream of tuples containing the units of the objects among which the nearest neighbors are to be be found. These tuples must arrive ordered by starting time of the units. Besides this, the attribute name of the unit attribute and the number k is given. Now, a plane sweep is performed to find intersections of distance functions from the units to the query *mpoint*. As we only want to compare the distances, the computation of the square root to get the exact distance is unnecessary and therefore omitted. For this reason, the distance functions are polynomials of degree two. Figure 3.7 shows the distance functions of four units and the two nearest neighbors.

The plane sweep algorithm is a variant of the well-known line segment intersection algorithm by Bentley and Ottmann. Due to space limita-

tions, further details are omitted here. The **knearest** operator returns all (parts of) units belonging to the k moving points closest to p in its output stream.

An example query for the operator is:

```
[htbp]
query UnitTrains feed sortby[UTrip] knearest[UTrip, train7, 3] consume
```

This will return the three nearest neighbors to train7 during the complete lifetime of train7 as a relation containing *upoints*.

The computation of the nearest neighbors is complex and if many objects are inside the query set, a lot of distance computations must be performed. To reduce the complexity, another operator is implemented in SECONDO acting as a filter for the **knearest** operator. It removes such tuples from the input stream which cannot contribute to the final result. After that, the **knearest**-operator is applied on the reduced data set.

The basic idea is to use an R-tree over the units to prune units far away from the query moving point q. In the following, we give a sketch of the algorithm. We store the units into a 3D-R-tree. We can prune a node p in the R-tree, if for the time interval i covered by this node, there are other nodes N closer than p to the query object containing more than k units. For example in Figure 3.8, the node p can be pruned for $k = 2$ because the node n is closer to q and has more than two units during the complete lifespan of p.

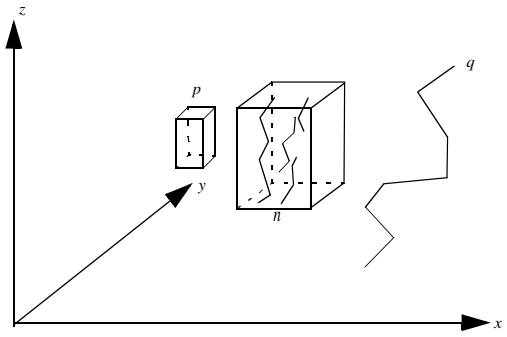

Figure 3.8 Prune a node

The closeness condition is fulfilled, if the minimum distance from p to q is greater than the maximum distance of n (from N) to q. To find out whether the set N contains at least k units during i, another

precomputed relation is used. In principle, this relation contains for each node of the R-tree a moving integer describing the number of units present at each instant of time. We call this number *coverage number*. Unfortunately, this number may have many units. The condition that enough units must be present must hold for the complete time interval covered by the node. If we would use the exact numbers, this check would be expensive due to the large numbers of units. For this reason, we coarsen the coverage number to have a maximum of three units. The coarsened number value is at most the original one for all instants to avoid to pruning nodes still needed.

The tree is then traversed, using the precomputed coverage numbers to prune irrelevant subtrees. More details can be found in the original literature.

An example query for this operation is:

```
query UTOrdered_rtree UTOrdered Numbers_btree Numbers
  knearestfilter[UTrip, train7, 3] knearest[UTrip, train7, 3]
  consume
```

Here, *UTOrdered* corresponds to *UnitTrains* sorted by *UTrip* and *Numbers* contains the coverage numbers for the nodes of *UTOrdered_rtree*, an R-tree built over this relation. *Numbers_btree* is a B-tree index created over the *Numbers* relation to speed up finding the coverage number for a certain node.

3.7 Spatiotemporal Pattern Queries

SECONDO recently provides a querying facility to find trajectories matching so-called *spatio-temporal pattern* predicates. Such a predicate specifies several time dependent conditions together with restrictions on the fulfillment time intervals of these conditions. Time dependent predicates may be, for example:

- The speed is higher than 120 km/h.
- The car is in a forest
- The altitude of an air plane is less than 500 meters.

