

The *POINT* Approach to Represent *now* in Bitemporal Databases

Bela Stantic¹, Abdul Sattar¹, Paolo Terenziani²

¹Institute for Integrated and Intelligent Systems

Griffith University, Queensland Australia

²Department of Computer Science, University of Piemonte Orientale, Alessandria, Italy

June 9, 2008

Abstract

Most modern database applications involve a significant amount of time dependent data and a significant portion of this data is *now-relative*. *Now-relative* data are a natural and meaningful part of every temporal database as well as being the focus of most queries. Previous studies indicate that the choice of the representation of *now* significantly influences the efficiency of accessing bitemporal data. In this paper we propose and experimentally evaluate a novel approach to represent *now* that we termed the *POINT* approach, in which now-relative facts are represented as *points* on the transaction-time and/or valid-time line. Furthermore, in the *POINT* approach we propose a logical query transformation that relies on the above representation and on the geometry features of spatial access methods. Such a logical query transformation enables off-the-shelf spatial indexes to be used. We empirically prove that the *POINT* approach is efficient on *now-relative* bitemporal data, outperforming the maximum timestamp approach that has been proven to be the best approach to now-relative data in the literature, independently of the indexing methodology (B^+ - tree vs R^* - tree) being used. Specifically, if spatial indexing is used, the *POINT* approach outperforms the maximum timestamp approach to the extent of factor more than 10, both in number of disk accesses and CPU usage.

1 Introduction

Standard databases usually only capture a snapshot or current state of the real world. A transaction then changes the database from one state to another by replacing the old values with new ones. However, there are many application domains where it is necessary to keep the old database states. In addition, many databases contain some amount of time dependent data and most database technology applications are temporal in nature, e.g., scheduling, financial and scientific applications.

In order to cope with temporal data in a general (not ad-hoc way), *temporal databases* have been introduced. A database is considered temporal if it is able to manage time varying data and it supports some time domain distinct from the user-defined time. In temporal databases time can be captured along two distinct and orthogonal time lines: transaction time and valid time [11, 23]. The valid-time line represents when a fact is valid in the modelled reality and the transaction-time line represents when a database transaction was performed. A bitemporal database is a combination of valid-time and transaction-time databases and records the database states with respect to both valid and transaction time. Many bitemporal database models, such as, e.g., TSQL2 [22] distinguish between STATES and EVENTS, stating that the former are durative and the latter instantaneous. It is important to note that our discussion applies to STATES only.

In temporal databases *now-relative* information play an important role. A temporal piece of data (tuple) is now relative if one of the following conditions (or both) holds:

- it is part of the current status of the database, i.e., it has been inserted in the past, and has not been deleted yet; In such a case, the ending point of its transaction time is usually set to *until changed* (*uc*);
- it is currently valid, i.e., the fact it describes holds at the current time; In such a case, the ending point of its valid time is set to *now*, to state that it is currently valid.

An efficient treatment of *now* in temporal databases is very important, since *now-relative* facts may be very frequent, and are likely to be accessed more frequently. As a matter of fact, it has been shown that the choice of the physical value for *now* significantly influences the efficiency of accessing temporal data [14].

There are two mainstreams in the treatment of now-relative data in temporal databases. In the first mainstream, variables, e.g., “*now*”, “until-changed”, “forever”, “ ∞ ”, “@”, and “-”; are introduced, leading to *Variable databases* [4]. However, *Variable databases* requires a significant departure from the “consensus” *relational* model (since they are not supported by the domain of SQL:1999 values [17]), which is not likely to occur in practice, due to the large commercial investments, both in terms of developed code and expertise [12]. Since we want to adhere to the relational model, we do not follow such a line of research in our approach.

In the second mainstream, the relational model has been extended. The literature has concentrated on three basic approaches: firstly using NULL, secondly using the smallest timestamp the minimum value approach *MIN* approach and thirdly using the largest timestamp supported by the particular RDBMS *MAX* approach, in order to denote the value *now*. It has been shown that the *MAX* approach outperforms the NULL and *MIN* approaches [14], so that it will be taken as a the reference approach in this paper.

We propose a novel approach called the *POINT* approach, to represent now-relative data, and we show that it outperforms the *MAX* approach both in case standard B^+ -tree indexing is adopted, and in case the more efficient spatial R^* -tree indexing is used to index bitemporal data.

We build on top of the preliminary results in [24, 25]. The core idea of the *POINT* approach, firstly presented in [25], is simple: now-relative data are represented as points along the time dimension in which *now* appears. However, such an idea has important practical implications in terms of physical disk I/O’s performance. In [25] such a new representation has been used in combination with B^+ -tree indices. The experimental evaluation in [25] showed that such approach is more efficient of the *MAX* approach by a factor of two, considering disk physical I/O’s.

Existing research shows that regular indices such as B^+ -trees are unsuited for temporal data [21], and a number of indices for temporal data has been proposed in the literature [21]. Due to the similarities between bitemporal and spatial data, as transaction time and valid time are considered to be orthogonal [23], the combined valid and transaction time of a fact can be treated as a region in two-dimensional space, therefore spatial indexes such as R^* - tree [1] can be adapted for indexing bitemporal data. Several methods based on R^* - tree has been proposed in the literature [15, 16]. However, those methods fall short in efficiently supporting data related to the current time, i.e., data for which the end of the valid time or transaction time is now. To enable efficient management of now-relative bitemporal data in [2] an extension to R^* - tree has been proposed. However, that extension requires modification to the kernel of the underlying RDBMS to support growing stair shapes, which represent now-relative data. It was our intention to stay within the capabilities of commercial RDBMS and therefore as an alternative for growing stair shape is the *MAX* approach.

Nevertheless, the adoption of spatial indexes in combination with the *MAX* approach has two major limitations:

- first of all, by denoting *now* with the maximum time, it ambiguously *overloads* the meaning of such a maximum time (which ambiguously denotes both the maximum time and the current time);
- it is not efficient [3, 24]; as a matter of fact, using *MAX* in combination with R^* – *tree* results in very large and overlapping bounding boxes making query retrieval extremely inefficient (large bounding boxes often mean lots of “dead space”).

In [24] the *POINT* approach has been extended, to take advantage of spatial indexing (specifically, R^* – *tree*). The preliminary experimental evaluation in [24], considering only the **Select** operation, has shown that, using R^* – *tree*, the *POINT* approach is more efficient of the *MAX* approach by a factor of more than ten, considering disk physical I/O’s. In this paper we build on top of [24], where attention was only given to the **Select** operation; here we extended the experiments in [24], and present an in-depth and more systematic evaluation of the experimental results, identifying the reasons of the better performance of the *POINT* approach. Furthermore, we add completely new experiments, to evaluate the *POINT* approach on other SQL statements such as **Update**, **Insert**, **Delete** as well as **Create**.

By representing now-relative data as points, the *POINT* approach avoids ambiguous overloading and, above all, optimize spatial indexing. Therefore, in conjunction with suitable query transformations, it

provides crucial advantages with respect to the other approaches in the literature (and, specifically, with respect to the *MAX* approach).

Specifically, the main original contributions of the work described in this paper are:

- the introduction of a new representation for *now*, overcoming the above limitations;
- the definition of suitable logical query transformations, based on the new representation; specifically, we take into account the bitemporal timeslice queries that have been identified in [14] as a benchmark to evaluate the efficiency of the *MAX*, *MIN*, and *NULL* approaches.
- an extensive experimental evaluation and analysis showing that the *POINT* approach outperforms the *MAX* approach, which is its best competitor), independently of the indexing method (B^+ -tree or R^* -tree) being adopted
- experimental evaluations of the *POINT* approach for *Update*, *Insert*, *Delete* as well as *Create* SQL statements.

Finally, it is worth stressing that our approach performs query transformations without requiring modification to the kernel. This has important side effects, including the fact that it enables the usage of different off-the-shelf indexing methodologies.

The paper is organized as follows. In Section 2, we look more closely at some basic temporal data concepts. In Section 3, we present the “core” of the *POINT* approach by presenting the extension to the query notation and the logical query transformations. Section 4 explains the experiments taken to evaluate the query performance of the *POINT* and *MAX* approaches in real database environment. In Section 5, we present the performance study concerning the data manipulation operations. In Section 6, we extend the comparison between the *POINT* and *MAX* approaches to consider also B^+ -tree indexing. Finally, in Section 7, we present our conclusions and discuss possible extensions and future work.

2 Temporal Data Concepts

While being ever changing, time is an important aspect of all real-world phenomena. Each fact bears a time attached to it, sometimes in more than one form. Time marks the starting and ending of a fact and establishes the valid of data. Data valid today may have had no meaning in the past and may hold no identity in the future. Some data, on the other hand may hold a historical significance or may continue to be valid up to a predefined point in time. This relationship between time and data adds a temporal identity to most data and in this light it would be hard to identify applications that do not require or would not benefit from some database support for time varying data.

