

SQL Extension for Interval Data

Nikos A. Lorentzos and Yannis G. Mitsopoulos

Abstract—IXSQL, an extension to SQL, is proposed for the management of interval data. IXSQL is syntactically and semantically upwards consistent with SQL2. Its specification has been based both on theoretical results and actual user requirements for the management of temporal data, a special case of interval data. Design decisions and implementation issues are also discussed.

Index Terms—Data model, interval data, query language, spatial data, temporal data.

1 INTRODUCTION

THE term *interval* is quite generic and has many applications: The range within which the voltage, current, temperature etc. should restrict for the proper functioning of a machine are examples of *integer* or *real intervals*. *Time intervals* are used to mark the duration of events as it is the lifespan of a person or the time during which somebody was involved in a project. *Alphabetic intervals* are used in many cases, as it is the family names or the book titles in the range A-C. Finally, intervals over arbitrary, application-dependent sets are often used, as it is the car plates in the range AAB2000-BBD4000. Given the wide use of intervals in every day life, it is obvious that their representation and manipulation in a database is of major importance. Yet, commercial Database Management Systems (DBMS) do not support them directly: Intervals can only be represented in an indirect way, by maintaining two attributes to record their boundaries, but no operations are provided for their management. As a consequence, programming effort is required in order to answer even simple queries. To overcome this problem, in this paper we propose IXSQL, an *Interval Extension to SQL*, for the management of interval data. This extension has a major advantage, that its functionality, syntax and semantics is fully upwards consistent with SQL2 [1] (A detailed description of SQL2 can be found in [2].)

The characteristics of IXSQL can be summarized as follows. It provides direct support to a generic interval data type. It incorporates new predicates, value functions, and set functions. It extends the SQL manipulation operations and supports the concept of the key of an *interval* relation. It can be applied to relations with more than one attribute of an interval type. It has many application areas, one of which is the management of temporal databases or, according to a terminology recently proposed [3], *valid time databases*, an area in which a lot of research has been undertaken in recent years [4], [5], [6], [7], [8], [9], [10], [11]. The definition of IXSQL has been based both on theoretical research results [12] and also on actual user requirements

[13], for valid time support. Its evaluation has revealed that it is very powerful, in that it can answer a superset of those queries which were required by the application [13].

The remainder of this paper can be summarized as follows. In Section 2, we present a sample database with interval data, against which we demonstrate, in subsequent sections, the functionality of IXSQL. For each relation, we also define a primary key, as this term has been enhanced for interval relations [14]. In Section 3, we present a relational algebra, on whose definition IXSQL has been based. In Section 4 we formalize IXSQL. In Section 5 we justify certain specification decisions. In Section 6 we present further possible enhancements of the language. In Section 7, we discuss implementation issues. Conclusions and further work are presented in Section 8. The paper is followed by two appendices: In Appendix A, we define the functions of IXSQL and present the relevant interval predicates. In Appendix B we provide the IXSQL extensions to the SQL2 syntax.

2 A SAMPLE DATABASE AND RELEVANT SEMANTICS

In this section we provide the description of a database (Fig. 1) which is used in subsequent sections. Relations TRANSPLANTATION, DRUG, and INFECTION, have been borrowed from an actual medical application [13] which requires the management of valid time data, concerning patients who have had a liver transplant operation. The remainder relations are given in order to demonstrate the functionality of IXSQL. Except for TRANSPLANTATION, all the relations incorporate attributes of an *integer* or *date* or *string interval* data type. A date has format 'yyyy-mm-dd,' but, for simplicity reasons, it is represented by an integer, preceded by 'd'. Thus, *d30* could represent 1993-05-01. Similarly, [*d30, d40*) could represent the interval [1993-05-01, 1993-05-11). For each relation, we also provide the associated semantics and define its *key*, as this term has been extended in [14], so as to apply to relations which contain interval data.

In TRANSPLANTATION, we record the date at which a patient has had a liver transplant operation. For simplicity reasons, we assume that only one such operation takes place per patient, thus the primary key of the relation is <Name>.

• N.A. Lorentzos and Y.G. Mitsopoulos are with the Informatics Laboratory, Agricultural University of Athens, Iera Odos 75, GR 118 55 Athens, Greece. E-mail: lorentzos@auadec.aaa.ariadne-t.gr.

Manuscript received 12 May 1994; revised 29 Dec. 1994.

For information on obtaining reprints of this article, please send e-mail to: transkde@computer.org, and reference IEEECS Log Number K96099.

TRANSPLANTATION		DRUG			
Name	Date	Name	Drug	Level	Time
John	d30	John	Cyclosporine	121	[d30, d35)
Peter	d40	John	Cyclosporine	58	[d35, d40)
		John	Cyclosporine	110	[d40, d44)
		John	Azathioprine	110	[d42, d46)

DENSITY			CARPLATE	
Length	Width	Population	City	Number
[1, 100)	[1, 100)	3,000,000	Athens	[AAA0001, CCC0000)
[25, 75)	[25, 75)	1,250,000	Salonica	[CCC0001, EEE0001)

LAND				INFECTION		
Pno	Depth	Time	pH	Name	Cause	Time
1	[0, 20)	[d0, d80)	8.0	John	Enterococcus s.p.	[d31, d34)
1	[0, 100)	[d120, d200)	8.0	John	Enterococcus s.p.	[d40, d46)
1	[30, 120)	[d0, d80)	8.0	John	Enterococcus s.p.	[d50, d60)
1	[0, 30)	[d80, d120)	8.2	John	Proteus s.p.	[d33, d38)
2	[0, 60)	[d10, d50)	8.3	Peter	Proteus s.p.	[d45, d50)
2	[60, 100)	[d30, d90)	8.3			

Fig. 1. A sample database.

D1				D2			
Name	Drug	Level	Time	Name	Drug	Level	Time
John	Cyclosporine	121	[d30, d33)	John	Cyclosporine	121	[d30, d32)
John	Cyclosporine	121	[d32, d35)	John	Cyclosporine	121	[d32, d35)
John	Cyclosporine	58	[d35, d40)	John	Cyclosporine	58	[d35, d40)
John	Cyclosporine	110	[d40, d44)	John	Cyclosporine	110	[d40, d44)
John	Azathioprine	110	[d42, d46)	John	Azathioprine	110	[d42, d46)

Fig. 2. Two possible representations of interval data.

DRUG is a relation in which we record the dosage of drugs each patient was treated with, *on each of the days* in a particular date interval. For example, the first tuple shows that during [d30, d35) the *daily* dosage of Cyclosporine, John was treated with, was 121 units (121 units for d30, 121 units for d31, etc.). From the second tuple we see that during [d35, d40), this dosage changed to 58 units per day. The data in DRUG can be recorded in many distinct ways. Two of them are shown in Fig. 2. In both D1 and D2 we can see that the first two tuples have replaced the first tuple of DRUG. (In D1 we also notice that John's treatment with Cyclosporine on date d32 is repeated in the first two tuples.) However, in data modeling it is desirable to represent data in a unique, nonarbitrary, way. From all possible ways therefore, we adopt the one used in DRUG, which is the most reasonable. In particular, DRUG should not contain two tuples whose respective values for Name, Drug, and Level match, whereas their values for Time form touching or overlapping intervals (For relations like this, this principle has been adopted by all the researchers in the area of valid time data modeling [6], [8], [9], [11], [12], [15]). We denote this fact by saying that DRUG is *normalized* with respect to Time. (The concept of a *normalized* relation is formalized in the next section.) One reasonable assumption is that a patient may be treated with more than one drug on the same date. Clearly however, DRUG may not contain two tuples like (John, Cyclosporine, 121, [d30, d35)) and (John, Cyclosporine, 60, [d32, d40)). Indeed, from the former it can be deduced that on each of the dates d32, d33,

d34 the dosage of Cyclosporine for John was 121 units, but this contradicts the second tuple, according to which the dosage on these days was 60. This constraint can be enforced if we define that the *primary key* of DRUG is <Name, Drug, Time POINT>. Semantically this means that it is not allowed to record in DRUG two distinct tuples, from which it can be deduced that on a particular date some patient was treated with two different dosages of the same drug.

If a geographical area is partitioned into square kilometers, it is possible to maintain a relation, showing the population living in each orthogonal rectangle of this area. Thus, the first tuple of relation DENSITY shows that the rectangle with coordinates [1, 100), [1, 100) is inhabited by 3,000,000 people. Similarly, the rectangle in the second tuple is inhabited by 1,250,000. It is important to notice that the semantics of DENSITY are different than those of DRUG: A value for Population now concerns a *whole rectangle* (2-dimensional interval), not *each square kilometer of the rectangle*. As opposed to DRUG, we see that now it is possible to record two tuples whose respective values for both Length and Width overlap. Finally, each rectangle determines uniquely the population living in it. We enforce this constraint, by declaring that the key of the relation is <Length INTERVAL, Width INTERVAL>. Hence, both DRUG and DENSITY have columns of an interval type, yet each of these tables is associated with different semantics. This distinction in the semantics is achieved, in our approach, by the addition of an appropriate keyword (POINT or INTERVAL) after the column name. Hence, each of these

keywords relates only to the semantics associated with the column which precedes it, it is not related, in any way, either with the indices or with the scheme of the table. (The definition, of an IXSQL relation scheme is given in Section 4.9.)

In INFECTION we record the disease a patient was infected by, during a time interval. Thus, the first tuple shows that John suffered from *Enterococcus s.p.* on *each of the dates* in [d31, d34). Similar to DRUG, INFECTION must also be normalized. Since a patient may be suffering from two distinct diseases at the same time, the primary key of INFECTION is <Name, Cause, Time POINT>.

LAND is an example of a relation with *spatiotemporal* data [16]. It enables to record the soil pH of various profiles, at various depths and at various times. For example, the first tuple shows that at profile 1 the pH value was 8.0 at *each of the depths* in [0, 20) cm and *on each of the dates* in [d0, d80). LAND is normalized with respect to Depth, Time. Clearly, for a given profile, depth, and date, only one pH value should be recorded; therefore the primary key is <Pno, Depth POINT, Time POINT>.

Finally, in CARPLATE, we record the car plates which the authorities of every city are authorized to issue. Thus, the first tuple shows that the authorities of Athens can issue *each of the plates* AAA0001, AAA0002, ..., BBB9999. Again, CARPLATE must be normalized. Since, a plate may not be issued by two distinct authorities, the primary key is <Number POINT>.

Regarding the above relations, we notice the following:

- 1) A relation may contain valid time data, although it may not necessarily have attributes to record time intervals. A typical example is TRANSPLANTATION, an ordinary SQL2 relation.
- 2) It does make sense to consider relations of practical interest, which incorporate intervals, yet they do not contain valid time data (CARPLATE).
- 3) It does make sense to consider relations which incorporate more than one type of intervals (LAND).
- 4) The semantics associated with a valid time relation are similar with the semantics of a relation not containing such data. Consider, for example, the similarity of the semantics of DRUG and INFECTION on one hand, with those of CARPLATE on the other.
- 5) In all the valid time relations which were presented, an attribute is maintained to record either time points or time intervals. However, valid time may be *implicit* and *relative*. If we assume, for example, that in INFECTION we are interested in recording time intervals *relative* to the transplantation date, then the time intervals in attribute Time should be replaced by *integer intervals*. (Thus, the interval of the first tuple would be [1, 4).)
- 6) DENSITY allows the recording of overlapping *integer intervals*. In a similar manner, valid time relations can also be identified whose semantics allow the recording of overlapping *time intervals* [12]. The above remarks justify the necessity to formalize a model for the uniform treatment of interval data. A relational algebra, for such a model, is provided in the next section.

3 INTERVAL-EXTENDED RELATIONAL ALGEBRA

3.1 Preliminaries

In this section, we present an algebra [12], on whose definition IXSQL has been based. We begin with some basic definitions.

DEFINITION 3.1. A 1-dimensional (1-d) space is a nonempty finite set, $D = \{p_1, p_2, \dots, p_n\}$, totally ordered with respect to '<'. The elements of D are called 1-d points, or simply points, and it is assumed that $p_1 < p_2 < \dots < p_n$. Two points p_i and p_{i+1} are consecutive and p_i (p_{i+1}) is the predecessor (successor) of p_{i+1} (p_i), denoted by $\text{pred}(p_{i+1})$ ($\text{succ}(p_i)$).

