A Novel Approach to Model NOW in Temporal Databases

Bela Stantic, John Thornton, Abdul Sattar
School of Information Technology
Griffith University Gold Coast, Australia
{b.stantic, j.thornton, a.sattar}@griffith.edu.au

Abstract

In bitemporal databases, current facts and transaction states are modelled using a special value to represent the current time (such as a minimum or maximum timestamp or NULL). Previous studies indicate that the choice of value for now *(i.e. the current time) significantly influences the efficiency of accessing bitemporal data. This paper introduces a new approach to represent* now*, in which current tuples and facts are represented as* points *on the transaction time and valid time line respectively. This allows us to exploit the computational advantages of point-based query languages. Via an empirical study, we demonstrate that our new approach to representing* now *offers considerable performance benefits over existing techniques for accessing bitemporal data.*

1 Introduction

Relational data models and their implementations 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 or even store future states. In addition, most production 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 fact, it is difficult to identify any database application that does not require some form of time-varying data. Conversely, the built-in temporal support offered by commercial database products and the Relational Query language SQL 1999 [8] is limited to predefined, time-related data types. Some commercial databases include temporal extensions, e.g. the Oracle TimeSeries cartridge, Oracle 9i "Flash-Back", and the Informix TimeSeries Data-Blade, but these extensions still do not fully support the successful management of time-varying data. Research has demonstrated that applications can significantly benefit from using a temporal RDBMS, and also that temporal support is needed that goes beyond simple data types [7].

Associating data with time values and keeping a history of fact validity is not technically difficult even using non-temporal RDBMS technology [13] [4]. However, it is a difficult task to efficiently query such time-varying data and to identify integrity constraints that hold over several database states.

A database is considered temporal if it is able to manage time-varying data and it supports some time domain distinct from user-defined time. In temporal databases time can be captured along two distinct time lines: transaction time and valid time. A *bitemporal database* is a combination of valid time and transaction time databases and records the database states with respect to both valid time and transaction time. The valid time line represents when a fact is valid and the transaction time line represents when a transaction was performed. Recording bitemporal data generally requires that updates are appended to the database (rather than overwriting existing values). This can easily lead to the storage of large volumes of data, and consequently makes the selection of efficient access methods very important. Storing bitemporal data also requires the selection of appropriate time units (granularities). Without careful management, the informational benefits of bitemporal data can be easily outweighed by the costs of poor access times and difficulties in formulating queries [5].

In our current work we use the TQuel four-timestamp format to represent bitemporal data [11], where, in addition to non-temporal attributes, each tuple has four temporal attributes: Vt_s and Vt_e representing the starting and ending time points of fact validity in the modelled world, and Tt_s and Tt_e representing the time when a tuple is inserted in the database and the time it is logically deleted. Sample bitemporal data is shown in Table 1. A tuple is considered current if it is part of the current database state, i.e. it has not been logically deleted by assigning a timestamp to Tt_e different from the value of *now*. In the literature such a tuple, where the validity of a fact in the modelled world is valid up to the

current time, is called *now-relative*.

Despite two decades of research in temporal databases, relatively few papers have addressed the issue of indexing temporal data. Even less have addressed the issues of how to index now-relative data, or temporal data that are current or valid *now*. Existing research shows that regular indices, such as B^+- trees, are unsuited for temporal data [10], and recently several other indices have been proposed. Only a few index structures address the need to store the *current time*, a need which is accommodated by almost all temporal data models and is natural and meaningful for many kinds of applications. As valid time and transaction time are considered to be orthogonal [12], bitemporal data can be represented in two dimensional space, enabling us to apply spatial indexes. For bitemporal indices based on R-trees, the maximum-timestamp approach is a straightforward solution to the indexing of now-relative data. But it is obvious that in this approach, facts with now-relative valid-time intervals are represented using very large rectangles, and the resulting search performance is poor due to excessive dead space in the index nodes and overlap between nodes. We are aware of only a few structures that address the issues related to storing now-relative data in temporal databases [1], [2], [9]. Some of the proposals rely on special variables *until changed* and *now* that should be part of a not yet existing temporal relational model. Research also suggests that the widespread acceptance of such a model is unlikely, due to the large commercial investment in the existing relational model, both in terms of developed code and expertise [7]. At the same time, due to the significant drop in the price of disk storage, more and more database applications are using added temporal dimensions and, as a consequence, are facing increasingly poor response times.

