Querying Moving Objects in SECONDO

Victor Teixeira de Almeida, Ralf Hartmut Güting, and Thomas Behr
LG Datenbanksysteme für neue Anwendungen
Fachbereich Informatik, Fernuniversität Hagen
D-58084 Hagen, Germany
{victor.almeida, rhg, thomas.behr}@fernuni-hagen.de

Abstract

Representing descriptions of movements in databases and querying them is a basic capability required in mobile data management. In this demonstration, we show for the first time a prototype implementing a data model and query language for moving objects (trajectories) completely integrated into a DBMS environment, including query optimization and user interface issues such as animation.

1. Introduction

Location aware mobile devices have become a cheap commodity. For example, there are already millions of users of GPS-equipped PDAs or car navigation systems. Such systems can also record movements. RFID tags are used as well to track the movements of goods. Such trends will result in the collection of massive amounts of moving object data, sometimes called trajectories, in the near future. There is a great interest in being able to represent such movements in databases in order to perform analysis on them, data mining, as well as ad-hoc querying.

The research area of *moving objects databases* has addressed this need, and there has already been a lot of research in the last years ranging from data models and query languages to implementation aspects such as efficient index structures (see [GS05]).

A query language for moving objects based on the idea of spatio-temporal abstract data types has been developed in earlier work [GBE+00]. Implementation aspects such as data structures for the types and algorithms for the operations have also been addressed [FGNS00, CFG+03]. In this demonstration we show for the first time a prototype of this design. An algebra for moving objects has been implemented in the SECONDO extensible DBMS environment. The design could be implemented in a similar way in other object-relational or extensible systems.

SECONDO is a DBMS prototyping environment particularly geared for extension by algebra modules for nonstandard applications. It is complete in the sense that all aspects needed by such applications, ranging from efficient query processing in the system kernel through optimization to an extensible user interface are addressed.

Examples in this paper are based on the "Berlin" database. It contains various geographic data sets from the city of Berlin. To this we have added a synthetic data set for moving objects, namely a relation describing underground trains as moving points. The moving point data have been generated by matching train schedules to train line geometries. Whereas analyzing train movements is perhaps not the most exciting application, it is a scenario that is easily understood, and the query examples can easily be translated to other domains. Since we have the data available, the queries in the paper can indeed be demonstrated.

This demonstration should be interesting because:

- It is to our knowledge the first presentation of a system that implements a moving objects data model and query language completely integrated into a DBMS environment.
- All system levels including kernel, query optimization, and user interface with animation for moving objects are included and can be demonstrated.

The SECONDO system has been demonstrated before, with a focus on architecture and extensibility and the use for prototyping and teaching [GAA+05]. This is the first demo addressing moving objects.

The complete SECONDO system, including the moving objects algebra demonstrated here, is freely available for download at http://www.informatik.fernuni-hagen.de/secondo.

2. Algebra for Moving Objects

In this section we review the system for representing moving objects presented in [GBE+00, FGNS00]. The core of this system are the abstractions *moving point* and *moving region*, describing objects with time-dependent position such as vehicles and mobile-phone users, and objects where the shape and extent are also time dependent, such as hurricanes and oil spills. These abstract data types (and their discrete representations described in [FGNS00]) may be embedded as attribute types into OO- or ORDBMS, or implemented as extension packages into extensible DBMS. We use the latter approach with the SECONDO Extensible DBMS [GBA+04], which is the subject of the next section.

Temporal types use the *sliced representation*, which represents a time-dependent value as a sequence of *slices* (*temporal units*) such that within each slice, the development of the value can be represented by a "simple" func-

tion, the so-called *temporal function*. As an example, for values that can only change discretely (e.g. *int* and *bool*) a constant function is applied. For the moving real (*mreal*), the function is a quadratic polynomial or square root of such (Figure 1(a)). Points move linearly inside each slice in the moving point (*mpoint*) representation (Figure 1(b)).

