A SPATIO-TEMPORAL ALGEBRA IMPLEMENTATION *

Miguel Rodríguez Luaces

Praktische Informatik IV, Informatik Zentrum FernUniversität Hagen
D-58093 Hagen (Deutschland)
Phone: +49-2331-9874282
Email: Miguel.Rodriguez-Luaces@FernUni-Hagen.de

Laboratorio de bases de datos, departamento de computación,
facultad de informática de A Coruña
E-15071, A Coruña (Spain)
Email: luaces@mail2.udc.es

ABSTRACT

This paper describes the implementation of a spatio-temporal algebra. The relational algebra has been widely used during the last years due to its simplicity and ability to store and query data types needed by most of the business applications, but it is not suitable for some other data types like spatio-temporal ones. There are many applications that need to use spatio-temporal data types, for instance, traffic control, resource management, environmental control, etc. It is highly desirable to use a DBMS and its features to build these applications, but two characteristics are needed from the DBMS: support for spatio-temporal data types, and a spatio-temporal algebra that allows the user to store values of these data types and answer significant queries involving values of these data types.

In this paper, we report the last step in the development of this spatio-temporal algebra, its implementation using SECONDO, an extensible database system that allows the user to implement easily new algebras, extending with them the set of types and operations supported by the DBMS, and even changing the data model of the database.

1 INTRODUCTION

Two research fields that have been widely explored during the last decades are spatial and temporal databases, but only during the last few years they have joint to form the new concept of a spatio-temporal database as a system that stores spatial information that changes over time. The way geometries change varies from ”almost static” (e.g., country boundaries evolution) to ”highly dynamic” (e.g., whale tracing in an ecological application), but we focus our research in the later one since it arises the most challenging problems. We use the term *moving objects database* to refer to this type of databases, emphasizing the dynamicness of the objects they store by using *moving* instead of *temporal*.

The first research goal in that type of databases is to allow the user to model and query this kind of objects. Even though the relational algebra has been widely used during the last decades due to its ability to store the data needed by almost any business application, it cannot be used to query moving objects using only the traditional basic data types (`integer`, `char`, and so on). A new algebra must be designed trying to find a set of significative data types and operations that can be plugged in any data model designed to be extended. Following this approach, we develop a *general* extension module because it does not depend on a particular data model, therefore it can be implemented in any. This is not an unrealistic idea since many *extensible systems* have appeared lately, both commercial systems like Informix Universal Server or IBM’s DB2 Universal Database and research systems like SECONDO.

Now the question of *level of abstraction*, which was introduced in (Erwig et al., 1999), must be taken into account. By level of abstraction we mean the level at which we define the domains of the data types considered. A description at what we call *abstract level* describes the domains using infinite set points, whereas a description at the so-called *discrete level* uses finite representations. For instance, an abstract level domain definition of the *region* data type could be $dom \subseteq R^2$, and a discrete level one could be *the set of points enclosed in a polygon*. The main difference is that a description at an abstract level is clean, simple and focused on the concepts not in their implementation, but it has no straightforward implementation. On the other hand, a description at a discrete level may not be able to catch all the expressive power of the abstract one, but it can be implemented with the limited resources of a computer.

* This work was supported by the CHOROCHRONOS project, funded by the E.U. under the training and Mobility of Researchers Programme, Contract No. ERB FMRX-CT96-0056

It was concluded in (Erwig et al., 1999) that both levels are necessary: the discrete level modelling is essential because is the only one that can be finally implemented in a computer system, and the abstract level is necessary because restricting our attention only to the discrete data model a simple and elegant query model could be missed. Therefore, one must design first the abstract model trying to achieve some interesting properties such as closure on the data types under the operations, completeness, etc., and afterwards design a discrete level that captures as much of that model as possible.

In (Güting et al., 1998), an abstract data model for a spatio-temporal algebra was presented being the main concerns in its design orthogonality in the type system, genericity and consistency of operations, and closure and consistency between structure and operations of related temporal and non-temporal types. In (Forlizzi et al., 1999), a discrete model for it was presented defining data types in terms of finite representations, how they relate to the correspondent abstract data types and how they can be directly mapped into data structures. Here, we report on the last step in this evolution, the actual implementation of the algebra in a computer. We review both designs and give descriptions of the data structures and algorithms, with the focus set on their implementation.

One of the simplest spatio-temporal models is the snapshot model presented in (Langran and Chrisman, 1988). Temporal information is incorporated to a spatial data model by means of layers. Each layer describes the complete spatial situation in a particular time instant. This model is obviously not suitable to describe our dynamic situation because any change in a particular value obliges to produce a complete snapshot duplicating all the unchanged data. The same author proposes another model in (Langran and Chrisman, 1988), a partition of the space is computed resulting from all the intersections of all the spatio-temporal values. A particular spatial value at any moment in time can be formed by the composition of several of these areas. This model is more efficient storing the changes, but it cannot capture movement and a change in a instant of time *t* may make necessary to change the state of the database in previous time instants because the basic partition of space may have changed.

Another approach, followed in (Hunter and Williamson, 1990), consists in time-stamping each spatial object with its time of creation and its time of cessation. This solution is more efficient than the previous one and capable of storing changes in objects, but the evolution of an object is split in different versions instead of keeping it as a whole. Therefore, it is more complex to trace the evolution of an object.

Other models, such as those presented in (Peuquet and Wentz, 1994) or (Peuquet and Duan, 1995), try to capture the changes in objects representing them as events that are explicitly stored. The database will store only its actual state, and to obtain a historical state the chain of events has to be *rewound* up to the appropriate position.

These models do not consider continuous changes and movement, only situations almost static with discrete changes in the spatial objects. More recently, research has addressed more dynamic applications. In (Sistla et al., 1997), (Wolfson, Xu et al., 1998) and (Wolfson, Chamberlain et al., 1998), Wolfson and colleagues consider the management of moving points in the plane, but only taking into account the current and expected positions of a point. No trajectory of the moving objects is kept in the database and no complex structures such as lines or regions are considered.

The CHOROCHRONOS project, in which we participate, has addressed some issues related to moving object databases: conceptual modelling is discussed in (Tryfona and Hadzilacos, 1997), indexing in (Theodoridis et al., 1998) and the uncertainty in capturing moving point trajectories in (Pfoser and Jensen, 1999).

The constrain database paradigm can also be used, and references (Grumbach et al., 1998) and (Chomicki and Revesz, 1997) address spatio-temporal examples and models.

In the first part of this paper, we try to summarize all the efforts that led to the work reported in the second part. This previous work can be seen as two different roads approaching to a crossroad, on the one hand we have the work done to achieve an extensible database system easy to extend described in Section 2, on the other hand we have the work done to describe a modelling and query language for spatio-temporal data described in Section 3. Section 4 deals with the meeting point of these efforts, the actual spatio-temporal algebra implementation using SECONDO.

2 SECONDO

The target system chosen for the implementation of the executable algebra is SECONDO, a database system being developed by our group at Praktische Informatik IV in the University of Hagen. The goal is to achieve an architecture that is extensible at a much more fundamental level than other systems, allowing not only to extend the set of data types but also the data model, and keeping, nevertheless, the easiness of use and extension; all these issues supported by a strong formal basis.

