

A Split Operator for Now-Relative Bitemporal Databases

Mikkel Agesen Michael H. Böhlen Lasse O. Poulsen Kristian Torp

Department of Computer Science, Aalborg University, Fredrik Bajers Vej 7E, DK-9220 Aalborg Øst

{bateman,boehlen,lop,torp}@cs.auc.dk

Abstract

The timestamps of now-relative bitemporal databases are modeled as growing, shrinking, or rectangular regions. The shape of these regions makes it a challenge to design bitemporal operators that a) are consistent with the point-based interpretation of a temporal database, b) preserve the identity of the argument timestamps, c) ensure locality, and d) perform efficiently. We identify the bitemporal split operator as the basic primitive to implement a wide range of advanced now-relative bitemporal operations. The bitemporal split operator splits each tuple of a bitemporal argument relation, such that equality and standard nontemporal algorithms can be used to implement the bitemporal counterparts with the aforementioned properties. Both a native database algorithm and an SQL implementation are provided. Our performance results show that the bitemporal split operator outperforms related approaches by orders of magnitude and scales well.

1 Introduction

Time is a pervasive aspect of most real-world phenomena and many database applications record various temporal aspects [15]. In a *bitemporal database* each tuple is associated with two time dimensions: *valid* and *transaction time*. The valid-time represents the period when the tuple is true in the modeled mini-world, and the transaction-time represents the period in which the tuple was logically current in the database [8]. Valid time is specified by the user and can be in the past or the future. Transaction time is managed by the DBMS and cannot extend beyond the current time. Valid time is useful for modeling the history (and future) of the data, while transaction time is useful for modeling the history of changes.

In a *now-relative* bitemporal database, either the start- or endpoint of the valid time can be *NOW*, which means that the tuple is valid from now onwards or is valid up to now. Likewise, the endpoint of the transaction time can be *NOW*, which means that the tuple is part of the current database state [5]. Figure 1 illustrates a *now-relative*

bitemporal (NR2T) relation modeling the association between employees and departments. For both tuples the 2D region spanned by valid and transaction time is illustrated graphically in the figure to the right. The relation is an in-

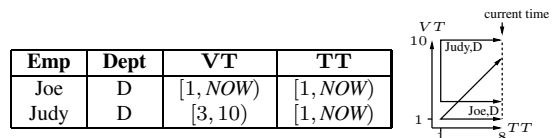


Figure 1: Joe and Judy working in Development (D).

stance of a NR2T schema with two explicit attributes, *Emp* and *Dept*, a valid-time interval *VT* and a transaction-time interval *TT*. The data records the following facts. At time 1 Joe starts working in development. At the same time it is planned, that Judy will be working in development from time 3 to 10. The *NOW* timestamp in valid time dimension means that the valid-time endpoint follows transaction time. The arrows that expand up to the current time in transaction time dimension indicate that the tuples remain current ($TT_E = NOW$) until they are deleted.

The contribution of this article is the specification, implementation, and empirical evaluation of the *split* operator. The split operator splits each bitemporal region into smaller bitemporal regions and can be used to define and implement a wide range of bitemporal operations. The split operator is the crucial step in answering NR2T queries because after splitting NR2T regions into smaller regions it is possible to compare them using equality (instead of overlap) and to use basic set operations to compute the result of bitemporal operations.

Salient properties of our split operator are that it a) is consistent with the point-based interpretation of a temporal database; this allows to view a temporal database as a set of nontemporal databases, b) preserves the identity of argument timestamps; this makes it possible to deal with any additional semantics the user has associated with the timestamps, c) ensures locality; this is important for the operator to scale up, and d) performs efficiently. All properties are essential to make the split operator generally applicable. We illustrate the properties in terms of the bitemporal difference

operation in Figure 2. Figure 2(a) shows the timestamps

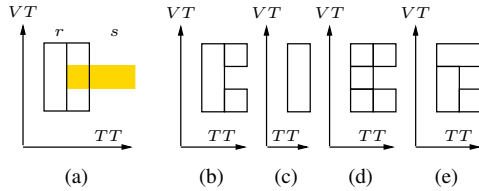


Figure 2: Possible Results of a Bitemporal Difference Operator

of the input tuples. Relation r contains two tuples, relation s one tuple. Only the result shown in Figure 2(b) is point-based, identity preserving, and respects locality. The result in Figure 2(c) is not point-based because the result region is not the result of subtracting the region of the s -tuple from the regions of the r -tuples. The result in Figure 2(d) does not respect locality because the region of an r -tuple has been split although the s -tuples does not overlap it. Figure 2(e) is not identity preserving because the timestamps of the two r -tuples have been merged and grouped differently.

The structure of this article is as follows. In Section 2, we discuss related research. Section 3 defines the NR2T data model and gives basic definitions. In Section 4, we introduce the split operator and use it to define now-relative bitemporal difference and aggregation. In Section 5, we formally specify the split operator. Sections 6 and 7 provide an SQL and a procedural implementation of the split operator, respectively. Section 8 presents performance results. Section 9 concludes the article and discusses future research.

2 Related Work

The efficient implementation (and semantics) of bitemporal relational operators has only received scant attention. All approaches use and acknowledge the importance of a split operator as a basic primitive to specify the semantics of and implement bitemporal relational operators [10, 11, 16, 17]. We first discuss all related approaches in turn and then point out the main differences of our approach presented in this paper.

Lorentzos et al. [11] proposes two relational operators, *fold* and *unfold*, for manipulating general multidimensional interval data. Fold merges adjacent points into intervals, and unfold expands intervals into points. Using fold and unfold it is possible to define point-based operators [3] for multidimensional interval data. The approach of unfolding intervals to points is good for specifying the point-based semantics. For an implementation it is prohibitively expensive and the authors use a global split operator [12] instead.

A formal specification of bitemporal aggregation is given in [16]. Both conventional and new temporal aggregates are presented. The specification is based on constant regions, i.e., regions that do not intersect the start or end

point of timestamps. The constant regions are found by globally partitioning the space spanned by the valid-time and transaction-time dimensions according to the start- and end-points in both time dimensions. The final aggregation is done over each region. Algorithms for efficiently implementing one-dimensional (valid-time) aggregation are proposed in [9]. This work is extended and parallelized in [6]. In both articles aggregates are computed over constant intervals [16].

Toman [18] discusses a normalization operation that makes it possible to compute point-based temporal queries. The normalization operation works on valid-time databases, and properties such a locality and identity preservation are not discussed.