Bitemporal tuples are associated with both valid time and transaction time. For both domains, as in most current approaches, we assume that the database has a limited precision; the smallest time unit (e.g., seconds, nanoseconds) is called *instant* (also called *chronon* in TSQL2 [13]). Valid time and transaction time are defined over the domain of time instants, which is ordered, discrete and isomorphic to the natural numbers. However, as in most approaches, we distinguish the possible values for transaction and valid time: while transaction time may only span from a chosen origin of the time domain to the current system time, since it encodes the time of the database transactions being performed future time is not allowed, valid time can also infinitely extend into the future. In the following, we will denote by T_t and T_v , the domains of transaction and valid time, respectively.

Following the TQel four timestamp format for temporal data [6], the valid-time interval VT can be represented as: $VT = [Vt^+, Vt^-)$ where the starting instant is Vt^+ while the ending instant is Vt^- . It is important to note that the interval is closed on left hand side and open on right hand side, which means that the starting instant is included, while the ending instant is not. Similarly, the transaction-time interval, TT can be represented as: $TT = [Tt^+, Tt^-)$ where starting instant is Tt^+ and ending instant is Tt^- and is not included.

For the moment, we introduce a symbol, “*now*”, in order to denote the current time. We follow the semantic of assigning the value *now* (to Vt^-) for valid time to indicate that the fact is valid up to and including the current time, and for transaction time (to Tt^-) to denote the fact that the tuple belongs to

<i>ID</i>	<i>Name</i>	<i>Position</i>	Vt^+	Vt^-	Tt^+	Tt^-
1	<i>Megan</i>	<i>DBA</i>	21.08.2006	10.11.2006	16.11.2006	20.01.2007
2	<i>Stephan</i>	<i>Teacher</i>	23.01.2007	now	01.07.2006	26.10.2006
3	<i>Mark</i>	<i>Admin</i>	10.01.2007	now	16.11.2006	now
4	<i>Steven</i>	<i>Officer</i>	21.02.2007	15.04.2007	13.12.2006	now

Table 1: Running sample of *now-relative* Bitemporal data

the current database state (in such a sense, our use of *now* for transaction time corresponds to the use of *UC* (Until Changed) in TSQL2 and many other approaches).

Definition 1 : Given a set of non-temporal attributes (A_1, \dots, A_n) defined over the domains (D_1, \dots, D_n) respectively, the domain D^{now} of bitemporal tuples having as non-temporal attributes (A_1, \dots, A_n) can be defined as follows:

$$D^{now} \cong \{ \langle A_1, \dots, A_n, Vt^+, Vt^-, Tt^+, Tt^- \rangle \mid \\ \in D_1 \times \dots \times D_n \times T_v \times (T_v \cup \{now\}) \times T_t \times (T_t \cup \{now\}) \mid \\ ((Vt^- = now \wedge Vt^+ \leq Vt^-) \vee Vt^+ < Vt^-) \wedge (Tt^- = now \vee (Tt^+ < Tt^- \leq now)) \}$$

Notice that the constraint $((Vt^- = now \wedge Vt^+ \leq Vt^-) \vee Vt^+ < Vt^-)$ grants that Vt^+ is strictly before Vt^- , except in the case in which $Vt^- = now$, in order to model the fact that only durative facts (STATES) are taken into account.

In Table 1 we show the bitemporal data used as a running sample in this paper. This bitemporal data represent the history of the employment of employees. For the sake of simplicity, in the rest of this paper we will replace absolute dates with valid-time values V_1 to V_6 and transaction-time dates with T_1 to T_5 . In bitemporal databases when a tuple is first inserted, its transaction time has the form $[Tt^+, now)$ indicating that the tuple is current and the ending time is unknown. Updates are allowed to be made on the most recent version of data only. No modification to the past is allowed, as the past cannot be changed. Deletion is logical, basically there are no physical deletions in bitemporal databases. When a fact is deleted, its transaction temporal attribute is changed from $[Tt^+, now)$ to $[Tt^+, Tt^-)$ where Tt^- is actually the time when the transaction is performed.

It is worth stressing that the representation for *now* we have adopted so far has just the purpose of describing the problem and the sample data. In the rest of the paper, we show how the above "abstract" representation of *now* can be efficiently implemented, showing also the advantages of our implementation with respect to alternative implementations.

3 The *POINT* Approach to Model *now*

In a discrete totally ordered model, a time interval, denoted as $[t^+, t^-)$, represents a set of a countably infinite equidistant time instants [5], where t^+ is the starting time instant and t^- is the ending time instant. These time instants are the smallest and largest values on the time line in the set of continuous time instants making up a given interval.

In such a model valid time *now-relative* data can be represented as $[Vt^+, now)$. Here Vt^+ represents the time point when the fact started to be true and *now* represents that the ending time continuously expanding and follows the current time.

Assuming that we are interested in using existing technology and given that the domain of SQL:1999 values does not contain a special value for *now*, the task is to select an appropriate value for *now* from an existing domain. This value should firstly satisfy the requirement that it cannot be used with some other meaning, otherwise the meaning becomes semantically ambiguous.

Previous work in the area has focused on three physical values to represent *now*: the *NULL* value, the smallest timestamp (*MIN*) and largest timestamp (*MAX*) supported by a particular RDBMS [14]. It is clear that whichever of these values is chosen, the domain of the data type becomes limited and a potential ambiguity is created. This is especially the case for the *NULL* value, as it is already overloaded in its normal usage. However, the *NULL* value does have the advantage that it takes up less space than a regular

ID	$Name$	$Position$	Vt^+	Vt^-	Tt^+	Tt^-
1	<i>Megan</i>	<i>DBA</i>	V_1	V_2	T_3	T_5
2	<i>Stephan</i>	<i>Teacher</i>	V_4	T_{max}	T_1	T_2
3	<i>Mark</i>	<i>Admin</i>	V_3	T_{max}	T_3	T_{max}
4	<i>Steven</i>	<i>Officer</i>	V_5	V_6	T_4	T_{max}

Table 2: Sample Bitemporal data with *MAX* representation of *now*

ID	$Name$	$Position$	Vt^+	Vt^-	Tt^+	Tt^-
1	<i>Megan</i>	<i>DBA</i>	V_1	V_2	T_3	T_5
2	<i>Stephan</i>	<i>Teacher</i>	V_4	V_4	T_1	T_2
3	<i>Mark</i>	<i>Admin</i>	V_3	V_3	T_3	T_3
4	<i>Steven</i>	<i>Officer</i>	V_5	V_6	T_4	T_4

Table 3: Sample Bitemporal data with *POINT* representation of *now*

timestamp value and can be processed faster [27]. Despite this, the crucial disadvantage of *NULL* is that columns that permit *NULL* values cannot be indexed [8], leading to potentially unacceptable access times. It is important to mention that using a *non NULL* value for *now* also can affect indexing. If, for example, a B-tree index on Vt^- or Tt^- is used to retrieve tuples with a time period that overlaps *now*, and *now* is represented with the *MIN* or *MAX* approach, tuples with the Vt^- or Tt^- attribute set to *now* will not be in the range retrieved, i.e., they will be at the extreme left or extreme right of the B-tree. We term this as the range indexing problem.

To simplify a graphical representation in our running example shown in Table 1, we will replace the absolute dates with values $V_1 - V_6$ and $T_1 - T_5$. Table 2 shows our running example of bitemporal data with *MAX* approach to represent the current time where T_{max} represents the maximum timestamp that the particular database supports.

Each of the previously discussed approaches to representing *now* has severe limitations in terms of indexing and/or semantic ambiguity. Of these approaches, the literature generally agrees that *MAX* is the best overall compromise [14], as it allows indexing and generally has better performance than *MIN*.

The core idea of our approach is simple: we propose a new representation for now-relative data, in which *now* is represented by making the end point of any current interval equal to the start point (i.e., $Vt^- = Vt^+$ and $Tt^- = Tt^+$) [25]. It is worth stressing that such "degenerate" intervals have no meaningful duration. As in several approaches we adopt the convention that such degenerate no-duration time intervals can be represented as a point on the time line [9] (we termed our approach *POINT* approach because of this consideration).

For instance, we represent the transaction time of current tuples with the notation $[n,n)$. We stress that it is only a notation, since, technically speaking $[n,n)$ is meaningless, since 'n' cannot be both included and excluded. In the notation $[n,n)$ the symbol "n" is overloaded. However (differently from the case of the *NULL*, *MIN* and *MAX* approaches mentioned above), it has not got an ambiguous meaning: whenever a notation $[n,n)$ occurs in our approach, "n" on the left always denotes the value n , while "n" on the right always denotes the value *now*.