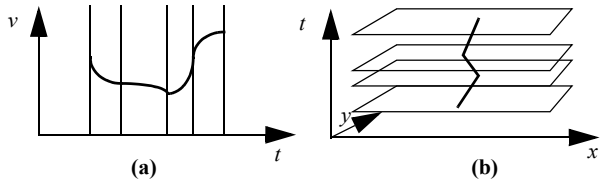

Figure 1: Sliced representation of a moving real and a moving point

For moving regions (*mregion*), vertices of regions also move linearly inside each slice, with several restrictions applied to ensure that, for every time instant inside the slice, a valid region is defined by the temporal function. More details about the representation of the moving object data types can be found in [FGNS00]. Figure 2 shows a sample slice of a moving region.

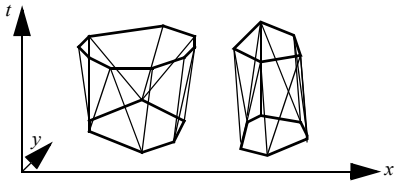

Figure 2: Sample slice of a moving region

Over these data types, a large set of operations is defined in [GBE+00]. First, generic operations on non-temporal data types are provided including predicates, set operations, aggregate operations, etc. Examples are:

point × *region*	→ *bool*	**inside**
region × *region*	→ *region*	**union**
line	→ *real*	**length**
point × *point*	→ *real*	**distance**

where **inside** checks whether a point is inside a region, **union** returns the region which is the union of the two argument regions, **length** returns the total length of a line, and **distance** computes the (Euclidean) distance between two points.

Then, by an approach called *lifting*, all operations defined in this first step are available for the corresponding temporal types. For example, the **inside** operator can be applied in the following ways

mpoint × *region*	→ *mbool*	**inside**
point × *mregion*	→ *mbool*	
mpoint × *mregion*	→ *mbool*	

where the arguments as well as the return value are lifted to their temporal counterparts.

Finally, special operators for temporal types are offered with projections into time and range of values, intersections with values or sets of values from time and range of values, and results that determine rate of change. Examples of such operators (appearing in queries below) are:

mpoint	→ *line*	**trajectory**
mpoint × *periods*	→ *mpoint*	**atperiods**
mpoint × *periods*	→ *bool*	**present**
mpoint × *instant*	→ *bool*	**present**
mpoint	→ *periods*	**deftime**
mpoint × *region*	→ *mpoint*	**at**
mpoint × *region*	→ *bool*	**passes**
mpoint × *point*	→ *bool*	**passes**
mpoint × *instant*	→ *ipoint*	**atinstant**
ipoint	→ *instant*	**inst**
ipoint	→ *point*	**val**

Here **trajectory** projects the moving point to the 2-d plane as a *line* value; **atperiods** restricts the movement to some period of time; **present** checks whether the moving object exists at a predefined period or instant of time; and **deftime** projects the movement to the time dimension. Operation **at** restricts a moving point to the times when it is inside a region, **passes** checks whether it is ever inside a region or at a point. Finally, **atinstant** evaluates the moving point at given instant of time, returning a pair consisting of the instant and a point, a value of type *ipoint*, for which **inst** and **val** return the components.

Now we are able to show how the abstract data types can be embedded into a (relational) DBMS data model and how the available operations can be used in queries. Assume that we have the following relations containing a set of underground trains and the train stations in Berlin. There are 562 trains and 173 stations. Each train contains about 100 temporal units. A larger version of this database is described in Section 4.

```
Trains(Id:int, Line:int, Up:bool, Trip:mpoint)
Stations(SName:string, Type:string, Loc:point)
```

A train system administrator could ask "Where exactly were the trains between 8:00 and 8:01 o'clock?":

```
SELECT  Id, Line,
    trajectory(Trip atperiods eight00) AS Stretch
FROM    Trains
WHERE   Trip present eight00;
```

where *eight00* is the period from 8:00 until 8:01 o'clock.

"At what times have trains passed through (underground) the park "Tiergarten"?

```
SELECT  Id, Line,
        deftime(Trip at tiergarten) AS Times
FROM    Trains
WHERE   Trip passes tiergarten;
```

Here *tiergarten* is a *region* value for the park area.