DEFINITION 3.2. If $p_i, p_j \in D$ and $p_i < p_j$, then an interval $[p_i, p_j]$ over a 1-d space D is the set of points

$$\{p_k \in D \mid p_i \leq p_k < p_j\}.$$

An interval can geometrically be represented by a line segment closed to the left and open to the right (see the figure in Appendix A). If v is an interval $[p_i, p_j]$, then p_i and p_j are said to be the *boundaries* of v and, in particular, its *start* and *stop* points, denoted by $\text{start}(v)$ and $\text{stop}(v)$, respectively. An interval over a 1-d space is thus uniquely defined by its boundaries. We denote by $I(D)$ the set of all intervals over D .

Two intervals can be merged into one, if their line segments touch or intersect. This is formalized in the following definition:

DEFINITION 3.3. Given two intervals v and w over D , v merges w if $\text{stop}(v) \geq \text{start}(w)$ and $\text{stop}(w) \geq \text{start}(v)$. If v merges w , the function *merge* yields a single interval as follows:

$$\text{merge}(v, w) = [\min(\text{start}(v), \text{start}(w)), \max(\text{stop}(v), \text{stop}(w))).$$

An interval is called *elementary* if it contains exactly one point. Clearly, a 1-d space D is isomorphic to $E(D)$, the set of *elementary* intervals over D , since a point p_i can be viewed as an elementary interval $[p_i, p_{i+1})$ and vice versa. This is termed the *duality principle*. Since $E(D) \subseteq I(D)$, then $D \subseteq I(D)$ up to this principle. We now generalize the notions of 1-d points and intervals to their n -d counterparts.

DEFINITION 3.4. If $D_1, \dots, D_n$ are 1-d spaces and $p_i \in D_i$ for all i , $1 \leq i \leq n$, then $(p_1, \dots, p_n)$ is an n -d point. If $v_1, \dots, v_n$ are intervals over $D_1, \dots, D_n$, respectively, then $(v_1, \dots, v_n)$ is an n -d interval. An n -d interval $(v_1, \dots, v_n)$ is *elementary* if each v_i is elementary. Given two n -d intervals $v = (v_1, \dots, v_n)$ and $w = (w_1, \dots, w_n)$, we define that v merges w if: 1) v_i merges w_i and 2) $v_j = w_j$ for all $j \neq i$. If v merges w , the function *merge_i* yields a single n -d interval as follows:

$$\text{merge}_i(v, w) = (v_1, \dots, v_{i-1}, \text{merge}(v_i, w_i), v_{i+1}, \dots, v_n).$$

3.2 Interval Relations

Let us now consider a database in which we record alphabetic, numeric or alphanumeric data. (For simplicity, we call *textual* any of these types of data.) We then notice the following: Firstly, the domain D of every attribute in this database can be seen as a 1-d space: Indeed, the domain of every attribute is a nonempty set. Also, this domain is totally ordered, since a default ordering is always in effect, for any type of textual data. Finally, this domain is practically finite: A character string is composed by a finite set of

characters (the ASCII characters) and its length may not exceed some upper bound n , usually fixed by the DBMS designer; an integer may not lie outside a prespecified range $[li, gi]$, where li (gi) is the lowest (greatest) integer, which can be represented in a particular computer. Finally, all reals also lie in a range $[lr, gr]$. Although the set of reals in this range theoretically form an uncountable infinite set, in practice only a finite number of them can be represented in a computer. Hence, for every domain D of textual data, the respective $I(D)$ can always be defined. Thus, $[1, 10]$, $[9.00, 20.50]$ and $['AAA', 'FFF']$ are examples of intervals over the set of integers, reals and $CHAR(3)$, respectively. Secondly, we recall the duality principle, according to which a point can be interpreted as an interval, up to an isomorphism. Making therefore use of the two observations, above, we define the concept of an *interval relation*:

DEFINITION 3.5. Given n 1-d spaces, $D_1, \dots, D_n$, an interval relation is a subset of $X(D_1) \times \dots \times X(D_n)$, where $X(D_i)$ represents exclusively D_i or $I(D_i)$.

The scheme of an interval relation R is a list of distinct attribute names $A_1, \dots, A_n$. The domain of each such attribute is exclusively either some 1-d space D or the respective $I(D)$. Thus, the scheme of some of the tables in Fig. 1 is:

TRANSPLANTATION(Name = CHAR(20), Date = DATE)
 DRUG(Name = CHAR(20), Drug = CHAR(30),
 Level = INTEGER, Time = I(DATE))
 DENSITY(Length = I(INTEGER), Width = I(INTEGER),
 Population = INTEGER)

3.3 Management of Interval Relations

The relational algebra operations [17] *union*, *except (difference)*, *project*, and *Cartesian product* remain the same for (interval) relations. *Selection* has been extended and incorporates relational operators which can be applied to pairs of intervals. For example, one operator cp (*common points*), when applied to two intervals v and w , evaluates to *true* if the two intervals have at least one point in common (they intersect). Thus, “[1, 15] cp [5, 25]” yields *true* whereas “[1, 15] cp [20, 25]” yields *false*. The set of all these new operators is given in Appendix A.

Finally, two more operations, *unfold* and *fold*, have been added to the relational algebra [17]. Here we restate their definitions and illustrate their use by way of examples.

DEFINITION 3.6. Given a relation $R(A_1, \dots, A_i = X(D_i), \dots, A_n)$, *unfold* $[A_i](R)$ yields a relation S with scheme $S(A_1, \dots, A_i = D_i, \dots, A_n)$ and with extension the set of tuples

$$\{(v_1, \dots, v_{i-1}, p, v_{i+1}, \dots, v_n) \mid (v_1, \dots, v_{i-1}, v_i, v_{i+1}, \dots, v_n) \in R \ \& \ p \in v_i\}.$$

Thus, *unfolding* R with respect to an attribute A_i replaces each tuple with a set of tuples containing only single points for A_i . For example, $D3 = \text{unfold}[\text{Time}](\text{DRUG})$ yields the table in Fig. 3. The definition is further extended so as to successively *unfold* a relation R with respect to a list of its attributes:

DEFINITION 3.7. If $B_1, \dots, B_m$ is a list of some of the attributes of a relation $R(A_1, \dots, A_n)$, we define

$$\text{unfold}[B_1, \dots, B_m](R) = \text{unfold}[B_m](\dots \text{unfold}[B_2](\text{unfold}[B_1](R)) \dots)$$

We now define the converse of *unfold*, namely the *fold* operation:

DEFINITION 3.8. Given a relation $R(A_1, \dots, A_n)$, *fold* $[A_i](R)$ yields a relation S with scheme $S(A_1, \dots, A_{i-1}, A_i = I(D_i), A_{i+1}, \dots, A_n)$ and extension T defined procedurally as follows:

begin

$T := R$;

while there exist distinct tuples r and $s \in T$ such that r merges $_i$ s **do** $T := (T - \{r, s\}) \cup \{\text{merge}_i(r, s)\}$

end

Thus, folding a relation R with respect to an attribute A_i (of a point or interval type) repeatedly takes pairs of tuples which are identical in all attributes other than A_i and merges them with respect to A_i , until no such pairs of mergeable tuples exist. For example, if D is any of the relations in Figs. 2 or 3 then *fold* $[\text{Time}](D)$ yields DRUG in Fig. 1. This definition is also extended, to successively *fold* a relation R with respect to a list of its attributes:

DEFINITION 3.9. If $B_1, \dots, B_m$ is a list of some of the attributes of a relation $R(A_1, \dots, A_n)$, we define

$$\text{fold}[B_1, \dots, B_m](R) = \text{fold}[B_m](\dots \text{fold}[B_2](\text{fold}[B_1](R)) \dots).$$

Contrary to *unfold*, the order in which attributes are *folded*, is significant, i.e., *fold* $[A, B](R) \neq (\text{fold}[B, A](R))$ [12]. As we can see, *unfold* can transform pure *interval data* to *point data*, whereas *fold* can perform the inverse. For this reason, the use of these operations, either in isolation or in combination, enables a high degree of data independence, as is shown in Section 4.6. At the same time, they have theoretical importance: They enable the easy formalization of the IXSQL manipulation operations (Section 4.10) as well as the formalization of the following algebraic operations:

normalize: If $B_1, \dots, B_m$ are some attributes of a relation $R(A_1, \dots, A_n)$ then

$$\text{normalize}[B_1, \dots, B_m](R) = \text{fold}[B_1, \dots, B_m](\text{unfold}[B_1, \dots, B_m](R))$$

The result relation, is called *normalized with respect to* $B_1, \dots, B_m$. As an example, if D is any of the relations in Figs. 2 or 3 then *normalize* $[\text{Time}](D)$ yields DRUG .

D3			
Name	Drug	Level	Time
John	Cyclosporine	121	d30
John	Cyclosporine	121	d31
...	...	...	...
John	Cyclosporine	121	d34
John	Cyclosporine	58	d35
...	...	...	...
John	Cyclosporine	58	d39
John	Cyclosporine	110	d40
...	...	...	...
John	Cyclosporine	110	d43
John	Azathioprine	110	d42
...	...	...	...
John	Azathioprine	110	d45

Fig. 3. $D3 = \text{unfold}[\text{Time}](\text{DRUG})$.

Finally, if $R(A_1, \dots, A_n)$ and $S(A_1, \dots, A_n)$ are two union-compatible relations and $B_1, \dots, B_m$ are some of their attributes, two more operations are defined:

$R \text{ **punion**}[B_1, \dots, B_m] S = \text{fold}[B_1, \dots, B_m]$
 $(\text{unfold}[B_1, \dots, B_m](R) \text{ **union** } \text{unfold}[B_1, \dots, B_m](S))$

$R \text{ **pexcept**}[B_1, \dots, B_m] S = \text{fold}[B_1, \dots, B_m]$
 $(\text{unfold}[B_1, \dots, B_m](R) \text{ **except** } \text{unfold}[B_1, \dots, B_m](S)).$

By their definition, these operations yield a relation *normalized* with respect to $B_1, \dots, B_m$. The practical importance of **normalize**, **punion**, and **pexcept** is shown in the next section, whereas the properties of all the operations, defined here, have been investigated in [18]. Incidentally, *unfold* (*fold*), originally presented in [19], were defined independently of *unnest* (*nest*) [20] and *unpack* (*pack*) [21]: The operations in [20], [21] are dedicated to the management of set-valued attributes, *not* the management of interval data. One major difference in the approaches is that if a relation $R(A)$ consists of three tuples with values 1, 2, and 5 then $\text{fold}[A](R)$ returns two tuples with values [1, 3], [5, 6], whereas *nest* (*pack*) return the set {1, 2, 5}.

4 EXTENSIONS TO SQL2

We now formalize IXSQL and provide examples of its functionality. The individual IXSQL syntactical features are given in **bold**, to make reading easier. We have avoided excessive complication of the IXSQL syntax, by giving, in written, all the rules which cannot be deduced by its BNF definition. The complete set of the extension is given in Appendix B.

4.1 Data Types

IXSQL supports every data type D (for example INTEGER, FLOAT, CHAR(10), DATE, TIME, TIMESTAMP, etc.) of SQL2 and the respective I(D) data type called, from now on, **INTERVAL OF D** . (It should be noted that although the keyword **PERIOD** could have been used in place of **INTERVAL**, in order to distinguish the relevant SQL2 keyword, in the present paper we retain **INTERVAL**, since an implementation, based on SQL89, uses this second word.)

4.2 Literals

IXSQL supports both the SQL2 literals and their *interval* counterparts. Their form is $[p_i, p_j]$, where p_i and p_j are SQL2 literals of the same type. Thus, [3, 5] is an **INTERVAL OF INTEGER** literal, ['3', '5'] and ['A', 'B'] are **INTERVAL OF CHAR(1)** literals and [date '1993-01-01', date '1994-01-01'] is an **INTERVAL OF DATE** literal. In the following, we shall denote a *generic* point (interval) by p_i ($[p_i, p_j]$) and a DATE point (DATE interval) by ' d_i ' ($['d_i', 'd_j']$).

4.3 Interval Predicates

The set of SQL2 predicates has been extended by the *interval predicates*, listed in Appendix A. This greatly simplifies the formulation of queries.