To illustrate the *POINT* approach to represent *now*, consider the relation in Table 3. Here, in tuple $ID = 3$, we can conclude that "Mark" has the current position of "Admin", firstly because Vt^- equals Vt^+ and represent point on the time line so do not have meaningful duration (representing that the fact is currently valid) and secondly because Tt^- equals Tt^+ (representing that the tuple has not been logically deleted). Tuple $ID = 1$ represents that "Megan" had the position of "DBA" from V_1 to V_2 and that was our past belief (tuple do not belong to the current database state as Tt^- is not equal Tt^+ , it was current only between T_3 and T_5). Hence, whenever the timestamp for Vt^- is the same as Vt^+ , it unambiguously means that a fact is valid *now*. Similarly when Tt^- is the same as Tt^+ it unambiguously means that a tuple belong to the current databases state.

Considering the above approach, we do no longer need an explicit *now* symbol in our representation. In particular, Definition 1 above can be simplified as follows in our approach:

Definition 2 : Given a set of non temporal attributes (A_1, \dots, A_n) defined over the domains (D_1, \dots, D_n) respectively, the domain D^{POINT} of bitemporal tuples having as non temporal attributes (A_1, \dots, A_n) in our approach can be defined as follows:

$$D^{POINT} \cong \{ \langle A_1, \dots, A_n, Vt^+, Vt^-, Tt^+, Tt^- \rangle \mid Vt^+ \leq Vt^- \wedge Tt^+ \leq Tt^- \}$$

It is worth stressing that, in our approach, no special symbol need to be included in the temporal domains in order to explicitly cope with *now*.

3.1 Query Transformations

A number of queries are chosen to measure the query performance of the spatial index on *now-relative* bitemporal data. Specifically, in this paper we consider the benchmark queries that have been introduced by the previous literature to measure the performance of the *MAX*, *MIN* and *NULL* [14]. Three types of queries are considered: the first type asks for current tuples valid at *now*, the second regards current tuples, holding in the past, and the third non-current tuples, holding in the past. Figure 1 represents the temporal semantics of the queries [24].

Before discussing such queries, it is worth stressing that the main goal of the approaches to now-relative data in the literature is that of maximizing efficiency for queries of the first type, which have been proven to be the most important in the literature, since it is more likely that users looks for data in the current version of the database, as well as valid at the current (query) time [14].

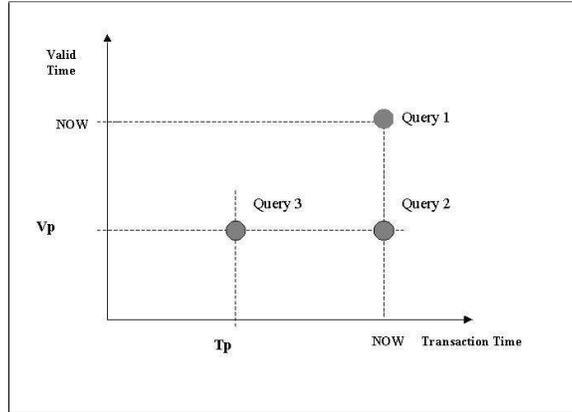


Figure 1: Time Slice Queries

Query 1 retrieves the current state in both transaction time and valid time. Semantically, it retrieves our current belief about tuples valid in the current state of the modelled reality. Let t_{now} denote the current instant (system clock time) when the query is performed. Query 1 selects tuples with valid-time intervals that overlap t_{now} and transaction-time intervals that meet t_{now} . It therefore selects tuples that have valid-time intervals starting at or before t_{now} and ending after t_{now} and transaction-time end equal to t_{now} .

$$(Vt^+ \leq t_{now} \wedge Vt^- > t_{now}) \wedge Tt^- = T_{max}$$

Since valid time and transaction time are orthogonal, they can be represented and addressed as a two dimensional space [23]. Bitemporal data from Table 2 represented in a two dimensional space are shown in Figure 2, values for $V_1 \dots V_6$ represent valid-time values from Table 2, similarly for transaction-time values $T_1 \dots T_5$. It is important to note that dots on axis represent that there is a big skip in order due to

introduction of T_{max} , which is significantly bigger than all other values. In order to satisfy the conditions $Vt^+ < t_{now}$ and $Vt^- \geq t_{now}$ and $Tt^- = T_{max}$, the geometries need to intersect the point in two dimensional space as show on Figure 2 (t_{now}, T_{max}). Therefore in Figure 2 we can see that only tuples ID=3 and ID=4 satisfy the criteria and are returned as result, since they intersect the point in the two dimensional space $[t_{now}, T_{max}]$

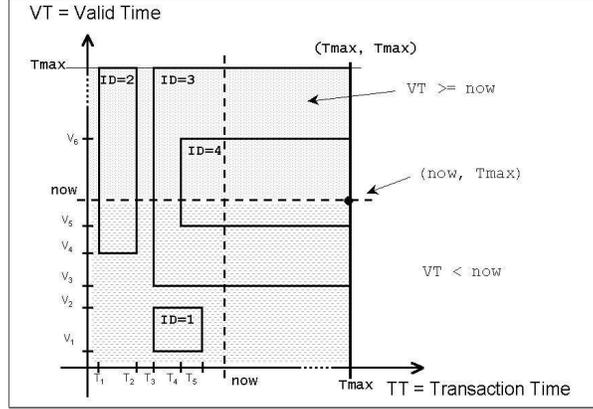


Figure 2: Query 1: *MAX* Approach

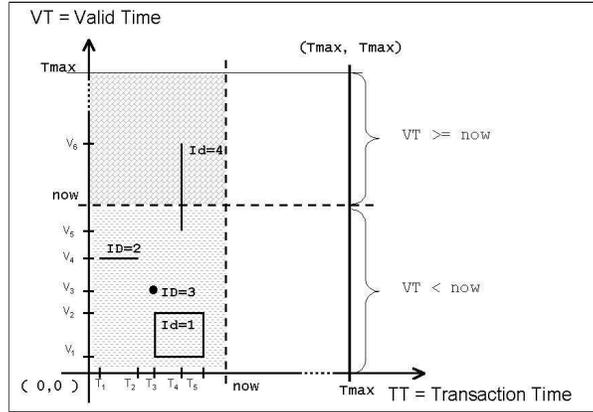


Figure 3: Query 1: *POINT* Approach

Using our *POINT* approach, Query 1 represents the domain of tuples that meets the following criteria:

$$(Vt^+ \leq t_{now} \wedge (Vt^- > t_{now} \vee Vt^- = Vt^+)) \wedge (Tt^- = T_{max})$$

The Spatial transformation of the *POINT* approach for Query 1, as shown in Figure 3, is a bit more complex. Since we look for all tuples where the transaction time is current (i.e., $Tt^- = T_{max}$), which means that the tuples of interest for this query can be represented as points or lines only, rectangles would not be part of the result.

Tuples where $[Vt^+ \leq t_{now} \text{ and } Vt^- = Vt^+]$ would be points (since $Tt^- = T_{max}$) and would lie below the line $[Vt = t_{now}]$, such as ID=3 shown in Figure 3 (notice that the tuples in which $Vt^- = Vt^+$ and $Vt^- > t_{now}$ are not of interest for Query 1). Also, geometries where $[Vt^+ \leq t_{now} \text{ and } Vt^- > t_{now}]$ would be line geometries, parallel to the valid-time axis VT , since $Tt^- = T_{max}$, which intersect the line $[VT = t_{now}]$ (ID=4). The *POINT* query thus returns all tuples representing point geometries (type 1) that intersect with the area under the line on $[VT = t_{now}]$ and line geometries (type 2) parallel to the VT -Axis intersecting with $[VT = t_{now}]$, therefore query will return tuples ID=3 and ID=4.

Figures 2 and 3 represent the semantics of Query 1 with tuples representing geometries $[ID=3, ID=4]$ being returned as part of the query result set, while tuples $[ID=1, ID=2]$ are not of interest for this query and are rejected.