For time dependent conditions A, B, C, constraints on fulfillment times may be:

- A then B then C
- A before B; A and B during C

In the model of Section 3.1, time dependent conditions are expressed as lifted predicates, i.e., predicates evaluating to *mbool*. Many such predicates can be expressed in the SECONDO implementation. A spatio-temporal pattern query can be expressed using an **stpattern** operator. For example:

```
query Trains feed filter[
    . stpattern[
      insnow: .Trip inside msnow,
      isclose: distance(.Trip, mehringdamm) < 10.0;
      stconstraint("insnow","isclose", vec("aabb")) ]]
  consume
```

This query finds trains that are in the snow area *msnow* before their distance to *mehringdamm* is smaller than 10 meters. The time interval constraint is expressed by a kind of graphical notation ("aabb" expresses that time interval a precedes time interval b).

A more detailed explanation of spatio-temporal pattern predicates can be found in Chapter ... on Air Traffic Analysis where it is employed to find flights exhibiting particular behaviours.

3.8 BerlinMOD

BerlinMOD is a benchmark for trajectory database systems. It comprizes a parametrizable data generator and a set of queries as a well-defined workload. Benchmarks are an accepted method to compare the efficiency of database systems. More than this, they are helpful to users and researchers in

1. evaluating a database system's overall performance
2. identifying a system's specific advantages and disadvantages
3. testing its optimization strategies
4. providing a standardized testing platform for new algorithms and index structures
5. comparing different trajectory/data representations

3.8.1 Data Generator

Benchmarking requires a well-defined dataset to which queries can be applied. One basic problem with trajectory databases is, that while large datasets are continually collected all over the world, few of them are made publicly available, usually due to economic interests and privacy

concerns. To tackle this problem, data generators have long been used to create synthetic data. However, there are more considerations regarding data for a benchmark:

1. The data must be scalable with regards to the number of observed objects and the observation time. While real data can only be restricted, synthetic data may be generated in accordance with the user's requirements.
2. The data should be representative, i.e. it should either be real or at least resemble the properties of real data, so that results can be transferred to typical real-world use cases.
3. The data set should be well-defined, so that experiments using it are repeatable. If real data is used, it must be made available in total, if a data generator is used, it should create the same data each time the same parameters are used.
4. Trajectory data should be augmented with standard data (e.g. *int*, *string*, *bool* etc.).
5. When using a data generator, it should be widely parametrizable, since many people are interested in using data adapted to some concrete situation, e.g. trajectories within their own city.

Respecting these points, generators can be an accepted alternative or supplement to native data collections. The BerlinMOD data generator addresses these needs. It uses real world parameters — the road network of Berlin and statistical data on local population and employer distributions — together with a well-defined algorithm to simulate the car trips of an arbitrary number of "persons". Each person is assigned a home location and a work location using distributions defined by the road network or the statistical data. Some additional standard type attributes (licence plate number, car manufacturer and car type) are associated with each person. The simulator then creates trips between these pairs of locations (commuting trips) on each working day. For weekends and in the person's leisure time, additional round trips starting at the home location and visiting several intermediate locations on the network are created. These round trips tend to be local, so that visited locations are more likely in the vicinity of a person's home. The simulation respects the geometry of the roads and the network (speed limits, curve angles, different types of crossings) and adds according acceleration, deceleration and stop events. Interactions between simulated persons are neglected, since otherwise the goal to achieve scalability would be endangered.

Besides the trajectory data, also sets of points, regions, observation periods, instants, and person's identities are created. These datasets are used by the workload queries to define query points and windows. Usually, the generated data are mapped to the road network, but optionally, noise can be added to the position data. This simulates small random errors, which are typical for positioning devices.

The following relations representing the generated trajectories are created:

```
dataScar(Licence: string, Model: string, Type: string,
  Trip: mpoint)
dataMcar(Licence: string, Model: string, Type: string)
dataMtrip(Licence: string, Trip: mpoint)
```

dataScar is a relation with one tuple per created person, using the compact representation and the object-based approach. The other two relations represent the same data, also using the compact representation, but following the trip based approach, i.e. *dataMcar* contains one tuple with the standard attributes for each person, while *dataMtrip* contains several tuples with a trip for each person. $\{Licence\}$ is a key for *dataScar* and *dataMcar* — and an external key of *dataMtrip*.

```
QueryPoints(Id: int, Pos: point)
QueryRegions(Id: int, Region: region)
QueryInstants(Id: int, Instant: instant)
QueryPeriods(Id: int, Period: periods)
QueryLicences(Id: int, Licence: string)
```

These five relations provide data to generate meaningful queries.