Common to all related approaches is that they globally partition intervals into smaller intervals. In its most extreme form intervals are decomposed into their constituting points. This can be used to concisely specify the semantics but does not offer an efficient, or even feasible, implementation strategy. Our approach pursues a new direction in that we go beyond the purely point-based semantics and also guarantee *locality* and *identity preservation*. Locality is important for algorithms to scale up (cf. Section 8), and identity preservation makes it possible to exploit any additional semantics the user might associate with timestamps. This is the first paper that focuses on the split operator as the crucial primitive for temporal operators and comprehensively explores it. Our split operator is the first one that supports now-relative data [2, 5, 19], which is abundant in temporal databases.

3 Basic Definitions

A now-relative bitemporal schema \mathbf{R} consists of a set of explicit attributes and two intervals representing valid and transaction time: $\mathbf{R} = (A_1, A_2, \dots, A_n, VT, TT)$. We write \mathbf{A} for the explicit attributes A_1, A_2, \dots, A_n , and use VT_S (VT_E) to denote the start (end) point of the half-open interval $VT = [VT_S, VT_E)$. Two tuples t and u are *value equivalent* iff $t.\mathbf{A} = u.\mathbf{A}$. A time interval $[s, e)$ with $s, e \in \mathbb{Z} \cup \{NOW\}$ includes all time points between s and e : $[s, e) = \{x \in \mathbb{Z} \mid s \leq x < e\}$. *NOW* is a variable that evaluates to the wall-clock (current) time when the tuple is accessed [5]. A tuple is called now-relative if either a start or end point of a timestamp is equal to *NOW*. A tuple in a bitemporal relation represents a *shrinking*, *growing*, or *rectangular* region in bitemporal space as illustrated in Figure 3. Given a tuple t and a transaction time point $x \in t.TT$, the extract operator, $\epsilon_x(t)$, returns the valid time of tuple t at transaction time x . The sloped lines of shrinking and growing regions *must* lie on the diagonal $VT = TT$ because the *NOW* timestamp in valid time dimension means that valid time follows transaction time [5]. In contrast, a

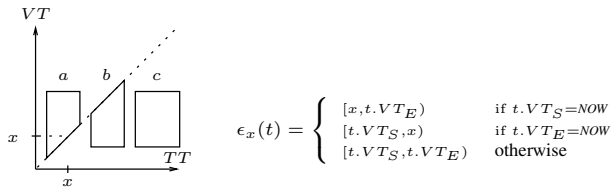


Figure 3: Basic Bitemporal Regions

rectangular region can be placed anywhere in the bitemporal space. Note that our definition is wider than previously published definitions for now-relative bitemporal schemas [5], as we allow *NOW* as the valid-time start value, meaning that data is valid from this moment onwards (shrinking regions). This extension is essential because the bitemporal difference operator returns a shrinking region when computing the difference of a rectangular and growing region.

According to the point-based view, the timestamp of a tuple in a NR2T relation can be viewed as set of points in the bitemporal space. The snapshot operator, \mathcal{S} , formalizes this. It is a variant of the unfold operator [11], and unfolds a now-relative bitemporal region into its constituting points.

$$\mathcal{S}(r) = \{ \langle \mathbf{a}, vt_p, tt_p \rangle \mid \exists t (t \in r \wedge \mathbf{a} = t.\mathbf{A} \wedge tt_p \in t.TT \wedge vt_p \in \epsilon_{tt_p}(t)) \}$$

An n -ary temporal operator \mathcal{O} is point-based iff $\mathcal{O}(r_1, \dots, r_n)$ and $\mathcal{O}(r'_1, \dots, r'_n)$ span the same snapshot points whenever r_i and r'_i span the same set of snapshot points, i.e.,

$$\begin{aligned} \mathcal{S}(r_1) = \mathcal{S}(r'_1) \wedge \dots \wedge \mathcal{S}(r_n) = \mathcal{S}(r'_n) &\Rightarrow \\ \mathcal{S}(\mathcal{O}(r_1, \dots, r_n)) = \mathcal{S}(\mathcal{O}(r'_1, \dots, r'_n)) & \\ \Leftrightarrow \mathcal{O} \text{ is point based.} & \end{aligned}$$

Throughout we use a few auxiliary predicates and functions: $ovlp(u, v)$ is true iff the timestamp of tuples u and v share at least one time point, $iovlp(i, j)$ is true iff intervals i and j share at least one point, and t^+ and t^- return the successor and predecessor of time point t , respectively.

4 The Split Operator and its Applications

To illustrate NR2T operations, consider the NR2T relation in Figure 1. Assume the following queries at time 8: 1) When was Joe thought to be working alone? and 2) How many people were thought to be working in development at any time? Query 1 is answered by a bitemporal difference, i.e., the times Joe was working except the times Judy was working. The result is shown in Figure 4(a). Query 2 is answered by a bitemporal aggregation query, i.e., count the number of tuples overlapping each time instant in the bitemporal space. The result is shown in Figure 4(b). Both queries are easily answered using the split operator followed by a standard relational difference and aggregation, respectively.

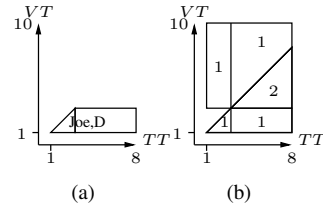


Figure 4: Bitemporal (a) Difference and (b) Aggregation

The *split* operator is a binary relational operator, $r \boxplus_{\mathbf{G}} s$, where r and s are NR2T relations and \mathbf{G} is a subset of the common explicit attributes of r and s . If a tuple $t \in r$ is overlapped by a tuple $u \in s$ and $t.\mathbf{G} = u.\mathbf{G}$, then t is disjointly partitioned into smaller regions that are either completely contained in u or do not overlap u at all. If $r = s$ (called a *self split*), then any two overlapping tuples in $r \boxplus_{\mathbf{G}} s$ where $t.\mathbf{G} = u.\mathbf{G}$ will completely overlap each other, i.e., they have the exact same timestamps. \mathbf{G} denotes the set of grouping attributes and reduces the number of tuples to split with. For aggregation, \mathbf{G} is set to the explicit grouping attributes, so that splitting is only performed within each explicit group. Figure 5 illustrates the split operator applied to different subsets of the NR2T relation from Figure 1. In the first example, the Joe tuple is split according to the Judy tuple. The result is shown in tabular form on the left. The figure to right graphically illustrates the bitemporal regions of the tuples. In the second example the Judy tuple is split according to the Joe tuple. The third example shows a self split of the relation in Figure 1.

