Representation of Periodic Moving Objects in Databases

Thomas Behr
thomas.behr@fernuni-hagen.de

Victor Teixeira de Almeida
victor.almeida@fernuni-hagen.de

Ralf Hartmut Güting
rhg@fernuni-hagen.de

FernUniversität in Hagen
Faculty of Mathematics and Computer Science
Database Systems for New Applications
D-58084 Hagen, Germany

ABSTRACT

In the real world, lots of objects with changing position can be found. Some of them repeat the same movement several times, called periodic movements. Examples include airplanes, trains, planets, and marine turtles. This paper describes a model for representing the periodic movements to be stored in a database system, exploiting the information about the repetitions. The model is generic enough to represent any kind of movement, not being restricted to objects with repetitions in their movement. We present algorithms to detect the repetitions and to convert to the periodic representation as well as the implementation of some operations on such representation. We claim that the data volume can be drastically reduced when repetitions in movements occur. Moreover, some operations can take advantage on the data representation and therefore have their performance improved. We show, in an experimental evaluation against the so-called flat representation, that the approach presented in this paper significantly improves the performance of query processing in a database system when dealing with objects with some periodic movement. We also show that, for the worst case where the objects do not follow any periodic movement at all, our approach still performs acceptably.

1. INTRODUCTION

Commercial database management systems (DBMS) offer the possibility to handle more than standard data types like integers, real numbers, and short strings. They can also store complex types like pictures, videos, XML files, spatial data, and spatio-temporal data. Examples include Oracle *inter*Media, Oracle Spatial, IBM DB2 XML Extender, IBM DB2 Spatial Extender, Full-Text Search of Microsoft SQL Server, etc. Independently of the technique for including a new data type into an DBMS, a model must be designed. Such a model is developed in this paper for periodic moving objects.

In the real world, we can see that many objects change their properties in a periodic way. As an example a (german) traffic light has the state 'green' for a given time, then it switches to the states 'yellow', 'red', and 'red/yellow'. This sequence is repeated many times per day. During the night, the traffic light may be switched off. We can model each lamp of the traffic light as a periodically changing boolean value, or all the lamps together as a periodically changing integer value, where each lamp has a bit in the integer number value. Another example is a ticker in a shop window. Such devices can be seen as periodically changing strings.

The main application field of the model developed here is for periodic moving objects. These are objects which change their positions continuously and periodically over time. Examples include planets, trains, aircrafts as well as all other public transportation vehicles using a time table. All these objects can be represented as periodic moving points within our model.

In [13, 14, 15], a model for describing continuously changing spatial data in databases is presented. This model stores the current state of an object together with its motion vector for describing the object properties at the current time and in the near future. This approach reduces the number of necessary updates drastically. An update is required when the difference of the actual position and the expected position of this object exceeds a certain threshold. By a user query, the expected position of the object is computed using the motion vector. Because repetitions can be only detected when the history of an object is known, this model is inappropriate for storing periodic moving objects.

A data model capturing complete histories of continuous movement together with a related query language was developed in [8, 6, 3]. An algebra is provided with data types such as *moving point* and *moving region* together with a comprehensive set of operations. This model is capable to store periodic movement, but is not able to take advantage from the fact that repetitions exist. Since our work is compared against this data representation, we review the main concepts of this spatio-temporal data model in Section 2.

In [10], cyclic changes of objects are discussed, where cyclic intervals are introduced as well as relations between them using a binary matrix. However, nested repetitions, which arise frequently in practice, are not allowed. Spatial objects are not handled, and therefore the proposed ap-

.

proach does not seem to be appropriate for defining periodic moving objects.

An approach for modeling changing n-dimensional rectangles is described in [2]. Each coordinate of such a rectangle is defined by a function of time. The functions of all coordinates as well as an interval build a unit of a parametric rectangle. A set of such units represents the complete movement. Because the functions are not restricted to be linear, the changes of the rectangles are very flexible. The periodicity is stored on every unit allowing periodic movements of parametric rectangles. However, nesting of periodic movements is not allowed, restricting the movement to simple periodic movements. The periodic movement example used in Section 3 cannot be represented using periodic parametric rectangles.

Periodic rectangles are extended to support acyclic movements in the data model called periodic spatiotemporal objects (PSO) databases in [11, 12]. An acycle periodic movement is the composition of a cycle periodic and a linear movement, i.e. is a function $f(t) = g(t) + h(t)$, where $g(t)$ is a cyclic function and $h(t)$ is a linear function. Compared to the parametric rectangle representation, cyclic functions are stored instead of the combination of simple functions and the periodicity. Again, the PSO model is not able to represent complex movements with nested periodicities such as the simple example in Section 3. Acyclic functions are not handled in our approach and are subject to future work.

In [1] the author introduces a model for the representation of periodic events. By defining temporal relations between events it is possible to represent the order of events and periodic repetitions of them. Here, also nested repetitions are representable. Unfortunately, this model does not allow continuous movement, which is a basic requirement for representing moving objects.

In this paper, we describe a model for representing the periodic movements. The main focus is to be able to store and efficiently query such movements in a database system, exploiting the information about the repetitions. The model presented is generic enough to represent any kind of movement, not being restricted to objects with repetitions in their movement. Moreover complex repetitions are allowed with the notion of nested repetitions.

We present algorithms to detect the repetitions from historical data sets and to convert from what we call the flat representation to the periodic representation. Algorithms for the implementation of some operations on such new representation are also provided.

We claim that the data volume can be drastically reduced when repetitions in movements occur. Moreover, some operations can take advantage on the data representation and therefore have their performance improved. We show, in an experimental evaluation against the flat representation that the approach presented in this paper significantly improves the performance of query processing in a database system when dealing with objects with some periodic movement. We also show that, for the worst case where the objects do not follow any periodic movement at all, our approach still performs acceptably.