Bitemporal databases store past, present and even future facts in either logically deleted or current tuples. To represent that a fact is current *now*, or that a tuple has not been logically deleted, requires the storing of a value representing the current time. In the literature, several concepts to represent current time have been proposed by including special variables, such as : "now" , "until-changed", "forever", "∞", "@", and "-". However, the same basic issue applies to any approach, i.e. how physically to store that concept in the database [3]. As *now* is not part of the domain of SQL1999 values [8], it is necessary to represent the current time by some other value, in such a way that the chosen value is not *overloaded* (i.e. does not have more than one meaning).

It has been shown that the choice of the physical value for *now* significantly influences the efficiency of accessing bitemporal data [14]. Currently, the literature has concentrated on three basic approaches: firstly using NULL, secondly using the smallest timestamp and thirdly using the largest timestamp supported by the particular RDBMS. A disadvantage of using NULL is that columns that permit NULL values prevent the RDBMS from using indexes. Conversely, using a non-NULL value can also affect indexing badly. For example, when an index is used to retrieve tuples with a time period that overlaps *current time*, and *now* is represented with smallest or largest timestamp value, tuples with the Vt_e (Valid time end) or Tt_e (Transaction time end) attribute set to *now* will not be in the range retrieved.

We have seen that current facts are represented by assigning the value *now* to Vt_e, and that by assigning *now* to Tt_e we can represent the belief that a tuple is current (or not logically deleted). This shows importance of *now* in bitemporal databases. Further, the importance of *now* increases when we consider that current tuples and current facts are likely to be accessed more frequently. While issues related to storing *now* are discussed in the literature in the context of temporal databases, this equally applies to conventional relational DBMS technology.

In this paper we propose a new approach to modelling current time in temporal databases that overcomes the limitation of an attribute set to *now* not being in the range retrieved. In addition, our approach has significant computational advantages over the previously proposed methods and, to the best of our knowledge, it is the only approach to representing *now* that ensures the value *now* is not overloaded.

In the remainder of the paper, we first look more closely at previous approaches to modelling current time and highlight their limitations and disadvantages. Then, in section 3, we present the "core" of the new proposed method for representing current time, and empirically evaluate our approach in comparison to two of the standard existing approaches. Finally, in section 4, we present our conclusions and discuss possible extensions and future work.

2 Traditional representations of current time

As time seems to be continuous, and current time is ever-increasing, a significant question in computer science, particularly with respect to databases, is how to store the value of current time (or *now*).

In line with the existing research, we have accepted a discrete model of time. Since digital computers only support a limited granularity for real numbers, most proposals for adding time to the relational model are based on a discrete, totally ordered set of time instants to represent both the transaction and valid time dimensions. This ordering is defined as follows:

For valid time :

$D_v = \{t_0, t_1, \cdots, t_i, \cdots, now\} \cup \{\infty\}$
$\forall t^{'}, t^{''} \in D_v / \{\infty\} : (t^{'} < t^{''} < \infty) \vee$
$(t^{''} < t^{'} < \infty) \vee (t^{'} = t^{''} < \infty)$

For Transaction time :

$D_t = \{t_0, t_1, \cdots, t_i, \cdots, now\}$
$\forall t^{'}, t^{''} \in D_t / \{now\} : (t^{'} < t^{''} < now) \vee$
$(t^{''} < t^{'} < now) \vee (t^{'} = t^{''} < now)$

In a discrete totally ordered model, a time interval, denoted as [Vt_s , Vt_e), represents a set of a countably infinite equidistant time instants [4], where Vt_s is the starting time instant and Vt_e is the ending time instant representing the starting and ending boundaries respectively. 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. Also, a time interval [Vt_s , Vt_e) is *closed* on the left-hand side and *open* on the right. This means that the start point of the interval = Vt_s and the end point < Vt_e, i.e. the interval *includes* the point Vt_s and *excludes* the point Vt_e.

In such a model, now-relative data contained in tuples that are currently valid, can be represented as [Vt_s , *now*). Here Vt_s represents the time point when the fact started to be true and *now* represents that the fact is still current and that the time validity of fact is continuously expanding (i.e. its end is unknown).

Assuming we are interested in using existing technology and given that the domain of SQL1999 values does not contain a special value for *now*, the task becomes one of selecting 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 ambiguous and the value is overloaded.