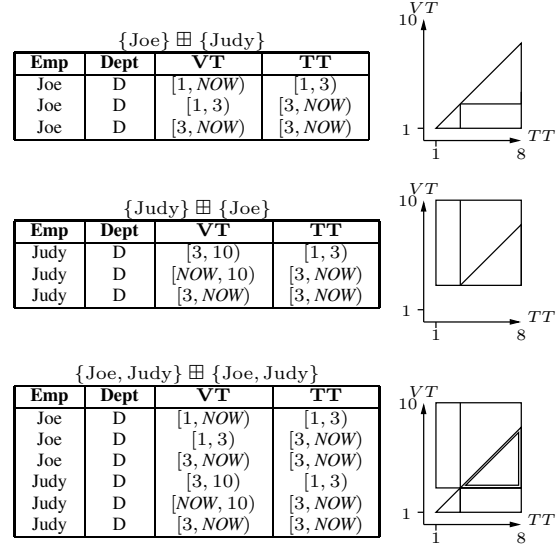


Figure 5: Illustration of the Split Operator

Using the split operator we can define bitemporal operations in terms of standard relational algebra operators. Below, we exemplify this for bitemporal difference and aggregation (cf. Figure 4). We prove that the definitions are consistent with the point-based interpretation of a NR2T

database. Identity preservation and locality follow directly from the specification of the split operator in Section 5: the identity is preserved because each tuple in r is split individually, and locality is ensured because the split lines are extended conservatively and cropped so that they never extend beyond the boundaries of the bitemporal region of the r -tuple (cf. Section 5.3).

The NR2T difference operator removes tuples from $r \boxplus_{\mathbf{A}} s$ that overlap tuples of s with the same explicit attributes

Definition 1 *The now-relative bitemporal difference operator, \setminus^{\square} , is defined as:*

$$r \setminus^{\square} s := (r \boxplus_{\mathbf{A}} s) - \Pi_{r.*} (s \bowtie_{r.\mathbf{A}=s.\mathbf{A} \wedge \text{ovlp}(r,s)} (r \boxplus_{\mathbf{A}} s))$$

Reading from the right of the expression, we start by splitting the tuples of the r relation with the tuple of the s relation that have the same explicit attribute values. We join that with the s relation so that tuples overlap and have the same explicit attribute values. Next, we project the result of the join so that we only keep the attributes of r . This part of the expression identifies what should be removed from r . To actually remove it, we split r with s again, and remove the tuples using an ordinary relational set difference.

Theorem 1 *The NR2T set difference operator, \setminus^{\square} , is consistent with the point-based interpretation of a bitemporal database.*

Proof: (Sketch) A tuple $t \in r$ may be overlapped by several value equivalent tuples in s . The split $r \boxplus_{\mathbf{A}} s$ partitions t into regions that are either completely contained in a value equivalent tuple $u \in s$, or do not overlap a value equivalent tuples of s . Thus, when we remove exactly those regions of the partitioned t that have an overlapping tuple in s we span the set of snapshot points $\mathcal{S}(\{t\}) \setminus \mathcal{S}(s)$. Since this holds for all $t \in r$ we get: $\mathcal{S}(r \setminus^{\square} s) = \mathcal{S}(r) \setminus \mathcal{S}(s)$. Thus, for any r' and s' with $\mathcal{S}(r) = \mathcal{S}(r')$ and $\mathcal{S}(s) = \mathcal{S}(s')$ we get $\mathcal{S}(r' \setminus^{\square} s') = \mathcal{S}(r') \setminus \mathcal{S}(s') = \mathcal{S}(r) \setminus \mathcal{S}(s) = \mathcal{S}(r \setminus^{\square} s)$. \square

NR2T aggregation aggregates with respect to a number of aggregate functions and associated attribute names [14]: ${}_{G_1, G_2, \dots, G_l} \mathcal{G}_{\mathcal{F}_1 B_1, \mathcal{F}_2 B_2, \dots, \mathcal{F}_m B_m}^{\square}(r)$ or, using vector notation, ${}_{\mathbf{G}} \mathcal{G}_{\mathcal{F}\mathbf{B}}^{\square}(r)$, where r is a NR2T relation and $B_i, G_i \in r.\mathbf{A}$.

Definition 2 *The now-relative bitemporal aggregation operator, ${}_{\mathbf{G}} \mathcal{G}_{\mathcal{F}\mathbf{B}}^{\square}(r)$, is defined as:*

$${}_{\mathbf{G}} \mathcal{G}_{\mathcal{F}\mathbf{B}}^{\square}(r) := {}_{\mathbf{G}, VT, TT} \mathcal{G}_{\mathcal{F}\mathbf{B}, VT, TT}(r \boxplus_{\mathbf{G}} r)$$

Ordinary relational aggregation groups a relation by the \mathbf{G} attributes and applies the aggregate functions \mathcal{F} to the \mathbf{B} attributes within each group. NR2T aggregation works similarly. First, it splits the relation with itself within the

groups defined by \mathbf{G} . Next, the result of the split is grouped by the \mathbf{G} attributes and the interval timestamps. The aggregate functions \mathcal{F} are then applied on the \mathbf{B} attributes. The result of the operation is the aggregate values together with the \mathbf{G} attributes and the interval timestamps of the regions in the group.

Theorem 2 *The NR2T aggregation operator, ${}_{\mathbf{G}} \mathcal{G}_{\mathcal{F}\mathbf{B}}^{\square}(r)$, is consistent with the point-based interpretation of a bitemporal database.*

Proof: (Sketch) In a self-split, $r \boxplus_{\mathbf{G}} r$, two tuples $t, u \in r$ with $t.\mathbf{G} = u.\mathbf{G}$ either have the same timestamps or do not overlap at all. Thus, for any $t, u \in r$ with $t.\mathbf{G} = u.\mathbf{G}$, the tuples t and u are partitioned in the same way in the area where they overlap (cf. third case in Figure 5). Therefore, grouping $r \boxplus_{\mathbf{G}} r$ by the VT - and TT -interval timestamps corresponds to a grouping by overlap. The aggregated attribute \mathbf{a} of any tuple $\langle \mathbf{a}, vt_p, tt_p \rangle \in \mathcal{S}({}_{\mathbf{G}} \mathcal{G}_{\mathcal{F}\mathbf{B}}^{\square}(r))$ therefore only depends on the set of explicit attributes of the tuples in r that overlap this (vt_p, tt_p) -point. For any r' with $\mathcal{S}(r') = \mathcal{S}(r)$ this set of explicit attributes will be the same and therefore $\mathcal{S}({}_{\mathbf{G}} \mathcal{G}_{\mathcal{F}\mathbf{B}}^{\square}(r)) = \mathcal{S}({}_{\mathbf{G}} \mathcal{G}_{\mathcal{F}\mathbf{B}}^{\square}(r'))$. \square

5 Specification of the Split Operator

In this section we develop a declarative specification of the split operator. Throughout, we use safe range first order calculus queries [1] to facilitate the translation to SQL. To illustrate the specifications we use relations r and s . Relation r consists of one tuple, $r = \{t\}$, and relation s consists of three tuples, $s = \{u, v, w\}$. The timestamps of the tuples are depicted in Figure 6(a). We assume value equivalent