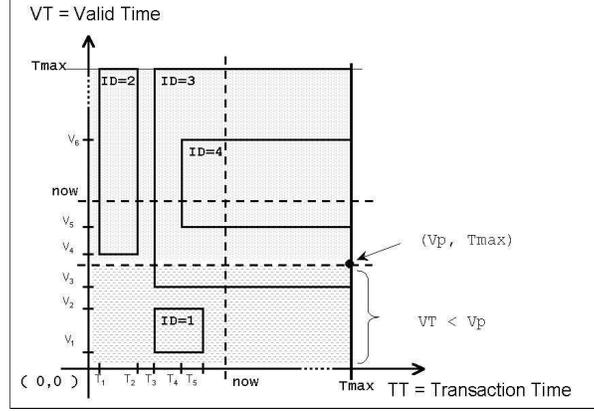


Figure 4: Query 2: Spatial *MAX* Approach

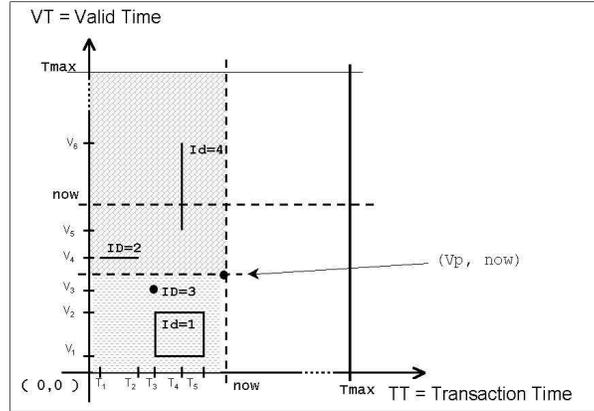


Figure 5: Query 2: Spatial *POINT* Approach

Query 2 timeslices the relation as of current time in the transaction-time domain and as of a past time in the valid-time domain, denoted by V_p in the following, and in Figures 4 and 5. Semantically, it retrieves our current belief about a past state of the world. For the *MAX* approach, the query condition is represented by:

$$(Vt^+ \leq V_p \wedge Vt^- > V_p) \wedge Tt^- = T_{max}$$

while for the *POINT* approach the query condition is:

$$(Vt^+ \leq V_p \wedge (Vt^- > V_p \vee Vt^- = Vt^+)) \wedge Tt^- = Tt^+$$

The Spatial representation for Query 2 is similar to Query 1 and the geometries satisfying the query in the *MAX* approach would have to intersect the point (V_p, T_{max}) . Similarly, for the *POINT* approach query would be satisfied by all *point* geometries that intersect with the area under the line $[VT=V_p]$ and all *Line* geometries parallel to the *VT*-Axis intersecting with, but not ending at, $[VT=V_p]$.

Figures 4 and 5 represent the semantics of Query 2 with the tuple representing geometry $[ID=3]$ being returned as part of the result, while tuples representing geometries $[ID=1, ID=2, ID=4]$ are rejected.

Query 3 timeslices the relation as of a past time V_p and T_p in both the transaction-time and valid-time domains. Semantically, it retrieves our past belief about a past state of the world. The *MAX* query condition would therefore be represented as:

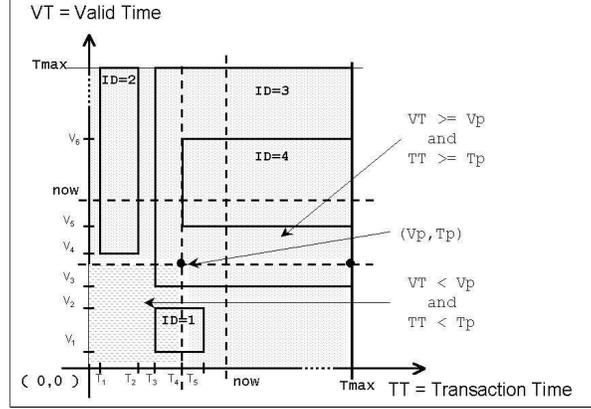


Figure 6: Query 3 : Spatial *MAX* Approach

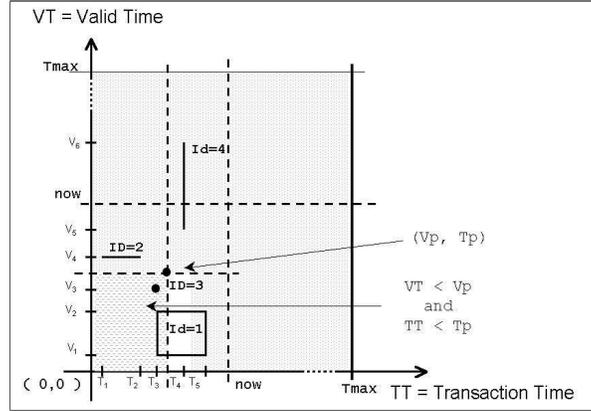


Figure 7: Query 3: Spatial *POINT* Approach

$$(Vt^+ \leq V_p \wedge Vt^- > V_p) \wedge (Tt^+ \leq T_p \wedge Tt^- > T_p)$$

while for the *POINT* representation:

$$(Vt^+ \leq V_p \wedge (Vt^- > V_p \vee Vt^- = Vt^+)) \wedge (Tt^+ \leq T_p \wedge (Tt^- > T_p \vee Tt^- = Tt^+))$$

Spatially geometries satisfying Query 3 in the *MAX* representation would have to pass through the point (V_p, T_p) .

Similarly, the *POINT* query would be satisfied by all *Point* geometries lying in the rectangle $(0, V_p, 0, T_p)$, all *Line* geometries parallel to the VT-Axis intersecting with, but not ending at $[VT=V_p]$, all *Line* geometries parallel to the TT-Axis intersecting with, but not ending at $[TT=T_p]$ and all *Rectangle* geometries intersecting with the point (V_p, T_p) . Figures 6 and 7 represent the semantics of Query 3 with the tuple representing geometry $[ID=3]$ being returned as part of the result, while tuples representing geometries $[ID=1, ID=2, ID=4]$ are rejected.

4 Experiment

In this section, we empirically compare the performances of our *POINT* approach with respect to other approaches in the literature. We have not considered the approaches modelling *now* using variables [4], since we aim at proposing an approach which does not require any modification of the kernel so that, e.g., off-the-shelf indexing techniques can be used. Considering other alternative approaches, we have chosen to

<i>Time dimension</i>	<i>Approach</i>	<i>Minimum</i>	<i>Maximum</i>
<i>Valid time</i>	<i>Point</i>	16 – 11 – 1997	11 – 12 – 2008
<i>Valid time</i>	<i>MAX</i>	16 – 11 – 1997	31 – 12 – 9999
<i>Transation time</i>	<i>Point</i>	16 – 11 – 1997	14 – 02 – 2007
<i>Transation time</i>	<i>MAX</i>	16 – 11 – 1997	31 – 12 – 2008

Table 4: Domains of the temporal data

compare our *POINT* approach just to the *MAX* approach, since previous studies have shown that the *MAX* approach outperforms *MIN* approach and NULL [14].

For each approach (*MAX* and *POINT*) we generated three relations differing in percentage of *now-relative* data. Then on each relation we performed three different representative time slice queries shown in Figure 1. We have chosen time slice queries because of their recognised importance in temporal databases [26]. For each approach we created three relations with one million randomly generated tuples, having 10%, 20% and 40% of the tuples overlapped with the current time in both transaction and valid-time domains.

4.1 Data sets

In absence of real data we randomly generated data sets with different data distributions to simulate a real-world scenario. We created number of tables for both method evaluated with different percentage of now-relative data 10%, 20% and 40%. The structure of the tables are identical with the sample data shown in Table 1, where beside the *ID*, *Name* and *Position* attributes there are four temporal attributes *vts*, *vte*, *tts*, and *tte* representing start and end time of valid and transaction time. Every table contain one million tuples. The size of the table is 132 MB.

When generating data we have taken into consideration that valid time can extend to future while transaction time cannot, because it is maintained by the system not by users. The temporal range of valid time and transaction time in our experimental data are shown in Table 4.

The starting time of the intervals were always uniformly distributed on the interval domain, while the duration and percentage of *now-relative* data was varied. The following data distributions have been considered:

- Uniformly distributed start and exponentially distributed length according to the exponential distribution function $y = e^{-0.00053*x}$ with 10% of uniformly distributed *now-relative* data.
- Uniformly distributed start and exponentially distributed length according to the exponential distribution function $y = e^{-0.00041*x}$ with 20% of uniformly distributed *now-relative* data.
- Uniformly distributed start and exponentially distributed length according to the exponential distribution function $y = e^{-0.00029*x}$ with 40% of uniformly distributed *now-relative* data.

We have chosen a uniform distribution of interval start and an exponential distribution of the duration because it reflects most real-world applications where short intervals are more likely to occur than long intervals [20]. We have chosen to test different percentage of *now-relative* data to investigate how it influences the results.

4.2 Measurables

For the qualitative assessment and quantitative evaluation, we used the following criteria proposed by Gaede and Gunther [7]:

- Dynamics: The ability to keep track of changes as tuples are inserted and deleted from the database in any order,
- Broad Range of Supported Operations : Ability to efficiently support *Insert*, *Update*, *Delete*, and *Query* processes,

- **Simplicity:** Ability to integrate easily and work efficiently in large scale applications,
- **Time Efficiency:** Ability to perform fast searches as well as inserts, updates and deletions,
- **Space Efficiency:** Especially as compared to the data that it indexes,
- **Concurrency and Recovery:** Ability to efficiently manage concurrent access and support recovery operations,
- **Minimum Impact:** Ability to integrate well into an existing system with minimum impact on existing parts of the system.

Our approach doesn't require any modification to the database kernel or additional implementation logic and it is within the existing capability of commercial RDBMS. Such facts ensure the Broad Range of Supported Operations (*Insert, Update, Delete*), Concurrency and Recovery, and to a large extent Dynamics, so that these criteria will be neglected in the discussion below.

