Indexing Valid Time Databases Via B^+-trees – The MAP21 Approach

Mario A. Nascimento and Margaret H. Dunham

March 9, 1998

TR- 26

A TIMECENTER Technical Report

Title	Indexing Valid Time Databases Via B^+-trees – The MAP21 Approach
Author(s)	Mario A. Nascimento and Margaret H. Dunham
Publication History	March 1997. A Southern Methodist University Technical Report (97-CSE-08). March 1998. A TIMECENTER Technical Report.

TIMECENTER Participants

Aalborg University, Denmark
Christian S. Jensen (codirector)
Michael H. Böhlen
Renato Busatto
Heidi Gregersen
Dieter Pfoser
Kristian Torp

University of Arizona, USA
Richard T. Snodgrass (codirector)
Anindya Datta
Sudha Ram

Individual participants
Curtis E. Dyreson, James Cook University, Australia
Kwang W. Nam, Chungbuk National University, Korea
Mario A. Nascimento, State University of Campinas and EMBRAPA, Brazil
Keun H. Ryu, Chungbuk National University, Korea
Michael D. Soo, University of South Florida, USA
Andreas Steiner, ETH Zurich, Switzerland
Vassilis Tsotras, University of California, Riverside, USA
Jef Wijsen, Vrije Universiteit Brussel, Belgium

For additional information, see The TIMECENTER Homepage:
URL: <http://www.cs.auc.dk/general/DBS/tdb/TimeCenter/>

The TIMECENTER icon on the cover combines two "arrows." These "arrows" are letters in the so-called *Rune* alphabet used one millennium ago by the Vikings, as well as by their precedessors and successors. The Rune alphabet (second phase) has 16 letters, all of which have angular shapes and lack horizontal lines because the primary storage medium was wood. Runes may also be found on jewelry, tools, and weapons and were perceived by many as having magic, hidden powers.

The two Rune arrows in the icon denote "T" and "C," respectively.

Abstract

We present an approach named MAP21 which uses standard B$^+$-trees, in a multiple disks single processor architecture, to provide efficient indexing of valid time ranges. The approach is based on mapping one dimensional ranges to one dimensional points where the lexicographical order among the ranges is preserved. We compare MAP21 to the Time Index and the R*-tree and we show that MAP21 is comparable to or outperforms those, depending on the number of trees utilized, the degree of parallelization among these and the type of query. The main contribution of this paper is to show that standard B$^+$-trees, available in virtually any DBMS, can be used to provide an efficient temporal index. Although our discussion is made in terms of valid time databases, MAP21 can be used (or extended to be used) within other application domains as well.

1 Introduction

This paper deals with the problem of indexing one dimensional ranges. Assuming our domain of application to be the one of Temporal Databases (TDBs), we focus on the problem of indexing valid time databases (VTDBs), also referred to as historical databases [SA86]. In VTDBs each version of record k (denoted by R^k) has, besides the non-temporal attributes, a valid time range $V^k = [V_s^k, V_e^k]$. This means that R^k was valid in the modeled world during that time range [JCG$^+$94]. VTDBs also support predictive and retroactive updates and thus do not require any ordering among the temporal ranges at input time. On the other hand they do not allow two versions of the same record Throughout the paper we consider a record version to be a record by itself, i.e., we assume tuple-versioning instead of attribute-versioning.

There has been a respectable amount of published research devoted to TDB and its applications [McK86, Soo91, Kli93, ÖS95, TK96]. However, relatively few have addressed the issue of indexing temporal data. In what follows we briefly review some of the work done in the area. We do not intend for this review to be exhaustive though. For a good survey we refer the reader to [ST94].

The Time Index [EWK93], Time Index$^+$ [K$^+$94], TP-index [SOL94], Interval B-tree [AT95], and the B$^+$-tree based TP-index[1] [G$^+$96] all index valid time ranges, therefore no ordering in the ranges input is imposed. The drawback of the Time Index is the size of the index itself which is quadratic on the number of indexed ranges. The Time Index$^+$ is an improved Time Index, which does reduce substantially the storage needed by the original Time Index while speeding up query processing time. However, in the worst case its storage requirement remains very large. The TP-index maps a (one-dimensional) range into a point in the two-dimensional space, and the query processing is reduced to a spatial search problem. It is space-wise more efficient than the Time Index, but is biased towards some types of queries. The Interval B-tree overcomes both shortcomings above but requires a complex nested data structure, which makes it difficult to be implemented. The B$^+$-tree based TP-index assigns an ordering among the two dimensional space created by the original TP-index, and this ordering is mapped into a B$^+$-tree. This approach is similar to the one we propose, but there are some fundamental differences. We discuss these differences in more detail in Section 5, after presenting our approach.