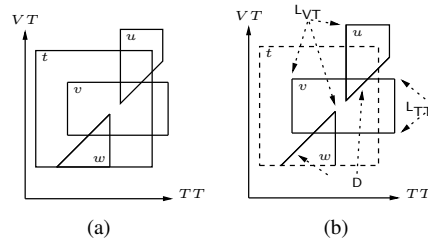


Figure 6: (a) Input Relations and (b) Line Types (L_{VT} , L_{TT} , D)

tuples and do not consider the explicit attributes explicitly. Intuitively, $r \boxplus s$ splits each tuple $t \in r$ with those tuples in s that overlap t . To split a tuple we perform the four steps illustrated in Figure 7. We (a) identify relevant horizontal, vertical, and diagonal lines of tuples in s that overlap t , (b) create additional vertical valid-time lines where diagonals intersect horizontal transaction-time lines, (c) adjust valid-time lines by extending them until they intersect a transaction-time line or a diagonal, or reducing them to fit

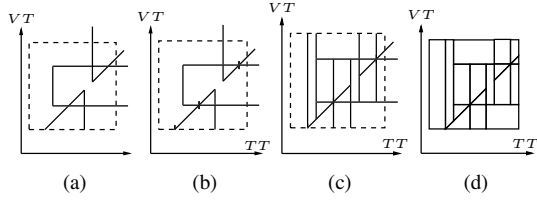


Figure 7: Overview of Split Process

inside t , and (d) construct NR2T result regions by combining valid-time lines, transaction-time lines, and diagonals. Each of these steps is and explored in detail in the following subsections.

5.1 Relevant Lines

We use the relations in Figure 6 for illustration purposes. For every tuple t in the relation r we consider each tuple $u \in s$, and identify the lines of u that overlap the inside of t . $L_{TT}(t, s)$ denotes transaction-time lines, $L_{VT}(t, s)$ valid-time lines, and $D(t, s)$ diagonals.

A line $l \in L_{VT}$ is represented by a tuple $\langle [S, E], O \rangle$ consisting of a transaction-time interval $[S, E]$ and the valid-time coordinate O . We write $l.S$, $l.E$, and $l.O$ to access the respective components. A tuple in $L_{VT}(t, s)$ consists of the left or right valid-time line of a bitemporal region of a tuple $u \in s$ with a transaction-time start or end that lies within t 's transaction-time interval and a valid-time interval that overlaps the valid-time interval of t :

$$L_{VT}(t, s) = \{ \langle \epsilon_{u.TT_S}(u), u.TT_S \rangle \mid u \in s \wedge t.TT_S < u.TT_S < t.TT_E \wedge \text{iovlp}(\epsilon_{u.TT_S}(t), \epsilon_{u.TT_S}(u)) \} \\ \cup \{ \langle \epsilon_{u.TT_E}(u), u.TT_E \rangle \mid u \in s \wedge t.TT_S < u.TT_E < t.TT_E \wedge \text{iovlp}(\epsilon_{u.TT_E}(t), \epsilon_{u.TT_E}(u)) \}$$

L_{TT} and D are defined in a similar fashion. The set $AllL_{TT}(t, s)$ is defined as the union of $L_{TT}(t, s)$ and the transaction-time lines of t itself. Since all diagonals lie on the main diagonal ($VT = TT$) they can be represented by a tuple $\langle [s, e] \rangle$ consisting of the transaction-time interval that the diagonal spans.

5.2 Additional Valid-Time Lines

Because wide angles ($> 90^\circ$) are not allowed in NR2T regions, they are divided into a sharp angle ($< 90^\circ$) and a right angle. This is done by introducing additional valid-time lines where transaction-time lines intersect diagonals within t 's region (cf. Figure 8). When a line from $AllL_{TT}$ intersects a diagonal from D inside the region of t ($in.t =$

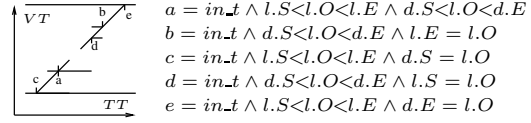


Figure 8: Origins of Additional Valid-Time Lines

$t.TT_S < l.O < t.TT_E$), a new valid-time line $\langle [s, e], o \rangle$ of minimal length, located in the intersection point of the diagonal and the transaction-time line, is created. The additional valid-time lines are denoted $NewL_{VT}$. Figure 8 shows the different possibilities for a transaction-time line l and a diagonal d to intersect, together with the detailed intersection conditions.

$$NewL_{VT}(t, s) = \{ \langle [l.O^-, l.O^+], l.O \rangle \mid \exists d(l \in AllL_{TT}(t, s) \wedge d \in D(t, s) \wedge a) \} \\ \cup \{ \langle [l.O, l.O^+], l.O \rangle \mid \exists d(l \in AllL_{TT}(t, s) \wedge d \in D(t, s) \wedge b) \} \\ \cup \{ \langle [l.O, l.O^+], l.O \rangle \mid \exists d(l \in AllL_{TT}(t, s) \wedge d \in D(t, s) \wedge c) \} \\ \cup \{ \langle [l.O^-, l.O], l.O \rangle \mid \exists d(l \in AllL_{TT}(t, s) \wedge d \in D(t, s) \wedge d) \} \\ \cup \{ \langle [l.O^-, l.O], l.O \rangle \mid \exists d(l \in AllL_{TT}(t, s) \wedge d \in D(t, s) \wedge e) \}$$

5.3 Adjusting Valid-Time Lines

To be able to split a NR2T region into a set of growing, shrinking, and rectangular bitemporal regions we either have to extend transaction or valid-time lines. We choose to extend (and shorten) valid-time lines because this preserves the valid-time intervals, which are user specified [3]. Each valid-time line in L_{VT} and $NewL_{VT}$ is adjusted using $adjS_{VT}$ and $adjE_{VT}$. The adjusted lines together with the left and right valid-time lines of t itself are denoted by $AllL_{VT}$. Figure 9 gives an example.