Since we considered different distributions of *now-relative* data, in our analysis we have also taken into account the dependence of performance on different data sets, especially considering different amounts of *now-relative* data.

The Theory of Indexability [10] identifies *I/O* complexity cost, measured by the *number of disk accesses*, as one of the most important factors for measuring query performance. The *Oracle 9i Performance Tuning Guide* [8] also establishes the importance of disk accesses and thus we used physical disk I/O to assess the *Query* performance. Other measures of importance such as *CPU usage* are also used in conjunction with the number of disk accesses to assess the performance of the *Create, Insert, Update* and *Delete* statements.

4.3 Creating the Spatial Index

Function-based spatial indexing has been chosen in the implementation in order to eliminate the need for an additional column to store the spatial geometry. A sample index statement is shown below:

```
create bitmap index Same40_spatial_bmp_idx
on TABSame40( GET_SAME_GEO (vts, vte, tts, tte).get_gtype())
nologging parallel;
```

A dedicated function is created for each method considered. This function receives parameters ($vts = Vt^+$, $vte = Vt^+$, $tts = Tt^+$, $tte = Tt^+$) as inputs. Initially, each geometry is represented as a spatial rectangle having (Vt^+, Tt^+) and (Vt^-, Tt^-) as lower left and upper right vertices respectively. A resultant geometry is then returned by the dedicated function *Get_Same_Geo* that ensures that the geometry types - *Point, Line* or *Rectangle* are explicitly chosen depending on the relation between Vt^+ and Vt^- and between Tt^+ and Tt^- . If $Vt^+ = Vt^-$ **or** $Tt^+ = Tt^-$ the resulting geometry is explicitly defined as a line; If $Vt^+ = Vt^-$ **and** $Tt^+ = Tt^-$ the resulting geometry is explicitly defined as a point in space (instead of being defined as rectangles with zero length and/or width). This ensures that less data is stored in leaf nodes. Part of the *Get_Same_Geo* function related to the point geometry types is shown below:

```
IF (vts=vte and tts=tte) THEN RETURN
MDSYS.SDO_GEOMETRY ( 2001, -- 2 represents two dimensional space and 001 point geometry
NULL, MDSYS.SDO_POINT_TYPE ( vts, tts, null ), NULL, NULL);
```

The spatial component of a Spatial feature is the geometric representation of its shape, referred to as its geometry and stored in the Spatial data type *Sdo_Geometry*, which acts as a container for storing points (type 1), lines (type 2) and polygons (type 3; see Table 5). We used this feature in the implementation to enhance the computational advantage of the *POINT* approach. *This obtains a major significance in the POINT approach since now-relative valid-time and transaction-time tuples are represented and stored as points, while now-relative valid-time or transaction-time tuples are represented and stored as lines.*

Shape	Geometry Type
Point	Type 001
Line	Type 002
Rectangle	Type 003

Table 5: Spatial geometry types of interest

4.4 Environment

Our implementation has been carried out on a four 450MHZ CPU - SUN UltraSparc II processor machine with 4096 Megabytes memory, running Oracle 10g RDBMS, with a database block size of 8K using Oracle Spatial [18], hereon referred to as Spatial. To ensure that there is no effect of environment parameters on the results, the Oracle Instance has been tuned appropriately. Parameters such as SGA (System Global Area) size and tolerance are varied to study their effect on performance, however it was noted that the variation of size of SGA and tolerance does not significantly influence the results. Empirical results provided in this paper consider a SGA of 100MB and *Sdo_Tolerance* of 0.5, as recommended for spatial index with granularity of 1 day in [18]. The SGA was locked into memory to ensure that paging does not affect results.

Considering that the size of every particular table is 132 MB it is obvious it cannot fit totally in SGA. In fact, SGA size in our experiments is 100MB, and, besides the data table, SGA needs to store index structures and BLOB tables related to the spatial index. To ensure that the logical read of data already in SGA does not influence the results we flushed the the database buffer cache in SGA before every particular test. Considering that the spatial index is forced to be used by "Optimizer Hints", as explained in subsection 4.3, it is ensured that the full table scan is not performed.

4.5 Results and analysis

In the following, we first presents the results concerning the three different types of queries separately, and then we draw some general conclusions. In all cases, we consider a database containing different ratio's of now-relative data. This is important in order to show to what extent our results depend on the presence of such data in the database. As suggested in the specialized literature [14], we consider three different cases: 10%, 20% and 40%. Both physical disk I/O's and CPU usage are taken into account in our analysis. It is worth stressing that, according to the analysis in [10], physical disk I/O's is the most important parameter. As a matter of fact, physical disk accesses are considered to be the bottleneck because CPU time might be reduced through the introduction of more powerful CPU's, and/or with an extension of the number of CPU's.

In order to better explain our experiments, it is worth mentioning some basic features of the spatial indexing method, which affect our experimental results.

The spatial indexing method automatically generates Minimum Bounding Regions (MBR) covering data. In the case of the *MAX* approach, now-relative data result in large rectangles, each extending to the maximum timestamp. This means that, from one side, such rectangles overlap, and from the other they contain a lot of "dead space". On the other hand, our *POINT* approach models now-relative data as points in the two dimensional space. Using features of spatial geometries it is ensured that the *point* is explicitly defined as single point rather than as rectangle with Zero length and/or width. Such points are represented as geometry Type 1 (Table 5) and because those points need less space therefore can fit in a relatively smaller number of MBR's, ensuring the higher fan-out (since more geometries type 1 can fit into the same MBR than the geometries type 2 or 3) [19].

A part of the spatial indexing method, that is very relevant in this context, is the automatic treatment of geometry types. Points, lines and rectangles are associated with a tag representing their type, so that queries fetching the same geometry (e.g., all points) are coped with in an efficient way. Such a feature may be very useful during *pruning* (i.e., the process of selecting records asked for in the query out of all the records contained in a given MBR).

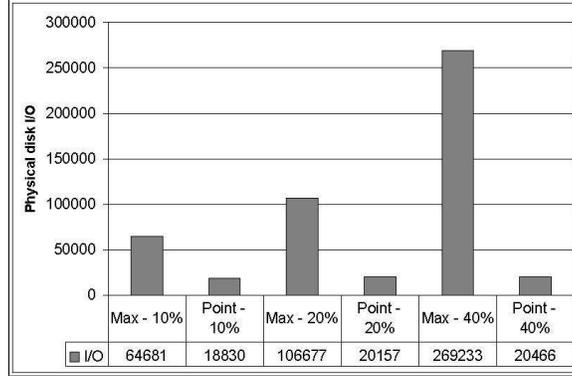


Figure 8: Query 1 : Disk Accesses

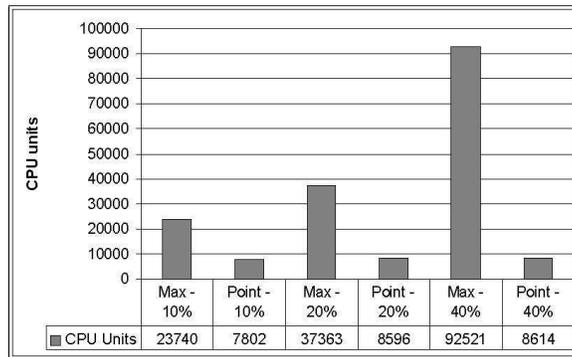


Figure 9: Query 1 : CPU Usage

Let us move now to the analysis of the experimental data.

Query 1

Figures 8 and 9 compare the results of the *POINT* and the *MAX* approaches considering physical disk I/O's and CPU usage respectively.

As shown in Figure 8, our approach outperform the *MAX* approach as regards disk accesses, especially with an increased percentage of now-relative data. In particular, with 40% now-relative data, the improvement is a factor of more then ten. Analogous advantages also regards CPU usage, as shown in Figure 9: with 40% now-relative data, the improvement is a factor of more then ten.

Analyzing the results from Figure 8 and 9 in more detail, it is important to notice that:

- (i) the disk I/O and the CPU usage for the *POINT* approach is only slightly increasing with the increase of the amount of now-relative data;
- (ii) the disk I/O and the CPU usage for the *MAX* approach is almost linearly increasing with the increase of the amount of now-relative data;

(i)

Query Q1 asks for the current and valid tuples (i.e., it looks for a point in our representation). As a consequence, in the case of query Q1, 10%, 20% and 40% represent also the answer size. It is important to note that in Figure 8 the answer size is increasing with increasing of the percentage of now-relative data, because it is natural that more and more data satisfy the criteria for time slice query, as more and more data are current and valid at given time. As explained above, points represented as geometry Type 1 require less space, so that the *POINT* approach has a better utilization of MBR's, since MBR's are smaller and contain more records. A better utilization of MBR's is possible with a higher percentage of now-relative

Approach	<i>now-relative</i> data	Physical Disk Accesses	CPU usage
Max	10%	138147	34111
Point	10%	127030	45071
Max	20%	157983	27681
Point	20%	67598	42462
Max	40%	186681	22925
Point	40%	30507	38605

Table 6: Query 2: Disk Accesses and CPU usage

data, so that physical disk I/O does not change significantly when increasing the dimension of the answer size. However, despite the answer size is linearly increasing physical disk I/O is only slightly increasing, which demonstrate the advantage of the point approach. For the same reason, also CPU usage does not significantly increase when augmenting the percentage of now-relative data. As a matter of facts, in the *POINT* approach, all now-relative data are represented by points fitting in a relatively limited number of MBR's, mostly containing points. Therefore, relatively few MBR's are fetched, and pruning is efficient, also thanks to the facilities for checking geometry types.