The Monotonic B$^+$-tree [EWK93], the Append-Only Tree [GS93] and the Snapshot Index [TK95] also aim at indexing valid time ranges. However they require the input to be given with the valid start time in increasingly monotonic order, which is a transaction time characteristic. This prevents these approaches from handling either predictive or retroactive updates.

The Time-Split B-tree [LS93] indexes a transaction time database (also known as a rollback database [SA86]). It models transaction time ranges assuming they are stepwise constant (i.e., the transaction end time of one version is the transaction start time of the next version). The approach is unique in the sense that it indexes temporal and non-temporal data under the same data structure. Although the structure behaves well in terms of space and query processing, it models transaction time and therefore does not handle predictive or retroactive updates either.

Lastly, there are the access structures for spatial data, where the R-tree [Gut84] and its derivatives, (R$^+$-tree [SRF87] and R*-tree [BKSS90]) are well known representatives. Although the structures based on the R-tree index aim primarily at indexing spatial data via N-dimensional minimum bounding rectangles, they can also be used to index one dimensional ranges, e.g., valid time ranges. There are other techniques for indexing spatial data, such as the Z-ordering [Ore86] and DOT [FR91]. We also discuss those in more detail later in Section 5.

The contribution of this paper is to address the problem of indexing VTDBs by using the proposed MAP21 approach. MAP21 makes use of a standard B$^+$-tree (for an introduction refer to [EN94, Chapter 5] among many

[1] We adopted this name as the authors do not give an explicit name to their approach.

other texts), which is available in virtually any existing DBMS. Therefore, our approach does not require the implementation of any novel data structure. Moreover, previous research results regarding issues such as concurrency control of B$^+$-trees [JS93] are therefore inherited. The basic (and realistic) assumption we need in order to offer efficient query processing is that an upper bound for the length of the indexed ranges is kept. Due to the underlying B$^+$-tree structure, MAP21 uses $O(N_r)$ space and requires $O(\log_B N_r)$ I/Os to process an update, where N_r is the number of indexed ranges. Furthermore, we show that we can enhance query processing time by making use of multiple trees, which can be parallelized.

Although some application domains can be well modeled using transaction time only, we believe that supporting valid time, i.e., allowing retroactive and predicte updates is very important, indeed: "... One limitation in supporting transaction time is that ... Errors can sometimes be overridden (if they are in the current state), but they cannot be forgotten ..." [SA86]. MAP21 does allow predictive and retroactive updates, therefore ruling out the limitations imposed by those structures aimed at transaction time or monotonic valid time, e.g., Time-Split B-tree, Snapshot Index, Monotonic B$^+$-tree and Append-Only Tree.

In the next sections we focus on the range indexing problem. To accomplish that, we introduce the MAP21 approach in Section 2, and also discuss algorithms to process some types of queries. In Section 3 we extend MAP21 to make use of multiple, and potentially parallel, trees in a multiple disks single processor architecture. Section 4 discusses how MAP21 can be incorporated into a standard (i.e., non-temporal) DBMS. Experiments comparing MAP21, Time Index and the R*-tree are presented in Section 5. (A brief review of the Time Index and the R*-tree is presented in the Appendix). We conclude in Section 6 summarizing the contributions presented in this paper and offering directions for future research.

2 Indexing Valid Time Ranges – The MAP21 Approach

MAP21 maps the two end points of a valid time range $V^k = [V_s^k, V_e^k]$ into a single value and uses this one as an indexing value for the range. We assume the following:

Definition 1 *α is the maximum number of digits needed to represent any time value.*

Definition 2 *The starting and ending points of an "indexable" valid time range are: (a) Non-negative integer values and (b) Upper-bounded (due to Definition 1). In other words, $V^k = [V_s^k, V_e^k] \Rightarrow 0 \leq V_s^k \leq V_e^k \leq 10^\alpha - 1$.*

Note that Definition 1 is not restrictive, all it states is that the user must know beforehand the domain of the attribute he/she is indexing, in fact that is the case with any attribute when the user defines an attribute of a relation. Assuming Definitions 1 and 2 any range $V^k = [V_s^k, V_e^k]$ can be indexed using a unique value provided by the following mapping function:

$$\Phi(V^k) = \Phi(V_s^k, V_e^k) = V_s^k 10^\alpha + V_e^k \tag{1}$$

Note that, in pratical terms, what the function $\Phi(V^k)$ does is to "left-shift" the valid start time. This implies the following [Nas96]:

Proposition 1 *The above defined function $\Phi(.)$ maps distinct ranges into distinct points, i.e., given $V^k = [V_s^k, V_e^k]$ and $V^l = [V_s^l, V_e^l]$ then $\Phi(V^k) = \Phi(V^l) \Leftrightarrow V_s^k = V_s^l$ and $V_e^k = V_e^l$.*

Proposition 2 *The ordered points in the resulting index represent a lexicographical order of the ranges, i.e., given $V^k = [V_s^k, V_e^k]$, $V^l = [V_s^l, V_e^l]$, and $\Phi(V)$ as defined above, $V_s^k < V_s^l \Rightarrow \Phi(V^k) < \Phi(V^l)$; and if $V_s^k = V_s^l$ then $V_e^k < V_e^l \Rightarrow \Phi(V^k) < \Phi(V^l)$.*

It is straightforward to obtain the original range given its mapping, namely: $V^k = [\Phi_s^{-1}(\Phi(V^k)), \Phi_e^{-1}(\Phi(V^k))]$ where: $\Phi_s^{-1}(\Phi(V^k)) = \frac{\Phi(V^k) - \Phi(V^k)\%10^\alpha}{10^\alpha}$, $\Phi_e^{-1}(\Phi(V^k)) = \Phi(V^k)\%10^\alpha$ and % is the traditional remainder operator. Finally:

Definition 3 *Given a set of (indexed) ranges V^k let $\Delta = \max_k\{V_e^k - V_s^k\}$.*

We assume that the value of Δ, i.e., the maximum length over the indexed valid time ranges, is known. The Δ value of the indexed ranges can be maintained in some sort of dictionary by the DBMS (see Section 4 for a discussion). Furthermore, in temporal databases in general there is no physical deletion of data, as all history is to be retained, hence the maintained Δ is an exact (and dynamic) upper bound. Even though the higher the number of indexed ranges (N_r), the higher the likelihood of having a large value for some valid time length, Δ does not vary as a function of N_r. Note that the approach we propose does not limit the size of the valid time ranges, we simply assume Δ is known. We believe, however, that in many if not most application domains, the value of Δ should not be too large compared to the whole time frame modeled. For instance, if one is modelling a hospital database for 10 years or possibly more, it is hardly expected that patients are hospitalized for more than, say 2 years. Nevertheless, a large Δ may exist and we do propose a framework that handles well such case.

Using MAP21 one can process several types of ranges-based queries, we will present details about three particular types of queries, the intersection query, the inclusion query and the containment query (following the naming in [AT95]). They are illustrated in Figure 1. Given a reference valid time range, these queries return all the indexed ranges that respectively intersect, are included within or contain the given reference. A fourth type of query, the location query, which returns all ranges that contain a given time point (instead of a range) can be simulated using either the intersection or containment query when the reference range degenerates to a point, and thus we do not treat it explicitly. Proposition 1 will be important as it guarantees that each range appears once and only once in the indexing tree. Proposition 2 ensures that the search in the tree can be done in one-pass. And finally, Definition 3 allows us to state exactly where to start and end the leaves searching during query processing time.

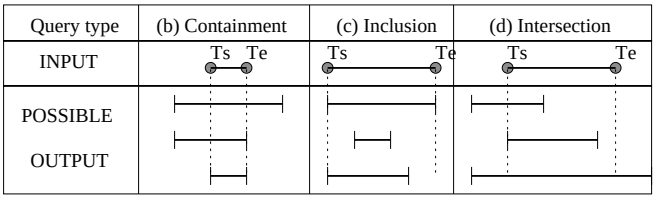


Figure 1: Queries used to benchmark MAP21's performance.

The algorithms we present next show how to collect all pointers needed to actually access the data records, hence we are not describing how to access the data records themselves. As we discuss later in the paper, the physical organization of the data is a task left to the DBMS, which manages the underlying indexing structure. In what follows we use the following notation:

- $\lceil\lceil T \rceil\rceil$ ($\lfloor\lfloor T \rfloor\rfloor$) is the smallest (greatest) indexed value greater (smaller) than or equal to T.
- A MAP21 tree, denoted by M, is a B$^+$-tree indexing point values which represent ranges, and were created using the $\Phi(.)$ function described above.