2.1 SECOND-ORDER SIGNATURE

The formal basis for SECONDO is *second-order signature*, which is described in (Güting, 1993). The main idea is to use a couple of many-sorted signatures, the first one describing type constructors and the second one using terms of the first signature to describe polymorphic operators. A *signature* consists of two sets of symbols called *sorts* and *operators*, being the operators defined by a term described as a regular expression over sorts: $\times$ is used to describe a term that is the Cartesian product of the subterms in the expression, $^+$ to describe a term that is a list of subterms, $\cup$ to describe a term that may be exactly one of a list of subterms, and $\rightarrow$ to describe a term that is a function.

The following signature defines the behaviour of the operator + between the types *integer* and *real* (the sorts of the signature):

sorts integer, real

operators

$integer \times integer$	$\rightarrow$	*integer*	+
$integer \times real$	$\rightarrow$	*real*	+
$real \times integer$	$\rightarrow$	*real*	+
$real \times real$	$\rightarrow$	*real*	+

Now, we can use a signature to describe a type system by making the operators of the signature play the role of *type constructors* and terms of the signature the role of *types*. Sorts will be used to partition the possible types that can be constructed, and will be called *kinds*. They will be used later on to describe polymorphic operations. As an example, consider the following relational data model description:

kinds IDENT, ATTR, TUPLE, REL

type constructors

	$\rightarrow$	IDENT	*Ident*
	$\rightarrow$	ATTR	*Int, real, string, bool*
$(\underline{ident} \times \text{ATTR})^+$	$\rightarrow$	TUPLE	*Tuple*
TUPLE	$\rightarrow$	REL	*Rel*

Values of type *ident* are identificators with the usual syntax that can be used as attribute names. Similarly, the types *int*, *real*, *string* and *bool* are the usual attribute data types. The type constructor *tuple* constructs tuple types consisting of a list of pairs *(attribute_name, attribute_type)*, and the type constructor *rel* uses a tuple type to indicate the type of the tuples that form the relation. With this type system one can describe the typical employee relation with the following term of the signature (and hence a type): *rel(tuple(<(name, string), (salary, integer), (dept, integer)>))*.

Once a type system has been defined, we can use the terms of this signature to specify operations on those types. Moreover, the presence of kinds partitioning types into different related sets provides an excellent tool to describe polymorphic operators over the types in the kind. Another signature is used to describe the operators, using as sorts of the signature terms constructed by the type specification signature. For instance, consider the following operator specification:

operators

$\forall$ attr in ATTR. attr $\times$ attr	$\rightarrow$	*bool*	$=, \neq, \leq, <, >, \geq$
$\forall$ rel: *rel*(tuple) in REL. rel $\times$ (tuple $\rightarrow$ *bool*)	$\rightarrow$	rel	**select**

Words in italics and underlined are, as above, type constructors, and words in normal face are variables that can be instantiated for each possible term that fits with the type specification described above. The first line defines the Boolean operators by means of *quantification over kinds*. For each possible instantiation of the formula (changing the variable *attr* for a type in the kind ATTR), we obtain an operator specification. The second line defines the **select** operator as an operator that takes a relation *rel* (with type *rel*(*tuple*)) and a function transforming *tuple* (the tuple type of the relation considered in a particular instantiation) in a *bool* value and returns a value of the same type *rel*.

For more details of this notation, please refer to (Güting, 1993) where these concepts are described in a formal manner; in addition to that, an optimizer for such algebras and considerations about query languages are also described.

2.2 SYSTEM ARCHITECTURE

The strategy followed to achieve a system with huge capabilities to be extended is to separate the data model independent components and mechanisms of a DBMS from the dependent ones. The later ones will be structured as algebra modules providing a description of a data model and a query language, and a collection of data structures for the types in the data model and algorithms for the operators in the query language. The data model independent components of the data base are integrated into a well-defined system frame that offers clear interfaces to plug in the algebra modules as well as tools to implement them and a fixed set of commands to use all the possible extensions in a consistent way.

The fact that a fixed set of commands is offered proofs that the system remains easily usable though its enormous extension capabilities. This set includes commands to manage databases (create, destroy, open, close, etc.), types (create, destroy, etc...), and values (create, update, destroy), commands to bulk-load and save database contents as well as single values, to manage transactions and to query the database.

A complete description of the system can be found in (Dieker and Güting, 1999) and (Güting et al., 1999), we will give here a brief summary. Fig. 1 shows an architectural overview of the SECONDO system where white boxes are part of the system frame and grey-shaded boxes represent the extensible part of the system. The system is built on top of the storage manager component of the SHORE system (Carey et al., 1994) to offer a full-fledged database system with concurrency control, transaction management and crash recovery. At the most basic level that SECONDO itself offers, we can find a variety of tools such as nested lists (the generic format used to pass values and type information), catalog tools, a tuple manager, a simplified storage manager interface, a SOS parser and some others.

On top of the tool level, we find the algebra module level. Each algebra module implements a particular data model and query language, whose description is supported by the formal basis of a second-order signature (Güting, 1993). For each type constructor, the algebra must provide a data structure to represent values and support functions to create, destroy, update and query them; besides, a model data structure must be provided to support optimization. For each operator, functions must be provided to support type mapping and argument checking, to evaluate a particular combination of argument types and to select the appropriate evaluation function given a combination of argument types. In addition to that, model

mapping and cost functions must also be provided to support optimization.

Modules can also be connected or disconnected for a particular version of the database system, allowing for different flavours of the system in different situations. In Fig. 1, modules 1, 2, and *n* are connected to the frame, thus *active*, while module 3 is *inactive*. Possible examples of algebra modules are: an algebra with the standard data types, a package of stream operations for streams of tuples, a B-Tree package, a relational algebra, an object-oriented algebra, spatio-temporal data types, etc.

The next level consists of the query processor, described in (Güting et al., 1999), and the system catalog. The first one takes an executable query, builds an evaluation tree and calls the appropriate functions in the algebra modules to execute it. The later one stores catalog information for all the types and values in the system.

In level 4, we find the query optimizer that transforms a query in a descriptive language into an efficient executable query by means of a two-step process. In a first step, algebraic rewriting is done over the query in descriptive language to achieve an equivalent but optimized query. In a second step, this query is transformed into an executable query using the cost models offered by the algebras.

The command manager in level 5 provides an application programming interface to access the functionality of the lower levels by means of a fixed set of commands. Every front-end for the system frame is expected to be developed on top of this interface without worrying about the details of the lower levels, just using this simple, easy to use set of commands.

Two different front-ends are currently offered with the system, a single-user interface and a multi-user client-server interface. The first one is intended to be used for debugging, the second one offers a full-fledged multi-user interface that can be accessed through a network interface by client software. To support the implementation of user clients, SECONDO provides comfortable client libraries for C++ and Java.