This paper is organized as follows: Section 2 reviews the spatio-temporal model (flat representation) presented in [8, 6, 3]. In Section 3 we introduce the representation of the periodic movements together with algorithms to detect the repetitions from historical data sets and to convert from them to the periodic representation. Algorithms of some of the operations on periodic moving objects are also addressed in this section. Section 4 addresses some implementation issues in the SECONDO extensible database system ([4, 7, 9]), while in Section 5 we evaluate our approach using data sets with some and without any periodicity. Finally, Section 6 concludes the paper pointing out some future work.

2. MOVING OBJECTS REPRESENTATION

In this section we review the system for representing moving objects presented in [8, 6, 3]. 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 [6]) may be embedded as attribute types into OO- or ORDBMS, or implemented as extension packages into extensible DBMS.

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" function, the so-called *temporal function* ι. 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)).

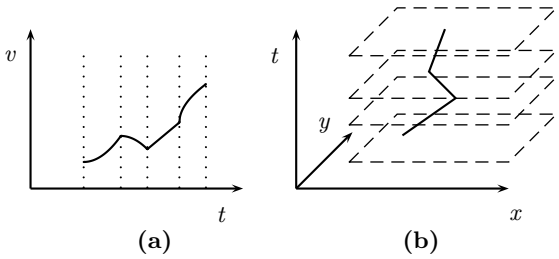

Figure 1: Sliced representation of (a) a moving real and (b) a moving point

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 [8].

Over these data types, a large set of operations is defined in [3]. 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 are:

mpoint	→ *line*	**trajectory**
mpoint × *periods*	→ *bool*	**present**
mpoint × *instant*	→ *bool*	**present**
mpoint	→ *periods*	**deftime**
mpoint × *region*	→ *mpoint*	**at**
mpoint × *region*	→ *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. Efficient algorithms for the operations are presented in [3].

3. PERIODIC MOVEMENT REPRESENTATION

In this section we present the model for representing periodic movements. Section 3.1 discusses the representation of time, while Section 3.2 presents the representation of the periodic movements. Finally, algorithms to detect repetitions and to convert from the flat representation to the one presented in Section 3.2 are given in Section 3.3.

The following simple example is used throughout this section for ease of understanding the concepts and algorithms. Assume an underground train starts at 8:00 AM moving from station A to station B within 10 minutes. Afterwards it goes to station C in 13 minutes. Then it comes back to station A using the same trajectory and times. At every station, the train stops for 1 minute. This is repeated until 4:20 PM. The train runs from Monday to Friday. At the weekend it stays at station A. The first run of this train was at January, 1st. 2000 and its lifetime is 10 years.

For simplicity, we assume that the rails between the stations can be modeled as a single line segment and the train goes with a constant speed. Furthermore, we exclude any maintenance work in the train. The location of every station $S \in \{A, B, C\}$ is (x_S, y_S), but for simplicity we use just the abbreviation S in our example.

3.1 Representation of Time

In this paper we use two data types for representing time, namely *instant* and *duration*. An *instant* value represents a point in time, also called a timestamp.

An instant can just be viewed as an element of $\mathbb{R}$ as in [6]. Unfortunately $\mathbb{R}$ is an infinite set and cannot be completely covered in a computer. For this reason we decided to use a discrete representation of time. Any two consecutive instants have the same distance and we have selected one millisecond as the value for this distance. Besides the numerical robustness, we can define an adjacency relation between two instances. An *instant* value is adjacent to another one iff their absolute distance is exactly one (millisecond).

Therefore, the *instant* type is modeled just by an integer value. In fact, in the actual implementation of the *instant* type we use two integer values, because the range of a single integer value in a resolution of one millisecond only allows one to represent a range of about 50 days, which is too short for practical applications.

To get a human readable date time value (given in a *year-month-day-hour:minute:second.milliseconds* format), we interpret the value of this integer as the distance in milliseconds to an anchor date and convert the instant into the Gregorian format with additional time information. We decided to use as the anchor date "2000-01-03-00:00:00.0000". A Monday is selected in order to facilitate the computation of weekdays.

A value of type *duration* describes a directed distance between two *instant* values. Negative values are allowed ensuring closure of operations. A *duration* value is also represented using integers holding the count of milliseconds of the duration.

Arithmetic operations are allowed using these data types, which are shown below:

instant × *instant*	→ *duration*	−
duration × *duration*	→ *duration*	−
instant × *duration*	→ *instant*	−
instant × *duration*	→ *instant*	+
duration × *duration*	→ *duration*	+
duration × *int*	→ *duration*	∗
duration × *real*	→ *duration*	∗
duration × *duration*	→ *int*	/
duration × *duration*	→ *duration*	%

One can create a *duration* value by calculating the difference between two *instant* values. One can add or subtract a *duration* value from an *instant* to move the timestamp to the future or to the past direction, respectively. Arithmetic operations on *duration* values are also provided.

Usually, an interval is described by its two end points with additional flags whether it is open or closed in the respective end point. This interval representation describes a fixed (also called *anchored*) period of time.