4.4 Interval Value Functions

SQL2 has been extended by the following functions, formally defined in Appendix A. They all obey the rules of SQL2 for value functions. The first two are particularly useful in temporal databases.

mindate() returns the least date supported by the DBMS.

maxdate() returns the greatest date supported by the DBMS.

start($[p_i, p_j]$) returns the start point of $[p_i, p_j]$:

$\text{start}([p_5, p_8]) = p_5.$

stop($[p_i, p_j]$) returns the stop point of $[p_i, p_j]$:

$\text{stop}([p_5, p_8]) = p_8.$

topoint($[p_i, p_{i+1}]$) is applied to an elementary interval and returns the corresponding point:

$\text{topoint}([p_5, p_6]) = p_5$, $\text{topoint}([p_5, p_{10}])$ is undefined.

tointerv(p_i) transforms a point p_i to the respective elementary interval $[p_i, p_{i+1}]$:

$\text{tointerv}(p_5) = [p_5, p_6].$

interv(p_i, p_j) returns the interval $[p_i, p_j]$, if $p_i < p_j$:

$\text{interv}(p_5, p_8) = [p_5, p_8]$, $\text{interv}(p_5, p_3)$ is undefined.

intervsect($[p_i, p_j], [p_r, p_s]$) computes the intersection of two intervals:

$\text{intervsect}([p_5, p_{10}], [p_7, p_{12}]) = [p_7, p_{10}]$, $\text{intervsect}([p_5, p_8], [p_{10}, p_{12}])$ is undefined.

succ(p_i, j) returns a point which is j places away from p_i :

$\text{succ}(p_{10}, 3) = p_{13}$, $\text{succ}(p_{10}, -3) = p_7.$

dur($[p_i, p_j]$) computes the number of points in $[p_i, p_j]$:

$\text{dur}([p_5, p_8]) = 3.$

span(p_i, p_j) computes the *relative* number of points from p_i to p_j :

$\text{span}(p_8, p_5) = 3$, $\text{span}(p_5, p_8) = -3.$

dist(p_i, p_j) computes the *absolute* number of points from p_i to p_j :

$\text{dist}(p_8, p_5) = 3$, $\text{dist}(p_5, p_8) = 3.$

middle($[p_i, p_j]$) returns the *middle* point of an interval:

$\text{middle}([p_5, p_{10}]) = p_7$, $\text{middle}([p_5, p_{11}]) = p_7.$

merge($[p_i, p_j], [p_r, p_s]$) returns the interval into which the two interval arguments merge:

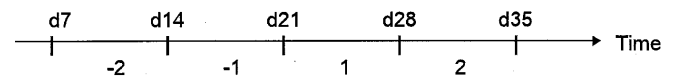
$\text{merge}([p_1, p_{10}], [p_5, p_{15}]) = [p_1, p_{15}]$, $\text{merge}([p_1, p_{10}], [p_{25}, p_{30}])$ is undefined.

window(p_i, n, m):

Starting from point p_i , it returns the m th interval, away from p_i , of duration n .

windowno(p_i, n, p_j):

Starting from point p_i , it returns the ordinal number of the interval of duration n , in which p_j belongs.



To provide examples of the last two functions, consider the axis of Time, above, and assume that the points d_i represent dates. Assume also that we start from an arbitrary date d_{21} and split the axis into intervals of seven days each. Then $[d_{21}, d_{28}]$, $[d_{28}, d_{35}]$, $[d_{14}, d_{21}]$, and $[d_7, d_{14}]$ represents the *1st*, *2nd*, *-1st*, and *-2nd* interval of a 7-day duration, respectively. Thus, $\text{window}('d_{21}', 7, 1) = [d_{21}, d_{28}]$, $\text{window}('d_{21}', 7, 2) = [d_{28}, d_{35}]$, $\text{window}('d_{21}', 7, -1) = [d_{14}, d_{21}]$ and $\text{window}('d_{21}', 7, -2) = [d_7, d_{14}]$. (Note that *window* is different than the **WINDOW** clause, defined in [8].)

Function *windowno* is the *converse* of *window*: If we consider the above split, we can see that *d25* lies in the first time-interval. Thus, *windowno*('d21', 7, 'd23') = 1. Similarly, *windowno*('d21', 7, 'd30') = 2 and *windowno*('d21', 7, 'd18') = -1.

It is worth noting that *window* and *windowno* had to be defined in order to satisfy *actual* user requirements for temporal data management [13]. Queries showing their practical interest, are the following:

"Give the patients who have had some infection during the first 30 days after their transplant operation."

```
SELECT T.Name
FROM TRANSPLANTATION T, INFECTION I
WHERE T.Name = I.Name AND window(Date, 30, 1) cp Time
```

"Give the week numbers after the transplant operation, in which John started being administered with Azathioprine."

```
SELECT DISTINCT windowno(T.Date, 7, start(D.Time))
FROM TRANSPLANTATION T, DRUG D
WHERE T.Name = 'John' AND T.Name = D.Name AND
      D.Drug = 'Azathioprine'
ORDER BY 1
```

The answer to the first query consists of two tuples, one for John and one for Peter. The answer to the second is 2.

4.5 Interval Set Functions

Two new interval set functions have been defined (Appendix A), *countap* and *countdp*: Function **countap** (*count all points*) returns the number of points (of integer type) that are contained in a selected column or value expression of an interval type. Duplicate points are also counted. Function **countdp** (*count distinct points*) returns the number of *distinct* points (of integer type):

"Give the number of days John was administered with some drug."

```
SELECT countap(Time)
FROM DRUG
WHERE Name = 'John'
```

In relation DRUG, we can see that the four intervals for John, recorded in attribute Time, consist of 5, 5, 4, and 4 days. Thus, the result returned equals 18, the summation of these days. However, there are days during which John was administered with both Cyclosporine and Azathioprine (dates *d42*, *d43*). By using *countap*, these days are counted twice. To avoid counting a point more than once, then *countdp* has to be used. Thus, *countdp*(Time), in the above query, returns 16.

Functions *countap* and *countdp* obey all the rules of the SQL2 set functions. One additional restriction is that the SQL2 functions *avg*, *max*, *min*, and *sum* may not be applied to elements of an interval type.

4.6 Interval-Extended SELECT

SELECT has been extended by a REFORMAT AS and a NORMALIZE ON expression, which enable the incorporation of the operations of the algebra defined in Section 3. Its syntax has thus become

```
SELECT [<set qualifier>] <select list> (1)
FROM <table ref list> (2)
[WHERE <search condition> ] (3)
[GROUP BY <grouping column ref list> ] (4)
[HAVING <search condition> ] (5)
[REFORMAT AS <reformat item> ] (6)
[NORMALIZE ON <reformat column list> ] (7)
[ORDER BY <sort spec list> ] (8)
```

Lines (1)–(5) are executed as in SQL2 and lines (6)–(8), if they exist, are executed in this order. **Line 6 (Reformats a table with respect to a sequence of columns): REFORMAT AS** introduces a sequence of *unfold* or *fold* operations which have to be performed on the table retrieved by the execution of the code in lines (1)–(5). The syntax is as follows:

```
<reformat item> ::= FOLD <reformat column list>
                  [<reformat item>]
                  | UNFOLD [ALL] <reformat column list>
                  [<reformat item>]

<reformat column> ::=
                  <column name>
                  | <table name>.<column name>
                  | <correlation name>.<column name>
                  | <unsigned integer>
```

Since REFORMAT AS is applied to the table obtained by the execution of lines (1)–(5), the <reformat column list>, which follows an UNFOLD or FOLD, must be a sublist of the columns in <select list>. In particular, if R is the table obtained by the execution of lines (1)–(5) then **FOLD** A1, A2, ..., An and **UNFOLD** A1, A2, ..., An are, respectively, equivalent to the algebraic operations *fold*[A1, A2, ..., An](R) and *unfold*[A1, A2, ..., An](R). One example is

```
REFORMAT AS UNFOLD Time1, 3, R.Time2
FOLD Time3, 5 UNFOLD Time4
```

EXAMPLE.

"Give the level of Cyclosporine administered to John, on each of the dates in [d32, d38)."

```
SELECT intervsect(Time, ['d32', 'd38']), Level
FROM DRUG
WHERE Name = 'John' AND Drug = 'Cyclosporine'
      AND Time cp ['d32', 'd38')
REFORMAT AS UNFOLD 1
```

	Level
[d32, d35)	121
[d35, d38)	58

	Level
d32	121
d33	121
d34	121
d35	58
d36	58
d37	58

The execution of the first three lines of this query gives the left table above. After the execution of the last line, the right table, above, is obtained. REFORMAT AS provides a high degree of data independence, since the user can view data in a way which best suits his requirements, no matter how this data has physically been stored. For example, one user may see relation LAND exactly as it is in Fig. 1; another one may view this data for each individual day

(**REFORMAT AS UNFOLD** Time) or for each individual depth (**REFORMAT AS UNFOLD** Depth) or for each individual day and depth (**REFORMAT AS UNFOLD** Time, Depth). Finally,

```
SELECT      *
FROM        LAND
REFORMAT AS UNFOLD Depth, Time FOLD Depth, Time
```

yields the table in Fig. 4. As has been shown in [12], this table and LAND differ in the semantics.

Pno	Depth	Time	pH
1	[0, 20)	[d0, d80)	8.0
1	[0, 100)	[d120, d200)	8.0
1	[30, 120)	[d0, d80)	8.0
1	[0, 30)	[d80, d120)	8.2
2	[0, 60)	[d10, d30)	8.3
2	[0, 100)	[d30, d50)	8.3
2	[60, 100)	[d50, d90)	8.3

Fig. 4. Example of a table restructuring.

Whereas relational theory does not allow tables with duplicate rows, such rows are by default retrieved in SQL2 (unless *DISTINCT* is specified after the keyword **SELECT**). To maintain therefore compatibility with SQL, two versions of UNFOLD have been defined in IXSQL, the simple **UNFOLD** (described above) and **UNFOLD ALL**. The latter allows duplicate rows.

EXAMPLE.

"List the days within [d31, d35) on which John was suffering from some disease."

If the query is formulated as below, the result table will contain one row for each of the values d31, d32, d33, d33, d34. Because of **UNFOLD ALL**, we can see that d33 is repeated.

```
SELECT      intersect(Time, ['d31', 'd35'))
FROM        INFECTION
WHERE       Name = 'John' AND Time cp ['d31', 'd35')
REFORMAT AS UNFOLD ALL 1
```

Line 7 (Normalizes a table with respect to certain columns): If <reformat column list> consists of the columns of the table derived after the execution of lines (1)–(6) (therefore <reformat column list> is a sublist of the columns referenced in <select list>), then **NORMALIZE ON** <reformat column list> is a shorthand for **REFORMAT AS UNFOLD** <reformat column list> **FOLD** <reformat column list>. (Perhaps some other keyword, e.g., *CANONICAL*, should have been used in place of **NORMALIZE**, since the latter is also related with database design. In the present paper, we are using **NORMALIZE**, since this keyword has also been incorporated in an implementation.)

EXAMPLE.

"Give the time during which John suffered from Proteus s.p., and the drugs he started being administered with, from the infection time, onwards. For each drug, give the corresponding treatment time."

```
SELECT      I.Time, D.Drug, D.Time
FROM        INFECTION I, DRUG D
WHERE       I.Name = 'John' AND I.Name = D.Name
AND         I.Cause = 'Proteus s.p.' AND D.Time follows I.Time
```

NORMALIZE ON D.Time

I.Time	Drug	D.Time
[d33, d38)	Cyclosporine	[d35, d40)
[d33, d38)	Cyclosporine	[d40, d44)
[d33, d38)	Azathioprine	[d42, d46)

I.Time	Drug	D.Time
[d33, d38)	Cyclosporine	[d35, d44)
[d33, d38)	Azathioprine	[d42, d46)

The execution of the first four lines yields the first table, above. Clearly, however, we would prefer to combine its first two tuples into one. This is obtained by the **NORMALIZE ON** expression, which is executed next, yielding the second table above. As another example, the table, in Fig. 4, can also be obtained by the statement

```
SELECT      *
FROM        LAND
NORMALIZE ON Depth, Time
```

The practical importance of **NORMALIZE ON** is that it yields tables normalized with respect to a sequence of columns and each such sequence is related to different semantics [12]. At the same time, it minimizes the typing effort. Specifically, our experimentation has shown that after a table has been reformatted with respect to certain columns, it is usually the case that a Normalization with respect to some other columns, is necessary. Thus, the user can type the first query below, instead of the second one.

```
SELECT      A1, A2, A3, A4
FROM        R, S
REFORMAT AS FOLD A1
NORMALIZE ON A2, A3, A4
```

```
SELECT      A1, A2, A3, A4
FROM        R, S
REFORMAT AS FOLD A1 UNFOLD A2, A3, A4
FOLD A2, A3, A4
```

Line 8 (Sorts rows on the basis of columns): Relatively to the fact that SQL sorts on the columns of a conventional data type with respect to the *less than* (<) predicate, IXSQL sorts, in addition, interval columns with respect to the *precedes* predicate: Intervals are sorted with respect to their *start* point; intervals with the same *start* are subsequently sorted with respect to their *stop* point. Thus, if $R(A, B)$ is the table derived by the execution of lines (1)–(7) and it consists of three tuples, (a, [5, 8)), (a, [1, 10)), (a, [1, 8)) then **ORDER BY** A, B arranges them in the order (a, [1, 8)), (a, [1, 10)), (a, [5, 8)).