(ii) In contrast, in the *MAX* approach, now-relative data results into large and overlapping rectangles. This situation increases linearly with the number of now-relative tuples. Of course, large and overlapping rectangles require more MBR's, resulting in an increased physical disk I/O's, as well as in an increased CPU usage for pruning.

The above discussion throw light on the results of our experiments: the *POINT* approach outperforms the *MAX* approach considering the query type 1, especially with a higher percentage of now-relative data. The better performance of the *POINT* approach has been achieved by reducing the dimensions of the Spatial geometries. In the *POINT* approach, *now-relative* data, are represented as points, which take up less space in the leaf nodes, leading to an higher fan-out. The smaller dimensions of these geometries means there is less overlap between the MBRs, which form the boundaries of each node. In addition, since the representation of *now-relative* data in general does not extend beyond the current time, the amount of dead space (nodes that are searched that do not contribute to the answer) is reduced. Another explanation for the better performance of the *POINT* approach is the reduction of the search space, due to logically dividing the total space to the areas of interest and identifying only the geometry types of interest, which improves the performance significantly.

Query 2

Table 6 compare the results of the *POINT* and the *MAX* approaches considering both physical disk I/O's and CPU usage.

As regards physical disk I/O's, the experiments show that the *POINT* approach is slightly better than the *MAX* approach with a small fraction of now-relative data (i.e., 10%). However, in the *MAX* approach, disk I/O's increase when the percentage of now-relative data increases, due to the fact that a larger number of large rectangles are present. On the other hand, one of the distinguishing features of our *POINT* approach is the fact that, in our approach, physical disk I/O decreases with the increase of now-relative data. This advantage is obtained thanks to the fact that, in the *POINT* approach, now-relative data are modeled as points and lines (instead than as large rectangles as in the *MAX* approach), thus leading to smaller and more efficiently manageable structures, and therefore to a better fan-out.

On the other hand, in our *POINT* approach, disk I/O's decrease with the increase of now-relative data (since more points and lines are present, leading to a better fan-out). As a consequence, the *POINT* approach outperforms the *MAX* approach out to a factor of six, for 40% of now-relative data.

As regards CPU usage, for both the *POINT* approach and the *MAX* approach CPU usage decreases when now-relative data increases, due to the smaller answer size. In fact, since query Q2 retrieves current data not valid at "now", now-relative data are not part of the result. Therefore, the answer size become smaller when now-relative data increases. On the other hand, for any percentage of now-relative data, the *POINT* approach is slightly more CPU demanding than the *MAX* approach. This is due to the fact that the *POINT*

Approach	now-relative data	Disk Accesses	CPU usage
Max	10%	180353	30291
Point	10%	142478	47740
Max	20%	165980	29538
Point	20%	92560	45366
Max	40%	75038	12134
Point	40%	35155	31374

Table 7: Query 3: Disk Accesses and CPU usage

approach requires the query transformations. However, it is worth stressing once again that, according to the Theory of indexability, disk I/O's is the most important factor to be taken into account [10].

Query 3

Table 7 compare the results of the *POINT* and the *MAX* approaches considering both physical disk I/O's and CPU usage.

As regards CPU time, query Q3 behave as query Q2 above. As regards physical disk I/O's, as above the *POINT* approach is better than the *MAX* approach, and its advantage increases with the percentage of now-relative data. On the other hand, differently from query 2 above, also in the *MAX* approach, physical disk I/O's decrease with an increasing percentage of now-relative data. This is due to the nature of query Q3, which only retrieves data in the past, as regards both the transaction and the valid time. As a consequence, although the ratio of large rectangles increases with the increase of the percentage of now-relative data (as in the case of query Q2 above), in this case such rectangles are much less likely to contain record in the answer. This means that the answer size significantly decreases, and influence the physical disk I/O's more than the effect due to large rectangles.

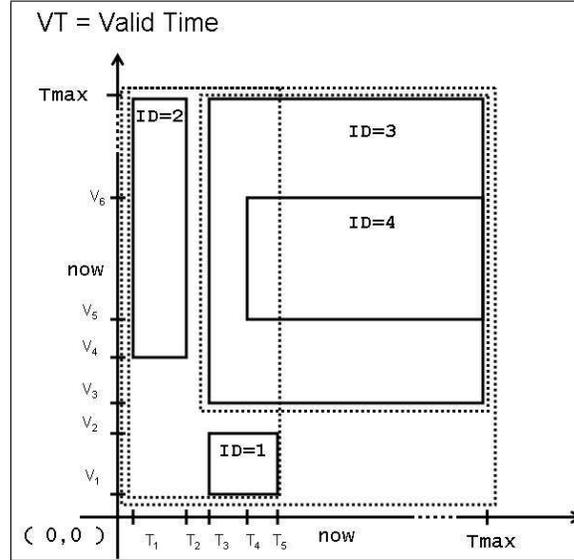


Figure 10: MBR for MAX approach

To summarize, large rectangles of current tuples in the *MAX* approach cause index entries to be spread across many nodes. In contrast the *POINT* approach can store many current tuples index entries in the same node. In addition the *POINT* approach enables current entries (long lived) to be stored efficiently with short lived entries.

The *POINT* approach has a smaller total area of the MBRs needed to cover the same tuples and needs relatively fewer leaf nodes, as shown in Table 8.

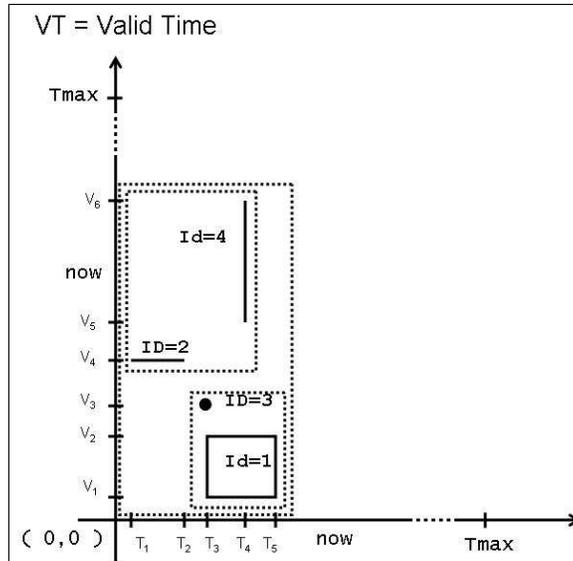


Figure 11: MBR for POINT approach

Approach	now-relative data	Fan-Out	Number of Nodes	Tree Height
Point	40%	69	18,841	4
Max	40%	37	34,540	5

Table 8: Spatial Index details

As a simple graphical example, Figures 10 and 11 show the MBRs needed in order to cope with the same data set in the *MAX* and *POINT* approach respectively. The sample data set includes the data shown in the running example shown in Tables 2, 3. Specifically, the geometries corresponding to the tuples with *ID*s from 1 to 4 have been marked in the figures.

Even in this simple example with only four tuples it can be seen that there is less overlap between the MBRs. Less overlap results in better performance as reduces the total number of disk accesses required to answer the query resulting in better performance. In Table 8 we present details, obtained from the dictionary view *all_sdo_index_metadata*, related to the spatial indexes physical structure for two methods tested and for 40% of now relative data.

Figure 12 represents the Spatial geometries of the *POINT* approach in the experiment using 40% of *now-relative* data, drawn using the *Spatial Index Advisor* of the *Oracle Enterprise Manager*. Ranges of the time axis are shown in Table 4. It is important to remind that relation has one million tuples and still it is possible to visually represent spatial geometries for the *POINT* approach. However, the Spatial Index Advisor fails in drawing the geometries from any of the *MAX* relations, even for a smaller subset of data. This is because *now-relative* data for *MAX* approach are represented with big rectangles or lines ending at '31-DEC-9999' while none *now-relative* data are represented with relatively very small rectangles or lines in two dimensional space, due to the relative scaling. This is the reason why Spatial Index Advisor is not able to draw and graphically show the geometries.