3.8.2 Workload

Besides the data generator, BerlinMOD comes with a workload of 16 different types of window queries (BerlinMOD/R), plus 9 different types of nearest neighbor queries (BerlinMOD/NN). The queries operate on the relations listed before. For BerlinMOD/R, the queries are provided in SQL (augmented with the operations for spatio-temporal data types). For the BerlinMOD/NN queries, formalized SQL statements are not yet available, and we will focus on BerlinMOD/R in the remainder of this section.

The benchmark queries have been carefully selected in order to cover all significant types of queries. There are queries with and without aggregations, point- and various (temporal, spatial, different kinds of spatio-

temporal) window queries, but also queries regarding the standard type attributes.

As an example, we show a slight variation of Query 10: "When and where did the vehicles with licence plate numbers from *QueryLicences* meet other vehicles (distance *lt* 3m) and what are the latters' licences?":

```
SELECT V1.Licence AS QueryLicence, V2.Licence AS OtherLicence,
  (V1.Trip atperiods (deftime(
    (distance(V1.Trip, V2.Trip) <= 3.0) at TRUE))) AS Pos
FROM dataScar V1, dataScar V2, QueryLicences LL
WHERE V1.Licence = LL.Licence
  AND V2.Licence <> V1.Licence
  AND sometimes(distance(V1.Trip, V2.Trip) <= 3.0);
```

In order to achieve a good performance for queries, index support is required for selections, especially on the spatial and/or temporal dimension (at scale factor 1.0, BerlinMOD uses 19 GB of data, representing observations of 2,000 persons over 28 days). It is in the responsibility of the benchmark user to provide required indexes. He or she can then translate the queries into the data base system's executable language — either by himself or herself or using the system's query optimizer.

For example, given that a B-tree index *dataScar_Licence_btree* indexing *dataScar* by *Licence* is available, a valid manual translation of Query 10 into a SECONDO executable query plan is:

```
query QueryLicences feed head[10]
  loopsel[ dataScar_Licence_btree dataScar exactmatch[.Licence]
    project[Licence, Trip] V1
  ]
  dataSCcar feed project[Licence, Trip] V2
  symmjoin[.Licence_V1 # ..Licence_V2]
  filter[(everNearerThan(.Trip_V1, .Trip_V2, 3.0))]
  projectextend[; QueryLicence: .Licence_V1,
    OtherLicence: .Licence_V2,
    Pos: .Trip_V1 atperiods
deftime((distance(.Trip_V1, .Trip_V2) <= 3.0) at TRUE)
  ]
  filter[not(isempty(deftime(.Pos)))]
  project[QueryLicence, OtherLicence, Pos]
  consume;
```

The meaning of this plan is that first the trajectories belonging to vehicles listed in *QueryLicences* are selected using the B-tree, then the result is joined with all other vehicles (*.Licence_V1 # ..Licence_V2*, since {Licence} is a key for *dataScar*). The result is filtered using a predicate with operator **everNearerThan** which performs a parallel scan over both of its *mpoint* arguments and immediately returns TRUE

when both *mpoint*s ever get closer to each other than the given threshold. Passing pairs are extended with an *mpoint* attribute Pos, which contains the restriction of the query vehicle's trajectory ($Trip\_V1$) to the *periods* (and hence also to the locations) where it is nearer than 3 meters to its join partner's trajectory ($Trip\_V2$). The last **filter** makes sure that tuples with empty instances of Pos are rejected.

The SECONDO optimizer accepts SQL queries in a syntax slightly adapted to PROLOG:

```
select [v1:licence as querylicence, v2:licence as otherlicence,
  v1:trip atperiods(deftime(
    (distance(v1:trip,v2:trip) <= 3.0) at testtrue)) as pos]
from [datasccar as v1, datasccar as v2, querylicences as ll]
where [v1:licence = ll:licence, not(v2:licence = v1:licence),
  sometimes(distance(v1:trip, v2:trip) <= 3.0)]
```