For modeling repetitions, is should be possible to reuse a time interval at different times. Therefore in this model we move to *relative* time intervals. We still keep the flags about open or closed end points. The representation of a relative time interval is

$$RelInterval = \{(l, lc, rc) |\ l \in \underline{duration}, lc, rc \in \underline{bool}\}$$

Similar to [6] we define for each data type α its domain of possible values, called its *carrier set*, D_α. Hence for type *relinterval* we have

$$D_{\underline{relinterval}} = RelInterval$$

To fix such an interval in time, we assign an *instant* as an anchor to it, which represents the start time of the interval. Note that the anchor is not a part of the interval itself. The assignment of an anchor t to an interval I is denoted as $(I \leftarrow t)$. This corresponds to a conversion from a relative interval into a fixed interval as follows.

$$(t, t+l, lc, rc) = ((l, lc, rc) \leftarrow t)$$

where $(t, t+l, lc, rc) \in Interval(\underline{instant})$, $(l, lc, rc) \in RelInterval$, and $t \in \underline{instant}$.

We define some operations between relative intervals which are required in the remainder of this paper.

$\underline{relinterval} \times \underline{duration}$	$\rightarrow \underline{bool}$	**contains**
$\underline{relinterval} \times \underline{real}$	$\rightarrow \underline{relinterval}$	$*$
$\underline{relinterval} \times \underline{duration}$	$\rightarrow \underline{real}$	**fraction**

Their respective semantics are

$$contains(I,\ d) := \begin{cases} \text{false} & \text{if } d < 0 \vee d > I.l \\ \text{true} & \text{if } 0 < d < I.l \\ I.lc & \text{if } d = 0 \\ I.rc & \text{if } d = I.l \end{cases}$$

$$(l,\ lc,\ rc)\ *\ f := (l * f,\ lc,\ rc)$$

$$fraction(\ (l,\ lc,\ rc),\ \mathrm{d}\ ) := \begin{cases} d/l & \text{if } 0 < d < l \\ 0 & \text{if } l = 0 \wedge d = 0\ \wedge lc \\ 1 & \text{if } l = d \wedge rc \\ \perp & \text{otherwise} \end{cases}$$

In particular **contains** checks if the interval contains a certain duration, the $*$ operator increases the duration of a relative interval by a given factor, and the **fraction** operation returns the factor which decreases a relative interval to a given duration, or undefined ($\perp$) if the given duration is not contained in the interval.

3.2 Representation of Movements

In Section 2 we have already mentioned the sliced representation presented in [8]. The movement of an object is divided into small slices where, in each slice, a simple temporal function is applied. The whole movement is represented by an (ordered) set of disjoint slices in the time dimension. This is what we call the flat representation.

For the representation of the periodic movements we propose to reuse units that belong to some repetition, in order to avoid repeting the units in the representation. Compared to the flat representation, the periodic movement representation is a tree containing nodes of the following types: basic, composite, and repetition nodes, which represent three different kinds of movements that are explained in Sections 3.2.1, 3.2.2, and 3.2.3, respectively. Additionally, a root node is provided for the complete movement in Section 3.2.4.

3.2.1 Basic Movement

A basic movement corresponds to a unit similar to the one in [8], but using a relative interval. The general unit representation is the following:

$Unit(S) = RelInterval \times S$

where S is a set whose values are discrete representations of unit functions.

An instantiation of this generic temporal unit must provide a set S_α. The unit function represented by a value of S_α is given by the *temporal evaluation function*

$\iota_\alpha : S_\alpha \times D_{duration} \rightarrow D_\alpha$

which must also be provided as part of the instantiation. Note that the temporal function ι accepts now a *duration* value instead of an *instant*.

These definitions will become clearer when we instantiate S_α and ι_α for the different data types. For discretely changing objects, e.g. $\alpha =$ *bool*, *int*, or *string*, the representation of a unit is

$$D_{u\alpha} = RelInterval \times D_\alpha$$

i.e. a relative interval and a value of type α. One should note that now we allow a unit to contain an undefined value of α. The need for this will be clearer in the following sections, but what we can briefly comment here is that, since we use relative intervals and only one anchor for the whole object, one needs to fill the gaps where no movement exists with units containing empty values. Units must exist in these gaps because otherwise it would not be possible to compute the absolute times for the rest of the movement.

The temporal function is

$$\text{For } (i, v) \in D_{u\alpha}, p \in D_{\underline{duration}}$$

$$\iota_\alpha((i, v), p) = \begin{cases} v & \text{if } i \textbf{ contains } p \\ \perp & \text{otherwise} \end{cases}$$

For the *real* data type, a non-constant and non-linear function is used. It contains a function representing a quadratic polynomial or a square root of such (see [6]). Within the unit, only the coefficients of the polynomial and a boolean flag indicating the use of the square root are stored.

Most of the movements of objects are representable by a linear approximation. The advantage is that continuous change can be approximately modeled together with a simple implementation. We will use units with a linear function for the representation of the movements of spatial objects, moving objects for short.

We represent a function describing linear movement inside an unit just by the state of the moving object at the begining and the end of the interval. When a value within the interval is needed, linear interpolation between these values is used. A unit for moving points is modeled as

$$D_{upoint} = RelInterval \times D_{point} \times D_{point}$$

The ι_{point} function is then

$$\text{For } (i, (x_1, y_1), (x_2, y_2)) \in D_{upoint}, p \in D_{\underline{duration}}$$

$$\iota_{point}((i, (x_1, y_1), (x_2, y_2)), p) = \begin{cases} (x_r, y_r) & \text{if } p \in i \\ \perp & \text{otherwise} \end{cases}$$

where

$x_r = x_1 + \textbf{fraction}(i, p)(x_2 - x_1)$

$y_r = y_1 + \textbf{fraction}(i, p)(y_2 - y_1)$.