The system is still being developed, but at the moment all levels but the optimizer have been implemented and two algebra modules providing basic data types and a relational algebra are working. In addition to the spatio-temporal algebra module described in this paper, a graph and object oriented algebra module is also being developed.

The work described in this paper consists of an algebra module that is expected to be used with the relational algebra module that SECONDO currently offers. The algebra module consists of a set of data types and operations that can be used to model and query spatio-temporal data, and it fits in level 2 of Fig. 1.

2.3 ALGEBRA MODULES

Even though SECONDO's user has full freedom to implement type constructors and operators, as long as the support functions are provided and SHORE persistent objects are used in the end to keep values under the scope of the concurrency and recovery system, the system offers some tools that make the implementation easier. Here, we will describe some of this tools that are used in the implementation of our algebra later on in Section 4.

A FLOB (standing for Faked Large OBject) is a large object that can be used as part of the representation of a data type with the particularity that it automatically switches its representation according to its size from being stored with the data type representation to be stored as a stand-alone entity. A large object is not only used in the implementation of new data types for a database system because it provides a way to store objects larger than a page, but also because it allows to store objects whose size may vary. For some types, the size of the variable part varies between very small and very big (as an example consider a polygon that may have three vertices or three hundred thousand). Since storage systems usually work over the basis of pages (the smallest block of storage space that is transferred is a page), when the size of the value is small it is in general more efficient to store the size-varying part together with the rest of the value representation, to save a disk access in case the size-varying had to be used. But if the size of this part is very big, it is more efficient to store it as a stand-alone object that is accessed (and therefore loaded) only if needed.

Another tool of the system, the Tuple Manager, provides an abstraction of tuples as aggregations of attributes. It takes a set of data type representations, assembles a contiguous byte string with the values and saves it to disk, making a persistent object whose identificator is returned to the user of the tuple manager. This identificator can be used later on to restore the tuple and the attribute data type representations in main memory. The Tuple Manager is responsible of managing FLOB values used in the attributes, deciding whether they should form part of the aggregation or be stored as a stand-alone object.

These two concepts were first introduced in (Dieker and Güting, 2000) and we refer the reader to that paper to find more information on the subject, in particular an analytical deduction of the optimum threshold size for a single FLOB.

In Fig. 2, we see a conceptual representation of this situation. The thick solid line represents a tuple of the tuple manager, it consists of attributes that are drawn with solid thin lines. Each attribute contains the static part of the memory representation of a data type. Some of them have FLOBs, which are represented with thin dashed lines. Dotted lines represent FLOBs bigger than the space used to draw them. FLOBs consist of a FLOB handle, which stores some information about the FLOB itself, and a FLOB value, which is stored with the tuple (like the FLOB of the second attribute) or as a persistent object (like the FLOB of the fourth attribute) depending on its size and access probability.

As a result of choosing second-order signature as the formal basis, there are constant and non-constant type constructors in algebra modules for SECONDO. A constant type constructor does not take any arguments to construct

the type, hence the type constructed by it is always the same; a clear example is the type constructor *int*. On the other hand, a non-constant type takes arguments to construct the new type, hence the type constructed varies depending on them; an example is the type constructor *tuple*. The main difference between both types is that the memory representation of the first one is (structurally) always the same, whereas the memory representation of the second one may change according to the particular arguments.

In addition to that, a type constructor in SECONDO must declare when registered whether it constructs persistent or non-persistent types. A persistent type is one whose values are still available in different sessions with a database, and a non-persistent type is one whose values do not survive the end of the session. Persistent types must have as memory representation SHORE persistent objects or tuples of the tuple manager, non-persistent objects must use plain structures in main memory (without pointers) using FLOBs for the size-varying parts.

During the implementation of the algebra that we are describing here, and because it is the first algebra to be implemented that it is not a data model on its own but a new set of types for an existing one, the need for new functionalities in the system tools was discovered. In particular, new functionalities for FLOBs. The first functionality that we need is the ability to nest FLOBs. When complex type constructors have to be implemented, they often need to store a value of another type in its memory representation (an *inner value*). To keep the type constructor implementation general, no assumption can be made on the memory representation of the inner value, therefore, the inner value has to be stored in a FLOB. If this value may use FLOBs, the tool must be prepared to manage the situation of nested FLOBs.

This functionality has been now integrated in the FLOB interface. When a situation like the one depicted in Fig. 3 is found and the tuple is saved, all the FLOBs are recursively saved. As an example of a situation in this algebra where we need this functionality, we could mention the type constructor *const* that will be described later.

Another new functionality needed is the ability to share the same storage space between different FLOBs. Some types have a *set behaviour*, e.g., *tuple*, *set*, or in the case of the algebra we are describing here, *mapping*. It is very common for a value of this type to have some operations that access all the individual values one after the other. If these individual values use FLOBs, they may be scattered across the secondary storage making inefficient for the storage manager to retrieve them in order. To improve efficiency in this case, we can use *FLOB clusters*. Instead of storing each FLOB value independently of the others, we create a shared space where all the FLOBs are stored. The differences with a normal FLOB are two: first, the value is stored all in a persistent object, therefore, when the FLOBs that compose the FLOB cluster are read one after the other, the same persistent object is read and we can expect the storage manager to be able to optimize the access. Second, the decision to keep the FLOB value together with the tuple or in a different object is considered for the whole FLOB cluster, not for each individual FLOB.

The representation of a FLOB cluster can be seen in Fig. 4, we can see in it a tuple consisting of attributes, some of them having FLOBs. These FLOBs are stored in a FLOB cluster.

One of the advantages of the FLOB implementation as an untyped byte string is that many different structures can be implemented on top of it having the same properties. As an example, we have implemented for this algebra the concept of a database array.

A database array (DBArray for short) is an open array of elements of equal size called *slots*. One can write in any slot and the size of the DBArray changes if a non-existing slot was referred, one can also read from any existing slot and resize the DBArray at any moment. DBArrays are implemented using FLOBs hence they have the same properties.

3 SPATIO-TEMPORAL QUERY LANGUAGE

3.1 DATA TYPES

In (Güting et al., 1998), an algebra to model and query moving objects consisting of data types and operators was presented. The data types can be described using the following second-order signature (see (Güting, 1993) and Section 2.1):

kinds BASE, SPATIAL, TIME, TEMPORAL, RANGE

type constructors

	$\rightarrow$ BASE	*int*, *real*, *string*, *bool*
	$\rightarrow$ SPATIAL	*point*, *points*, *line*, *region*
	$\rightarrow$ TIME	*instant*
BASE $\cup$ SPATIAL	$\rightarrow$ TEMPORAL	*moving*, *intime*
BASE $\cup$ TIME	$\rightarrow$ RANGE	*range*

The base types are the same as in the relational algebra with the addition of an undefined value. The spatial types are *point* (a two dimensional point), *points* (a finite set of *point* values), *line* (a finite set of continuous curves in the two dimensional space) and *region* (a finite set of disjoint faces with holes). The first one of these types can also be undefined, the last three are sets of values and the empty set represents the undefined value. An example of a value of type *point* could be a city in a large scale map, a value of type *points* could be the location of all the book shops in a city, a value of type *line* could be the sewage network in a city, and a value of type *region* could be the extension of a particular country. Finally, time (*instant* type) is considered isomorphic to real numbers.