4.7 Nested Queries

REFORMAT AS and **NORMALIZE ON** may also appear in a subquery. As in SQL2, the execution of a nested query starts with the subquery at the lowest level and proceeds upwards, to the query at the highest level. The execution of a subquery is similar to the description given in the previous section, that is, a **REFORMAT AS** or **NORMALIZE ON** expression (if present) are executed, in this order, at the end.

EXAMPLE.

"Give the patients who were administered with Cyclosporine for all the days in [d30, d41)."

```

SELECT DISTINCT Name
FROM DRUG D1
WHERE ['d30', 'd41') subinterv ANY
      (SELECT Time
       FROM DRUG D2
       WHERE D1.Name = D2.Name AND
             D2.Drug = 'Cyclosporine'
       NORMALIZE ON Time)

```

4.8 Nonjoin Query Expressions

The syntax of the SQL2 UNION and EXCEPT have been extended in IXSQL, in order to support **union**, **punion**, **except** and **pexcept**, as defined in Section 3. Their IXSQL syntax is:

<query exp 1> {**UNION** | **EXCEPT**} [<reformat column list>]
 <query exp 2>

where <query exp i>, i = 1, 2, yield two union-compatible tables and <reformat column list> is a subset of their columns. If <reformat column list> is missing then an SQL2 UNION or EXCEPT is executed, otherwise the statement is semantically equivalent to

<query exp 1> {**punion** | **pexcept**} [<reformat column list>]
 <query exp 2>

EXAMPLES. Let a table I1, union-compatible to INFECTION contain the single tuple (John, Enterococcus s.p., [d34, d55]) and let a query be

“Combine the data of INFECTION and I1 into a single table.”

```

SELECT *
FROM INFECTION
UNION Time
SELECT *
FROM: I1

```

This yields a table with three tuples, (John, Enterococcus s.p., [d31, d60]), (John, Proteus s.p., [d33, d38]) and (Peter, Proteus s.p., [d45, d50]).

“Give the time intervals during which each patient suffered from Enterococcus s.p. but was not, at the same time, being administered with Cyclosporine.”

```

SELECT Name, Time
FROM INFECTION
WHERE Cause = 'Enterococcus s.p.'
EXCEPT Time
SELECT Name, Time
FROM: DRUG
WHERE Drug = 'Cyclosporine'

```

The result table consists of the tuples (John, [d44, d46]), (John, [d50, d60]). We notice that by using only SQL2, it would not be possible to obtain these tuples.

REFORMAT AS or NORMALIZE ON may be contained in a query expression, in order that two tables become union-compatible, before UNION or EXCEPT is applied to them. For example, assume that we want to combine the data in INFECTION with that in another relation I1, with tuples (John, Enterococcus s.p., d_i), i = 34, 35, ..., 55. Depending on how we would like to view the result data, this can be achieved by either of the following ways:

```

SELECT *
FROM INFECTION
UNION Time
SELECT *
FROM I1
NORMALIZE ON Time

SELECT *
FROM INFECTION
REFORMAT AS UNFOLD Time
SELECT *
FROM I1

```

If <query exp 1> and <query exp 2> yield two union compatible tables, R and S, respectively, then the SQL2 statement R INTERSECT S yields the common rows of R and S. This operation is equivalent to R EXCEPT (R EXCEPT S). In IXSQL, INTERSECT has been extended as

<query exp 1> **INTERSECT** [<reformat column list>]
 <query exp 2>

If <reformat column list> is missing, then an SQL2 INTERSECT is executed, otherwise the statement is semantically equivalent to

<query exp 1> **pexcept** [<reformat column list>]
 <query exp 1> **pexcept** [<reformat column list>]
 <query exp 2>

In this way, if the first expression contains the row (a, [1, 10]) and the second one contains (a, [5, 15]), then the resulting table contains the row (a, [5, 10]), where [5, 10] is the intersection of [1, 10] and [5, 15]. Any of <query exp 1> and <query exp 2> may again contain REFORMAT AS or NORMALIZE ON.

4.9 Extension to CREATE TABLE

The SQL2 CREATE TABLE statement has been extended by one more clause. Two other extensions concern the primary key, which matches the definition given in Section 2, and the UNIQUE constraint. Here, we restrict ourselves to the enhanced features of the syntax, which thus becomes as follows:

```

<table definition> ::= CREATE TABLE <table name>
(<column name> <data type> [{UNIQUE | PRIMARY KEY}
  [{INTERVAL | POINT}]]
{, <column name> <data type> [{UNIQUE | PRIMARY KEY}
  [{INTERVAL | POINT}]] ...}
[, NORMALIZED (<normalized column list>)]
[, PRIMARY KEY (<key column list>)]
[, UNIQUE (<unique column list>)]

```

<key column list> ::=
 <key column name> [INTERVAL | POINT] {,
 <key column name> [INTERVAL | POINT] ...}

<unique column list> ::=
 <unique column name> [INTERVAL | POINT] {,
 <unique column name> [INTERVAL | POINT] ...}

(Note that only one primary key may be declared.) The <normalized column list> is a sublist of the columns of the table, of an interval data type, with respect to which a Normalization must always take place whenever data is inserted, deleted or updated. If a column of an interval type

appears in either the <key column list> or the <unique column list>, it must be followed by one keyword, **INTERVAL** (default) or **POINT**. The rules, to be satisfied by this syntax are the following:

- 1) The <normalized column list> must be followed only by columns of an interval type.
- 2) If **NORMALIZED** (<normalized column list>) has been declared, then the table is always normalized with respect to the columns in <normalized column list>, whenever an insertion, deletion or update takes place. The order of the columns in <normalized column list> also determines the sequence of Normalization.
- 3) If a <key column name> or a <unique column name> is not followed by either **POINT** or **INTERVAL** then **INTERVAL** is assumed.
- 4) If a column appears in <normalized column list> and it also participates in the key then its name, in the definition of the primary key, must be followed by the keyword **POINT**. (Thus the information in the PRIMARY KEY clause is self-contained.)
- 5) Every column which participates in the key and is followed by the keyword **POINT**, must also be present in <normalized column name>.

Using the above rules, the tables in Fig. 1 can be declared to the DBMS as follows:

```
CREATE TABLE TRANSPLANTATION
```

```
(Name CHAR(20),
 Date DATE,
 PRIMARY KEY (Name))
```

```
CREATE TABLE DRUG
```

```
(Name CHAR(20),
 Drug CHAR(30),
 Level INTEGER,
 Time INTERVAL OF DATE,
 NORMALIZED (Time),
 PRIMARY KEY (Name, Drug, Time POINT))
```

```
CREATE TABLE LAND
```

```
(Pno INTEGER,
 Depth INTERVAL OF INTEGER,
 Time INTERVAL OF DATE,
 pH FLOAT,
 NORMALIZED (Time, Depth),
 PRIMARY KEY (Pno, Time POINT, Depth POINT))
```

```
CREATE TABLE DENSITY
```

```
(Length INTERVAL OF INTEGER,
 Width INTERVAL OF INTEGER,
 Population INTEGER,
 PRIMARY KEY (Length INTERVAL, Width INTERVAL))
```

```
CREATE TABLE INFECTION
```

```
(Name CHAR(20),
 Cause CHAR(30),
 Time INTERVAL OF DATE,
 NORMALIZED (Time),
 PRIMARY KEY (Name, Cause, Time POINT))
```

```
CREATE TABLE CARPLATE
```

```
(City CHAR(15),
 Number INTERVAL OF CHAR(7),
 NORMALIZED (Number),
 PRIMARY KEY (Number POINT))
```

Explanations on the above examples are the following: Firstly, TRANSPLANTATION is a valid SQL2 table declaration. Secondly, the primary key of DENSITY consists of columns Length and Width. Exactly as it is the case in SQL2, this does not allow the recording of two distinct tuples whose values for these columns are pairwise identical. Thirdly, DRUG contains "NORMALIZED (Time)". This implies that an automatic Normalization on Time always takes place in data insertion, deletion or update. (The effect of insertions, deletions, and updates is described in the next Section.) The domain of Time is the set of the intervals of dates and we notice that, in the primary key, Time is followed by the keyword **POINT**. Hence, as already described in Section 2, no instance of DRUG may contain two tuples whose values for Name and Drug are identical and their values for Time contain the same date. A final example demonstrates the extension to the UNIQUE constraint. Consider the declaration

```
CREATE TABLE OWNERSHIP (Name CHAR(15),
 Length INTERVAL OF INTEGER,
 Width INTERVAL OF INTEGER,
 PRIMARY KEY (Name, Length, Width),
 UNIQUE (Length POINT, Width POINT))
```

In this table, we can record pieces of land (orthogonal rectangles with sides Length and Width) owned by various people. A piece of land can be owned by more than one person and a person may own more than one such piece. Hence, the primary key is <Name, Length, Width>. The declaration does not include a NORMALIZED(Length, Width). This way, two adjacent pieces of land, owned by the same person, may not be *normalized* into one. Finally, two distinct pieces of land should not overlap. This constraint is enforced by the last line in the table declaration.

In the context of valid time databases, relations like those, above, capture all the valid time semantics discussed in [22] (in fact, these semantics have been extended, in the present work, to application areas other than valid time databases), except one, *semantics for continuous time*. For this latter, interpolation functions are proposed [22]. In their majority, however, all temporal extensions to the relational model, including TSQL2 [10], consider time as discrete, for practical reasons. Finally we notice that the incorporation of interpolation functions is a more general problem, not dedicated solely to the management of valid time data or interval data. As a final remark, we also notice that the primary key of a table relates completely to the semantics which must be captured. As such, it should not be confused with issues of fast data retrieval. For this reason, IXSQL supports completely, and without any changes, the CREATE INDEX statement of SQL2.

4.10 Extension to the Data Manipulation Statements

In this section, we formalize the functionality of the IXSQL manipulation statements. The examples provided should be seen in conjunction with the declaration of the tables in the previous section.

4.10.1 Data Insertion

The syntax of the SQL2 INSERT statement is

```
INSERT INTO <table name> (<column name list>) <query exp>
```

and remains exactly the same in IXSQL. However, according to the explanations already provided in CREATE TABLE, the way data is inserted, depends completely on how a table has been declared: If the table definition does not contain a NORMALIZED expression, then INSERT functions exactly the same way as in SQL, otherwise a Normalization takes place. To formalize the functionality of INSERT, we initially recall that in SQL2 a transaction, to insert the contents of a table S into another table R fails completely, if either of the following occurs:

- 1) The key value of a tuple of R is repeated in a tuple of S.
- 2) S has two tuples with the same key value.

For example, if R(A, B) has key A and consists of the tuple (a1, b1) and another table S consists of either 1) (a1, b1), (a2, b2) (key duplication in R and S) or 2) (a2, b2), (a2, b2), (a2, b3) (key duplication only in S) then the command INSERT INTO R SELECT * FROM S fails. This functionality has been extended in IXSQL. Formally, consider R(A1, A2, ..., An) and let S be another table, union-compatible to R, consisting of the data to be inserted into R. The insertion is then semantically equivalent to the following steps:

- 1) Set a *checkpoint*.
- 2) **If** in CREATE TABLE it has been declared that R is normalized with respect to a subset B1, B2, ..., Bm of its columns

then R := R *punion* [B1, B2, ..., Bm] S
else R := R *union* S

- 3) **If** key duplication (in either R and S or only in S) was identified in the previous step **then** rollback to *checkpoint*.

EXAMPLES.

“Insert into DENSITY the data ([25, 75), [25, 75), 1250000).”

INSERT INTO DENSITY VALUES ([25, 75), [25, 75), 1250000)

This is a typical SQL insertion. If the primary key of DENSITY had not been declared, the tuple would be inserted (though DENSITY would then contain duplicate data).

“Insert into DRUG the data of S, where S consists of the tuples: 1) (John, Azathioprine, 110, [d46, d50)), 2) (John, Azathioprine, 110, [d43, d50)), 3) (John, Azathioprine, 110, [d46, d50)), (John, Azathioprine, 110, [d49, d50)), and 4) (John, Azathioprine, 100, [d43, d50)).”

We recall that DRUG is normalized on Time. Hence, if the primary key has not been declared, then

INSERT INTO DRUG SELECT * FROM S

will function as follows: In Cases 1, 2, and 3, DRUG will contain (‘John’, ‘Azathioprine’, 110, [d42, d50)) in place of its last tuple. In Case 4 it will contain the initial tuples plus the new one.