As mentioned before, previous work has concentrated on three physical values to represent *now*: the NULL value , the smallest timestamp (MIN) and largest timestamp (MAX) supported by a particular RDBMS. It is clear that whichever of these values is chosen, the domain of the data type becomes limited and a potential for **overloading** 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 timestamp value and can be processed faster. Despite this, the crucial disadvantage of NULL is that columns that permit NULL values cannot be indexed by a conventional RDBMS, 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_e or Tt_e is used to retrieve tuples with a time period that overlaps *now*, and *now* is represented with the smallest or largest timestamp value, tuples with the Vt_e or Tt_e 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**. Hence all previous approaches to model current time using the largest timestamp value (MAX), the smallest timestamp value (MIN) or NULL have disadvantages related to indexing. This has been highlighted in the literature and is particularly relevant to accessing bitemporal data, where the choice of the value of *now* has been shown to significantly affect access efficiency [14].

3 A new approach to model now

Each of the previously discussed approaches to representing *now* has severe limitations in terms of indexing or overloading. 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. However, MAX and MIN have further performance problems navigating and updating time value indexes, due to the redundancy of the special timestamp value used to represent the current time. This means that *all* Tt_e and Vt_e values will be indexed to the *same* special value (i.e. MAX or MIN) causing the index to search *sequentially* through these records. We term this the **index redundancy problem**.

A major aim of our research is to overcome the limitations of previous approaches to representing *now*. These limitations have been identified as **overloading**, the **range index problem** and the **index redundancy problem**. From a consideration of the redundancy problem, it became clear that our solution should produce a value for *now* that is related to some distinct property of the tuple to which it refers. In this way the level of redundancy can be reduced. Also, to avoid the range index problem, a value is needed that is *contained* in the interval between the start time and the actual current time. We therefore concluded that the best solution would be to make the end point of any current interval *equal to the start point* (i.e. Vt_e=Vt_s and Tt_e=Tt_s). This representation therefore defines the actual interval between the start and end points of a current interval to be zero (i.e. it becomes a *point* on the time line rather than an interval).

A first objection to this approach could be to say that the value is overloaded, e.g. it becomes impossible to distinguish between an interval that actually started on the 12.12.01 and finished on the same day, from one that started on 12.12.01 and is still current (given that the granularity of our time value or chronon is one day). However, using the definitions of interval start and end points we can see that this objection is not valid: i.e. the interval [12.12.01, 12.12.01) is closed on the left (and so starts on the 12.12.01) and open on the right (and so finishes before 12.12.01). Therefore this interval has no meaningful duration. If it is required to distinguish the start and end times more finely, then the granularity of the time value must be changed (e.g. from days to hours). From this discussion, we can see that our new approach to representing current time also solves the overloading problem as any tuple where Vt_e=Vt_s or

ID	$Name$	$Position$	$[Vt_s$	$Vt_e)$	$[Tt_s$	$Tt_e)$
1	$Megan$	DBA	21.08.00	10.05.02	**26.09.00**	**26.09.00**
2	$Stephan$	$Teacher$	23.07.00	30.01.01	01.07.00	26.10.00
3	$Mark$	$Admin$	**22.03.99**	**22.03.99**	**10.04.99**	**10.04.99**
4	$Steven$	$Officer$	**21.02.01**	**21.02.01**	13.02.01	23.02.01
...	...	...	...	...	...	...
$1,020,965$	...	...	...	...	...	...

Table 1. Sample Bitemporal data with POINT representation of *now*

Tt_e=Tt_s is *unambiguously* current.

To illustrate our new point-based approach to *now* (which we term **POINT**), consider the relation in Table 1. Here, in tuple ID = 3, we can conclude that Mark has the current position of "Admin", firstly because Vt_e equals Vt_s (meaning the fact is currently valid) and secondly because Tt_e equals Tt_s (meaning the tuple has not been deleted). Tuple ID = 1 represents that Megan had the position of "DBA" from "21.08.00" to "10.05.02" and because the tuple is current (as Tt_e equals Tt_s) this represents our *current* belief about Megan's past position. Tuple ID = 2 is logically deleted (as Tt_e differs from Tt_s), which means that we believed from "01.07.00" to "26.10.00" that Stephan was employed as a "Teacher" between "23.07.00" and "30.01.01". Tuple ID = 4 is logically deleted (as Tt_e differs from Tt_s), which means that we believed from "13.02.01" to "23.02.01" that Steven has a current employment as a "Officer" from "21.02.01". Hence, when the timestamp for Vt_e is the same as Vt_s it means that a fact is valid *now*. Similarly when Tt_e is the same as Tt_s it means that a tuple is current. Note also that in Table 1 we are displaying the data as it would be stored in the database, in actual practice we would expect the end time value when Vt_e=Vt_s or Tt_e=Tt_s to be displayed to the user as some special symbol (such as *now*, *until changed* or NULL).