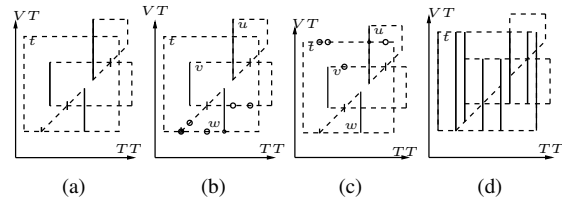


Figure 9: Adjusting Valid-Time Lines

$$AllL_{VT}(t, s) = \{ \langle [adjS_{VT}(t, l.S, l.O), adjE_{VT}(t, l.E, l.O)], l.O \rangle \mid l \in L_{VT}(t, s) \cup NewL_{VT}(t, s) \} \\ \cup \{ \langle \epsilon_{t.TT_S}(t), t.TT_S \rangle \} \cup \{ \langle \epsilon_{t.TT_E}(t), t.TT_E \rangle \}$$

Given a valid-time start point and a transaction-time coordinate, adjS_{VT} (cf. Figure 9(b)) identifies the point to which the valid-time start point should be adjusted. This is the largest point of the set consisting of: a) The valid-time coordinates ($l.O$) of all transaction-time lines that the valid-time line would intersect if it was extended downwards, b) the valid-time start point of t if it is not *NOW* (to make the line fit inside the boundaries of t), c) the valid-time coordinate of a diagonal that the valid-time line would intersect if it was extended downwards, and d) the valid-time start point if there exist two transaction-time lines that respectively begin and end where the valid-time line begins. adjE_{VT} (cf. Figure 9(c)) is defined in a very similar way.

$$\begin{aligned} \text{adjS}_{VT}(t, s, o) = & \\ \max \{ & \{ l.O \mid l \in \text{L}_{TT}(t, s) \wedge l.S < o < l.E \wedge l.O \leq s \} \quad \text{a)} \\ & \cup \{ t.VT_S \mid t.VT_S \neq \text{NOW} \} \quad \text{b)} \\ & \cup \{ o \mid t.VT_S = \text{NOW} \vee \exists d(d \in \text{D}(t, s) \wedge \\ & \quad \wedge d.S < o < d.E \wedge o \leq s) \} \quad \text{c)} \\ & \cup \{ s \mid \exists l_1, l_2(l_1, l_2 \in \text{L}_{TT}(t, s) \wedge l_1.O = l_2.O = s \wedge \\ & \quad l_1.E = l_2.S = o) \} \quad \text{d)} \end{aligned}$$

5.4 Constructing the Regions

In the final step, rectangular, shrinking, and growing bitemporal regions are identified by combining lines and diagonals from $\text{AllL}_{TT}(t, s)$, $\text{AllL}_{VT}(t, s)$, and $\text{D}(t, s)$. The lines in AllL_{TT} and D have to be coalesced [4], i.e., overlapping or adjacent lines have to be merged. The details are illustrated in Figure 10.

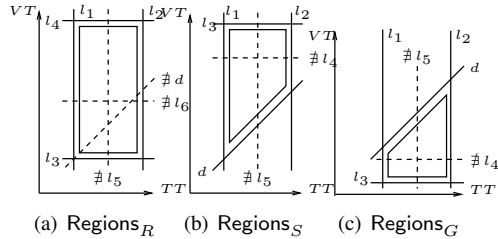


Figure 10: (a) Rectangular, (b) shrinking, and (c) growing regions

Regions_R constructs rectangular regions by combining the coordinates of intersecting pairs of valid-time lines and transaction-time lines that form a rectangular region that no other valid-time line, transaction-time line, or diagonal overlaps.

$$\begin{aligned} \text{Regions}_R(t, s) = & \\ \{ \langle t.A \mid & \mid [l_3.O, l_4.O], [l_1.O, l_2.O] \rangle \mid l_1, l_2 \in \text{AllL}_{VT}(t, s), \\ & l_3, l_4 \in \text{AllL}_{TT}(t, s) \wedge l_1.O < l_2.O \wedge l_3.O < l_4.O \\ & \wedge \text{isct}(l_1, l_3) \wedge \text{isct}(l_1, l_4) \wedge \text{isct}(l_2, l_3) \wedge \text{isct}(l_2, l_4) \\ & \wedge \nexists l_5(l_5 \in \text{AllL}_{VT}(t, s) \wedge l_3.O < l_5.E \\ & \quad \wedge l_5.S < l_4.O \wedge l_1.O < l_5.O < l_2.O) \end{aligned}$$

$$\begin{aligned} \wedge \nexists l_6(l_6 \in \text{AllL}_{TT}(t, s) \wedge l_1.O < l_6.E \wedge l_6.S < l_2.O \\ & \wedge l_3.O < l_6.O < l_4.O) \\ \wedge \nexists d(d \in \text{D}(t, s) \wedge l_1.O < d.E \wedge d.S < l_2.O \\ & \wedge l_3.O \leq l_1.O < l_4.O) \} \end{aligned}$$

The predicate $\text{isct}(m, n)$ is true iff the valid time line m and the transaction time line n share a time point. Regions_S and Regions_G are defined equivalently. Regions_R , Regions_S , and Regions_G are combined into Regions : $\text{Regions}(t, s) = \text{Regions}_R(t, s) \cup \text{Regions}_S(t, s) \cup \text{Regions}_G(t, s)$, and the result of the split operator is defined in terms of Regions : $r \boxplus s := \bigcup_{t \in r} \text{Regions}(t, s)$.

6 SQL Implementation

Since the specification in the previous section is based on safe range first order calculus queries, a translation to SQL is straightforward [1]. However, our initial performance results revealed that a direct translation is exceedingly inefficient. This section illustrates how to derive a feasible implementation from the specification. This is done for each of the four steps in turn. We assume r and s with schema $(A, VT_S, VT_E, TT_S, TT_E)$.

Relevant Lines Figure 11 illustrates the identification of relevant lines that participate in splitting a rectangle.

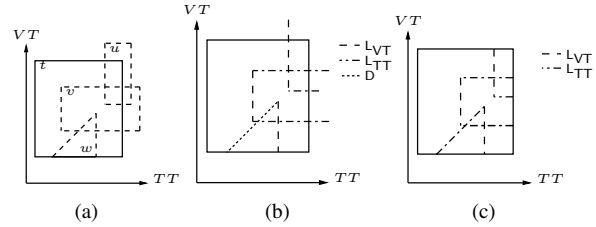


Figure 11: Identifying and Cropping Relevant Lines

To identify relevant lines, we use an R-tree to index the minimum bounding boxes of bitemporal regions [7]. Rectangular regions are indexed precisely whereas a conservative approach is taken to index growing regions. The indexing of growing regions can be further improved if we use the GR-tree [2], a generalization of the R-tree for now-relative data. We crop relevant lines to fit inside the region of t as illustrated in Figure 11(c). The cropping does not cost any extra disk I/O and eliminates subsequent computationally expensive steps. We give the SQL code to identify left side relevant valid-time lines:

```
INSERT INTO LVT (A,S,E,O)
SELECT t.OID,
       greatest(fix(t.VTs,u.TTs),fix(u.VTs,u.TTs)),
       least(fix(t.VTe,u.TTs),fix(u.VTe, u.TTs)), u.TTs
FROM r AS t, s AS u
WHERE t.TTs < u.TTs AND u.TTs < t.TTe
AND fix(t.VTs,u.TTs) < fix(u.VTe,u.TTs)
AND fix(u.VTs,u.TTs) < fix(t.VTe,u.TTs)
```

The `fix` macro is used to instantiate `NOW` to the current time: `fix(a, b)` returns `a` if `a` is different from `NOW` and `b` otherwise. The `greatest` and `least` macros are borrowed from Oracle. They return the greatest and least of their arguments, respectively. Above, they are used to crop the lines according to the boundaries of `t`. Right relevant valid time lines, transaction-time lines, and diagonals are found in a similar way. Diagonals are treated as transaction-time lines that have `NOW` as their valid-time coordinate. This significantly improves performance because transaction time lines and diagonals can often be treated uniformly, e.g., when adjusting valid time lines.