Processing an Intersection Query – The intersection query accepts as input a time range $T^q = [T_s, T_e]$ and returns the pointers to all records R^k such that $V^k \cap T^q \neq NULL$.

Lemma 1 *Given that the indexed ranges are in lexigraphical order in the leaf nodes of the MAP21 tree, to find all ranges that intersect with* $[T_s, T_e]$ *one needs only to scan the ranges between ranges* $[T_s - \Delta, T_s]$ *and* $[T_e, T_e + \Delta]$.

Proof: Given Definition 3 we know that no range starting before $T_s - \Delta$ can intersect with T^q, otherwise there would be a range larger than Δ, which there cannot be. No range starting after T_e can intersect with T^q either.

Algorithm 1 P = *MAP21-Intersection(*M, Δ, T_s, T_e*)*

1. TRAVERSE M down to the leaf entry indexing $I_v = \lceil\lceil (T_s - \Delta)10^\alpha + T_s \rceil\rceil$
2. DO traverse (in ascending order) each leaf of M indexing I_v

(a) IF $[\Phi_s^{-1}(\Phi(I_v)), \Phi_e^{-1}(\Phi(I_v))] \cap [T_s, T_e] \neq NULL$
THEN $P = P \cup \{$ pointers associated to such entry $\}$

3. UNTIL $I_v = \lfloor \lfloor T_e 10^\alpha + (T_e + \Delta) \rfloor \rfloor$
4. RETURN P

Processing an Inclusion Query – The inclusion query accepts as input a time range $T^q = [T_s, T_e]$ and returns the pointers to all records R^k such that $V^k \subseteq T^q$.

Lemma 2 *Given that the indexed ranges are in lexigraphical order in the leaf nodes of the MAP21 tree, to find all ranges that are contained in $[T_s, T_e]$ one needs only to scan the ranges between ranges $[T_s, T_s]$ and $[T_e, T_e]$.*

Proof: As we are interested in ranges that lie completely within T^q (= $[T_s, T_e]$) we are not interested in any range starting before T_s nor in any range starting after T_e, as those cannot be completely included in T^q.

Algorithm 2 P = *MAP21-Inclusion*(M, T_s, T_e)

1. TRAVERSE M down to the leaf entry indexing $\lceil \lceil T_s 10^\alpha + T_s \rceil \rceil$
2. DO traverse (in ascending order) each leaf entry of M indexing I_v
   (a) IF $\Phi_e^{-1}(\Phi(I_v)) \leq T_e$
   THEN $P = P \cup \{$pointers associated to this leaf entry$\}$
3. UNTIL $I_v = \lfloor \lfloor T_e 10^\alpha + T_e \rfloor \rfloor$.
4. RETURN P

Processing a Containment Query – An inclusion query accepts as input a time range $T^q = [T_s, T_e]$ and returns the pointers to all records R^k such that $V^k \supseteq T^q$. Notice that if $L = T_e - T_s > \Delta$ no indexed range may contain T^q, therefore in what follows we assume $L \leq \Delta$.

Lemma 3 *Given that the indexed ranges are in lexigraphical order in the leaf nodes of the MAP21 tree, to find all ranges that contain $T = [T_s, T_e]$ one need only to scan the ranges between ranges $[T_e - \Delta, T_e]$ and $[T_s, T_s + \Delta]$.*

Proof: Due to Definition 3, any range starting farther than $T_e - \Delta$ cannot contain T. Otherwise, there would be a range with length greater then Δ. It is similarly easy to see that any range starting after T_s or ending before T_e cannot contain T either.

Algorithm 3 P = *MAP21-Containment*(M, Δ, T_s, T_e)

1. IF $T_e - T_s > \Delta$
   THEN RETURN P
2. TRAVERSE M to the leaf entry indexing $\lceil \lceil (T_e - \Delta) 10^\alpha + T_e \rceil \rceil$
3. DO traverse (in ascending order) each leaf entry of M indexing I_v
   (a) IF $T_e \leq \Phi_e^{-1}(\Phi(I^k))$
   THEN $P = P \cup \{$pointers associated to this leaf entry$\}$
4. UNTIL the leaf entry indexing $\lfloor \lfloor T_s 10^\alpha + T_s + \Delta \rfloor \rfloor$
5. RETURN P

Remarks – Notice that the above algorithms may read ranges that do not belong to the answer, but no further overhead is imposed. Eventual pointers to "useless" data records are filtered out of the algorithms' output and thus only data records belonging to the actual response will be accessed. It is also worthwhile mentioning that the same behavior, i.e., reading pointers to useless data is also presented by the R-trees, and the R*-tree in particular (we discuss this issue in Appendix A.2).