To add temporal types, instead of adding a new temporal type for each non-temporal type that we wish to make temporal (as it would have to be done with a pure relational system), the orthogonal type constructor *moving* is introduced. It yields for a given type a new one

consisting of a mapping from time to this type. This mapping may be partial, and therefore the new type stores for each time instant a value from the type used to construct it or an undefined value. Because we may want to know the value of the temporal object at any specific point in time, even at an undefined position, we need the values of all the non-temporal types that have a temporal counterpart to be able to be undefined.

The spatial type *point*, which stores a single two dimensional point (or an undefined value), can be used to construct the spatio-temporal type *moving*(*point*). This type stores for each time instant a two dimensional point (or undefined). A value of this type can be used, for example, to store the position of the eye of a hurricane. Its value is undefined until the moment that the tropical storm reaches the category of hurricane, storing a position in a two dimensional space as long as the hurricane exists, and being undefined again for the rest of the time dimension. Notice that the hurricane could go back to the stage of tropical storm during its lifetime and reach again the category of hurricane later on. Our value would have a undefined period between two defined periods.

The type constructor *intime* yields a new type that associates an instant of time with a value of the type used to construct it. For instance, the type *intime*(*int*) has as possible values pairs formed by a time instant and an integer value. Its main benefit is to be used as a return type for some particular operations.

Some of the operations that we are interested in project a value from a temporal type into its orthogonal components (i.e., domain and range). Types must be provided to serve as return type of such operations. The domain of a temporal object is the set of time instants in which the object takes a value other than undefined. Similarly, the range of a temporal object is the set of values that the object takes discarding the undefined. To support these types, a *range* type constructor is introduced yielding for a type a new one consisting of a set of intervals over values of that type. This type can be used to store the result of a projection over the domain (a value of type *range*(*instant*)) or over the range (a value of type *range*(α) for the type *moving*(α)).

A straightforward implementation of this signature is not possible in a computer since it uses infinite sets to describe the domains of the data types. A discrete model for this abstract model was designed and presented in (Forlizzi et al., 1999). The type system changes to reflect the fact that the types are not the same but new ones that try (but not always achieve) to represent the types of the abstract model. The new signature is the following:

kinds BASE, SPATIAL, TIME, RANGE, TEMPORAL, UNIT, MAPPING

type constructors

	→	BASE	*int*, *real*, *string*, *bool*
	→	SPATIAL	*point*, *points*, *line*, *region*
	→	TIME	*instant*
BASE ∪ TIME	→	RANGE	*range*
BASE ∪ SPATIAL	→	TEMPORAL	*intime*
BASE ∪ SPATIAL	→	UNIT	*const*
	→	UNIT	*ureal*, *upoint*, *upoints*, *uline*, *uregion*
UNIT	→	MAPPING	*mapping*

The base types as well as the *instant* type and the spatial types *point* and *points* can be implemented easily with usual computer language types, for *line* and *region* linear approximations (polylines and polygons with polygonal holes) are introduced. Types in the kinds RANGE and TEMPORAL can also be implemented with computer language types.

The concepts of *temporal unit* and *mapping* are introduced to represent a moving object. As we said before, a moving object is a partial mapping from time to a particular type, but we cannot store this mapping because it implies to store a value for each time instant of an infinite set. Consequently, we split the temporal evolution of the object in slices where it is *coherent* with respect to a class of functions, and we store the mapping as a set of slices consisting each one of a time interval plus a function describing the evolution of the object in it; if the moving object is undefined, no unit is stored for that interval. We call temporal unit to the type of these slices. The type constructor *mapping* assembles them and makes sure that the time intervals are correct. Fig. 5 shows an example of a *moving*(*real*) value of the abstract level in the left part, and its sliced representation on the right side using linear functions as the class of functions chosen for each unit. Each portion of the line between vertical dashed segment can be represented with a function of the class:

$$f(t) = at + b \tag{1}$$

This function is used as the value of the temporal unit for the time interval between the dashed segments.

The election of the class of functions must be driven by the needs of the applications that use the algebra; the slicing of the value must be done according to some criteria of validity for the application.

As a consequence, for each type that the *moving* type constructor may be applied to, a unit type must be designed representing the time interval plus the function describing the evolution of the value in that time interval. Since there are some types (*int*, *bool* and *string*) that can only change in a discrete way, the function that describes perfectly the way they evolve is a value of that type itself, hence we introduce the type constructor *const* that constructs a unit associating a time interval with a value of that type. This type constructor will be used for any discrete-changing temporal value.

To summarize, the temporal behaviour of non-temporal data types once the *mapping* type constructor is applied is defined by the class of functions chosen for the temporal units of that type. There may be types whose temporal behaviour is only discrete, therefore no specific temporal unit is needed because the type constructor *const* is used in that case. For the other types a temporal unit type must be provided. The class of functions for a particular temporal type must be chosen according to the needs of the application that is going to use the algebra.

To avoid confusion with all these type constructors, the reader must remember that there are two levels of abstraction, the abstract level, whose types are going to be used in the descriptive algebra, and the discrete level, whose types are going to be used in the executable algebra. A value declared of type *moving(point)* in the descriptive level, is created of type *mapping(upoint)* in the executable level. All the type mappings can be seen in Table 1.

In (Forlizzi et al., 1999), and consequently in our implementation, linear functions have been chosen to approximate the behaviour of spatial data types. A quadratic function with the option of having a square root has been chosen to approximate the behaviour of a moving real, because it allows to represent exactly the result of the operation **distance** between two moving points.

3.2 OPERATIONS

A complete description of the operations will not be given here. We refer the reader instead to (Güting et al., 1998) where a complete reference of them can be found. The design of the operations in that paper was done in a way that the resulting operators were as general as possible to avoid the proliferation of many different operators, capturing all the interesting phenomena with the chosen subset. The process followed to achieve this was to define in a first step a set of important and significant non-temporal operations that were overloaded in a second step by means of a process called *lifting* to achieve the same set of operations for temporal and non-temporal types. In addition to that, some other independent and important temporal and non-temporal operations were added to this set.

The concept of lifting consists of extending systematically non-temporal operations to work with the temporal variants of the argument types. For instance, if we have an operation called **distance** that yields the distance between two *point* values as a *real*, by means of the lifting process we overload the **distance** operation with three new signatures, two of them dealing with a *moving*(*point*) and a *point* and the third one dealing with two *moving*(*point*) values (always yielding the distance between them as a *moving*(*real*)).