3.1 Experiments

In order to evaluate our POINT approach to representing *now* we decided to empirically compare POINT to both the MAX timestamp approach and to using NULL. In doing this we followed previous research in the area, but in contrast to previous work, we decided not to test the minimum timestamp value (MIN) as this has already been shown to be consistently worse than MAX [14]. For each of our methods we generated three relations, each differing only in the physical representation of the current time value. Then on each relation we performed three different representative time slice queries shown in Figure 1. We chose time slice queries because of their recognised importance in temporal databases [15].

Query One retrieves the current state in both transaction and valid time. It selects tuples with transaction and valid time intervals that both overlap with the current time, as well as retrieving tuples where valid time and transaction time ends at *now*. Basically, Query One retrieves what we currently believe about the current state of the world.

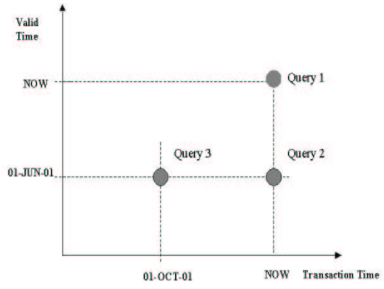

Figure 1. Types of Time slice Queries

Query Two time slices the relation as of *now* in transaction time and as of a past time in valid time. In other words it retrieves our current belief about a past state of the world. Query Three time slices the relation as of a past time in both transaction time and valid time, which means it retrieves a past belief about a past state of reality. Queries One and Two favour the current state, because this state is assumed to be accessed much more frequently than old states.

Results for 10% current data			
Disk	CPU	Duration	Exp. Type
7564	758	26	POINT10-1
14333	1565	29	MAX10-1
25666	1138	32	NULL10-1
22485	1855	36	POINT10-2
31827	2822	51	MAX10-2
48703	2607	74	NULL10-2
49571	1608	38	POINT10-3
52749	2150	57	MAX10-3
59382	1801	62	NULL10-3

Results for 20% current data			
Disk	CPU	Duration	Exp.Type
12641	1249	37	POINT20-1
27848	2671	47	MAX20-1
40818	1638	49	NULL20-1
35275	2575	56	POINT20-2
65448	4397	91	MAX20-2
82041	4154	98	NULL20-2
42762	2792	65	POINT20-3
86280	3168	81	MAX20-3
98033	2998	99	NULL20-3

Our tests were performed on a four 450MHZ CPU - SUN UltraSparc II processor machine, running Oracle 9.2.0 RDBMS, with a database block size of 8K. During the tests the server had no other significant load. We performed experiments using four different sizes of SGA (System Global Area): 30MB, 50MB, 100MB and 200MB in order to investigate the effects of aging buffers. We created B^*-tree composite indexes on Vt_e and Vt_s and a B^*-tree index on Tt_e for all tables. We also performed additional tests using function based indexes.

Results for 40% current data			
Disk	CPU	Duration	Exp.Type
22642	1872	48	POINT40-1
48188	4710	75	MAX40-1
54807	2426	70	NULL40-1
62848	4256	88	POINT40-2
124561	7356	145	MAX40-2
148594	6542	122	NULL40-2
76191	4790	111	POINT40-3
118334	6115	175	MAX40-3
109334	4336	154	NULL40-3

Our queries were executed on nine different bitemporal tables, three for each representation of *now*, with each table having a random distribution of one million temporal tuples and a granularity of one day. Within this data set, each representation of *now* was tested on three separate relations: In the first relation, 10% of the tuples overlapped with the current time in both transaction and valid time. In the second and third relations this percentage was 20 and 40, respectively.

The tables present our experimental results for 100MB SGA, where CPU usage is measured in CPU units, duration is measured in seconds and Experiment Type uses the notation METHODm-n, where METHOD represents the method used to model *now*, i.e. either our new POINT approach (where Vt_e = Vt_s and Tt_e = Tt_s) or MAX (using the Oracle max timestamp "31-DEC-9999") or NULL; m represents the percentage of the tuples in the experiment overlapped with the current time in both transaction and valid time (i.e. either 10, 20 or 40%); and n represents the time slice query type (i.e. either Query One, Two or Three).