2.1 Indexing Open-Ended ($V_e^k = NOW$) Ranges

In this section we treat the cases where the end point of a valid time range is not known. In the domain of temporal databases, it is common to make use of a special temporal variable for the end of a range where the actual valid end time is some time in the future but yet unknown. We denote such temporal variable by *NOW*. Those ranges with $V_e = NOW$ are valid from their valid start time (V_s) until the current time. Other researchers have proposed to use other representation to achieve the same (or closely related) semantics, such as $V_e = +\infty$, $V_e =$ *until-changed*, and others (see [CDI+94] for a good discussion on the topic).

What matters to MAP21 is that open-ended ranges would violate Definitions 2 and 3. We thus index all open-ended ranges under another tree, which we refer to as an Open-Ended Tree (OET). The OET is a standard B+-tree, where the indexed points are the valid start time of the open-ended ranges. As it is known that all ranges have the same valid end time (which is NOW), it need not be indexed. Furthermore once a range is "closed", i.e. its V_e^k is updated to a specific time point, it is fairly simple and not costly to move such range from the OET to the MAP21 tree. Hence, MAP21 is complemented with the OET in order to feasibly index all ranges in $[0, NOW]$. Figure 2 shows an example instance of a MAP21 tree and the OET indexing a set of data adapted from [EWK93], where EIJ stands for the J^{th} version of the I^{th} record and also serves as a unique identifier to the physical location of that record, i.e., its pointer address (for simplicity the internal nodes are omitted).

Data Set

ID	Dept	Vs	Ve
E11	A	0	3
E12	B	4	now
E21	B	0	5
E31	C	0	7
E32	A	8	9
E41	C	2	3
E42	A	8	now

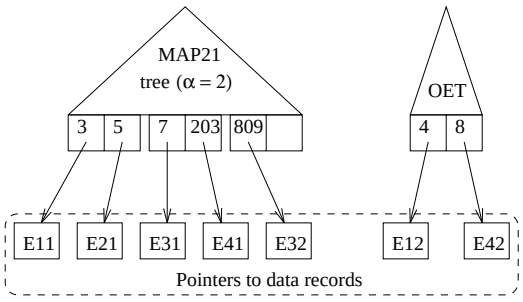


Figure 2: Example of the MAP21 indexing approach (Adapted from [EWK93]).

The earlier algorithms need to be extended to handle such open-ended ranges (and the extra OET) properly. Such extensions are rather simple and thus we do not present them in this paper (nevertheless the interested reader can find them in [Nas96]).

3 Using Multiple, Possibly Parallel, MAP21 Trees

We cannot deny that Δ plays an important role in the search performance. For the intersection and containment queries in particular the bigger the Δ the longer the linear scan on the leaves of the MAP21 tree and, as we pointed out earlier, a large value Δ of may exist in some application domains. In fact, such Δ may be due to as few a one single large indexed range. Fortunately we can improve the search performance by applying a simple idea.

Let us assume a set of ranges with a Δ value equal to $\mathcal{L}$. We may distribute the data set into N_T distinct MAP21 trees as follows. Let $\delta = \lceil \mathcal{L}/N_T \rceil$. We propose to partition the set of indexed ranges in such a way that each MAP21 tree M_j will index ranges with length within $[(j-1)(\delta+1), j(\delta+1)-1]$.

As an example consider the scenario depicted in Figure 3. Figure 3(a) shows a set of ranges with lengths equal to 0, 2, 5 and 10 time units. (Assume that the distance between two dashed vertical lines is equal to one time unit). If all those were to be indexed under one single tree, it would result in $\Delta = 10$. Let us chose $N_T = 3$, implying

$\delta = 4$. According to our discussion above, it would yield: M_1 indexing ranges with length in [0, 4], M_2 indexing ranges with length in [5, 9], M_3 indexing ranges with length in [10, 14]. That is, M_1, M_2 and M_3 will index the sets of ranges depicted in Figures 3(b), (c) and (d) respectively. However, the actual Δ value of each set may be smaller than the one yielded by the partitioning criteria. Indeed, by simple inspection, we can see that the actual Δ values for each those trees are 2, 5 and 10. Recall that, as we assume that the current Δ value for any MAP21 is to be maintained by the DBMS, the performance in any single tree will not be any worse than it actually need be. The range lengths determined by the above scheme serve as a guideline for the partitioning process only.