Additional Valid-time Lines We create additional valid-time lines where transaction-time lines and diagonals intersect within the boundaries of `t` as illustrated in Figure 8. To determine the intersection points L_{TT} is joined with itself. The `where` clause ensures that the two lines come from the splitting of the same region, and that one is a diagonal and the other is not.

```

INSERT INTO LVT (A,S,E,O)
SELECT l.ID,
CASE WHEN a OR d OR e THEN pred(1.O) ELSE 1.O END,
CASE WHEN a OR b OR c THEN succ(1.O) ELSE 1.O END,
1.O
FROM LTT AS d, LTT AS l
WHERE l.ID=d.ID AND d.O=NOW AND 1.O<>NOW
AND (a OR b OR c OR d OR e)

```

We use the predicates from Figure 8. Note though that because the lines have been cropped to the bitemporal region of `t`, the first condition (`in.t`) can be omitted, which also eliminates a join.

Adjusting Valid-time Lines In the third step valid-time lines have to be extended until they intersect a diagonal or transaction-time line or until they extend beyond the boundaries of `t`. Since transaction-time lines and diagonals are stored together and since we have cropped the lines to fit inside `t`, we only need to join the valid-time lines with the transaction-time lines to find all candidate start- and end-points.

```

INSERT INTO AllLVT (A,S,E,O)
SELECT lvt.ID, max(least(fix(1.O,lvt.O),lvt.S)),
min(greatest(fix(1.O,lvt.O),lvt.E)), lvt.O
FROM LinesVT lvt, LinesTT l
WHERE lvt.A=1.A AND 1.S<lvt.O AND lvt.O<1.E
AND (1.O>=lvt.E OR 1.O<=lvt.S)
GROUP BY lvt.A, lvt.S, lvt.E, lvt.O

```

L_{VT} is joined with L_{TT} to find every transaction-time line or diagonal (`l`) that the valid-time line (`lvt`) would intersect if it was extended. The join is assisted by composite indices on the `(A, O)` attributes of both tables. The adjusted start-point is the closest instantiated valid-time coordinate of `l` that lies below the original start point. The instantiation is done to deal with diagonals. Likewise the adjusted endpoint

is the closest valid-time coordinate of `l` that lies above the original endpoint.

Constructing Regions The final step is to combine transaction-time lines, adjusted valid-time lines, and diagonals to regions that form a partitioning of `t`. This is done in three steps as illustrated in Figure 12: determine left and right corners, determine left and right sides, and determine bitemporal regions. To find left corners we join valid-time

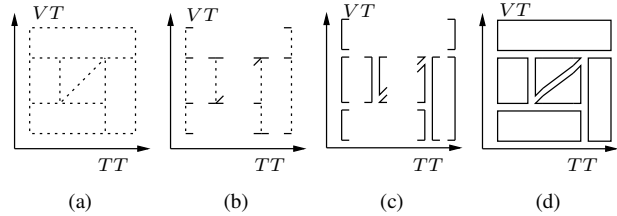


Figure 12: Constructing Regions

lines with transaction-time lines whose right endpoint is to the right of the intersecting valid-time line. For the intersection predicate we instantiate `NOW` in the valid-time coordinate of transaction time lines to handle diagonals. Right corners are found similarly. Adjacent corners are then combined to left and right sides by ensuring that no other corner is in-between (a 3-way join). Finally, left and right sides are matched to produce regions as shown in Figure 12(d). A left side and a right side match if they share the same transaction-time lines and no other right side is in-between (a 3-way join).

7 An Algorithm for the Split Operator

In this section the SQL implementation from Section 6 is subject to scrutiny and we show how the identification of relevant lines and the construction of result regions can be enhanced with algorithms that use sorting, iterations, and hash tables.

7.1 Identifying Relevant Lines

To identify all four sides of a region (the left and right side valid-time lines and the top and bottom transaction-time lines) `r` and `s` have to be joined multiple times. In SQL this cannot be improved because it is impossible to create multiple output tuples for a single input tuple. We can do better if we use an algorithm that joins `r` and `s` on overlapping bounding boxes of tuples (**FL1** below) and scans the result. For each `u ∈ s` that overlaps a tuple `t ∈ r` one or two valid-time lines and one or two transaction-time lines (or diagonals) are inserted into L_{VT} and L_{TT} , respectively (**FL2**).

FindLines

- FL1** For each pair $(t, u) t \in r, u \in s$ with overlapping bounding boxes
- FL2** If the left valid-time line of u overlaps t , add it to L_{VT}
 If the right valid-time line of u overlaps t , add it to L_{VT}
 If the lower transaction-time line of u overlaps t , add it to L_{TT}
 If the upper transaction-time line of u overlaps t , add it to L_{TT}

An R-tree index is used to find the tuples in s that overlap a tuple in r . This has to be done once for each tuple in r .

7.2 Constructing Regions

To construct NR2T regions we have to use 2-way joins to find corners, use 3-way joins to combine the corners to sides, and use yet another 3-way join to construct regions. Exploiting sorting and iteration we can eliminate all but a single 2-way join. Lines in L_{VT} (lvt) are joined with intersecting lines in L_{TT} (lft) and the result is sorted according to the ID of the region being split, the transaction-time coordinate of the valid-time line ($lvt.O$), the valid-time coordinate of the transaction-time ($lft.O$) line instantiated at $lvt.O$, and the uninstantiated $lft.O$. The last sorting attribute is needed to decide the ordering between transaction-time lines and diagonals in corners of growing and shrinking triangles.

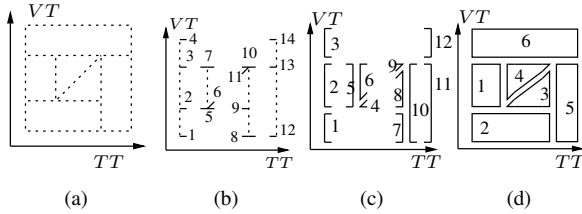


Figure 13: Constructing Regions (Procedural)

The sorting allows us to walk through the intersection points bottom up from left to right and find the corners. These corners and their ordering are illustrated in Figure 13(b). When we have found two adjacent left-side corners, we take the transaction time coordinate of the valid-time line shared by the two corners and insert it into a hash table with the id of the two transaction-time lines as key. When we have found two adjacent right-side corners we look up the corresponding left side in the hash table. Using the value from the hash-table as TT_S , the transaction-time coordinate of the valid-time line shared by the two corners as TT_E , and the valid-time coordinates of the two transaction-time lines as VT_S and VT_E we have all information to construct a region. Figure 13(c) shows the order in which the sides are found, and Figure 13(d) shows the new regions and the order in which they are found. The detailed algorithm is given below.