The following query will be used throughout the rest of the paper: "Where have the trains passing through the Mehringdamm station been at 6:50 am (as far as they are moving at this time):"

```
SELECT  Id, Line,
        val(Trip atinstant sixfifty) AS Pos
FROM    Trains, Stations
WHERE   Trip passes Loc AND
        SName contains "Mehringdamm" AND
        Trip present sixfifty
```

Here *sixfifty* is a value of type *instant*.

3. Moving Objects Algebra in SECONDO

In this section we present implementation issues of the moving object algebra in SECONDO, emphasizing the changes needed in order to accommodate the new data types and operations.

The goal of SECONDO is to provide a "generic" database

system frame that can be filled with implementations of various DBMS data models. For example, it should be possible to implement relational, object-oriented, temporal, or XML models and to accomodate data types for spatial data, moving objects, chemical formulas, etc. In this paper we deal with the relational data model with extensibility capabilities to provide the data types as attributes for moving objects.

The SECONDO system consists of three major components shown in Figure 3:

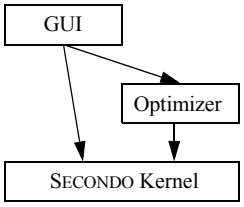

Figure 3: SECONDO Components.

- The SECONDO kernel implements specific data models, is extensible by algebra modules, and provides query processing over the implemented algebras. It is written in C++.
- The optimizer provides as its core capability conjunctive query optimization, currently for a relational environment, and also implements the essential part of SQL-like languages. It is written in PROLOG.
- The graphical user interface (GUI) is an extensible interface for an extensible DBMS such as SECONDO, where new data types or models can provide their own *viewers* or extend an existing viewer by display methods. It is written in Java.

3.1. The Kernel

A very rough description of the architecture of the SECONDO kernel is shown in Figure 4. A data model is implemented as a set of data types and operations. These are grouped into *algebras*. For example, there is an algebra with relations and tuples as data types and operations like projection or hashjoin. Index structures are also offered as algebras; currently SECONDO has an algebra for B-trees and another one for R-trees.

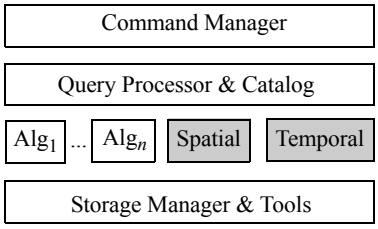

Figure 4: Rough architecture of the kernel

The focus of this paper is in the implementation of the *Spatial* and the *Temporal* Algebras. The Spatial Algebra implements the types *point*, *points*, *line*, and *region* following the implementation of the ROSE Algebra ([GRS95]). The Temporal Algebra mainly provides types for moving points and moving regions following the description in [FGNS00]. For every moving data type, a unit data type is provided implementing the corresponding temporal function, e.g. for the *mpoint* data type, the *upoint* is also provided. A subset containing the most important operators in [GBE+00] is implemented.

The kernel can evaluate a query plan, also called an *executable query*, or a query at the *executable level*, which is just a term of the implemented algebras. Query processing is performed as follows: the Command Manager receives an executable query, parses it and passes the result to the Query Processor. The Query Processor then evaluates the query by building an operator tree and then traversing it, calling operator implementations from the algebras. More details about this process can be found in [DG00]. SECONDO objects are stored (and retrieved) by the Storage Manager into a database and managed by the Catalog. As an example, one possible way of writing the last query of Section 2 at the executable level is:

```
Trains feed filter[.Trip present sixfifty]
Stations feed filter[.SName contains
"Mehringdamm"] symmjoin[.Trip passes ..Loc]
extend[Pos: val(.Trip atinstant sixfifty)]
project[Id, Line, Pos]  consume
```

where **feed**, **filter**, **symmjoin**, **extend**, **project**, and **consume** are operators of the Relational Algebra. **Feed** converts a relation into a stream of tuples and **consume** does the contrary, **filter** filters the stream of tuples given a condition, **extend** adds a calculated attribute to the tuple, and **project** projects the tuples to the given attributes. The dot is used to retrieve an attribute from the given tuple. **Symmjoin** is a symmetric variant of nested loop join, the double dot notation refers to an attribute of a tuple of the second argument.