Operations on Non-temporal Types

We distinguish six different categories:

- **Predicates:** in addition to the unary predicate isempty that yields true when the value to which it is applied is undefined (single values) or empty (set values), we find binary predicates by means of splitting predicates into three categories, set theory (=, ⊆, and ∩), order relationships (< and >) and topology (intersection of boundaries and interiors). Then, we consider for each category the possible relationships between two points, two sets and a point and a set in the respective space. The list of predicates found is the following: =, ≠, **intersects**, **inside**, <, ≤, >, ≥, **before**, **touches**, **attached**, **overlaps**, **on_border**, **in_interior**.
- **Set operations:** the common set operations plus operators with special names to yield values of a lower dimension. The intersection of two *region* values that only share a border yields an empty *region* value, but sometimes is interesting to retrieve this border as a *line* value, which is an object from a lower dimension. The operators are: **intersection**, **minus**, **union**, **crossings**, **touching_points**, **common_border**.
- **Aggregation:** operations reducing sets of values to only one. The operators are: **min**, **max**, **avg, center**, **single**.
- **Numeric properties of sets:** yielding well-known properties of sets. The list of operators consists of **no_components**, **size, duration, length, area**, **perimeter.**
- **Distance and direction: distance**, **direction**.
- **Base type specific operations:** the Boolean operations **and**, **or**, **not**.

Operations on Temporal Types

We have found the following categories:

- **Lifted operations:** We call kernel algebra to those operations defined in Section 3.2.1. We call *lifted operation* to a kernel algebra operation where one (or more) of the arguments has been converted in a temporal type. Therefore we have more than one lifted signature for each kernel operation, yielding always the temporal counterpart type of the original returned type.
- **Interaction with points and point sets in domain and range:** operations that relate values of moving types with values either in their domain or range. The operators are: **atinstant**, **atperiods**, **initial**, **final**, **present**, **at**, **atmin**, **atmax**, **passes**.
- **When operation:** restricts a time dependent value to the periods when its range value fulfils a property specified as a predicate.
- **Rate of change:** yield different measures of the rate of change of temporal types. The operations are: **derivative**, **speed**, **turn**, **velocity**.

3.3 EXAMPLES OF USE

The best way to get an idea of how this algebra works is to present a little example showing it. One of the possible fields where an algebra like this could be used is in meteorological applications. We show here a little example with very few features, it is not more than a toy application but sufficient to give a feeling of what the algebra can do.

Consider the following scheme:

```
country(name:string, area:region);
meteo_data(cloud_cover:moving(region),
           snow_cover:moving(region));
satellite(id:int, position:moving(point),
          photo_area:moving(region),
          photo_id:moving(int));
```

We store in the relation `country` the region that each country covers. In the relation `meteo_data`, we store the evolution of the cloud and snow cover over the earth for a period of time. We imagine that data come from a satellite that measures them in a continuous basis. In the relation `satellite`, we store the position of each satellite and the region that gets photographed by it. We also store a temporal identifier synchronized with the region photographed to retrieve the image from some other relation.

Possible queries to this database could be:

Q1: Retrieve all the existing photographs of a hurricane.

parameter eye:*moving*(*point*);
select *atperiods*(photo_id,
deftime(*at*(*in_interior*(eye,
photo_area),
TRUE)))
from satellite;

The time periods when the eye is in the interior of an existing photograph are computed and used to restrict to that periods the *moving*(*int*) that stores the identificators of the photographs. The resulting value can be used to extract the photographs from another relation (not shown here).

Q2: Show the evolution of the cloud cover in a particular country.

parameter countryname:*string*;
select *size*(*intersection*(country.area,
meteo_data.cloud_cover))
from meteo_data, country
where country = contryname;

From the Cartesian product of relations `country` and `meteo_data`, we keep only those tuples with the country we are studying in them. Then, the intersection between the area of the country and the cloud cover is computed, resulting in a *moving*(*region*) representing the evolution of the cloud-covered area in the country. Finally, the size of this moving region is computed and a *moving*(*real*) is printed as result.

Even though we have shown with these query examples that a system like is this is very desirable, in a first step we are not going to implement all its features. We will implement only a reduced set of data types and the operations that have them in their signature. The signature of the discrete model is now:

kinds BASE, SPATIAL, TIME, RANGE, TEMPORAL, UNIT, MAPPING

type constructors

	→	BASE	*int*, *real*, *string*, *bool*
	→	SPATIAL	*point*, *points*, *line*
	→	TIME	*instant*
BASE ∪ TIME	→	RANGE	*range*
BASE ∪ *point*	→	TEMPORAL	*intime*
BASE ∪ *point*	→	UNIT	*const*
	→	UNIT	*ureal*, *upoint*
UNIT	→	MAPPING	*mapping*

The reason for this change is that we want to show in a first step of implementation that this algebra can be used, and discover with its implementation some general algorithms that will be used by all the operators, as well as solving the problems that could occur in the implementation, without worrying too much with the most complex data types and algorithms for them. The reduced signature allows to use *moving*(*points*) only, not the rest of spatio-temporal data types. However, we think that this type is the most important one in a moving objects database. Even though we have reduced the set of types and operations, there are still applications that can benefit from it, as the following example shows.

In the last years, the number of accidents involving ships and whales in the sea corridor between the two main islands of the Canarian archipelago has raised. The problem occurs when a whale, after looking for food at deep waters, runs out of oxygen and rushes to the surface to breathe where it collides against a passing ship. The reason for this collision is that the whales are no longer sensitive to the noises produced by the ships, so they cannot know that the ship is there. A possible solution would be to use the sound produced by these animals to locate them and store their position together with the positions of the ships. The historical data resulting from this database can be used, for instance, to study the evolution of the group of whales living in that area or to plan safer new routes for ships. The database can also be used to detect the possible collisions and have the colliding ship change its bearing.

An scheme for this application could be:

```
whale(id:int, position:moving(point));
ship(id:int, position:moving(point));
```

The relation `whale` stores for each animal a *moving*(*point*) with its moving position. The relation `ship` stores the trips of the ships that cross the area. Possible queries could be:

Q1: Tell me the position and heading of whales that are nearer than a safety distance to the position where I am.

parameter safetydistance:*real*;
parameter myposition:*point*;
select *distance*(myposition,
atinstant(whale.position, NOW)),
direction(whale.position)
from whale
where *distance*(myposition,
atinstant(whale.position, NOW))
< safetydistance;

From the `whale` relation, the current positions of the whales are retrieved and the distance and direction of those whales whose distance is less than the safety distance are printed.

Q2: Could I have collided with a whale?

parameter mytrip:*int*;
select *intersection*(ship.position, whale.position)
from ship, whale
where ship.id = mytrip
and *isempty*(*at*(*intersects*(ship.position,
whale.position), TRUE))

= FALSE;

The Cartesian product of both relations is computed and only those tuples whose ship identificator is that of the ship queried are kept. Then, only the tuples whith an intersection point between my ship and the whale are kept. The lifted predicate **intersects** is used, and the results are restricted to the periods when the value is true. If the result is empty, there is no intersection between the ship and the whale. Finally, the intersection points between the ship and each whale are shown.

This little schema has many other applications, but a tool more powerful than a query language is needed. For instance, a statistical analysis tool to analyse the evolution of the points (e.g., trying to find a relation between the number of whales and the intensity of the traffic) or a browser for results to do this analysis in an informal manner.