All units are stored together in an array structure (see Section 4) without any temporal order. In our example, the units are

Index	Unit	Remark
0	((600 F T) A B)	going from A to B
1	((60 F T) B B)	staying at B
2	((780 F T) B C)	going from B to C
3	((60 F T) C C)	staying at C
4	((780 F T) C B)	going from C to B
5	((600 F T) B A)	going from B to A
6	((60 F T) A A)	staying at A
7	((56400 F T) A A)	staying at A (night)
8	((142800 F T) A A)	staying at A (weekend)

Here the durations in the relintervals are given in seconds rather than milliseconds for better readability.

The basic node of the unit index 0 is shown in Figure 2. It contains the node type 'L', which stands for *link*, and the index of the unit. One should note that this kind of node is present just at the conceptual level, and does not exist in the implementation. A father node pointing to a simple node actually points directly to the unit of the simple node (stored in the array).

Figure 2: A basic movement node

3.2.2 Periodic Movement

A periodic movement describes a repetition of a single movement, which can be a basic or a composite movement (Section 3.2.3). The number of repetitions is stored in the periodic movement node and must be at least two. If there is no repetition, then a periodic movement is not necessary.

Since a periodic movement contains only a single sub-movement, we do not allow this sub-movement to be periodic too, because in this case both nodes could be merged into only one with their numbers of repetitions multiplied.

In our implementation, additional (redundant) information, e.g. the total duration of the movement, is stored to speed up some operations.

Let $I = (l, lc, rc)$ be the complete interval of the sub-movement. We require that lc and rc are different, or formally, $lc \oplus rc$. This is due to the fact that conceptually the sub-movements are concatenated r times to form the complete movement, where r is the number of repetitions of the periodic movement, and to be able to concatenate them, their interval boundaries should not match.

The node of the periodic movement for a daily trip of the train in our example is shown in Figure 3. It contains the node type 'R', which stands for *repetition*, the number of repetitions, and a pointer to its son node, represented here by an arrow. Every two-way trip takes 50 minutes and the train makes 10 two-way trips per day.

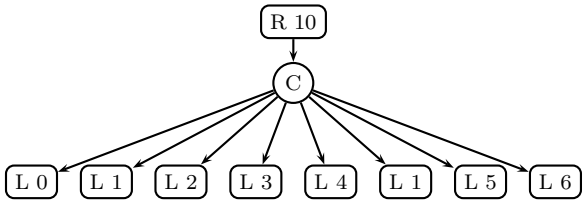

Figure 3: A periodic movement node

3.2.3 Composite Movement

A composite movement summarizes two or more other movements. It can be viewed as a list of sub-movements where each one is a basic or periodic movement. It is not allowed for a sub-movement itself to be a composite one, in order to avoid chains of composite movements. This does not reduce the power of this model because we can just include the sub-movements of a composite node directly into the father composite node.

Within a composite movement, the order of the contained sub-movements determines the temporal order of the movements. Hence, by using relative intervals, holes in the definition time of a moving object are represented by units containing undefined values.

For two consecutive sub-movements S_i and S_j, $j = i + 1$, with relative intervals $I_i = (l_i, lc_i, rc_i)$ and $I_j = (l_j, lc_j, rc_j)$, respectively, the following conditions must hold:

- $rc_i \oplus lc_j$
- $summarize(S_i, S_j) = \perp$ (explained below)

The first condition ensures connected intervals and the second one ensures a unique representation of a composite movement.

The *summarize* function tries to connect two movements into only one. Three cases can occur depending on the types of the sub-movements:

- S_i and S_j are basic movements. The summarize function in this case tries to extend the interval of the second argument with the interval of the first argument if the temporal function is able to represent both units together.

More formally, we have

$$\begin{array}{l} \text{For } u_i = ((l_i, lc_i, rc_i), f_i) \in D_{\underline{u\alpha}}, \\ \qquad u_j = ((l_j, lc_j, rc_j), f_j) \in D_{\underline{u\alpha}}, \\ summarize(u_i, u_j) \\ \qquad = \begin{cases} ((l_i + l_j, lc_i, rc_j), f_i) & \text{if * holds} \\ \perp & \text{otherwise} \end{cases} \end{array}$$

where
$* : rc_i \oplus lc_j \ \wedge \ \forall x \in [0, l_j] : \iota_\alpha(u_i, l_i + x) = \iota_\alpha(u_j, x)$

- S_i is a basic movement and S_j is a periodic movement and vice-versa. Let r_j be the number of repetitions in S_j. If S_j is a repetition of the basic movement S_i, then the result of the *summarize* function is a periodic movement of the basic movement S_i with the number of repetitions equal to $r_j + 1$. Otherwise an undefined movement ($\perp$) is returned.
- S_i and S_j are both periodic movements. Let S_i' and S_j' be the movements that are repeated in S_i and S_j, and let r_i and r_j be the number of repetitions, respectively. If S_i' and S_j' are equal movements, then the result of the *summarize* function is a periodic movement of the movement S_i' (or S_j') with the number of repetitions equal to $r_i + r_j$. Otherwise an undefined movement ($\perp$) is returned.

The total duration of a composite move is the sum of the durations of the intervals of its components. As done for

the periodic movement, this value is stored in the composite movement node to speed up computations.

The node of the composite move for a one-way trip of the train in our example is shown in Figure 4. It contains the node type 'C', which stands for *composite*, and pointers to its son nodes, represented here by arrows.

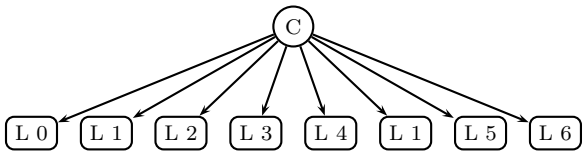

Figure 4: A composite movement node

3.2.4 The Complete Representation

Until now, all presented movements use relative intervals. This means by combining basic, composite, and periodic movements, we can define complete movements free in time. The complete movement is a node that defines a global anchor for its sub-move. Thereby anchors for all involved sub-movements are implicitly defined. We treat the movement tree in depth-first manner, i.e. the first movement starts at the global anchor, and any next movement starts at the end of its previous movement, i.e. the start of the previous movement plus its duration. The complete representation of our example is shown in Figure 5.