In Figures 13 and 14 we show the query execution plans for the *POINT* and *MAX* representations of current time generated by the *Oracle Enterprise Manager*. The plans highlight the benefits of the query transformations performed by our *POINT* approach. In fact, in our approach, it is sufficient to look only for geometry types points or lines. Additionally, it is important to stress that searching for a particular geometry type bitmap index can be efficiently accomplished. In the query execution plan it is important to notice that the allocation of the records which satisfy the criteria can be done at the bit level within the index. This fact positively reflects on costs (which is only 77 in the *POINT* approach). On the other hand,

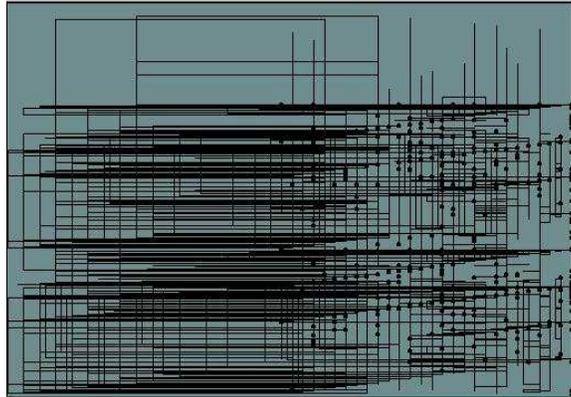


Figure 12: The Spatial Geometries of the POINT approach with 40% of new relative data

Step Name	Step #	Cost
SELECT STATEMENT	15	77
SORT (AGGREGATE)	14	
SYSTEM.TABSAME40 TABLE ACCESS (BY INDEX ROWID)	13	77
BITMAP CONVERSION (TO ROWIDS)	12	
BITMAP OR	11	
BITMAP AND	5	
BITMAP CONVERSION (FROM ROWIDS)	3	
SORT (ORDER BY)	2	
SYSTEM.TABSAME40_SPATIAL_IDX DOMAIN INDEX	1	
SYSTEM.TABSAME40_SPATIAL_BMP_IDX BITMAP INDEX (SINGLE VALUE)	4	
BITMAP AND	10	
BITMAP CONVERSION (FROM ROWIDS)	8	
SORT (ORDER BY)	7	
SYSTEM.TABSAME40_SPATIAL_IDX DOMAIN INDEX	6	
SYSTEM.TABSAME40_SPATIAL_BMP_IDX BITMAP INDEX (SINGLE VALUE)	9	

Figure 13: Query execution plan for the POINT approach and query one

Step Name	Step #	Cost
SELECT STATEMENT	4	1,371
SORT (AGGREGATE)	3	
SYSTEM.TABMAX40 TABLE ACCESS (BY INDEX ROWID)	2	1,371
SYSTEM.TABMAX40_SPATIAL_IDX DOMAIN INDEX	1	

Figure 14: Query execution plan for the MAX approach and query one

Approach	now-relative data	Disk Accesses	CPU usage
Max	10%	387072	516494
Point	10%	324962	229699
Max	20%	392458	519635
Point	20%	312453	219452
Max	40%	397527	522152
Point	40%	296458	201458

Table 9: Disk Accesses and CPU usage for Index creation

Approach	Insert size %	Disk Accesses	CPU usage
Max	2%	7634	5689
Point	2%	6534	2653
Max	5%	14835	11152
Point	5%	12898	5456

Table 10: Disk Accesses and CPU usage for Insert operation

the *MAX* approach entirely relies on spatial index which has primary and secondary pruning of records that satisfy the query condition, so that the cost is much higher (1371, as can be seen in Figure 14).

5 Performance study of Insert, Delete and Update Statements

In addition to the evaluation and comparison between the *POINT* and *MAX* approaches for different query types and different percentage of *now-relative* data, presented in section 4, we performed an extensive experimental evaluation for different DML (Database Manipulation Language) statements: **Insert**, **Update** and **Delete**. It is important to note that we considered a temporal version of these DML statements. Specifically, **Insert** has two additional parameters, to represent the start and the end of the valid time. The start of the transaction time is automatically set to the current system time, while the end of transaction time is 'now'. On the other hand, **Delete** has no additional parameter. However, it does not involve any physical deletion of the tuple from the database; instead, the end of the transaction time of the tuple is automatically set to the current system time.

We have investigated the cost of the initial creation of spatial indexes for both *MAX* and the *POINT* approach, results are presented in Table 9.

The experiments about the initial creation of the indexes showed that the *MAX* approach is more than two times computationally more expensive than the *POINT* approach in terms of *CPU* usage and slightly more expensive in terms of Disk accesses. It is important to note that the *CPU* usage and physical disk I/O are slightly increasing with the increasing percentage of *now-relative* data for the *MAX* approach. This is due to more overlap between the MBR's and more dead space with the increasing percentage of *now-relative* data. In contrast, both *CPU* usage and Physical disk I/O are slightly decreasing with the increasing percentage of *now-relative* data. This is due to the better utilization of MBR's because of smaller sizes of the stored objects (*now-relative* data represented as points or lines).

The presented results of this and of the subsequent experiments are the average values of 10 different tests in order to avoid affect of the distribution of data, also results are obtained on tables with 10% of *now-relative* data, since the qualitative difference between the two approaches is similar for other percentages.

Considering the **Insert** DML statement we performed performance evaluation for different insert sizes. We randomly selected starting and ending points of time intervals in both time dimensions.

For small insert sizes (less than 1% of the the total number of tuples) the **Insert** performance exhibited by the indexes was independent of representation and of amount of *now-relative* data. Therefore we considered larger insert sizes namely, 2% and 5%.

Approach	Delete size %	Disk Accesses	CPU usage
Max	2%	3235	2635
Point	2%	3139	2354
Max	5%	6234	5154
Point	5%	5945	4431

Table 11: Disk Accesses and CPU usage for Delete operation

Approach	Update size %	Disk Accesses	CPU usage
Max	2%	10248	7942
Point	2%	8119	3944
Max	5%	24825	17011
Point	5%	16835	7756

Table 12: Disk Accesses and CPU usage for Update operation

As shown in Table 10, physical disk I/O and CPU usage for the **Insert** statement both follow a pattern similar to the one exhibited by the index creation. The *MAX* approach is more than two times computationally more expensive than the *POINT* approach in terms of *CPU* usage and is slightly more expensive in terms of disk accesses. The experiments have also shown that the CPU usage and physical disk I/O are slightly increasing with the increasing percentage of *now-relative* data for the *MAX* approach. This is due to more overlap between the MBR’s and more dead space with the increasing percentage of *now-relative* data. In contrast, both CPU usage and Physical disk I/O are slightly decreasing with the increasing percentage of *now-relative* data. This is due to the better utilization of MBR’s because of smaller sizes of stored objects (*now-relative* data represented as points or lines).

In order to test the performance of the **Delete** statement we randomly selected objects for deletion. Once again, with a small number of deleted objects (less than 1%) we could not identify any significant difference between the *POINT* and the *MAX* approaches, and results were also quite insensitive with respect to the percentage of *now-relative* data. Therefore, similarly as for the **Insert** statement, we have considered larger delete size, consisting of 2% and 5% of the the total number of tuples, results are presented in Table 11.

Qualitatively, the results we have obtained follow the same pattern as for the **Insert** statement, due to the relative smaller number of MBR’s used for the *POINT* approach. However, it is important to note that, in the case of the **Delete** statement, both CPU usage and Physical disk I/O are approximately a half with respect to the case of the **Insert** statement. This is due to the fact that the less work must be done in order to reorganize MBR’s during the **Delete** operation.

We tested the performance of the update operation for different update sizes.

For the **Update** DML statement, we randomly selected tuples and changed their ending time to randomly generated values. For small update size we could not notice any significant difference, both in CPU usage and Physical disk I/O, between the *POINT* and the *MAX* approaches. In Table 12 we present the results considering updates of a quite relevant size, namely concerning 2% and 5% of the the total number of tuples.

Considering the Physical disk I/O for **Update** statement the *MAX* approach is only slightly more expensive than the *POINT* approach. However, the CPU usage is constantly twice as more expensive for the *MAX* approach. This is due to the need of more MBR’s to resize in case of the *MAX* approach. Also it is important to note that the results for **Update** statement shown in Table 12 are almost sum of **Insert** and **Delete** statements. This is because in temporal databases an **Update** statement is a combination of a **Delete** statement followed by an **Insert** statement.