3.2 Analysis

The results show that the new POINT representation for *now* clearly outperforms both MAX and NULL in terms of disk reads, CPU usage and query duration across the full range of our problem set. Looking more specifically, Query One most accurately measures the effect of varying the percentage of tuples overlapped with current time, as it retrieves all such tuples. On this measure we can see all techniques start slowing down as the percentage of overlap increases (i.e. as we move from 10-1 to 20-1 and 40-1) but also that the relative advantage of POINT over the other techniques grows as the overlap increases. For instance, at 10-1 the duration times for POINT MAX and NULL are 26, 29 and 32 respectively, whereas at 40-1 this has changed to 48, 75 and 70 (i.e. POINT has gone from being roughly comparable to NULL and MAX to performing almost as twice as fast). This demonstrates the distinct advantage POINT has in retrieving current records, due to POINT representing current records as single points on the valid and transaction time lines rather than as the intervals defined by differing start and end points used by MAX.

The performances on Queries Two and Three again show POINT to be consistently superior, although here the relative performance of the three techniques remains relatively stable as the overlap percentage increases. This is because Queries Two and Three are more concerned with past states of the database and so are not so affected by the proportion of tuples that overlap the current time.

Also, the disk access results show POINT to be consistently better than MAX or NULL especially on the 10-1 problems for Query One where the duration measures are fairly similar. According to the Theory of Indexability [6], the I/O complexity cost measured by the number of disk accesses for updating and answering queries is one of the most important factors for measuring performance. This is because, as technology advances, CPU speeds tend to increase relatively faster than disk I/O speeds. In Figures 2, 3 and 4 we graphically represent our results for the case when m = 10% (i.e. 10% of tuples overlap with current time in both valid and transaction time with SGA again set at 100MB). This gives a clearer representation of the relative performance of each technique and highlights the dominance of POINT especially in terms of disk reads on Query One.

We also investigated, through further experimental study, several possible factors that could interfere with our results, but due the limitations of space we will only briefly mention them. One of the factors that can affect the number of physical disk reads is the size of SGA. As previously

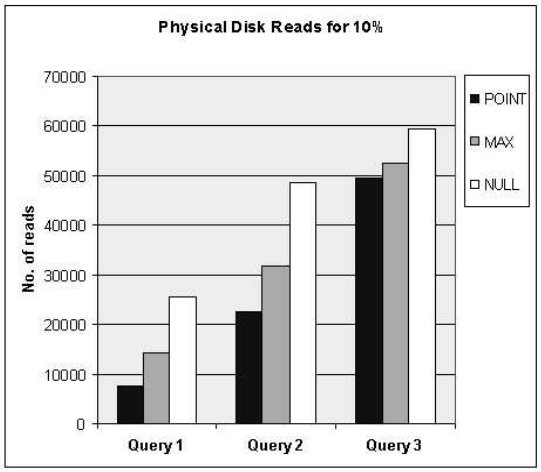

Figure 2. Number of Physical disk reads

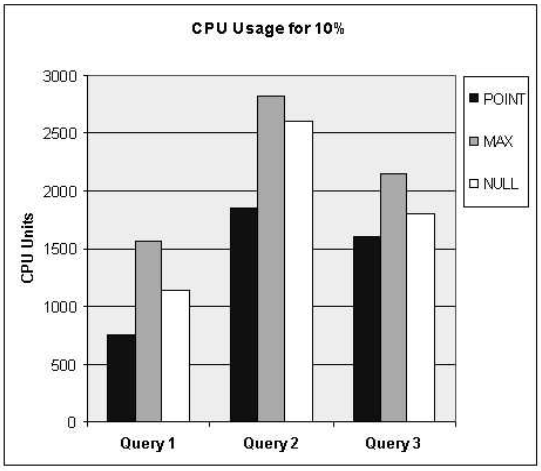

Figure 3. CPU Usage

mentioned, we looked at the effects of aging buffers by varying the size of SGA through 4 values: 30Mb , 50Mb, 100Mb and 200Mb. Our results showed small differences between approaches and favoured larger values of SGA. The main difference noted was that a smaller SGA size tends to favour the NULL representation of *now*, resulting in relatively fewer disk reads. This is because a NULL value uses less space than a timestamp and so the chance of the buffer aging is reduced. Apart from this, we did not notice any significant effects on the results when comparing the different representations of *now* on different sizes of SGA, i.e. changes in the number of physical disk reads, CPU usage and query duration are virtually linear for all the approaches considered.