The optimizers recommended plan is:

```
query dataScar feedproject[Trip, Licence] v2
  QueryLicences feedproject[Licence] ll
  loopjoin[dataScar_Licence_btree dataScar exactmatch[.Licence_ll]
    project[Trip, Licence] v1 ]
  symmjoin[not((.Licence_v2 = ..Licence_v1))]
  filter[sometimes((distance(.Trip_v1,.Trip_v2) <= 3.0))]
  extend[Querylicence: .Licence_v1, Otherlicence: .Licence_v2,
    Pos: (.Trip_v1 atperiods
      deftime(((distance(.Trip_v1,.Trip_v2) <= 3.0) at testtrue)))]
  project[Querylicence, Otherlicence, Pos]
  consume;
```

This is — neglecting the reversed, insignificant order of arguments to the **symmjoin** — almost the same as the man-made plan. To be honest, the recommendation is the result after applying the BerlinMOD benchmark to SECONDO and enhancing the optimizer's cost functions. The first plan did not perfom an unequality join on *Licence*, but directly applied the distance criterion when joining the trajectories. Scanning pairs of *mpoint*s is likely to be more expensive than testing two *string*s for equality. Investigations in the optimizer resulted in a refinement of the cost function for the **symmjoin** and allowed to generate better plans. But there is still some potential for further optimization, since the optimizer could be instructed to also use operator **everNearerThan**(x, y, z) instead of the expression **sometimes**(**distance**$(x, y) \leq z$), which is forced to always (1) compute the complete distance function (an *mreal*) and from this (2) the lifted comparison (an *mbool*). This is only one simple example on how using a benchmark may help to improve a database system.

3.9 Hermes

Another system dealing with moving objects is *Hermes* [13]. It is implemented on top of the Oracle 10g database system using PL/SQL as a programming language. Beside the core system of Hermes, there is an implementation of a web based query builder and viewer. Hermes does not implement own data structures for spatial objects, rather it uses the spatial objects of the underlying system.

Because Hermes implements the same data model as SECONDO does, the data types and operations on them are quite similar. Additionally to the types provided by SECONDO, Hermes has implementations for moving circles, moving rectangles, and moving collections (sets of moving objects of different types).

As SECONDO, Hermes uses the sliced representation for representing moving objects. Units belonging to a moving object are stored within a nested table.

Besides the moving data types, Hermes contains a TB-tree implementation. This structure supports the standard operations for this index (point query and range query), but also k-NN and similarity queries.

Hermes' query language is SQL extend by spatio-temporal operations. Although SQL is familiar to the most database systems users, formulating complex temporal queries in SQL is a hard task and queries tend to degenerate to deeply nested function calls.

3.10 Bibliographic Notes

The data model of Section 3.1 was developed in a series of papers [4, 8, 5, 10]. Whereas [4] describes the approach, in [8] type system and operations are carefully designed. The discrete model is defined in [5] and algorithms for the operations are presented in [10]. The model was extended to a network-based representation of moving objects (or trajectories) in [9].

The conceptual framework underlying the Secondo system, second-order signature, is given in [6]. The system is freely available for download from the web site [15] where a lot of further documentation can be found. The Hermes system which also partially implements the model of Section 3.1 is decribed in [12].

The spatial join algorithm described in Section 3.5 is roughly based on [11]. The algorithm for continuous nearest neighbor queries of Section

3.6 is presented in detail in [7]. The algorithm for line segment intersection mentioned in that section is due to Bentley and Ottmann [1]. Spatiotemporal pattern queries are described in [14]; see also Chapter [Air Traffic Analysis].

Finally, the BerlinMOD benchmark is presented in [3]. A web site providing scripts and further documentation on the BerlinMOD benchmark is [2].

References

[1] Jon Louis Bentley and Thomas Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Trans. Computers*, 28(9):643–647, 1979.

[2] BerlinMOD Web Site. http://dna.fernuni-hagen.de/Secondo.html/BerlinMOD/BerlinMOD.html.

[3] Christian Düntgen, Thomas Behr, and Ralf Hartmut Güting. BerlinMOD: a benchmark for moving object databases. *VLDB J.*, 18(6):1335–1368, 2009.

[4] Martin Erwig, Ralf Hartmut Güting, Markus Schneider, and Michalis Vazirgiannis. Spatio-temporal data types: An approach to modeling and querying moving objects in databases. *GeoInformatica*, 3(3):269–296, 1999.