If the key (Name, Drug, Time **POINT**) has been declared then Case 1 will work again, as described above. In contrast, in Case 2 and Case 4 the insertion will fail (key duplication on DRUG and S, concerning John’s treatment on each of the dates d43, d44, and d45). Case 3 will also fail (key duplication in S, concerning John’s treatment on date d49).

4.10.2 Data Deletion

The SQL2 DELETE statement has been extended as

```
DELETE FROM      <table name>
[PORTION        <normalized column value list>]
[<where clause>]
```

<normalized column value> ::=

<normalized column name> = <value exp>

The PORTION clause is optional. It may appear if and only if the table declaration contains a NORMALIZED clause. In this case, a <normalized column name> must be the name of a column on which Normalization occurs. The <value exp> must be any expression evaluating to an interval type, compatible with that of the preceding column name. For an informal description, let a deletion be

```
DELETE FROM      INFECTION
PORTION          Time = ['d34', 'd55')
WHERE            Name = 'John'
```

Initially, the tuples of INFECTION are identified, which satisfy the <where clause> (tuple nos. 2, 3, 4). Because of the PORTION clause, we next select, from these tuples, those whose values for Time intersect with [d34, d55) and delete them from INFECTION (tuple nos. 2, 3, 4). From these, tuple no. 2,

(John, Enterococcus s.p., [d40, d46))

has a characteristic, that [d40, d46) is a subinterval of [d34, d55). Tuple no. 3 can similarly be *split with respect to* [d34, d55), into two tuples,

(John, Enterococcus s.p., [d50, d55)),
 (John, Enterococcus s.p., [d55, d60)).

The interval [d50, d55) of the first tuple, above, is again a subinterval of [d34, d55). Finally, interval [d33, d38) of tuple no. 4, can also be split, in a similar way, into two tuples,

(John, Proteus s.p., [d34, d38)),
 (John, Proteus s.p., [d33, d34)).

From this set of five tuples above, created this way, the first, second, and fourth contain Time intervals which are subintervals of [d34, d55). These tuples represent exactly the data that has to be deleted, therefore, we eliminate them all. In contrast, the remaining two tuples do not intersect with [d34, d55), thus we insert them into INFECTION and we next normalize INFECTION with respect to Time. Then INFECTION becomes

Name	Cause	Time
John	Enterococcus s.p.	[d31, d34)
John	Enterococcus s.p.	[d55, d60)
John	Proteus s.p.	[d33, d34)
Peter	Proteus s.p.	[d45, d50)

Hence, the command deletes from each tuple of a relation R, which satisfies the deletion criteria, *that portion of data which is explicitly referenced in the list, after the keyword PORTION*. For a formal description, equivalent to the procedure described above, we identify two distinct cases below, depending on whether it has been declared that R is not (is) normalized:

Case 1. It has *not* been declared to the DBMS that R is normalized. Then the syntax of the command is identical to that of SQL2,

```
DELETE FROM      R
[<where clause>]
```

If <where clause> evaluates to S, the result obtained is semantically equivalent to $R := R \text{ except } S$, that is, the command functions as in SQL.

Case 2. It has been declared to the DBMS that R is normalized. Let its scheme be $R(A_1, A_2, \dots, A_p, B_1, B_2, \dots, B_q)$ and that it has been declared that R is normalized with respect to $B_1, \dots, B_q$. If $\{W_1, W_2, \dots, W_r\} \subseteq \{B_1, B_2, \dots, B_q\}$ we define that the result obtained by

```
DELETE FROM      R
[PORTION        W1 = w1, W2 = w2, ..., Wr = wr]
[<where clause>]
```

is semantically equivalent with the table obtained, as follows:

1. If “**PORTION** $W_1 = w_1, W_2 = w_2, \dots, W_r = w_r$ ” is missing, then apply Case 1, above.
2. If “**PORTION** $W_1 = w_1, W_2 = w_2, \dots, W_r = w_r$ ” is present, then perform steps 2.1–2.3, below:

- 2.1 Let S be a table consisting of the rows of R which satisfy

“(<where clause>) **and** ($W_1 \text{ cp } w_1 \text{ and } W_2 \text{ cp } w_2$
and ... and } W_r \text{ cp } w_r)”

(If the <where clause> is missing, this search statement is adjusted appropriately.)

- 2.2 Replace each $S.W_i$ value of S by the respective *intersect* ($S.W_i, w_i$), $i = 1, 2, \dots, r$.

- 2.3 $R := R \text{ pexcept } [B_1, B_2, \dots, B_q] S$

It should also be noted that an interval attribute may appear in both the <where clause> and the <normalized column value list>.

EXAMPLES. All the deletion commands which follow are executed against the contents of LAND, in Fig. 1. The examples show that, in general, the functionality introduced by the PORTION clause cannot be replaced by that of a WHERE clause.

“Delete all the rows of profile 2, whose value for Depth has points in common with [40, 80) and their value for Time has points in common with [d50, d100).”

Its formulation is shown below and results in the deletion of the last row of LAND.

```
DELETE FROM      LAND
WHERE           Pno = 2 AND Depth cp [40, 80) AND
                Time cp [d50, d100)
```

“From all the rows of profile 2, whose value for Depth has points in common with [40, 80), delete the data for all the times in [d50, d100).”

After the execution of the command following, LAND contains (2, [60, 100), [d30, d50), 8.3) in place of its last tuple.

```
DELETE FROM      LAND
PORTION         Time = [d50, d100),
WHERE           Pno = 2 AND Depth cp [40, 80)
```

“From all the rows of profile 2, delete the data for all the times in [d50, d100) and for all the depths in [40, 80).”

After execution of the command below, LAND contains the tuples (2, [60, 80), [d30, d50), 8.3) and (2, [80, 100), [d30, d90), 8.3) in place of its last tuple.

```
DELETE FROM      LAND
PORTION         Time = [d50, d100), Depth = [40, 80)
WHERE           Pno = 2
```

4.10.3 Data Update

The syntax of UPDATE is analogous to that of DELETE. It has been extended as

```
UPDATE          <table name>
[PORTION       <normalized column value list>]
SET            <set clause list>
[<where clause>]
```

where, again, the rules for the PORTION expression obey exactly those for DELETE. For an informal description, consider

```
UPDATE          INFECTION
PORTION         Time = [d34, d38)
SET            Cause = 'Enterococcus s.p.'
WHERE          Name = 'John' AND Cause = 'Proteus s.p.'
```

As in DELETE, the tuples are firstly identified which satisfy the <where clause> (fourth tuple). Because of the PORTION clause, we next select from them, those tuples whose value for Time intersects with [d34, d38) (the same tuple) and delete them from INFECTION. Similarly to the deletion, this selected tuple can then be split into the tuples

(John, Proteus s.p., [d34, d38)),
(John, Proteus s.p., [d33, d34))

The first tuple has a Time interval, which is a subinterval of [d34, d38). The update concerns exactly this tuple. Hence, this tuple is updated. The second tuple remains unchanged. This yields

(John, Enterococcus s.p., [d34, d38)),
(John, Proteus s.p., [d33, d34))

These tuples are inserted into INFECTION. The table is next normalized, thus becoming

Name	Cause	Time
John	Enterococcus s.p.	[d31, d38)
John	Enterococcus s.p.	[d40, d46)
John	Enterococcus s.p.	[d50, d60)
John	Proteus s.p.	[d33, d34)
Peter	Proteus s.p.	[d45, d50)

Therefore, for each row of R which satisfies the update criteria, the extended statement updates that portion, which is explicitly referenced in the list, after the keyword PORTION. Before we formalize UPDATE, we recall that in SQL it fails, if the primary key is duplicated either in the base table and the updated tuples or only in the updated tuples. For example, assume that the key of a relation $R(A, B)$ is A and let (1, 2) be a tuple of it. Assume also that, as a consequence of an update, either 1) some tuple (3, 3) of R becomes (1, 3) or 2) each of two tuples (3, 3) and (4, 4) of R become (2, 3). In either case, the transaction fails. This func-

tionality has been extended in IXSQL. As an example, if the first command below is issued

```
UPDATE      INFECTION
PORTION     Time = ['d33', 'd35')
SET         Cause = 'Enterococcus s.p.'
WHERE       Cause = 'Proteus s.p.'
```

```
UPDATE      INFECTION
PORTION     Time = ['d45', 'd46')
SET         Name = 'Alex'
```

then, similarly with the previous example, the fourth tuple of INFECTION is deleted. Subsequently, this tuple is split. After the update, two tuples are obtained,

```
(John, Enterococcus s.p., [d33, d35))
(John, Proteus s.p., [d35, d38))
```

However, the key of INFECTION is (Name, Cause, Time POINT) and, as can be seen, there will be a duplication of the key values if these tuples are inserted into INFECTION. (The data of the first tuple above for date d33, is also repeated in the first tuple of INFECTION.) Hence, the update transaction will be rejected. Similarly, the second update, above, will fail. Formally, the functionality of UPDATE, as described above, is semantically equivalent to the result obtained by the following sequence of steps, depending on whether a table R is not (is) normalized.

Case 1. If it has *not* been declared to the DBMS that R is normalized, the syntax is

```
UPDATE      R
SET         <set clause list>
[<where clause>]
```

and the following steps describe completely the functionality of the SQL UPDATE:

- 1) Set a *checkpoint*.
- 2) Let S consist of the rows of R to be updated
- 3) $R := R \text{ except } S$
- 4) For each assignment $A = a$ in <set clause list> *replace* by a the value of each row of S for column A.
- 5) $R := R \text{ union } S$
- 6) If a duplicate key value was identified in R, in the previous step, then rollback to *checkpoint*.

Case 2. It has been declared to the DBMS that R is normalized with respect to certain attributes. Assume that its scheme is $R(A_1, A_2, \dots, A_p, B_1, B_2, \dots, B_q)$ and that it has been declared that R is normalized with respect to $B_1, \dots, B_q$. If $\{W_1, W_2, \dots, W_r\} \subseteq \{B_1, B_2, \dots, B_q\}$, we define that the result obtained by

```
UPDATE      R
[PORTION     $W_1 = w_1, W_2 = w_2, \dots, W_r = w_r$ ]
SET         <set clause list>
[<where clause>]
```

is semantically equivalent to the following sequence of steps:

- 1) If “**PORTION** $W_1 = w_1, W_2 = w_2, \dots, W_r = w_r$ ” is missing then perform steps 1.1–1.3 below:
 - 1.1 Perform steps 1–4 of **Case 1**.
 - 1.2 $R := R \text{ punion } [B_1, \dots, B_q] S$

- 1.3 If a duplicate key value was identified in R, in the previous step, then rollback to *checkpoint*.

- 2) If “**PORTION** $W_1 = w_1, W_2 = w_2, \dots, W_r = w_r$ ” is present, then perform steps 2.1–2.7, below.

- 2.1 Set a *checkpoint*.
- 2.2 Let S be a table, union-compatible to R, consisting of the rows of R which satisfy “(<where clause>) **and** ($W_1 \text{ cp } w_1$ **and** $W_2 \text{ cp } w_2$ **and... and** $W_r \text{ cp } w_r$).” (If the <where clause> is missing, this search statement is adjusted appropriately.)
- 2.3 Replace each $S.W_i$ value of S by the respective *intersect*($S.W_i, w_i$).
- 2.4 $R := R \text{ pexcept } [B_1, B_2, \dots, B_q] S$
- 2.5 For each assignment $A = a$ in <set clause list> *replace* by a the value of each row of S for column A.
- 2.6 $R := R \text{ punion } [B_1, B_2, \dots, B_q] S$
- 2.7 If a key duplication was identified in the previous step, then rollback to *checkpoint*.

It should also be noted that an interval attribute may participate in both the <where clause> and the <set clause list> and the <normalized column value list>.

EXAMPLES. The following updates are executed against LAND in Fig. 1.

“Change the interval of dates [d30, d90] of profile 2, to the correct one, [d10, d50].”

After the execution of the command below, the last two rows of LAND are replaced by (2, [0, 100], [d10, d50], 8.3).

```
UPDATE LAND
SET      Time = ['d10', 'd50')
WHERE    Time = ['d30', 'd90') AND Pno = 2
```

“Update LAND to record that for all the days in [d30, d50] the pH value of Profile 2 was 8.1.”

```
UPDATE      LAND
PORTION     Time = ['d30', 'd50')
SET         pH = 8.1
WHERE       Pno = 2
```

Its execution replaces the last two tuples of LAND by (2, [0, 60], [d10, d30], 8.3), (2, [0, 100], [d30, d50], 8.1), and (2, [60, 100], [d50, d90], 8.3).