ConstructRegions

- CR1** Join L_{VT} (lvt) with intersecting L_{TT} (lft)

Sort the result by $lvt.OID, lvt.O, fix(lft.O, lvt.O), lft.O$

While tuples remaining:

Fetch tuple (lvt, lft)

- CR2** If $lvt.OID \neq last_id$: /* we are traversing a new valid-time line */
 $last_r = NULL; last_l = NULL; last_id = lvt.OID$
- CR3** If $lft.E > lvt.O$: /* we have reached a left corner */
 If $last_l \neq NULL$: /* we have two left-side corners - remember TT_S */
 $TT_S = lvt.O$
 $key = (last_l.OID, lft.OID)$
 $hashtable.add(key, TT_S)$
 $last_l = lft$
- CR4** If $lft.S < lvt.O$: /* we have reached a right corner */
 If $last_r \neq NULL$: /* we have two right-side corners - build a region */
 $A = lvt.ID$
 $VT_S = last_r.O; VT_E = lft.O$
 $key = (last_r.OID, lft.OID)$
 $TT_S = hashtable.get(key)$
 $hashtable.remove(key)$
 $TT_E = lvt.O$
 add ($A, VT_S, VT_E, TT_S, TT_E$) to result
 $last_r = lft$

8 Performance

This section reports the performance of two implementations of the local split operator that have been developed for PostgreSQL 7.0 [13]. The optimized SQL implementation described in Section 6 and the implementation of the algorithm from Section 7 written in C using the Server Processing Interface (SPI) of PostgreSQL. For comparison purposes we have also implemented a global split operator [12] both in SQL and as an algorithm using SPI. All experiments were run three times and averages computed. All experiments were made as self splits on a single relation, i.e., $r \bowtie r$. The experiments were run on an Intel Pentium III 450 Mhz PC with 128 MB RAM running RedHat Linux 6.1. All measurements on elapsed time are in seconds.

A NR2T data generator provides data for the experiments. The generator accepts the parameters n : the number of tuples to generate, w : the size of the world, i.e., the maximum value for VT_E and TT_E (minimum value is 1), l : the maximum length for the intervals in a tuple; the same maximum length is used for both transaction time and valid time (minimum value 1), and r : the percentage of growing regions (the rest will be rectangular). The data generator creates n tuples, and for each tuple VT_S and TT_S are chosen randomly between 1 and w . VT_E and TT_E are found by adding a length chosen randomly between 1 and l to the start value, thus the average length will be $l/2$. Each tuple has r percent probability of having VT_E replaced by NOW , thus becoming a growing region.

8.1 Test Results

The first test illustrates how the SQL and SPI implementations of local and global split scale up as a sparsely populated bitemporal database grows with time. The test data simulates a bitemporal database in which a new tuple of length 100 ($l = 100$) is inserted for every ten elements of time, i.e., the size of the world is ten times the number of tuples ($w = 10 \times n$). There are no now-relative tuples, i.e., all regions are rectangular, because global split does not work on now-relative data. Because the number of tuples and the size of the world grows together, the data density (i.e., the amount of overlap between tuples) remains almost constant. The number of tuples in the result of the local split is approximately 105% higher than the number of tuples in the input relation for all n . For the global split the number of tuples in the result is approximately 1000 times the number of tuples in the input relation for all n . Figure 14(a) shows

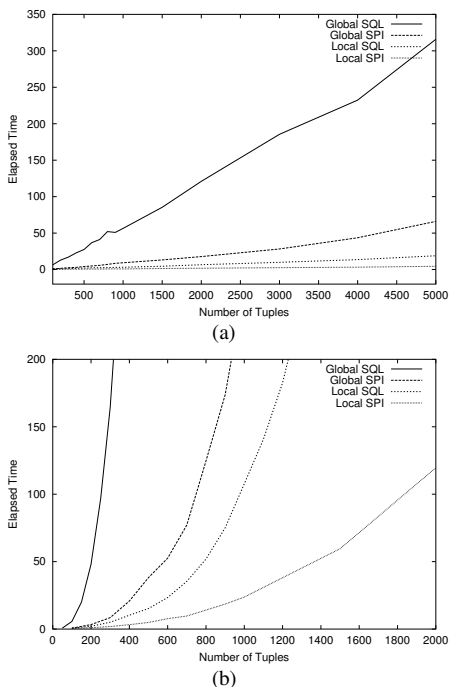


Figure 14: (a) Scalability and (b) Overlap Test for Rectangular Data

the elapsed time for all implementations for an increasing number of input tuples. The local split implementations scale up significantly better than the global split implementations, and both SPI implementations perform better than their corresponding SQL implementations.

The second test illustrates how SQL and SPI implementations of local and global split react to increasing overlap between tuples, simulated by a relation of fixed size ($w = 1000$) with an increasing number of tuples of fixed

size ($l = 100$). Again there are no now-relative tuples ($r = 0$) because global split does not support now-relative data. Figure 14(b) shows the elapsed time for all implementations for an increasing number of input tuples. Like in the first test, the local split implementations scale up significantly better than the global split implementations, and the SPI implementations perform better than their corresponding SQL implementations. The SQL implementations perform worse than in the first test, because of the increasing overlap. The more overlap there is, the more intermediate data there is for the 3-way joins (cf. Section 6).

Figure 15 illustrates how well the SQL and SPI implementations of local split scale up as a *now-relative* bitemporal database grows with time. The test data simu-

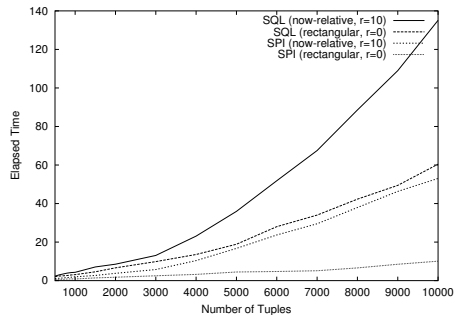


Figure 15: Scalability Test

lates a NR2T database in which a new tuple of length 100 ($l = 100$) is inserted for every ten elements of time, i.e., the size of the world is ten times the number of tuples ($w = 10 \times n$). Ten percent of the tuples are now-relative ($r = 10$). Because the number of tuples and the size of the world grow together, the data density (i.e., the amount of overlap between tuples) remains almost constant, thus the number of tuples in the result is approximately 130% of the number of tuples in the input relation for all n . Figure 15 shows the elapsed time for an increasing number of now relative input tuples ($r = 10$) as well as the elapsed time for both implementations for an increasing number of rectangular tuples ($r = 0$). As can be seen, the local SQL and SPI implementations scales up nicely for both rectangular and now-relative data. Now-relative data takes longer to process because now-relative tuples are populated around the diagonal. Therefore, the overlap between tuples increases. As we saw in the second test, increasing overlap increases the elapsed time.