4 ALGEBRA IMPLEMENTATION

After designing the algebra to be implemented and deciding the tool to implement it, data structures and algorithms to implement the algebra have to be designed. Even though data structures were roughly sketched in (Forlizzi et al., 1999) to show how easy the mapping between the discrete model and the implementation is, we will show here a more detailed description focused on the implementation.

4.1 DATA STRUCTURES

The basic types (*int*, *real*, *bool* and *string*) are implemented with a structure containing the correspondent programming language type (`int`, `float`, `bool`, and `char[256]` respectively) and a flag indicating whether the value is undefined. These are the data structures:

```
class int_t {
  int Value;
  bool Undefined;
}
class real_t {
  float Value;
  bool Undefined;
}
```

```
class bool_t {
  bool Value;
  bool Undefined;
}
class string_t {
  char[256] Value;
  bool Undefined;
}
```

The implementation of the spatial types is the following:

```
typedef point
float[3];
class point_t {
  point Value;
  bool Undefined;
}
class points_t {
  int NoOfPoints;
  DBArray Value;
}
```

```
typedef segment
point[3];
class line_t {
  int NoOfSegments;
  DBArray Value;
}
```

A *point* value consists of a structure with two float values for the co-ordinates and a flag indicating whether the value is undefined. A *points* value is represented by means of a structure with two members, one of them is the number of points that the value has, and the other one is a DBArray storing in each of its slots a pair of point co-ordinates. To ensure unique representations and hence efficient equality comparisons, the points in the value have to be lexicographically ordered.

A *line* value is a structure with the number of segments in the value and a DBArray storing a segment in each slot. Each segment consists of two points, the starting point and the ending point, and we require, in order to achieve uniqueness of the representation, the starting point to be smaller than the ending point. The segments in the value are stored in lexicographical order and it must be ensured for any pair of different segments that if they intersect they are not collinear (this restriction ensures that the maximal representation of the line value has been chosen). Efficient and robust functions to compute intersection and collinearity of segments can be found in (O'Rourke , 1994).

The time type, *instant*, is implemented easily with a structure containing a float value and another flag to indicate whether the value is ∞ or -∞ or none of them, and a flag to indicate whether the value is undefined. The type *intime* is the first non-constant type constructor that we find in this description, it is implemented with a structure containing a time value and a FLOB to store the representation of the type to which the type constructor is applied. For instance, if the structure is used to represent the value of the type *intime*(*point*), the FLOB will contain the memory representation of the type *point*. Here is an example where FLOBs show valuable, they allow the implementor to store size-varying objects while keeping the efficiency of access. If the object is small (as it happens with a point value), it will stored together with the tuple it belongs to and no other disk access is required to use the FLOB value; if the object is big, it will be stored in a different object, saving disk accesses when retrieving the tuple in case the object is not needed. These are the structures for the types:

```
typedef time_t
struct {
  float Time;
  bool Infty;
}
class instant_t {
  time_t Value;
  bool Undefined;
}
```

```
class intime_t {
  time_t Instant;
  FLOB Value;
}
```

A *range* value consists of a set of intervals, stored each one in a slot of a DBArray. Each interval is represented as a structure with a pair of FLOBs that will hold the limits of the interval, and two Boolean values to indicate whether the extremes of the interval are closed or open. To ensure uniqueness of the representation we store the intervals ordered by its starting value and we require for any pair of intervals to be disjoint and not adjacent.

```
typedef interval
struct {
  FLOB Start;
  FLOB End;
  bool LC;
  bool RC;
}
```

```
class range_t {
  DBArray Value;
}
```

A unit type constructor must be provided for those types that will have a temporal continuous version. In our

case, these types are *real* and *point*. The function that we use to describe the evolution of a real value in a temporal unit is a function like:

$$f(t) = \begin{cases} at^2 + bt + c & \text{if SqRt} = \text{false} \\ \sqrt{at^2 + bt + c} & \text{if SqRt} = \text{true} \end{cases} \quad (2)$$

The representation of the unit is:

```
typedef timeint_t
struct {
  time_t Start;
  time_t End;
  bool LC;
  bool RC;
}
```

```
class ureal_t {
  timeint_t
Interval;
  float a;
  float b;
  float c;
  bool SqRt;
}
```

An structure for a time interval is defined, with the starting and ending time instants of the interval and two flags to indicate whether the interval is left or right closed. The temporal unit consists (besides of a time interval) of a real value for each coefficient of the quadratic equation and a flag indicating whether the square root is used.

The function used for a moving point is a linear function describing the behaviour of each point co-ordinate independently. We can describe the function as:

$$f(t) = (f_x(t), f_y(t)), \text{ where } \begin{cases} f_x(t) = x_1 t + x_0 \\ f_y(t) = y_1 t + y_0 \end{cases} \quad (3)$$

The representation of the unit stores the time interval and four real values that stand for the four coefficients in the equations.

```
class upoint_t : unit_t {
  timeint_t Interval;
  float X1;
  float X0;
  float Y1;
  float Y0;
}
```

The type constructor *const* is a non-constant one. Its values store a time interval and a value of the data type used to construct the type. The value is stored in a FLOB and it could use, in turn, FLOBs to store its value. The existence of nested FLOBs proves valuable in this case, since the implementor does not have to provide an ad hoc solution to manage them.

```
class const_t {
  timeint_t Interval;
  FLOB Value;
}
```

Finally, the type constructor *mapping* stores a set of units, they may be from one of the types that we have just shown or from a new type introduced by another implementor who needs different basic and temporal types. For this reason, the memory representation of a mapping must be based in FLOBs.

```
class mapping_t {
  int NoOfUnits;
  DBArray UnitArray;
}
```

The structure consists of an integer storing the number of units that form the value and a DBArray with a unit value in each slot. For each FLOB that the unit type has, a FLOB cluster is created and all the FLOB values of the unit are stored in it. This can be seen in Fig. 6.

With this structure, all the FLOB values are clustered in a bigger structure, which is handled by the storage manager as such. We can expect that consecutive reading of the mapping value is more efficient than if each FLOB were stored and accessed independently. The units in the value are ordered by time instant, and they do not overlap; besides if two units are adjacent, their value must be different.

4.2 OPERATORS

We cannot describe here all the algorithms for the operators because once the polymorphic operators are expanded we get around a hundred operations. In addition to that, the non-temporal operations are either very simple to implement, or described in other papers, so we will not describe them here.

The important algorithms that we have to describe are those that deal with temporal types. A particularly important algorithm is that of *lifting*. Lifting is the process to obtain for an operator *op*(arg_1:$type_1$, ..., arg_n:$type_n$) all the possible operators where any of the arguments and the result type are substituted by its temporal counterpart. For instance, for the operator:

point × *point* → *real* **distance**

we obtain by means of lifting the following three operators:

moving(*point*) × *point* → *moving(real)* **distance**
point × *moving*(*point*) → *moving(real)* **distance**
moving(*point*)×*moving(point*) → *moving(real)* **distance**

They all yield the temporally changing distance between two points.

Lifted operations can be implemented using a general algorithm that we call *temporal sweeping*. We will describe here how it works for a lifted operation with two temporal arguments because all our operations have at most two arguments, and an operation with a temporal argument and a non-temporal one can be easily implemented with the operation for the general case as we describe below.