“Update LAND to record that for all the days in [d50, d90] and all the depths in [60, 100] the pH of profile 2 was 8.1.”

```
UPDATE      LAND
PORTION     Depth = [60, 100), Time = ['d50', 'd90')
SET         pH = 8.1
WHERE       Pno = 2
```

After execution, the last tuple of LAND is replaced by (2, [60, 100], [d30, d50], 8.3) and (2, [60, 100], [d50, d90], 8.1).

4.11 Enhanced Views

The syntax of the view definition remains exactly the same as in SQL2, except that now a *selection* may also include a REFORMAT AS or a NORMALIZE ON expression. For example, assume that a doctor is interested for the drugs

administered to each patient, but not for the particular drug dosages. The relevant view is defined as:

```
CREATE VIEW      DRUG1 AS
SELECT          Name, Drug, Time
FROM            DRUG
NORMALIZE ON    Time
```

5 SPECIFICATION DECISIONS

We are not aware of any other SQL extension for interval data management. Dedicated extensions have however been proposed for temporal data management. Some of them incorporate only one type of interval data, *time interval*. In this section therefore, we justify certain IXSQL specification decisions and clarify their difference from valid time extensions to the relational model.

5.1 Representation of Intervals

An easy way, to model time intervals, is to maintain two attributes, *start* and *stop*, to record the boundaries of a time interval [8], [9], [15]. A common characteristic in these approaches is that all relations incorporate *only one* valid time interval. In our approach an interval is supported as a primitive data type, for the following reasons:

- 1) In our opinion this is more natural, in that it is closer to the human perception.
- 2) The DBMS takes automatic action for the maintenance of interval attributes. For example, it disallows the recording of an interval v , if $start(v) > stop(v)$ (*invalid interval*). In contrast, if two attributes are maintained, the same result can only be achieved if the user declares explicitly to the DBMS the pair of attributes which form intervals.
- 3) Using interval types, certain queries are easier to formulate.
- 4) In our approach a relation may consist of many, not only one interval attribute. If start and stop attributes were used for each distinct interval attribute, it is likely that the user would easily lose overall control of the attributes which have to be considered in pairs.

5.2 Interval Extension to the Relational Model

In our approach we have extended Codd's relational model [17]. In temporal databases, however, valid time relational extensions are classified into two categories, *nongrouped* and *grouped* models [23]. One basic difference between the two is that the former satisfy first-normal form, whereas the latter do not. In our opinion, a good grouped model is more powerful than a nongrouped. This is also true in a comparison between nongrouped and grouped interval extensions to the relational model. Our decision, to define an SQL extension for a *nongrouped model*, has been based on a simple reason: There are various proposals for a non-1NF model [20], [21] and extensions to SQL [20], [24], but no *reference* non-1NF model and SQL extension has yet been stabilized. Some initial attempts of such an extension, concerning an object-oriented SQL, can be found in [25] but, still, it will take some time before a final version is reached. In addition, indexing techniques and optimization strategies for such models are

still open to research, and they have to be seriously considered, before a commercial product is developed. In contrast, IXSQL is a minimal interval extension, semantically and syntactically consistent with SQL2. As such, the implementation effort to extend SQL2 can also be minimized. At the same time, IXSQL can easily be adopted by relevant applications, since it contributes substantially in the minimisation of the programming effort for interval data management. Finally, in our view a complete non-1NF extension to IXSQL should be compatible with the final version of SQL3.

5.3 Representation of Time

In the majority of temporal extensions to the relational model, valid time is represented either as a point or as an interval. However, there are extensions in which time is represented either as a set of time points [10] or as a union of disjoint, nonintersecting intervals [6], [10], [26]. Our approach, however, is not dedicated to the management of valid time data. Therefore, if we were to support these structures of valid time then, for uniformity reasons, we should also support similar structures for other types of data, such as sets of points of CHAR(10) or union of intervals over CHAR(10). We then would end up, again, with a grouped model. However, the formalization of such a model has been avoided for reasons explained in Section 5.2. For the same reason, in our approach we do not distinguish between *user defined time* and *valid time* [15] or between *implicit valid time* [15] and *explicit data*, distinctions made in almost all valid time models.

5.4 Incorporation of a Single Qualification Condition

Valid time extensions to SQL are dedicated to the management of only one particular type of data, *valid time data*. This might be a justification of the fact that some of these extensions incorporate both a *<where-clause>* and *<when clause>* for retrievals [8], [15]. However, IXSQL aims at handling *any* type of *point* or *interval* data. In this sense, it would not be reasonable for its syntax to include an individual *<when clause>*, specially tailored only for temporal qualifications.

5.5 Nonautomatic Normalization

In valid time SQL extensions, the SQL operations have been extended appropriately, in a way which achieves an automatic normalization [8], [10], [26]. This has an advantage, that a query can be expressed in a simpler way. Contrasting to this, in IXSQL a NORMALIZE ON expression has to be explicitly included in both nonnested and nested queries. Roughly speaking, a normalization in IXSQL is, in general, required on those attributes, with respect to which it has been declared that a relation is normalized. In our approach, it was a hard decision to sacrifice the simplicity of other approaches, but this was based on the following justification:

Firstly, a relation may have many interval attributes. Since the sequence in which normalization occurs is associated with relevant semantics [12], the user should have direct control of this normalization sequence.

Secondly, an automatic normalization may weaken the power of the language, even for a valid time relation. As an example, consider the query

“list the time intervals having the property that during each of them, the daily Cyclosporine level, John was administered with, was steady.”

Its formulation is shown below, and yields the first three Time intervals. It is obvious that if we had allowed an automatic Normalization, the result would be a single interval, [d30, d44), which *does not* answer the query.

```
SELECT      Time
FROM        DRUG
WHERE       Name = 'John' AND Drug = 'Cyclosporine'
```

Incidentally, it should be noted that the inverse of the above example may also occur: A relation may not be declared to be normalized, yet a Normalization may be necessary in a query. To make it clear, consider the query

“list the subrectangles of [1, 200)X[1, 200), for which no population has been recorded in DENSITY.”

The query is shown below and yields two tuples, ([100, 200), [1, 100)) and ([1, 200), [100, 200)).

```
VALUES      ([1, 200), [1, 200))
EXCEPT    Length, Width
SELECT      Length, Width
FROM        DENSITY
```

5.6 Support of Relations with More than One Valid Time Attribute

Consider the query

“list the time John was administered with Azathioprine and the time he was infected by Proteus s.p.”

Using IXSQL (SQL2, in fact), it is formulated as shown below and yields the left table. (It should be noted that in a further extension of IXSQL, supporting grouped data, a better representation, could have been obtained, the right table, below.)

```
SELECT      D.Time, I.Time
FROM        DRUG D, INFECTION I
WHERE       D.Name = 'John'
            AND I.Name = 'John'
            AND Drug = 'Azathioprine'
            AND Cause = 'Proteus s.p.'
```

[d42, d46)	[d33, d38)	[d42, d46)	[d33, d38)
[d42, d46)	[d45, d50)		[d45, d50)

The basic characteristic of the result table is that it consists of *two* attributes, each of which contains *only* valid time. In our opinion, this flexibility of IXSQL is an advantage, if compared with other valid time extensions: Except perhaps [9], in all other approaches a table may not, by definition be composed of two distinct valid time attributes. For example, in TSQL2 [10], this query cannot be formulated unless *valid time* is transformed to *user-defined time*. More generally, a relation may not have more than one valid time attribute in the proposed valid time extensions.

5.7 Participation of Time in the Key

As has been shown in IXSQL, interval data, and subsequently valid time, can participate in the key. Now, assume that we are interested in recording, in a table, the times at

which processes are assigned to the CPU, in a multiprogramming, single processor environment. In IXSQL the table can be declared as

```
CREATE TABLE ASSIGNMENT
(Processid      INTEGER,
Time           INTERVAL OF TIMESTAMP(6),
               NORMALIZED(Time),
PRIMARY KEY (Time POINT))
```

Thus, the DBMS automatically disallows the recording of two rows with overlapping time intervals. This is a reasonable requirement, since two processes may not appear as being in execution mode, at the same time. In our opinion, this property is also an advantage over other valid time extension approaches. In particular, valid time may not participate in the key, in any of them. As a consequence, the recording of nonoverlapping valid time intervals can only be achieved, in these approaches, by the assertion of one additional integrity constraint. To our knowledge, however, such constraints have not been proposed in any of these extensions, including TSQL2 [10].

5.8 Support of Span Data Types

If we *subtract* a time point p_i from another point p_j , we obtain a result which represents the *relative* number of points from p_i to p_j . For example, ‘1994/01/05’ – ‘1994/01/01’ = 4 DAYS, ‘1994/01/01’ – ‘1994/01/05’ = – 4 DAYS, and ‘1994/01/04, 5 HOURS’ – ‘1994/01/01, 3 HOURS’ = 3 DAYS, 2 HOURS. According to the SQL2 [1] (TSQL2 [10]) terminology, the result obtained this way is of an *interval (span)* type. Contrary to these types, in the present work we defined functions which return a result of an *integer* type (e.g., *dist(date ‘1994/01/05’, date ‘1994/01/01’)* = 4). In our opinion, the support of an SQL2 *interval* (TSQL2 *span*) type represents only one special case of the necessity to support a more generic type, *composite numbers*. Values of such a type are not only 4 DAYS, 3 HOURS, but also 4 POUNDS, 2 OUNCES and 4 FEET, 2 INCHES, with practical importance in many applications. A formalization of such a parametric type and relevant operations have been investigated in [27]. Furthermore, operations between them and compatible *compound* data types (*composite* 1-d spaces), such as DATE:HOURL:MINUTE, have been investigated in [28].

6 FURTHER EXTENSION TO IXSQL

IXSQL supports only intervals closed to the left and open to the right. Consequently, all other interval types are supported indirectly. (For example, the intervals [1, 10], (1, 10), and (1, 10] can be expressed in IXSQL as [1, 11), [2, 10), and [2, 11), respectively.) In practice, however, a user should have the flexibility to directly declare the type of interval which fits best his application. IXSQL can easily be extended to this end. The necessary extensions are the following:

- Specification of the interval type in CREATE TABLE: For example, A INTERVAL OF INTEGER(C, O) can be used to denote intervals attribute A, *closed* on the left and *open* to the right.

- Specification of the interval type in operation **fold**: If **fold** is applied to an attribute *A* in which *point* data has been recorded, then the syntax of **fold** can be extended as, **fold(o, o)** *A*, **fold(o, c)** *A*, **fold(c, c)** *A*, or **fold(c, o)** *A* to indicate the type of interval to be produced.
- Adjustment of the definitions of predicates and functions: Their definition should be adjusted appropriately, so as to yield the intended result.
- Finally, four new value functions, **co**(*<interval expression>*), **cc**(*<interval expression>*), **oc**(*<interval expression>*), and **cc**(*<interval expression>*), should be defined, to perform transformations between various types of intervals.

7 IMPLEMENTATION ISSUES

Language design, including extensions to SQL [8], [9], [24], focuses mainly on the functionality of the language, whereas implementation issues are usually neglected. Although the implementation of IXSQL is beyond the scope of the present paper, in this section we report briefly on a prototype implementation and on certain other implementation issues.

A substantial implementation of IXSQL has been successfully completed in ORES (Greek word for *HOURS*), an ESPRIT III project, aiming at the management of valid time data. Some characteristics of this implementation are the following:

Contrary to the formalization proposed in the present paper, the implementation was restricted to an extension of SQL89, given that, at the time the project was initiated, SQL2 was not yet available. The implementation took place on a Sun workstation, with SunOS 4.1.3. Only one interval type is supported, DATEINTERVAL, whose format is [yyyy-mm-dd, yyyy-mm-dd). DATEINTERVAL, as well as the interval predicates and functions, have been implemented directly within the kernel of INGRES, release 6.4. The semantics captured by a valid time relation, as well as the functionality of operations insert, delete and update, as described in the present paper, are also supported.