Approach	Indexing method	Disk Accesses	CPU usage
Max	Spatial	90,889	31,928
Point	Spatial	6,863	2,886
Max	B^* -tree	47,583	4,158
Point	B^* -tree	21,586	1,958

Table 13: Query 1: Disk accesses and CPU usage for Spatial and B^+ -tree index for 40% of now relative data

6 Comparing *POINT* and *MAX* approaches adopting B^+ – tree indexes

The experimental results described in Sections 4 and 5 clearly demonstrate that our *POINT* approach outperforms its best competitor in the literature (i.e., the *MAX* approach) when spatial indexing is used. However, it is important to emphasize that the advantages of our approach are mostly independent of the specific indexing methodology being used. In order to substantiate this claim, in this section we extend our experimental work by comparing the *POINT* and *MAX* approaches in case conventional B^+ -tree indexing is adopted. (Our discussion here is quite concise, since we have already published a more detailed study based on conventional B^+ -tree one dimensional indexing in a conference paper [25]). In Table 13 we compare the *POINT* and *MAX* approaches considering both the R^* -tree and the B^+ -tree indexes, on the basis of the time slice query Type 1 introduced in subsection 3.1, and considering tables with 40% of now relative data. As regards the B^+ -tree, we have performed several experiments, adopting different combinations of indexes. Results in Table 13 concern the case which has experimentally proven to provide the best performances, namely the case in which we have created a composite B^+ -tree index on *vts*, *vte* and also one index on *tte*. Since in all our experiments the answer size almost linearly influenced the CPU usage and physical disk I/O of all methods being taken into account, in Table 13 we have focused our attention only on one time slice query randomly chosen along the time lines.

Results in Table 13 substantiate our claim that the *POINT* approach is advantageous with respect to its best competitor (i.e., the *MAX* approach) independently of the indexing methodology being used. Specifically, also in case conventional B^+ -trees are adopted on bitemporal data, the *POINT* approach outperforms the *MAX* approach as regard both disk accesses and CPU time. Moreover, it is worth emphasizing two further considerations.

- First of all, as expected, spatial methods, widely adopted in this paper, had a better performance than the usage of one-dimensional B^+ -tree indexes. This is not surprising, since the previous literature has demonstrated that spatial indexing provides crucial advantages when coping with bitemporal data. As a matter of facts, bitemporal data can be represented in a two-dimensional space, which is the specific environment that spatial indexing has been devised for.
- Second, it is important to note that the advantages of the *POINT* approach with respect to the *MAX* approach are greater in case spatial indexing is adopted. Specifically, roughly speaking, Table 13 shows that *POINT* outperforms *MAX* of a factor of about two when B^+ -trees are used, and of a factor of about fifteen in case spatial indexes are used. This seems to us an additional positive result of our approach, since it outperforms the *MAX* approach especially in case the most efficient indexing techniques are used for bitemporal data.

7 Conclusions

Many database applications involve a significant amount of time dependent data. Often, a significant part of such data is now-relative. However, current temporal relational approaches have proven to be quite inefficient when coping with now-relative data, whose access has proven to influence significantly the efficiency of accessing temporal data [14].

In this paper we have presented a new approach overcoming such a limitation. Specifically, the main original contributions of the work described in this paper are:

- we have proposed a new way of representing *now* in relational temporal databases, which does not introduce overloading with semantic ambiguity;
- considering different types of queries, we have introduced logical query transformation in order to properly cope with the new representation;
- the proposed approach is based only on logical transformations of queries; it does not require modification to the kernel, so an off-the-shelf indexing methodologies can be used.
- in the experimental study we have demonstrated that our *POINT* approach outperforms the *MAX* approach mostly independently of the indexing methodology being adopted. Specifically, we have taken into account both "standard" B^+ - trees and spatial R^* - trees, and considered different types of benchmark queries. Moreover, we have also experimentally shown that our *POINT* approach outperforms the *MAX* approach considering the performance of data manipulation operations.

There are number of issues remain open and need to be addressed in future. For example, it would be interesting to investigate how the *POINT* approach could also be extended in order to cover a wider range of temporal database queries, and to what extent the efficiency improvement extends to such a broader context. Additionally, it would be interesting to investigate how the *POINT* approach would affect the performance of other Bitemporal access methods that do not rely on R-tree spatial indexes.

Acknowledgements

The authors are very grateful to the anonymous referees, for their in-depth review of the paper, and for their constructive and inspiring criticism.

References

- [1] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R^* -Tree: An Efficient and Robust Access Method for Points and Rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, pages 322–331. ACM Press, 1990.
- [2] Rasa Bliujute et al. Light-Weight Indexing of General Bitemporal Data. In *Statistical and Scientific Database Management*, pages 125–138, 2000.
- [3] Rasa Bliujute, Christian S. Jensen, Simonas Saltenis, and Giedrius Slivinskas. R-Tree Based Indexing of Now-Relative Bitemporal Data. In *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, New York, USA*, pages 345–356, 1998.
- [4] J. Clifford, C. Dyreson, T. Isakowitz, C. S. Jensen, and R. T. Snodgrass. On the semantics of "Now" in databases. *ACM Transactions on Database Systems (TODS)*, 22(2):171–214, 1997.
- [5] C.J. Date, H. Darwen, and N.A. Lorentzos. *Temporal Data and the Relational Model*. Morgan Kaufmann, ISBN: 1-55860-855-9, 2002.
- [6] Richard T. Snodgras. The Temporal Query Language Tquel. *ACM TODS*, 12(2):247–298, 1987.
- [7] Volker Gaede and Oliver Gunther. Multidimensional Access Methods. *ACM Computing Surveys (CSUR)*, 30(2):170–231, 1998.
- [8] Connie Dialeris Green. Oracle9i Database Performance Tuning Guide and Reference, Release 2 (9.2), Part No. A96533-02. url=http://download-west.oracle.com/docs/cd/B10501_01/server.920/a96533.pdf, 2002.

- [9] Pat Hayes. A Catalog of Temporal Theories. *Technical Report UIUC-BI-AI-96-01, University of Illinois*, 1996.
- [10] J.M. Hellerstein, E. Koutsupias, and C.H. Papadimitriou. On the Analysis of Indexing Schemes. *16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 249–256, 1997.
- [11] C. S. Jensen and R. Snodgrass. Temporal Data Management. *IEEE Transactions on Knowledge and Data Engineering*, 11(1):36–44, 1999.
- [12] Christian S. Jensen. Introduction to Temporal Databases, Research. url=<http://www.cs.auc.dk/csj/Thesis/pdf/chapter1.pdf>, 2000.
- [13] Christian S. Jensen, Richard T. Snodgrass, and Michael D. Soo. The tsql2 data model. In *The TSQL2 Temporal Query Language*, pages 157–240. 1995.
- [14] Kristian Torp and Christian S. Jensen and Michael H. Bohlen. Layered Temporal DBMS: Concepts and Techniques. *Database Systems for Advanced Applications*, pages 371–380, 1997.
- [15] A. Kumar, V.J. Tsotras, and C. Faloutsos. Designing Access Methods for Bitemporal Databases. *IEEE Transactions on Knowledge and Data Engineering*, 10(1):1–20, 1998.
- [16] Anil Kumar, Vassilis J. Tsotras, and Christos Faloutsos. Access Methods for Bi-Temporal Databases. In *Proceedings of the International Workshop on Temporal Databases*, pages 235–254, 1995.
- [17] Jim Melton and Alan R. Simon. *SQL:1999 - Understanding Relational Language Components*. Morgan Kaufman, ISBN: 1-55860-456-1, 2002.
- [18] Chuck Murray. Oracle Spatial User’s Guide and Reference, Release 9.2, Part No. A96630-01. url=http://download-west.oracle.com/docs/cd/B10501_01/appdev.920/a96630/title.htm, 2002.
- [19] Oracle. Oracle Spatial 8.1.6 Performance-Related Characteristics, An Oracle Technical White Paper. url= http://www.oracle.com/technology/products/spatial/pdf/spatial_perf_twp.pdf, 2000.
- [20] Robert Fenk and Volker Markl and Rudolf Bayer. Interval Processing with the UB-Tree. *Proceedings of the 2002 International Symposium on Database Engineering and Applications*, pages 12–22, 2002.
- [21] Betty Salzberg and Vassilis J. Tsotras. Comparison of Access Methods for Time Evolving Data. *ACM Computationg Surveys*, 31(2):158 – 221, 1999.
- [22] Richard T. Snodgrass. *The TSQL2 Temporal Query Language*. Kluwer Academic, ISBN:0-7923-9614-6, 1995.
- [23] Richard T. Snodgrass and I. Ahn. Temporal Databases. *IEEE Computer*, 19(9):35–42, 1986.
- [24] B. Stantic, S. Khanna, and J. Thornton. An Efficient Method for Indexing Now-relative Bitemporal data. In *Proceeding of the 15th Australasian Database conference (ADC2004), Denidin, New Zealand*, 26(2):113–122, 2004.
- [25] B. Stantic, J. Thornton, and A. Sattar. A Novel Approach to Model NOW in Temporal Databases. In *Proceeding of the 10th International Symposium on Temporal Representation and Reasoning (TIME-ICTL 2003), Cairns*, pages 174–181, 2003.
- [26] Vassilis J. Tsotras, Christian S. Jensen, and Richard T. Snodgrass. An Extensible Notation for Spatiotemporal Index Queries. *ACM SIGMOD Record*, 27(1):47–53, 1998.
- [27] C. Zikopoulos, G. Baklarz, D. deRoos, and R.B. Melnyk. *DB2(R) Version 8: The Official Guide - Chapter 4*. IBM Press, ISBN: 0131401580, 2003.