The conceptual idea of the algorithm that implements the lifting process is the following: given a function that works with two static values and two mappings, construct a third mapping as the result of applying in each time instant the given function to the values. It is obviously impossible to do this literally, the values are structured as sets of temporal units and we have to take them as basis.

The algorithm traverses both mappings extracting in each iteration a temporal unit from each one, and applying a unit-based version of the operations to them. This algorithm does not work either, because the time partition that each mapping imposes does not have to be the same and we could be applying the function to units with different time interval.

To solve this problem, a refined partition must be computed as a superset of both partitions (as shown in Fig. 7) and then applied to both mapping to achieve two new mappings with the same time partition. The resulting mapping may need a normalizing step after its

computation finishes because there may be adjacent units with the same value, which are not allowed.

An optimization to the algorithm consists in computing the refined partition *on the fly*. Instead of traversing both mapping values three times to build the refined partition, apply it to both values and then apply the function, we use a *temporal sweeping*. Each iteration of the algorithm starts with a unit that has to be processed. Then, another unit must be extracted from one of the values. Fig. 8 shows the possible temporal relationships between these units. The first row is the unit to be processed in the current iteration of the algorithm, the second row represents the unit extracted in the iteration, and the third one shows the time intervals of the refined partition and below them the units to which the unit-based function must be applied. We show with an arrow the unit that has to be processed in the following operation. The unit-based function must be prepared to deal with unit-arguments that are NULL. The second case can be implemented using the third one because once the first time interval is computed we have the same situation as in the third case. The fourth case is a particular case of the second one, no implementation is needed for it.

Not all lifted operations can be implemented with this algorithm, only those whose arguments are all mappings. Nevertheless, this algorithm is general and new operations can be implemented easily with it just providing an appropriate unit-based function.

In addition to that, this algorithm can be used to implement *partially lifted functions*. A partially lifted function is a function with only one of its arguments substituted by its temporal counterpart. This type of functions can be implemented with a special algorithm developed for it or with the general algorithm that we have just described. To do that, the value passed in the static argument is substituted by a static moving object (i.e., a moving object that has the same value for all the time axis) and both mappings are passed to the algorithm. The only extra operation done with this approach and not with the former one is the conversion to a static moving value, which is not a specially expensive process.

We have not mentioned robustness and topological problems in this description. These problems appear due to the use of limited precision numbers as computer floating point numbers. The imprecision accumulated could lead to errors in operations, such as yielding a intersection point that does not belong to any of the lines intersected. This is a problem unsolved at the moment, and the most conservative solution (with a serious drawback in the efficiency) is to use exact computations. In our case, we use exact real numbers in all the operations dealing with geometries and computer real numbers in the rest of cases.

5 CONCLUSIONS AND FUTURE WORK

In this paper, we have reviewed the steps that led to the implementation of a spatio-temporal algebra. First we have shown SECONDO, a powerful database system that can be extended easily in many different ways keeping an easy-to-use interface. Then, we have shown the algebra that we want to implement. It was described first in (Erwig et al., 1999), (Güting et al., 1998) and (Forlizzi et al., 1999), but we have condensed here the main issues in an informal way. Finally, we have shown the most important details of the implementation, the data structures and an algorithm to implement the temporal lifted operations.

This paper is not the end of the evolution but the beginning of a new phase, now it is time to analyse whether the algebra can be used with real world applications, to provide friendly query interfaces for the user, and to implement the full set of types. As future work, we can remark:

- Implementation of the full set of types: the implementation of the temporal units for *region* and *line* will raise the need for new complex algorithms for unit-based operations such as intersection. Besides, with the full set of types more complex applications can be implemented with the database system.
- Performance analysis: the efficiency, speed and other parameters of the implementation have to be tested with real or artificial data sets.
- Robustness analysis: different solutions to the robustness problem have to be tested to analyse their advantages and drawbacks (e.g. space vs. time overhead, etc.).
- Spatio-temporal objects browser: a browser that makes possible the visualization of continuously evolving spatio-temporal objects has to be implemented to allow the user to deal with the result in a much more comfortable way.

REFERENCES

M.J. Carey, D.J. DeWitt, M.J. Franklin, N.E. Hall, M.L. McAuliffe, Je.F. Naughton, D.T. Schuh, M.H. Solomon, C.K. Tan, O.G. Tsatalos, S.J. White, M.J. Zwilling. Shoring Up Persistent Applications. In *Proc. of the 199 ACM SIGMOD Intl. Conference*, pp. 383-394, Minneapolis, May 1994.

J. Chomicki and P. Z. Revesz. Constraint-Based Interoperability of Spatiotemporal Databases. In *Proceedings of the 5th International Symposium on Large Spatial Databases*, p.142-161, 1997

S. Dieker and R.H. Güting. Plug and Play with Query Algebras: SECONDO. A Generic DBMS Development Environment. FernUniversität Hagen, Informatik-Report 249, February 1999.

S. Dieker and R.H. Güting. Efficient Handling of Tuples with Embedded Large Objects. *Data & Knowledge Engineering*, to appear.

M. Erwig, R. H. Güting, M. Schneider, and M. Vazirgiannis. Spatio-Temporal Data Types: An Approach to Modeling and Querying Moving Objects in Databases. *GeoInformatica*, Vol. 3, No. 3, 1999, to appear.

S. Grumbach, P.Rigaux and L. Segoufin. The DEDALE System for Complex Spatial Queries. In

Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD-98), ACM SIGMOD Record, Vol. 27,2, pp. 213-224, ACM Press, June 1-4 1998.

L. Forlizzi, R.H. Güting, E. Nardelli, M. Schneider. A Data Model and Data Structures for Moving Objects Databases. FernUniversität Hagen, Informatik-Report, 1999.

R. H. Güting. Second-Order Signature: A Tool for Specifying Data Models, Query Processing, and Optimization. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pp. 277-286, 26-28 May 1993.

R.H. Güting, M.H. Boehlen, M. Erwig, C.S. Jensen, N.A. Lorentzos, M. Schneider, and M. Vazirgiannis. A Foundation for Representing and Querying Moving Objects. FernUniversität Hagen, Informatik-Report 238, September 1998.

R.H. Güting, S. Dieker, C. Freundorfer, L. Becker, and H. Schenk, SECONDO/QP: Implementation of a Generic Query Processor. In *10th Int. Conf. on Database and Expert System Applications (DEXA'99)*, LNCS 1677, Springer Verlag, 66-87, 1999.

G. Hunter and I. Williamson. The Development of a Historical Digital Cadastral Database. *International Journal of Geographical Information Systems*, 4(2), pp. 169-179, 1990.

G. Langran and N. Chrisman. A Framework for Temporal Geographic Information. *Cartographica*, 25(3), pp. 1-14, 1988.

J. O'Rourke. Computational geometry in C. Cambridge Univ. Press, 1994.

D.J. Peuquet and E. Wentz. An Approach for Time-Based Analysis of Spatiotemporal Data. In *Sixth International Symposium on Spatial Data Handling*, Vol. 1, pp. 489-504, 1994.