Some optimization has been incorporated in the implementation. For example, although operations **unfold** and **fold** theoretically simplify the formalization of **normalize**, **union**, **except**, etc., the use of **unfold**, in an actual implementation is practically prohibitive. For this reason, the implementation incorporates, internally, another operation, **split** [18, 29], which reduces drastically the execution time. Main storage structures are also used, to reduce disk I/O operations [30]. Finally, certain **unfold** or **fold** operations, explicitly issued by the user, are internally eliminated, if they are determined to be redundant for the derivation of the result relation. Details of this implementation are reported in [31]. Clearly, there is still a lot of work to be undertaken. Some issues to be addressed are the following:

Support of Parametric Interval Data Types. In IXSQL, interval data is *parametric*, that is, for each type *x* of textual data (integers, strings etc.) the respective interval type is declared as **interval of x**. This has certain advantages:

- 1) *Simplicity.* The user does not have to memorize a distinct interval data type name, for each distinct type of textual data.
- 2) *Compliance with SQL2.* Most SQL92 data types are parametric (CHAR(*x*), VARCHAR(*x*), NCHAR(*x*), BIT(*x*), BIT VARYING(*x*), DEC(*p*, *s*), FLOAT(*x*), TIME(*precision*), TIMESTAMP(*precision*)). All such types have common properties. For example, every function which can be applied to a string of CHAR(5) can also be applied to a string of CHAR(10) or CHAR(15). The same is also true for intervals, no matter whether they are intervals of dates or integers or CHAR(10). Perhaps a sophisticated implementation, to support all the interval types, should consider the internal use of a limited number of *base* data types.

Support of Intervals for Each SQL2 Textual Data Type.

In our formalization an interval type has been defined for each type of textual data, assuming that this data can be ordered linearly. The major advantage of this approach is that, from a theoretical point of view, a high degree of uniformity is achieved: We did not have to distinguish between: 1) data types of SQL2 for which the respective interval types should be supported and 2) data types for which interval types should *not* be supported. From a practical point of view, however, it could be argued that certain interval types, such as **INTERVAL OF VARCHAR(*n*)**, *n* = 32,000, are not likely to be useful. On the other hand, however, it is most likely that, for *n* = 10, intervals of this type are useful in a number of applications. We believe, therefore, that the determination of a constant *n₀*, beyond which VARCHAR(*n*) intervals, for *n* > *n₀*, should not be supported, is a decision usually made by the DBMS designer.

Support of Unfold for Each Interval Type. Similarly with the previous discussion, there is a theoretical advantage, that **unfold** can be applied to *every* type of interval data. We notice, however, that although intervals of float can be useful in scientific applications, it is not obvious that applying **unfold** to them has some practical importance. In our opinion, therefore, this particular operation does not really have to be implemented for float intervals, unless justified from a practical point of view. The same remark also applies to various functions, such as *succ* and *countdp*.

Optimization. The advantage of REFORMAT is that distinct users can view the same interval data in different ways, independently of the way this data has actually been designed and physically stored. At the same time, optimization strategies can be integrated in the DBMS. It might be argued, however, that an **unfold**, implicitly contained in REFORMAT, may result in an overflow of the disk capacity. This is a query optimization problem, which is beyond the purposes of the present paper. However, the following can be noticed: Firstly, an initial investigation we have undertaken in ORES, has shown that, in SQL89, an **unfold**, within a subquery can actually be eliminated. This can be achieved by transforming the query issued by the user, to an equivalent one, whose execution does not necessitate the actual execution of **unfold**. Secondly, as reported earlier, some optimization has been undertaken within ORES, which eliminates the execution of redundant **unfold** operations. Thirdly, the execution of a query may virtually pro-

duce millions of *unfolded* tuples, but this is independent of the way the data is physically stored. For example, the space occupied by the data in the table

```
CREATE TABLE T AS
SELECT  A
FROM    R
REFORMAT AS UNFOLD A
```

does not necessarily have to be more than that required, to store the table obtained by the execution of the first three lines. The same is also true for the time taken to execute this query. Various alternatives, open to investigation, are: 1) Internal transformation of such queries to equivalent ones, not requiring the actual execution of *unfold*, and 2) Lazy execution of *unfold*, if it is determined that it cannot be eliminated, in conjunction with pipelining.

Embedded IXSQL. Existing programming languages do not support directly various types of interval data. Hence, provision has to be made, in the implementation of embedded IXSQL, for intervals to be transformed to valid data types of the language. For example, an interval of dates could be returned to the host language as a character string. More generally, however, embedded IXSQL is an issue to be further investigated.

8 CONCLUSIONS

We provided the specifications of IXSQL, an *Interval Extension* to SQL2, for the management of n-dimensional interval

data. Major effort was made for this extension to be both minimal and powerful: With respect to minimality, SELECT has been extended by two clauses, whereas each of CREATE TABLE, UPDATE, and DELETE has been extended by one clause. UNION, EXCEPT, and INTERSECT have also been slightly extended. Although extensions to the SQL2 JOIN and PRODUCT could have also been proposed, this was intentionally avoided, since it was estimated that these operations are already particularly complicated. The proposed extension is also powerful: It captures new semantics and can effectively handle various types of interval data. It has many applications too, one of which is the management of valid time data. The language has also been evaluated against actual user requirements for the management of such data [13], with outstanding results:

- 1) Only two more functions, *window* and *windowno* had to be added to the initial IXSQL specification.
- 2) It was identified that IXSQL can also be used for the formulation of queries which are more complicated than the user requirements [32].

A subset of IXSQL, dedicated to the management of valid time data, has been implemented, and it incorporates Optimization techniques [18]. In terms of implementation, further work includes the support of various interval types and the relevant operations, the execution of IXSQL in a multiuser environment, the support of embedded IXSQL and the update of views.

APPENDIX A IXSQL FUNCTIONS AND PREDICATES

Notation

The following notation is used in the sequel:

Sets

I	: The set of integers.		
R	: The set of reals.	$D = \{p_1, p_2, \dots, p_n\}$	: Some 1-d space.
$P(S)$	: The powerset of a set S.	$I(D) = \{[p_i, p_j] \mid p_i, p_j \in D, p_i < p_j\}$	: A set of intervals over D.

Functions

<i>trunc</i> (x)	: Returns the integer part of a real number x.
<i>D div d</i>	: Returns the integer quotient of the division of D by d, where $D, d \in I$.
$ x $	: Returns the absolute number of a number x.
$\ S\ $	: Returns the number of points in a set S.
<i>points</i>	: $I(D) \rightarrow D : points([p_i, p_j]) = \{p_i, p_{i+1}, \dots, p_{j-1}\}$.

VALUE FUNCTIONS

<i>mindate</i> ()	: Returns the least date supported by the DBMS.
<i>maxdate</i> ()	: Returns the greatest date supported by the DBMS.
<i>start</i>	: $I(D) \rightarrow D : start([p_i, p_j]) = p_i$
<i>stop</i>	: $I(D) \rightarrow D : stop([p_i, p_j]) = p_j$
<i>topoint</i>	: $I(D) \rightarrow D : topoint([p_i, p_{i+1}]) = p_i$
<i>tointerv</i>	: $D \rightarrow I(D) : tointerv(p_i) = [p_i, p_{i+1}]$
<i>interv</i>	: $D \times D \rightarrow I(D) : interv(p_i, p_j) = [p_i, p_j], i < j$
<i>intervsect</i>	: $I(D) \times I(D) \rightarrow I(D) : intervsect([p_i, p_j], [p_r, p_s]) = [max(\{p_i, p_r\}), min(\{p_j, p_s\})], [p_i, p_j] \cap [p_r, p_s] \neq \emptyset$
<i>succ</i>	: $D \times I \rightarrow D : succ(p_i, k) = p_{i+k}, 1 \leq i + k \leq n$
<i>dur</i>	: $I(D) \rightarrow I : dur([p_i, p_j]) = j - i$
<i>span</i>	: $D \times D \rightarrow I : span(p_i, p_j) = i - j$
<i>dist</i>	: $D \times D \rightarrow I : dist(p_i, p_j) = i - j $

$$\begin{aligned}
\text{middle} & : I(D) \rightarrow D: \text{middle}([p_i, p_j]) = p_{i+\text{trunc}((j-i)/2)} \\
\text{merge} & : I(D) \times I(D) \rightarrow I(D): \text{merge}([p_i, p_j], [p_r, p_s]) = [\min(\{p_i, p_r\}), \max(\{p_j, p_s\})], j \geq r, s \geq i \\
\text{window} & : D \times I \times I \rightarrow I(D): \text{window}(p_i, m, n) = [p_{j+m*(n-1)}, p_{i+m*n}], m > 0, n > 0 \\
& \quad \text{window}(p_i, m, n) = [p_{i+m*n}, p_{i+m*(n+1)}], m > 0, n < 0 \\
\text{windowno} & : D \times I \times D \rightarrow I: \text{windowno}(p_i, m, p_j) = ((j-i) \text{div } m) + 1, j \geq i, m > 0 \\
& \quad \text{windowno}(p_i, m, p_j) = -((i-j+m-1) \text{div } m), j < i, m > 0
\end{aligned}$$

SET FUNCTIONS

$$\text{countap}: P(I(D)) \rightarrow (I : \text{countap}(V) = \text{dur}(v_1) + \dots + \text{dur}(v_m),$$

where $V = \{v_1, \dots, v_m\}$

$$\text{countdp}: P(I(D)) \rightarrow I : \text{countdp}(V) = ||\text{points}(v_1) \cup (\text{points}(v_2) \cup \dots \cup \text{points}(v_m))||,$$

where $V = \{v_1, \dots, v_m\}$

INTERVAL PREDICATES

(1)	v	<i>before</i>	w
(2)	v	<i>meets</i>	w
(3)	v	<i>loverslaps</i>	w
(4)	v	<i>lcovers</i>	w
(5)	v	<i>covers</i>	w
(6)	v	<i>rcovers</i>	w
(7)	v	$=$	w
(8)	v	<i>rcovers</i>	w
(9)	v	<i>covered</i>	w
(10)	v	<i>lcovered</i>	w
(11)	v	<i>roverlaps</i>	w
(12)	v	<i>met</i>	w
(13)	v	<i>after</i>	w

(14)	v	<i>psubinterv</i>	$w \Leftrightarrow (6 \vee 9 \vee 10)$	$(v \text{ is a pure subinterval of } w)$
(15)	v	<i>subinterv</i>	$w \Leftrightarrow (14 \vee 7)$	$(v \text{ is a subinterval of } w)$
(16)	v	<i>psupinterv</i>	$w \Leftrightarrow (4 \vee 5 \vee 8)$	$(v \text{ is a pure superinterval of } w)$
(17)	v	<i>supinterv</i>	$w \Leftrightarrow (16 \vee 7)$	$(v \text{ is a superinterval of } w)$
(18)	v	<i>overlaps</i>	$w \Leftrightarrow (3 \vee 11)$	
(19)	v	<i>cp</i>	$w \Leftrightarrow (3 \vee 4 \vee \dots \vee 11)$	$(v \text{ has common points with } w)$
(20)	v	<i>merges</i>	$w \Leftrightarrow (19 \vee 2 \vee 12)$	
(21)	v	<i>precedes</i>	$w \Leftrightarrow (1 \vee 2 \vee 3 \vee 4 \vee 5 \vee 6)$	
(22)	v	<i>prequals</i>	$w \Leftrightarrow (21 \vee 7)$	$(v \text{ precedes or equals } w)$
(23)	v	<i>follows</i>	$w \Leftrightarrow (8 \vee 9 \vee 10 \vee 11 \vee 12 \vee 13)$	
(24)	v	<i>folequals</i>	$w \Leftrightarrow (23 \vee 7)$	$(v \text{ follows or equals } w)$
(25)	v	<i>adjacent</i>	$w \Leftrightarrow (2 \vee 12)$	

APPENDIX B

IXSQL EXTENSIONS TO SQL2

Notes

- 1) Only the extensions to SQL2 are given in the following, therefore it is assumed that the SQL2 documentation is available for reference.
- 2) Wherever <SQL2 literal> or <SQL2 data type> or <SQL2 value exp> appears in the following, *noninterval* literal or data type or value expression, respectively, is assumed.