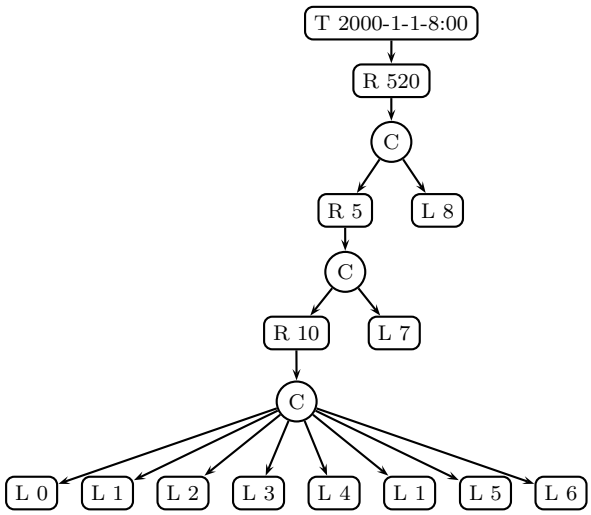

Figure 5: The complete (tree) representation of the train example.

This example illustrates the advantage of the model presented in this paper in comparison with the flat representation with fixed intervals. When we expand this tree in Figure 5 to the flat representation, we have to store more than 200,000 units. In the tree representation we have only 17 nodes and only 10 units.

3.3 Conversion from a Flat Representation

When the data set is created from a timetable, it is straightforward to detect the movement repetitions. But in some cases, the data come from observations, for example locations of moving objects determined by GPS devices. In these cases, data come in the flat representation. In this section, we provide an algorithm for detecting periods in a flat representation, i.e. to create the tree representation from data in the flat representation.

Instead of using a customized algorithm for each data type, we have developed an algorithm to find repetitions in a list of integer numbers. The algorithm therefore is performed in three steps:

1. Creation of an array of integers from the flat representation of a moving object (Section 3.3.1).
2. Detection of repetitions within the array of integers (Section 3.3.2).
3. Construction of a periodic moving object tree representation from the result of the last step (Section 3.3.3).

3.3.1 Converting a Moving Object into an Array of Integers

The algorithm used for constructing an integer array from a flat representation is shown in Algorithm 1. It tries to find holes in the temporal dimension and fills them with units with relative intervals with undefined values. For every unit, it tries to find equal ones in the result array of units U and if so, an equal integer number is added into the resulting array of integers R. In fact, the index of the unit in U is used as such integer number.

Algorithm 1 Algorithm *ConstructIntegerArray*

INPUT: a moving object o in the flat representation
OUTPUT: an array of integers R, an array of units U

```
1: for every unit u_i ∈ o do
2:    if there is a hole between u_i and the last unit u_{i-1}
      then
3:       let u' be the unit filling the temporal hole with an
         undefined value
4:       if u' ∉ U then
5:          insert u' into U
6:       end if
7:       insert into R the index of u' in the array U
8:    end if
9:    if  u_i ∉ U  then
10:      insert u_i into U
11:   end if
12:   insert into R the index of u_i in the array U
13: end for
```

Given that hash tables are used for searching equal units, this algorithm performs in $O(n)$ steps, where n is the number of units in the flat representation.

The problem with this algorithm is that, in some cases, no equality of units exists. A good example for that behaviour is when trying to convert the data of the train in our example that is captured using a GPS receiver. No periodicity can be found because of small differences in time with respect to the time table and by measurement errors of the GPS system. In this case, we propose to change the value of the measured time to a raster grid. For the spatial information we also use a grid with the cells dividing the railway lines. By changing the size of these raster grids, we can control the accuracy of the algorithm and the number of periods found.

3.3.2 Detecting Repetitions within an Integer Array

This algorithm is shown in Algorithm 2. It receives an integer array and constructs a structure very similar to the representation of periodic changing objects. This means, the result of the algorithm is a tree consisting of *basic*, *composite*, and *periodic* nodes.

Algorithm 2 Algorithm *DetectRepetitions*

INPUT: an array of integers I
OUTPUT: a tree of integers T

```
1: Let T be a composite node containing as sons basic
   nodes with all integer values of I
2: len ← 1
3: while len < (# of sons of T)/2 do
4:    Let N be the first son of T
5:    while N is not the last son of T do
6:       Find the repetitions of length len from node N
7:       if there are any repetitions then
8:          Let C be a compositive node with len sons con-
            taining the node N and the len−1 following ones,
            i.e. the pattern that is repeated
9:          Assign a unique identifier to the node C. If a
            node C' equal to C was already created, just re-
            use the identifier of C'. Otherwise generate a new
            unique identifier and assign it to C.
10:         Let P be a periodic node with only son C and
            containing the number of repetitions calculated
            before
11:         Assign a unique identifier to the node P in the
            same way as done for the node C
12:         Replace all nodes in T belonging to the repetition
            by the node P
13:         Let N be the next node after node P
14:      else
15:         Let N be the next node
16:      end if
17:   end while
18:   if the tree rooted by T has changed then
19:      len ← 1
20:   else
21:      len ← len + 1
22:   end if
23: end while
```

Each node contains an integer number which is unique for the represented subtree. Nodes of type *simple* have no sons, nodes with type *periodic* have exactly one son, and *composite* nodes have at least two sons. Besides the identifier, a *periodic* node contains a further integer representing the number of occurrences of the pattern.