3.2. The Optimizer

The optimizer provides as its core functionality cost-based optimization of conjunctive queries. That is, it receives a set of relations together with a collection of selection and join predicates, and produces a plan. It employs a novel algorithm for query optimization described in detail in [GBA+04], based on shortest path search through a predicate order graph. This technique is remarkably simple to implement, yet is efficient and is guaranteed to find the optimal plan even in the presence of expensive predicates[1].

The optimizer is written in PROLOG, using the SWI-PROLOG system. PROLOG is an excellent language for implementing optimizers, extensible optimizers in particular as new optimization rules can be formulated easily. It is also very efficient for this kind of task. The SECONDO optimizer handles queries with up to ten predicates in less than a second. The number of relations involved plays no role.

On top of the conjunctive query optimizer, the essential parts of an SQL-like language have been implemented. The SQL notation was slightly adapted so that queries can be written directly as PROLOG terms.

The algebra for moving objects, like other non-standard applications, poses the following requirements to an opti-

1. This is, of course, relative to the given cost functions, assuming correct estimations of selectivity and no correlations.

mizer:

- Selectivity estimation must work for complex data types and an extremely large set of operations. The traditional histogram-based approach does not scale to this case.
- Operations can be expensive; hence expensive predicates must be supported in optimization.

SECONDO provides selectivity estimation by sampling; for each relation a small materialized sample is kept. Unknown selectivities are determined in advance by sending selectivity queries to the kernel before starting the proper optimization process; they are then stored for later use.

The cost for expensive predicates is determined as well in the execution of the selectivity queries on samples by measuring the actual execution time, subtracting overhead.

The beauty of this scheme is that optimization works to a large extent automatically without manual work when a new algebra with non-standard types is added. What has to be provided manually are optimization rules for adding specialized indexes, and possibly syntax rules for operations (the latter is very easy).

An example interaction with the SECONDO optimizer is shown in Figure 5. It shows the example query from Section 2 as written by the user. Here all symbols are written in lower case, and the PROLOG notation for lists is used. For the two predicates trip passes loc and sname contains "Mehringdamm", selectivities and predicate evaluation costs are determined (for trip present sixfifty this was known already). Then the query plan is constructed and shown. The latter is a term of the executable level of SECONDO that is readable and which can as well be typed in directly, e.g. for experimenting.

```
 opt-server >
 optimization-input : select [id, line, val(trip atinstant sixfifty) as
pos] from [trains, stations] where [trip passes loc, sname contains
"Mehringdamm", trip present sixfifty]

Computing best Plan ...

Elapsed Time: 1218 ms
Predicate Cost: 0.0994104 ms
Selectivity : 0.119884

Elapsed Time: 62 ms
Predicate Cost: 0.001 ms
Selectivity : 0.00578035
Destination node 7 reached at iteration 5
Height of search tree for boundary is 2

 optimization-result : Trains  feed  project[Id, Line, Trip] Stations
feed  project[Loc, SName]  filter[(.SName contains
"Mehringdamm")] symmjoin[(.Trip passes ..Loc)] filter[(.Trip
present sixfifty)] extend[Pos: val((.Trip atinstant sixfifty))]
project[Id, Line, Pos]  consume


 opt-server >
```

Figure 5: Protocol of the SECONDO optimizer

3.3. The User Interface

A visualization of query results is possible in the graphical user interface "Javagui" of the SECONDO system. Javagui communicates with the system kernel and the optimizer via TCP/IP. It can be extended by viewers. Each viewer can display a set of different data types. In this way, Javagui is able to display each type implemented in the system kernel.