LEXICAL ELEMENTS

<IXSQL literal> ::= <SQL2 literal> | <IXSQL interval literal>
 <IXSQL interval literal> ::= [<SQL2 literal>, <SQL2 literal>]

SCALAR EXPRESSIONS

<IXSQL data type> ::= <SQL2 data type> | <IXSQL interval data type>
 <IXSQL interval data type> ::= INTERVAL OF <SQL2 data type>
 <IXSQL set function spec> ::= <SQL2 set function spec>
 | {COUNTDP | COUNTAP} ([ALL | DISTINCT] <IXSQL interval value exp>)
 <IXSQL interval value exp> ::= <value expression which returns an IXSQL interval type>
 <IXSQL value function> ::= <SQL2 value function> | <IXSQL value function>
 <IXSQL value function> ::= *mindate()*
 | *maxdate()*
 | *start* (<IXSQL Interval value exp>)
 | *stop* (<IXSQL interval value exp>)
 | *topoint* (<IXSQL interval value exp>)
 | *tointerv* (<SQL2 value exp>)
 | *interv* (<SQL2 value exp>, <SQL2 value exp>)
 | *intersect* (<IXSQL interval value exp>, <IXSQL interval value exp>)
 | *succ* (<SQL2 value exp>, <integer exp>)
 | *dur* (<IXSQL interval value exp>)
 | *span* (<SQL2 value exp>, <SQL2 value exp>)
 | *dist* (<SQL2 value exp>, <SQL2 value exp>)
 | *middle* (<IXSQL interval value exp>)
 | *merge* (<IXSQL interval value exp>, <IXSQL interval value exp>)
 | *window* (<SQL2 value exp>, <integer exp>, <integer exp>)
 | *windowno* (<SQL2 value exp>, <integer exp>, <SQL2 value exp>)

QUERY EXPRESSIONS

<IXSQL row value constructor> ::= (<SQL2 row value constructor>) [<reformat clause>] [<normalize clause>]
 <IXSQL table value constructor> ::= (<SQL2 table value constructor>) [<reformat clause>] [<normalize clause>]
 <IXSQL joined table> ::= (<SQL2 joined table>) [<reformat clause>] [<normalize clause>]
 <IXSQL query spec> ::= <SQL2 query spec> [<reformat clause>] [<normalize clause>]
 <IXSQL non-join query exp> ::= (<IXSQL non-join query term>) [<reformat clause>] [<normalize clause>]
 | (<query exp> UNION [ALL] [<reformat column list>] [<corresponding spec>] <query term>)
 | [<reformat clause>] [<normalize clause>]
 | (<query exp> EXCEPT [ALL] [<reformat column list>] [<corresponding spec>] <query term>)
 | [<reformat clause>] [<normalize clause>]
 <IXSQL nonjoin query term> ::= <nonjoin query primary>
 | <query term> INTERSECT [ALL] [<reformat column list>] [<corresponding spec>] <query primary>
 <IXSQL explicit table> ::= <SQL2 explicit table> [<reformat clause>] [<normalize clause>]
 <reformat clause> ::= REFORMAT AS <reformat item>
 <reformat item> ::= FOLD <reformat column list> [<reformat item>]
 | UNFOLD <reformat column list> [<reformat item>]
 <normalize clause> ::= NORMALIZE ON <reformat column list>
 <reformat column> ::= <column reference> | <unsigned integer>

PREDICATES

<IXSQL predicate> ::= <SQL2 predicate> | <IXSQL interval predicate>
 <IXSQL interval predicate> ::= *before* | *meets* | *loverslaps* | *lcovers* | *covers* | *rcovers* | *=* | *rcovered*
 | *covered* | *lcovered* | *roverslaps* | *met* | *after* | *psubinterv* | *subinterv* | *psupinterv* | *supinterv*
 | *overlaps* | *cp* | *merges* | *precedes* | *prequals* | *follows* | *folequals* | *adjacent*

SCHEMA DEFINITION AND MANIPULATION

<IXSQL table definition> ::=
 CREATE [{GLOBAL | LOCAL} TEMPORARY] TABLE <table name>
 <table element list> [ON COMMIT {DELETE | PRESERVE} ROWS]
 <table element> ::= <column definition> | <table constraint definition> | <table interval semantics definition>

<table interval semantics definition> ::= **NORMALIZED (normalized column list)**

<normalized column list> ::= <IXSQL interval type column name>

<IXSQL unique constraint definition> ::= <IXSQL unique spec> (<IXSQL unique column list>)

<IXSQL unique spec> ::= {UNIQUE | PRIMARY KEY} [POINT | INTERVAL]

<IXSQL unique column> ::= <column name of SQL2 data type>
| <IXSQL interval type column name>[POINT | INTERVAL]

DATA MANIPULATION

<IXSQL delete stat: searched> ::=

DELETE FROM <table name>

[PORTION <normalized column value list>]

[<where clause>]

<IXSQL update stat: searched> ::=

UPDATE <table name>

[PORTION <normalized column value list>]

SET <set clause list>

[<where clause>]

<normalized column value> ::= <IXSQL interval type column name> = <value exp>

ACKNOWLEDGMENT

The authors are grateful to the referees, whose comments helped in substantially improving the paper.

REFERENCES

- [1] "The International Organization for Standardization," *Database Language SQL*. ISO 9075:1992(E), 1992.
- [2] S.J. Cannan and G.A.M. Otten, *SQL: The Standard Handbook*. McGraw-Hill, 1993.
- [3] C. Dyreson, F. Grandi, W. Kater, N. Kline, N. Lorentzos, Y. Mitsopoulos, A. Montanari, D. Nonen, E. Peressi, B. Pernici, J. Roddick, N. Sarda, M. Scalas, A. Segev, R. Snodgrass, M. Soo, A. Tansel, P. Tiberio, and G. Widerhold, "A Consensus Glossary of Temporal Database Concepts," C. Jensen, J. Clifford, R. Elmasri, S. Gadia, P. Hayes, and S. Jagodia, eds., *SIGMOD Record*, vol. 23, no. 1, pp. 52-64, Mar. 1994.
- [4] G. Ariav, "A Temporally Oriented Data Model," *ACM Trans. Database Systems*, vol. 11, no. 4, pp. 499-527, Dec. 1986.
- [5] J. Clifford and A. Croker, "The Historical Relational Data Model (HRDM) and Algebra Based on Lifespans," *Proc. Third Int'l Conf. Data Eng.*, pp. 528-537, Feb. 1987.
- [6] S.K. Gadia, "A Homogeneous Relational Model and Query Languages for Temporal Databases," *ACM Trans. Database Systems*, vol. 13, no. 4, pp. 418-448, Dec. 1988.
- [7] E. McKenzie and R. Snodgrass, "Supporting Valid Time: An Historical Algebra," Technical Report TR87-008, Computer Science Dept., Univ. of North Carolina, Chapel Hill, Aug. 1987.
- [8] S.B. Navathe and R. Ahmed, "Temporal Extensions to the Relational Model and SQL," *Temporal Databases: Theory, Design, and Implementation*. J. Clifford and R. Snodgrass et al., eds., Benjamin Cummings, pp. 92-109, 1993.
- [9] N.L. Sarda, "Extensions to SQL for Historical Databases," *IEEE Trans. Knowledge and Data Eng.*, vol. 2, no. 2, pp. 220-230, June 1990.
- [10] R.T. Snodgrass, I. Ahn, G. Ariav, D. Batory, J. Clifford, C.E. Dyreson, R. Elmasri, F. Grandi, C.S. Jensen, W. Kafer, N. Kline, K. Kulkarni, T.Y.C. Leung, N. Lorentzos, J.F. Roddick, A. Segev, M.D. Soo, and S.M. Spirada, *The SQL2 Temporal Query Language*. R.T. Snodgrass, ed., Kluwer Academic Publishers, 1995.
- [11] A.U. Tansel, "A Historical Query Language," *Information Sciences*, vol. 53, pp. 101-133, 1991.
- [12] N.A. Lorentzos, "The Interval Extended Relational Model and Its Application to Valid Time Databases," *Temporal Databases: Theory, Design, and Implementation*. J. Clifford and R. Snodgrass, eds., pp. 67-91. Benjamin Cummings, 1993.
- [13] C.H. Salvator and A. Munoz, "TDBMS, User Requirements," ORES Deliverable B2, Dec. 1993.
- [14] N.A. Lorentzos, "Semantics in Databases with Interval Support," technical report, Informatics Laboratory, Agricultural Univ. of Athens, Sept. 1993.
- [15] R.T. Snodgrass, "The Temporal Query Language TQUEL," *ACM Trans. Database Systems*, vol. 12, no. 2, pp. 247-298, July 1987.
- [16] K.K. Al-Taha, R.T. Snodgrass and M.D. Soo, "Bibliography on Spatiotemporal Databases," *SIGMOD Record*, vol. 22, no. 1, pp. 59-67, Mar. 1993.
- [17] E.F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Comm. ACM*, vol. 13, no. 6, pp. 377-387, June 1970.
- [18] N.A. Lorentzos, A. Poulouvasilis, and C. Small, "Manipulation Operations for an Interval-Extended Relational Model," *Data and Knowledge Engineering*, vol. 17, pp. 1-29, 1995.
- [19] N.A. Lorentzos and R.G. Johnson, "TRA: Model for a Temporal Relational Algebra," *Proc. Temporal Aspects in Information Systems*, Sophia-Antipolis, France, May 1987. (in *Temporal Aspects in Information Systems*, C. Rolland, F. Bodart, and M. Leonard, eds., pp. 203-215, North Holland, 1988).
- [20] P. Dadam, K. Kuespert, F. Anderson, H. Blanken, R. Erbe, J. Guenauer, V. Lum, P. Piştor, and G. Walch, "A DBMS Prototype to Support Extended NF² Relations: An Integrated View on Flat Tables and Hierarchies," *Proc. ACM SIGMOD '86 Int'l Conf. Management of Data*, May 1986.
- [21] G. Ozsoyoglu, Z.M. Ozsoyoglu, and V. Matos, "Extending Relational Algebra and Relational Calculus with Set-Valued Attributes and Aggregate Functions," *ACM Trans. Database Systems*, vol. 12, no. 4, pp. 566-592, Dec. 1987.
- [22] A. Segev, and A. Shoshani, "Logical Modeling of Temporal Data," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, San Francisco, pp. 454-466, 1987.
- [23] J. Clifford, A. Croker, and A. Tuzhilin, "On Completeness of Historical Relational Query Languages," *ACM Trans. Database Systems*, vol. 19, no. 1, pp. 64-116, Mar. 1994.
- [24] M.A. Roth, H.F. Korth, and D.S. Batory, "SQL/NF: A Query Language for Non-1NF Relational Databases," *Information Systems*, vol. 12, no. 1, pp. 99-114, 1987.
- [25] "Database Language SQL (SQL3)," ISO-ANSI Working Draft. X3H2-93-091 and YOK-003, May 1993.
- [26] S.K. Gadia and G. Bhargava, "SQL-Like Seamless Query of Temporal Data," *Proc. Int'l Workshop An Infrastructure for Temporal Databases*, Arlington, Texas, June 1993.
- [27] N.A. Lorentzos, "DBMS Support for Non-Metric Measurement Systems," *IEEE Trans. Knowledge and Data Eng.*, vol. 6, no. 6, pp. 945-953, Dec. 1994.
- [28] N.A. Lorentzos, "DBMS Support for Time and Totally Ordered Compound Data Types," *Information Systems*, vol. 17, no. 5, pp. 347-358, Sept. 1992.
- [29] N.A. Lorentzos, A. Poulouvasilis, and C. Small, "Implementation of Update Operations for Interval Relations," *Computer J.*, vol. 37, no. 3, pp. 163-176, 1994.

- [30] N.A. Lorentzos and Y. Manolopoulos, "Efficient Management of 2-d Interval Relations," *Proc. Fifth Int'l Conf., DEXA '94, in Database and Expert Systems and Applications*, D. Karagiannis, ed., pp. 72–82, Springer-Verlag, Sept. 1994.
- [31] C. Vassilakis, N.A. Lorentzos, P. Georgiadis, and Y.G. Mitsopoulos, "ORES: Design and Implementation of a Temporal DBMS," technical report, Dept. of Informatics, Univ. of Athens, 1994.
- [32] N.A. Lorentzos and Y.G. Mitsopoulos, "Specification of Valid Time SQL," ORES Deliverable D2, Sept. 1993.



Yannis G. Mitsopoulos received his first degree in agriculture from the Agricultural University of Athens in 1990. He then specialized in computer science. In this area, he participated in a number of research projects funded either by CEC or the Greek public sector. Within ORES, he was actively involved in the specification of a temporal extension to SQL. His research interests include temporal/interval data modeling and GIS.



Nikos A. Lorentzos received the BSc degree in mathematics from Athens University in 1975, the master's degree in computer science from Queens College, City University of New York, in 1981, and the PhD degree in computer science from Birkbeck College, London University, in 1988. Today he is an assistant professor at the Agricultural University of Athens, Informatics Laboratory. He has been successfully involved in a number of CEC-funded research projects. He was the technical manager of ORES, an ESPRIT

III project aiming at the development of tools for the management of temporal data. Work completed under his close supervision includes the specification, design, and implementation of a temporal relational algebra, SQL extension and temporal DBMS. He has been active on the program committees of various conferences, including the 1995 International Workshop on Temporal Databases. He serves as a reviewer for major database journals. His research interests include temporal/interval data modeling, object-oriented databases, GIS, and image processing.