The algorithm searches for repeated cycles of growing length. Whenever a repetition is detected, all involved nodes are replaced by an appropriate sub-tree with a *periodic* node as root.

If a scan changes the current tree when a repetition is found, the length is set back to one, to be able to detect nested repetitions. The complexity of this algorithm is $O(n^3)$ in the worst case where no repetition of any length is found.

In the following, we try to clarify the algorithm using the sequence "0-1-2-3-2-3-2-3-1-2-3-2-3-2-3" as an example. In the initial step, a composite node containing nodes with these numbers as sons is created. During the first scan, the algorithm is not able to find any repetition of length one. In the next step, it finds the three-time repetition of the two element sequence "2-3" at two places. These repetitions are replaced by appropriate periodic nodes having the same identifier (5) because they represent equal sub-trees. The resulting tree is shown in Figure 6(a). Each node is labeled with its identifier. Nodes representing repetitions are rectangular and its label is extended by the number of repetitions.

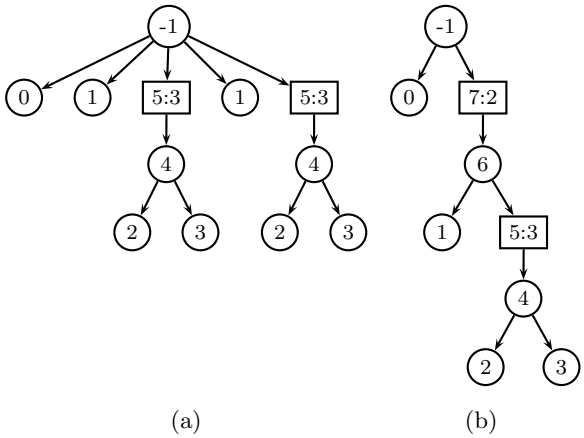

Figure 6: First change of the tree (a) and result of algorithm *DetectRepetitions* applied to the example sequence(b).

After this change, the algorithm searches for repetitions of length one again. Because this search gives no result, repeated sequences of length two are searched. We find exactly one repetition of the sequence "1-5". This is replaced by a new composite node with identifier 6 (Figure 6(b)) and the algorithm starts again searching for repetitions of length one. Because no further repeated sequences of any length are found, this replacement is the final result of the algorithm.

3.3.3 Converting the Repetition Tree into a Periodic Moving Object Representation

The structure of the repetition tree is equivalent to the one of the periodic moving tree. So, the time required for the creation of the periodic moving object is linear in the size of the repetition tree. This can be easily done by copying the structure of the tree got in step two into the nodes of the tree of the moving object. During the copy process, we can also compute additional information like node MBRs or total duration of intervals of composite and periodic moves.

3.4 Conversion into the Flat Representation

Sometimes it is necessary to convert from the periodic tree representation back to the flat represation, for example, for displaying a moving object on the screen. The algorithm for that is straightforward and will be omitted here because of space constraints. It basically traverses the tree in a depth-first manner expanding the repetition nodes and appending units to the result, converting the relative intervals into fixed ones. The units with undefined values are not inserted into the result.

4. IMPLEMENTATION ISSUES

In this section we point out some implementation issues. The approach presented in the previous sections is implemented inside a database system. Section 5 experimentally evaluates the approach and its implementation comparing periodic moving (point) objects to non-periodic ones.

4.1 Programming Environment

For implementing this model, we have used SECONDO ([4, 7, 9]), an extensible open source database system. SECONDO can be extended by algebra modules implementing new data types together with some operations. This is not performed at the application level. Rather, a new algebra module (written in C/C++) is linked to the system kernel directly. Besides the system kernel, the following existing algebras are used:

StandardAlgebra: containing standard data types like *bool*, *integer*, *real*, and *string*.

RelationAlgebra: including implementations for relations and tuple streams.

DateTimeAlgebra: providing the *instant* and *duration* data types.

SpatialAlgebra: making the 2-dimensional spatial types *point*, *points*, *line*, and *region* available.

TemporalAlgebra: implementing the flat representation for moving objects.

Dependencies to these algebras (excluding the RelationAlgebra) result from the fact that types defined in these algebras are used as arguments or result of implemented operators. The *RelationalAlgebra* is required to be able to use periodic moving objects as attributes in relations. Because a periodic moving object can be used as an attribute of a relation, we can handle a large set of such objects.

4.2 Data Structures and Representation

For a database system, the storing of pointer structures is very hard. This must be done by serialization of the data. To avoid a complex time consuming algorithm for converting a tree structure into serial data, we embed the tree into a fixed set of arrays. Pointers are emulated by a pair (*arraynumber*, *arrayindex*). Instead of using main memory based arrays, we use so called DBArrays, a sophisticated persistent data type providing the functionality of an array on top of FLOBS (see [5]).

The data structure for periodic moving objects is shown in Figure 7. It consists of a storage block, called the *root record*, containing a few fields of fixed size and some references to database arrays. The fields are the anchor time, the total duration of the movement, and a field *topmove* providing a logical pointer to the top node of the tree. Beyond that, there are four database arrays to represent the various types of nodes of the tree. Hence the field *topmove* contains an array identifier together with an index into that array.

Composite movements are in fact represented by the two database arrays C and S where the sons $s_1 \ldots s_n$ of a node c are represented as a subrange $S[i] \ldots S[i+n-1]$ of the array S. Hence a field of array C contains the two indices defining this subrange.

A field of array S contains a logical pointer to some other node (like *topmove*). Furthermore it contains a duration field storing the sum of the durations of its left neighbors within the same subrange. This information allows a binary search for a (relative) point in time within the sons of a composite node (see Algorithm 3 in Section 5).