Figure 16 illustrates that the non-linear increase in the execution time for increasing overlap between tuples is not due to an inefficient algorithms but rather a function of the size of the output. Figure 16(a) shows how SQL and SPI implementations of local split react to increasing overlap between tuples, simulated by a relation of fixed size

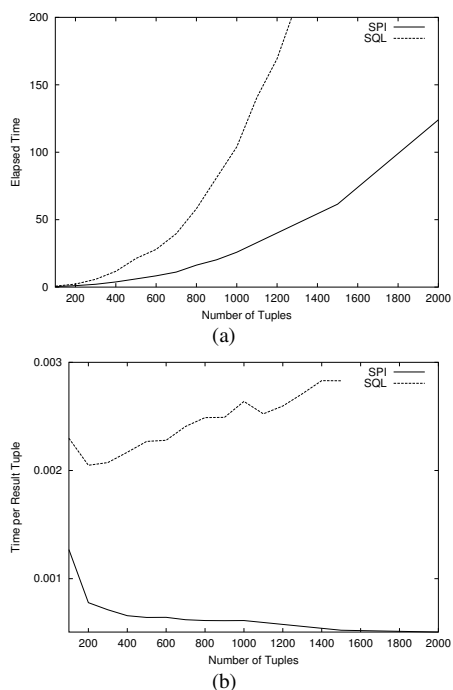


Figure 16: (a) Overlap Test, (b) Overlap Test (Normalized)

($w = 1000$) with an increasing number of tuples of fixed size ($l = 100$) of which 10% are now-relative ($r = 10$). Figure 16(b) shows the elapsed time normalized by the number of resulting tuples, i.e., the time it takes to produce a single output tuple. The time to produce a single tuple is almost constant for both implementations, slightly increasing for SQL and slightly decreasing for SPI, even though the elapsed time is increasing non-linearly.

9 Conclusion and Future Research

We have specified, implemented, and evaluated a now-relative bitemporal split operator. The split operator splits now-relative bitemporal regions into smaller regions such that standard relational algebra operators can be used to specify and implement the NR2T counterparts of a wide range of relational algebra operators. We demonstrated this for NR2T difference and aggregation. Distinct properties of our split operator are that it a) is consistent with the point-based interpretation of a temporal database, b) preserves the identity of the argument timestamps, c) ensures locality, and d) performs efficiently.

Our experiments show that a local split operator produces significantly fewer result tuples than a global split operator. This is a precondition for a split algorithm to scale up. The experiments show that the response time per result tuple is almost independent on the size of the input relation

and the overlap.

An interesting future research direction is the investigation of the semantics associated with the timestamps. The features of our split operator, in particular locality and identity preservation, make it possible to manage and exploit such semantics. It would also be interesting to parameterize the split operator to make it applicable to higher-dimensional temporal databases, one dimensional temporal databases, spatial databases, and general databases storing interval data. Finally, it is interesting to further investigate the applicability of the split operator to problems posed by the management of temporal data.

References

- [1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley Publishing Company, 1995.
- [2] R. Bliujute, C. S. Jensen, S. Saltinis, and G. Slivinskas. R-Tree Based Indexing of Now-Relative Bitemporal Data. In *Proceedings of VLDB*, pages 345–356, 1998.
- [3] M. H. Böhlen, R. Busatto, and C. S. Jensen. Point-Versus Interval-Based Temporal Data Models. In *Proceedings of ICDE*, pages 192–200, 1998.
- [4] M. H. Böhlen, R. T. Snodgrass, and M. D. Soo. Coalescing in Temporal Databases. In *Proceedings of VLDB*, pages 180–191, 1996.
- [5] J. Clifford, C. E. Dyreson, T. Isakowitz, C. S. Jensen, and R. T. Snodgrass. On the Semantics of “Now” in Databases. *ACM TODS*, 22(2):171–214, 1997.
- [6] J. A. G. Gendrano, B. C. Huang, J. M. Rodrigue, B. Moon, and R. T. Snodgrass. Parallel Algorithms for Computing Temporal Aggregates. In *Proceedings of ICDE*, pages 418–427, 1999.
- [7] A. Guttman. R-Trees: A Dynamic Index Structure For Spatial Searching. In *Proceedings of SIGMOD*, pages 47–57, 1984.
- [8] C. S. Jensen and et al. The Consensus Glossary of Temporal Database Concepts. In *Temporal Databases: Research and Practice*, volume 1399 of *Lecture Notes in Computer Science*, pages 367–405, 1998.
- [9] N. Kline and R. T. Snodgrass. Computing Temporal Aggregates. In *Proceedings of ICDE*, pages 222–231, 1995.
- [10] N. Lorentzos. The Interval-extended Relational Model and Its Application to Valid-time Databases. In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. T. Snodgrass, editors, *Temporal Databases: Theory, Design, and Implementation*, chapter 3, pages 67–91. Benjamin/Cummings Publishing Company, 1993.
- [11] N. A. Lorentzos and Y. G. Mitsopoulos. SQL Extension for Interval Data. *IEEE TKDE*, 9(3):480–499, 1997.
- [12] N. A. Lorentzos, A. Poulouvasilis, and C. Small. Manipulation Operations for an Interval-Extended Relational Model. *Data & Knowledge Engineering*, 17(1):1–29, 1995.
- [13] PostgreSQL Development Team. *PostgreSQL Manual*, 1996-2000. <http://www.postgresql.org/docs/postgres/index.html>.
- [14] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, 1997.
- [15] R. T. Snodgrass. *Developing Time-Oriented Database Applications in SQL*. Morgan Kaufmann, 1999.
- [16] R. T. Snodgrass, S. Gomez, and E. McKenzie. Aggregates in the Temporal Query Language TQuel. *IEEE TKDE*, 5(5):826–842, 1993.
- [17] D. Toman. Point-based vs Interval-based Temporal Query Languages. In *Proceedings of the 15th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 58–67, 1996.
- [18] D. Toman. Point-Based Temporal Extension of Temporal SQL. In *Proceedings of DOOD*, pages 103–121, 1997.
- [19] K. Torp, C. S. Jensen, and R. T. Snodgrass. Modification Semantics in Now-Relative Databases. Technical report, TimeCenter, 1999.