[5] Luca Forlizzi, Ralf Hartmut Güting, Enrico Nardelli, and Markus Schneider. A data model and data structures for moving objects databases. In *SIGMOD Conference*, pages 319–330, 2000.

[6] Ralf Hartmut Güting. Second-order signature: A tool for specifying data models, query processing, and optimization. In *SIGMOD Conference*, pages 277–286, 1993.

[7] Ralf Hartmut Güting, Thomas Behr, and Jianqiu Xu. Efficient k-nearest neighbor search on moving object trajectories. *VLDB J.*, 19(5):687–714, 2010.

[8] Ralf Hartmut Güting, Michael H. Böhlen, Martin Erwig, Christian S. Jensen, Nikos A. Lorentzos, Markus Schneider, and Michalis Vazirgiannis. A foundation for representing and querying moving objects. *ACM Trans. Database Syst.*, 25(1):1–42, 2000.

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

[10] José Antonio Cotelo Lema, Luca Forlizzi, Ralf Hartmut Güting, Enrico Nardelli, and Markus Schneider. Algorithms for moving objects databases. *Comput. J.*, 46(6):680–712, 2003.

[11] Jignesh M. Patel and David J. DeWitt. Partition based spatial-merge join. In *SIGMOD Conference*, pages 259–270, 1996.

[12] Nikos Pelekis, Elias Frentzos, Nikos Giatrakos, and Yannis Theodoridis. HERMES: aggregative LBS via a trajectory DB engine. In *SIGMOD Conference*, pages 1255–1258, 2008.

[13] Nikos Pelekis and Yannis Theodoridis. An oracle data cartridge for moving objects. Technical report, University of Peraeus, 2005.

[14] Mahmoud Attia Sakr and Ralf Hartmut Güting. Spatiotemporal pattern queries. *GeoInformatica*, 15(3):497–540, 2011.

[15] Secondo Web Site. http://dna.fernuni-hagen.de/Secondo.html/.

Verzeichnis der zuletzt erschienenen Informatik-Berichte

[347] Fechner, B.:
Dynamische Fehlererkennungs- und –behebungsmechanismen für verlässliche Mikroprozessoren

[348] Brattka, V., Dillhage, R., Grubba, T., Klutsch, A.:
CCA 2008 - Fifth International Conference on Computability and Complexity in Analysis

[349] Osterloh, A.:
A Lower Bound for Oblivious Dimensional Routing

[350] Osterloh, A., Keller, J.:
Das GCA-Modell im Vergleich zum PRAM-Modell

[351] Fechner, B.:
GPUs for Dependability

[352] Güting, R. H., Behr, T., Xu, J.:
Efficient k-Nearest Neighbor Search on Moving Object Trajectories

[353] Bauer, A., Dillhage, R., Hertling, P., Ko K.I., Rettinger, R.:
CCA 2009 Sixth International Conference on Computability and Complexity in Analysis

[354] Beierle, C., Kern-Isberner, G.:
Relational Approaches to Knowledge Representation and Learning

[355] Sakr, M.A., Güting, R.H.:
Spatiotemporal Pattern Queries

[356] Güting, R. H., Behr, T., Düntgen, C.:
SECONDO: A Platform for Moving Objects Database Research and for Publishing and Integrating Research Implementations

[357] Düntgen, C., Behr, T., Güting, R.H.:
Assessing Representations for Moving Object Histories

[358] Sakr, M.A., Güting, R.H.:
Group Spatiotemporal Pattern Queries

[359] Hartrumpf, S., Helbig, H., vor der Brück, T. , Eichhorn, C.:
SemDupl: Semantic Based Duplicate Identification

[360] Xu, J., Güting, R.H.:
A Generic Data Model for Moving Objects

[361] Beierle, C., Kern-Isberner, G.:
Evolving Knowledge in Theory and Application: 3rd Workshop on Dynamics of Knowledge and Belief, DKB 2011

[362] Xu, J., Güting, R.H.:
GMOBench: A Benchmark for Generic Moving Objects

[363] Finthammer, M.:
A Generalized Iterative Scaling Algorithm for Maximum Entropy Reasoning in Relational Probabilistic Conditional Logic Under Aggregation Semantics