Array B contains the unit representations for the basic movements.

A field of array P contains a logical pointer to some other node, the number of repetitions, and the total duration for this subtree.

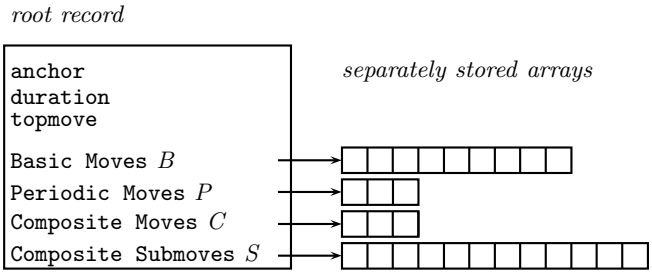

Figure 7: Data Structure for Periodic Moving Objects

By the separation between the root records and the different kinds of arrays, we have the possibility to partially load data. This means data from an array is only loaded from disk when it is accessed. So, we can retrieve the anchor time instant, the total duration of the movement, or whatever is stored in the root record without accessing the actual data of the periodic moving object. Furthermore, we can scan the units without loading data from the other arrays.

4.3 Implemented Data Types

The approach proposed here can be used to model periodic (or non-periodic) changes of any data type. Some of them are implemented in a SECONDO algebra module called *PeriodicAlgebra*. The selection is made to cover a broad range of different kinds of possible types.

The implemented data types are:

- *pmbool*. This data type represents a periodic moving boolean value. This is a candidate for a data type with only discrete changes where, in each unit, the value is constant.

- *pmreal*. A value of type *pmreal* models periodic changes of a value in $\mathbb{R}$. Changes within a unit are described by a set of parameters for a fixed polynomial function.

- *pmpoint*. A *pmpoint* value describes a point moving freely in space. The nodes of the repetition tree are extended by the bounding box of the appropriate subtree. Within a single unit the point moves linearly. It is sufficient to store the relative interval, the location at the start of the unit and the location at the and of the unit to describe a linear change of such a point.

5. EXPERIMENTAL EVALUATION

In this section we perform an experimental evaluation in order to show the good properties of the periodic moving

objects representation. We compare our approach with the flat representation using moving point (*mpoint*) data sets.

We use in this evaluation two data sets: a generated data set with periodicity and a real data set without any periodicity.

The first data set consists of a train line in the Berlin city. We expanded a schedule of such a train to a 2-dimensional data set with the help of the train railway (a *line* value). It is a relation containing 13 tuples where the trains with even identifiers stop on weekends and the trains with odd identifiers run on weekends in the same way as on working days. The total duration of the data set used is half a year.

Every day, the first train leaves the starting station at 6:03 AM and the time difference between two trains is 6 minutes, i.e. the second train leaves the starting station at 6:09 AM. The duration of the whole one-way trip is 40 minutes. The trains stop at every station for 10 seconds and between two stations the trains move with constant speed. Every train does 11 two-way trips, and the last train arrives at the starting station at 10:01 PM. The trains stop for the night and start again on the next day.

The second data set is a real one consisting of 47 tuples with GPS traces of several different trips from our group of research. It contains trips inside Hagen, trips from Dortmund to Hagen, and trips from Hagen to Gardelegen. No periodicity exists in this data set. The 4 relations are described as follows:

trains(no: *int*, trip: *mpoint*)
traces(name: *string*, trip: *mpoint*)
ptrains(no: *int*, trip: *pmpoint*)
ptraces(name: *string*, trip: *pmpoint*)

where the relations named with "p" as prefix use the periodic representations.

Table 1 shows the sizes of both data sets in terms of space used in MBytes and in terms of units. It can be seen in this table that the periodic representation is much more compact than the corresponding flat representation, in the trains data set, because many duplications of units are avoided in the repetitions.

Table 1: The two data sets.

Data set	Trains	GPS Traces
Size in the flat representation (Mbytes)	429.56	35.58
Size in the periodic representation (MBytes)	0.37	39.43
# of units in the flat representation	4,021,626	301,440
# of nodes in the periodic representation	3,037	301,487
# of basic nodes	2,958	301,440
# of periodic nodes	33	0
# of composite nodes	33	47

In order to evaluate the algorithm to detect periodicity presented in Section 3.3 we measured the time spent to convert the whole relation from the flat representation to the periodic representation using both data sets. Table 2 shows these results. The second data set represents the worst case of this algorithm, i.e. the case where no repetition is found and the result is a composite node with all units from the flat representation. We think that 223 seconds, which is almost 4 minutes to convert 47 tuples of approximately 6,414 units per tupleis reasonable.

Table 2: Time spent to convert from the flat to the periodic representation.

Data set	Time (s)
Trains	117
GPS Traces	223

Finally, we use two operations to evaluate the execution of queries in the periodic representation, namely **atinstant** and **trajectory**. The **atinstant** operation reduces the movement to the time instant passed as argument. The algorithm for that is presented in Algorithm 3. The algorithm simply traverses the tree until it finds the basic node with the unit that contains the argument time instant.