D.J. Peuquet and N. Duan. An event-based spatiotemporal data model (ESTDM) for temporal analysis of geographical data. *International Journal of Geographical Information Systems*, 9(1), pp. 7-24, 1995.

D. Pfoser and C. S. Jensen. Capturing the Uncertainty of Moving-Object Representations. In *Proc. of the 6th Intl. Symposium on Spatial Databases.* LNCS, Vol. 1651, 1999.

A. Sistla, O. Wolfton, S. Chamberlain and S. Dao. Modeling and Querying Moving Objects. In *Proceedings of the 13th International Conference on Data Engineering* (ICDE'97), pp. 422-433, IEEE, April 1997.

Y. Theodoridis, T. Sellis, A. Papadopoulos, and Y. Manolopoulos. Specifications for Efficient Indexing in Spatiotemporal Databases. In *Proceeding of the 10th International Conference on Scientific and Statistical Database Management (SSDBM98)*, July 1-3, 1998.

N. Tryfona and T, Hadzilacos. Logical Data Modeling of Spatio-Temporal Applications: Definitions and a Model. In *Proc. of the Intl. Database Engineering and Applications Symposium*, 1997.

O. Wolfson, B. Xu, S. Chamberlain, L. Jiang. Moving objects Databases: Issues and Solutions. In *Proceeding of the 10th International Conference on Scientific and Statistical Database Management (SSDBM98)*, July 1-3, 1998.

O. Wolfson, S. Chamberlain, S. Dao, L. Jiang, and G. Mendez. Cost and Imprecision in Modeling The Position of Moving Objects. In *Proc. of the 14th Intl. Conference on Data Engineering*, pages 588-596, 1998.

FIGURES

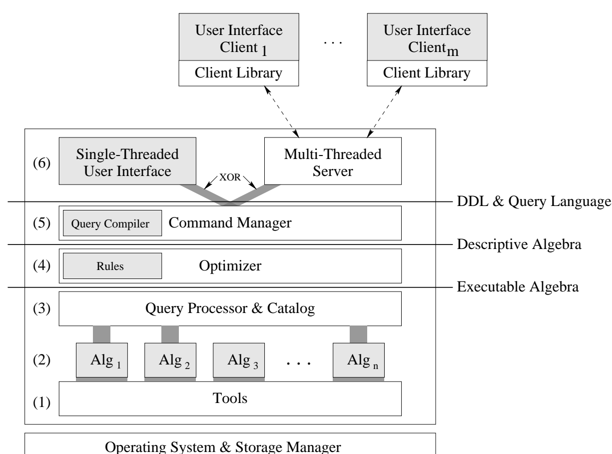

Fig. 1 SECONDO architecture

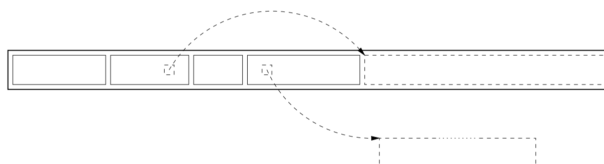

Fig. 2 A tuple with FLOBs

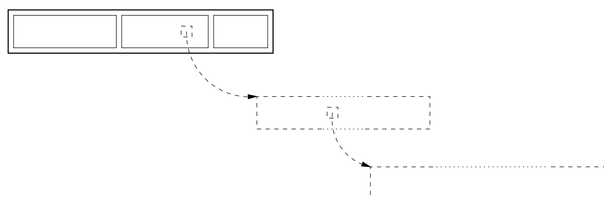

Fig. 3 A nested FLOB

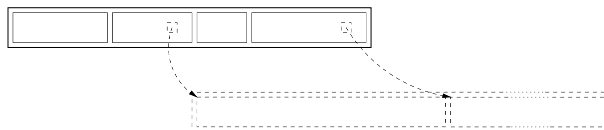

Fig. 4 A FLOB cluster

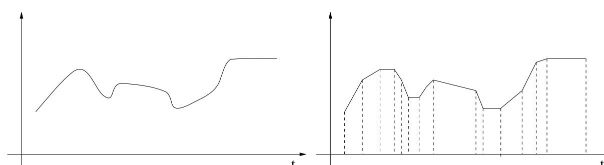

Fig. 5 A value and its discrete representation

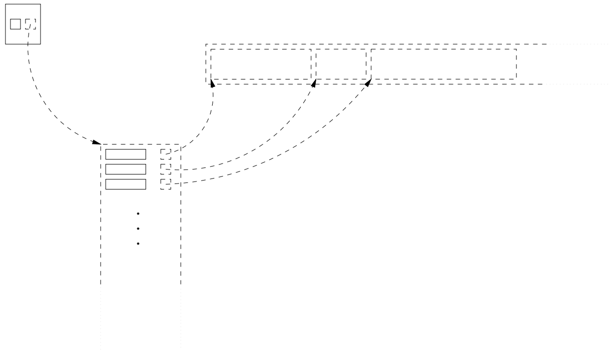

Fig. 6 A mapping structure

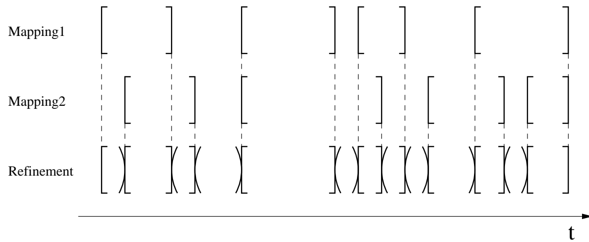

Fig. 7 Computing a refined time partition

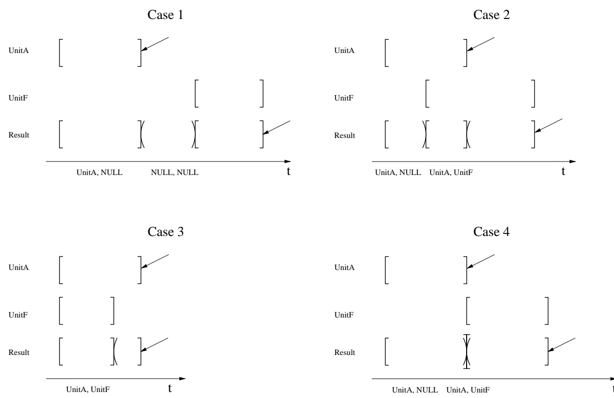

Fig. 8 Temporal relationships between units

TABLES

Abstract model (Descriptive level)	**Discrete model (Executable level)**
moving(*int*)	*mapping*(*const*(*int*))
moving(*string*)	*mapping*(*const*(*string*))
moving(*bool*)	*mapping*(*const*(*bool*))
moving(*real*)	*mapping*(*ureal*)
moving(*point*)	*mapping*(*upoint*)
moving(*points*)	*mapping*(*upoints*)
moving(*line*)	*mapping*(*uline*)
moving(*region*)	*mapping*(*uregion*)

Table 1 Equivalence between abstract and discrete model types