The user interface consists of three parts (see Figure 7), namely the command area (top-left), the object manager (top-right), and an area containing the current viewer (bottom). In the command area, the user can input queries and commands controlling Javagui. Javagui recognizes whether a query is given at the executable level or in the syntax of the optimizer. If the query is in optimizer syntax, Javagui sends it to the optimizer and receives a plan at the executable level. This plan is sent to the system kernel. The result of a query is delivered in a generic format based on nested list structures to the object-manager. It stores the result of the query, selects a viewer able for displaying the result, and finally it transfers the query result to this viewer for further processing.

The HoeseViewer (named by its author) is a fairly sophisticated viewer for spatial and spatio-temporal data. This viewer can be extended for displaying further data types using display classes. Existing implementations include classes for displaying:

- simple types like integer and string
- spatial types like point, line, and region
- temporal types, e.g. moving reals
- spatio-temporal types, for example, moving points and moving regions

The HoeseViewer contains in principle three areas displaying different informations about query results. At the left, textual information is shown. The right part is divided into a big area displaying spatial and spatio-temporal objects and a smaller area for temporal data, e.g. periods or moving reals (this area is not shown in Figure 7).

Depending on the type to display, each display class converts an object given as a nested list into an internal format, e.g. a string or a geometrical object. For spatio-temporal objects, a display class has to provide a method taking an instant and returning the shape of this object at this instant or nothing when the object is not defined at this time.

If an object does not fit well into existing areas, the class implementer is free to create a new window. This is done for example within display classes for text, pictures and moving reals.

For spatial and spatio-temporal objects, the appearance (linewidth, filling etc.) can be changed to the user's preferences. This and further functionality like zooming and labeling of objects are part of the HoeseViewer. A display class must not worry about such things.

Moving spatial objects are animated. The animation is controlled using a few buttons and a slider (Figure 6).

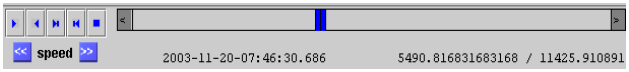

Figure 6: Controlling the animation

Using the time slider, any instant can be selected. The ani-

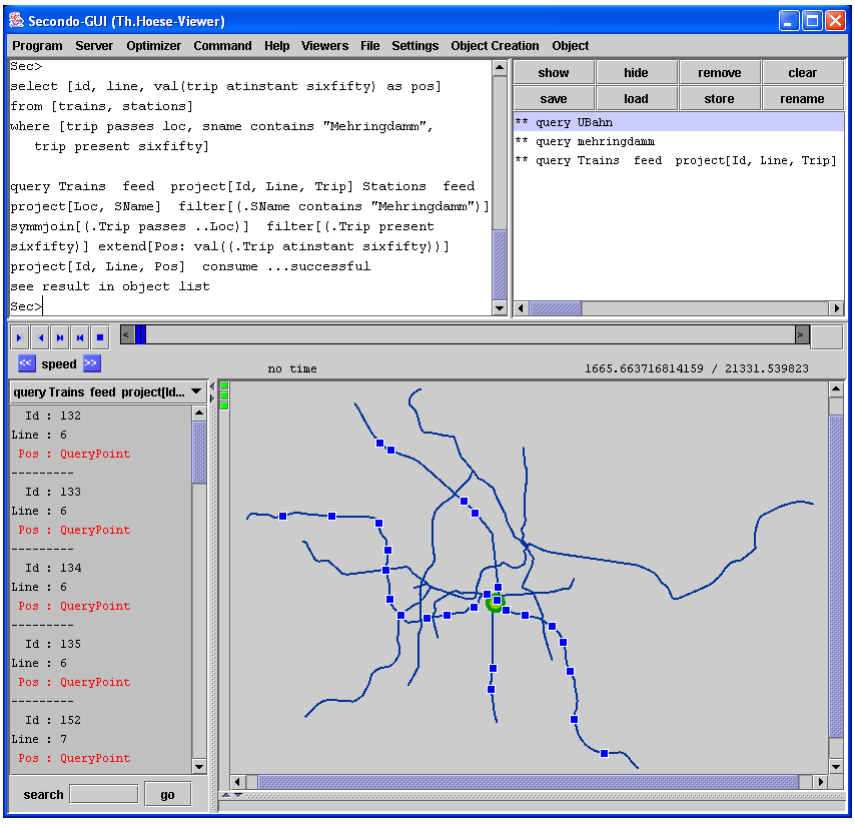

Figure 7: The graphical user interface

mation speed can be doubled or halved by the corresponding buttons. By the remaining buttons, the animation can be started/stopped or set to its begin or to its end. Below the time slider, the current time of the animation and the spatial position of the cursor can be seen.