Algorithm 3 Algorithm **atinstant**

INPUT: a periodic moving point object o, an instant t
OUTPUT: a ipoint (t, p) (i.e. a pair consisting of instant t and point p)

1: **if** the time instant t is contained inside the interval $[o.anchor, o.anchor + o.duration]$ **then**
2: Let R be the root node of o
3: Return $(t,\ atinstantRec(\ R,\ t - o.anchor\ ))$
4: **else**
5: Return $(t, \perp)$
6: **end if**

Algorithm 4 Algorithm *atinstantRec*

INPUT: a node of a periodic moving point object R, a duration d
OUTPUT: a point p

1: **if** the node R is a basic node **then**
2: Compute the state of the unit at duration d calling the ι temporal function. Return $\perp$ for units with undefined values.
3: **else if** the node R is a composite node **then**
4: Determine the correct sub-movement S with binary search
5: Return $atinstantRec(\ S,\ d - \sum_{S' < S \in R} S'.duration\ )$
6: **else if** the node R is a periodic node **then**
7: Let S be the only son of R
8: Return $atinstantRec(\ S,\ d - \lfloor d/S.duration \rfloor\ )$
9: **end if**

Table 3 shows the result of the time spent in seconds on the following query

```
SELECT val(atinstant(trip, t)) FROM R
```

where R can be any of the four relations and t some time instant.

We can see in this table that the **atinstant** operator takes advantage of the periodic representation for the data with

Table 3: Time spent on queries with the atinstant operator.

Data set	Trains	GPS Traces
Flat representation	6.7	0.58
Periodic representation	0.0023	0.33

periodicity. For the data without periodicity, the results are similar.

The **trajectory** operation computes the 2-dimensional line that represents the movement of the point object. This operation is even simpler, it just scans the array containing the units, because the complete spatial information of the movement is contained in this array. Table 4 shows the time spent to perform the following query with the **trajectory** operator:

```
SELECT trajectory(trip) FROM R
```

This is another operation that takes advantage of the compact representation proposed in this paper for data with periodicity. Again, for the data set without periodicity, the performance of both approaches is still similar.

Table 4: Time spent on queries with the trajectory operator.

Data set	Trains	GPS Traces
Flat representation	149.00	11.4
Periodic representation	0.25	11.2

6. CONCLUSIONS

In this paper we have proposed an approach for representing periodically changing objects. The model is able to handle both nested periods and linearly changing objects. The model is independent of the underlying data type. An implementation of this model within the SECONDO framework was also described. We have presented an algorithm for detecting periods within linear data. It was shown that the size of the data which must be stored in a database or must be sent to a database client can be reduced drastically by exploiting periodic properties of changing objects. We have also shown that some operators can be implemented more efficiently than in a linear model.

7. REFERENCES

[1] L. Anselma. Recursive representation of periodicity and temporal reasoning. In *Proc. of the 11th Intl. Symp. on Temporal Representation and Reasoning (TIME)*, pages 52–59, 2004.

[2] M. Cai, D. Keshwani, and P. Z. Revesz. Parametric rectangles: A model for querying and animation of spatiotemporal databases. In *Proc. of the 7th Intl. Conf. on Extending Database Technology (EDBT)*, pages 430–444, 2000.

[3] J. A. Cotelo Lema, L. Forlizzi, R. H. Güting, E. Nardelli, and M. Schneider. Algorithms for moving objects databases. *The Computer Journal*, 46(6):680–712, 2003.

[4] S. Dieker and R. H. Güting. Plug and Play with Query Algebras: SECONDO-A Generic DBMS Development Environment. In *Proc. of the Intl. Symp. on Database Engineering & Applications (IDEAS)*, pages 380–392, 2000.

[5] S. Dieker, R. H. Güting, and M. R. Luaces. A tool for nesting and clustering large objects. In *Proc. of the 12th Intl. Conf. on Scientific and Statistical Database Management (SSDBM)*, pages 169–181, 2000.

[6] L. Forlizzi, R. H. Güting, E. Nardelli, and M. Schneider. A data model and data structures for moving objects databases. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 319–330, 2000.

[7] R. H. Güting, T. Behr, V. T. Almeida, Z. Ding, F. Hoffmann, and M. Spiekermann. SECONDO: An extensible DBMS architecture and prototype. Technical Report 313, Fernuniversität Hagen, Fachbereich Informatik, 2004. Available at http://www.informatik.fernuni-hagen.de/import/pi4/papers/Secondo04.pdf.

[8] 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 (TODS)*, 25(1):1–42, 2000.

[9] R. H. Güting, V. T. de Almeida, D. Ansorge, T. Behr, Z. Ding, T. Höse, F. Hoffmann, M. Spiekermann, and U. Telle. SECONDO: An extensible dbms platform for research prototyping and teaching. In *Proc. of the 21st Intl. Conf. on Data Engineering (ICDE)*, pages 1115–1116, 2005.

[10] K. Hornsby, M. J. Egenhofer, and P. J. Hayes. Modeling cyclic change. In *Proc. of the 1st. Intl. Workshop on Evolution and Change in Data Management (ECDM)*, pages 98–109, 1999.

[11] P. Z. Revesz and M. Cai. Efficient querying of periodic spatiotemporal objects. In *Proc. of the 6th Intl. Conf. on Principles and Practice of Constraint Programming (CP)*, pages 396–410, 2000.

[12] P. Z. Revesz and M. Cai. Efficient querying and animation of periodic spatio-temporal databases. *Ann. Math. Artif. Intell.*, 36(4):437–457, 2002.

[13] A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and querying moving objects. In *Proceedings of the 14th Intl. Conf. on Data Engineering (ICDE)*, pages 422–432, 1997.

[14] O. Wolfson, S. Chamberlain, S. Dao, L. Jiang, and G. Mendez. Cost and imprecision in modeling the position of moving objects. In *Proceedings of the 14th Intl. Conf. on Data Engineering (ICDE)*, pages 588–596, 1998.

[15] O. Wolfson, B. Xu, S. Chamberlain, and L. Jiang. Moving objects databases: Issues and solutions. In *Proc. of the 10th Intl. Conf. on Scientific and Statistical Database Management (SSDBM)*, pages 111–122, 1998.