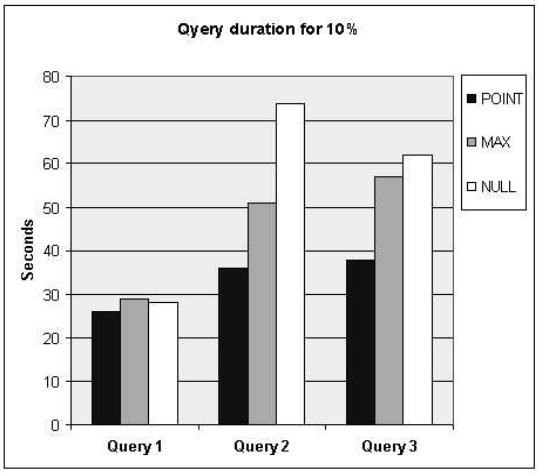

Figure 4. Duration of queries in seconds

4 Conclusion and Future work

This study makes the following contributions to the field:

- by investigating different representations of *now* in bitemporal databases, we presented a better understanding of the significance of modelling current time, particularly in the context of efficiently accessing bitemporal data;
- we identified limitations of previous approaches to representing *now*, namely *overloading*, the *range index problem* and the *index redundancy problem*;
- we proposed a new approach, called POINT, to represent current time in bitemporal databases; and
- in the experimental study we have demonstrated that the proposed POINT based approach not only overcomes the limitations of previous approaches, but also leads to a significant improvement in the efficiency of querying bitemporal databases in comparison to existing methods.

Currently, we are investigating the effect of the proposed POINT approach on the performance of multidimensional indexes, such as the spatial R-tree index.

Finally, the POINT-based method to represent *now* in temporal databases introduced in this paper, could significantly effect performance of previously proposed index structures for temporal data. It would be interesting to investigate the effect of POINT on various index structures for temporal data with respect to space usage and response time. It would be also interesting to look at the time required for index restructuring in relation to insertion, deletion and updating. This investigation should follow the directions used for the worst case scenario in [10].

References

[1] R. Bliujute, C. S. Jensen, S. Saltenis, and G. 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 City, New York, USA*, pages 345–356, 1998.

[2] R. Bliujute, C. S. Jensen, S. Saltenis, and G. Slivinskas. Light-weight indexing of general bitemporal data. In *Statistical and Scientific Database Management*, pages 125–138, 2000.

[3] 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.

[4] C. Date, H. Darwen, and N. Lorentzos. *Temporal Data and the Relational Model*. Morgan Kaufmann, 2002.

[5] C. E. Dyreson, R. T. Snodgrass, and M. Freiman. Efficiently supporting temporal granularities in a DBMS. Technical Report TR 95/07, 1995.

[6] J. Hellerstein, E. Koutsupias, and C. Papadimitriou. On the analysis of indexing schemes. *16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1997.

[7] C. S. Jensen. Introduction to temporal databases, research. *http://www.cs.auc.dk/ csj/Thesis/pdf/ chapter1.pdf*, 2000.

[8] J. Melton and A. R. Simon. *SQL:1999 - Understanding Relational Language Components*. Morgan Kaufman, 2002.

[9] M. A. Nascimento and M. H. Dunham. Indexing valid time databases via B^+-tree. *IEEE Transactions on Knowledge and Data Engineering*, 11(6):929–947, 1999.

[10] B. Salzberg and V. J. Tsotras. Comparison of access methods for time evolving data. *ACM Computiong Surveys*, 31(1), 1999.

[11] R. Snodgras and et al. The temporal query language TQEL. *ACM TODS*, 12(2):247–298, 1987.

[12] R. Snodgrass and I. Ahn. Temporal databases. *IEEE Computer*, 19(9):35–42, 1986.

[13] R. T. Snodgrass. *Developing Time-Oriented Database Applications in SQL*. Morgan Kaufmann, 2000.

[14] K. Torp, C. S. Jensen, and M. Bohlen. Layered implementation of temporal DBMS concepts and techniques. *A TimeCenter Technical Report TR-2*, 1999.

[15] V. J. Tsotras, C. S. Jensen, and R. T. Snodgrass. An extensible notation for spatiotemporal index queries. *ACM SIGMOD Record*, 27(1):47–53, 1998.