If an object is selected, the object is kept in the visible area of the animation. For seeing moving objects within a spatial context, a picture, e.g. a city map, can be used as background image.

Non-spatial temporal objects can be shown at the bottom right or in a new window. The default is the display of the single units within the HoeseViewer. For moving real values an implementation exists opening a new window showing the value of this object as a function of time.

4. What Will be Demonstrated

The demonstration will be focused on query execution and visualization, and will be divided into parts, using the following databases.

The Berlin Database. The Berlin database contains several relations with spatial objects such as streets, underground train lines, green and water areas, sightseeing spots, restaurants, etc. and a relation containing several lines of underground trains as moving points. We will show the capabilities of all three components of SECONDO performing several different queries in this database.

GPS Data. This database contains some real data about tracings collected using a GPS device. The main focus on this demonstration will be to show the animation of moving objects in the user interface of SECONDO.

Moving Region Data. This small sample database contains some regions, moving points, and moving regions. We will show with this example query processing using the moving region data type.

Big Berlin Database. We translated the Berlin database five times in all directions: *x*, *y*, and *time*. We then have a database that is 125 times larger than the Berlin database. With this database we can show how queries scale with bigger data sets and how indexes are used.

Acknowledgements

We thank Slaven Rezic for allowing us to use the Berlin database taken from the BBBike application (`http://bbbike.sourceforge.net`). We also thank everybody that has contributed in the development of SECONDO and the algebra for moving objects, especially Markus Spiekermann, Zhiming Ding, Frank Hoffmann, Thomas Höse, and Holger Münx.

References

[CFG+03] J. A. Cotelo Lema, L. Forlizzi, R. H. Güting, E. Nardelli, M. Schneider, Algorithms for Moving Objects Databases, *The Computer Journal*, 46(6), 2003.

[DG00] S. Dieker and R.H. Güting, Plug and Play with Query Algebras: SECONDO. A Generic DBMS Development Environment. Proc. IDEAS 2000, 380-392.

[FGNS00] L. Forlizzi, R.H. Güting, E. Nardelli, and M. Schneider, A Data Model and Data Structures for Moving Objects Databases. In Proc. ACM SIGMOD Intl. Conf. on Management of Data, 2000, 319-330.

[GAA+05] R.H. Güting, V.T. de Almeida, D. Ansorge, T. Behr, Z. Ding, F. Hoffmann, M. Spiekermann, and U. Telle, SECONDO: An Extensible DBMS Platform for Research Prototyping and Teaching. In Proc. 21st Intl. Conf. on Data Engineering (ICDE), 2005, 1115-1116.

[GBA+04] R.H. Güting, T. Behr, V.T. de Almeida, Z. Ding, F. Hoffmann, and M. Spiekermann, SECONDO: An Extensible DBMS Architecture and Prototype. Fernuniversität Hagen, Informatik-Report 313, 2004.

[GBE+00] R.H. Güting, M.H. Böhlen, M. Erwig, C.S. Jensen, N.A. Lorentzos, M. Schneider, and M. Vazirgiannis, A Foundation for Representing and Querying Moving Objects. *ACM Transactions on Database Systems*, 25(1): 1-42, 2000.

[GRS95] R.H. Güting, T. de Ridder, and M. Schneider, Implementation of the ROSE Algebra: Efficient Algorithms for Realm-Based Spatial Data Types. In Proc. 4th. Intl. Symp. on Advances in Spatial Databases (SSD), 1995, 216-239.

[GS05] R.H. Güting and M. Schneider, *Moving Objects Databases*. Morgan Kaufmann Publishers, 2005.