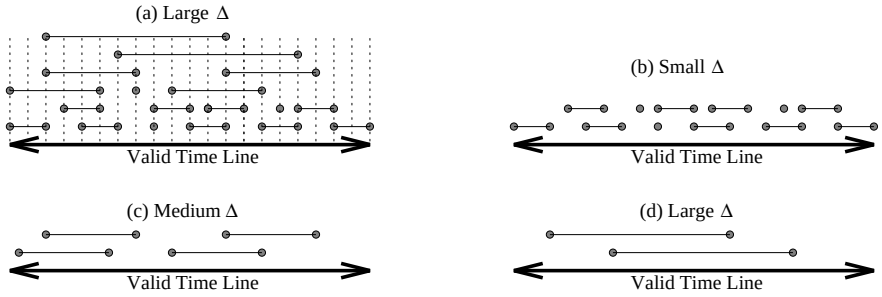


Figure 3: Illustration of the data partitioning based on the ranges length.

For the particular case of the intersection query, depending on which tree the algorithm is processing, a different value of (local) Δ should be used. Each tree M_j will be indexing ranges with lengths upper-bounded by $j(\delta + 1) - 1$. The actual local Δ for each tree, as discussed above, will thus be smaller than the original Δ value. Note that it is assumed that such local Δs are also maintained for each tree, just as the original Δ was maintained for the single tree. The only difference is that we now have several Δ values (i.e., $\delta_1, ..., \delta_{N_T}$) for each of the N_T trees. It is true that the last MAP21 tree, M_{N_T}, will have the same Δ as the original set, but that tree should also have fewer indexing points than the original set, as smaller ranges are now hosted under other trees. The algorithm for processing an intersection query using multiple MAP21 trees, built as described above, is as follows:

Algorithm 4 P = *N-MAP21-Intersection*($M_1, M_2, ..., M_{N_T}, \delta_1, \delta_2, ..., \delta_{N_T}, T_s, T_e$)

1. FOR j = 1, 2, ..., N_T
   (a) $P = P \cup$ MAP21-Intersection(M_j, δ_j, T_s, T_e)
2. RETURN P

On the other hand, in the case of processing an inclusion query the algorithm for multiple MAP21 trees may not need to traverse all trees. If L is the length of the range in the input query, it is clear that any tree indexing ranges with length strictly greater than L need not be traversed, as none of its ranges can possibly be included in a range of length L. Such observations lead to the following algorithm:

Algorithm 5 P = *N-MAP21-Inclusion*($M_1, M_2, ..., M_{N_T}, \delta_1, \delta_2, ..., \delta_{N_T}, T_s, T_e$)

1. FOR j = 1, 2, ..., $\lceil (T_e - T_s)/\delta \rceil$
   (a) $P = P \cup$ MAP21-Inclusion(M_j, T_s, T_e)
2. RETURN P

For the processing of containment queries, similarly to the rationale used for the inclusion query, MAP21 trees indexing ranges smaller than the queried range, need not be traversed. No range in tree M_j (which have their length upper bounded by δ_j) contain a range with length greater than δ_j. This yields the following algorithm:

Algorithm 6 P = *N-MAP21-Containment*($M_1, M_2, ..., M_{N_T}, \delta_1, \delta_2, ..., \delta_{N_T}, T_s, T_e$)

1. FOR j = $\lceil (T_e - T_s)/\delta \rceil$, ..., $N_T - 1$, N_T

(a) $P = P \cup$ MAP21-Containment$(M_j, \delta_j, T_s, T_e)$

2. RETURN P

Multiple trees can also be used to handle the case where a non-uniform distribution of the range length is encountered. When using a single MAP21 tree, few large ranges would slow the search performance, by forcing a longer linear scan in the leaf nodes (in the case of the intersection and containment queries). Using multiple MAP21 trees one may distribute the ranges in such a way that each tree ends up having a more uniform distribution of the ranges length. Likewise, even though we do not touch the issue, one may extend this idea into a dynamic approach.

Finally, parallel updates and searches can be feasibly implemented in a straigthforward manner, as the N_T trees are totally independent.

4 Incorporating MAP21 into a DBMS

We now discuss the implications of using MAP21 within the framework of a standard relational DBMS. The purpose of this section is to demonstrate the feasibility of MAP21's implementation. Examples should be seen as simple illustrations of how easy it is to use the MAP21 approach. Further research is needed to examine the best approach to be used for its implementation.

Figure 4 illustrates how we envision MAP21 would be used with a traditional DBMS which provides B^+-trees as indexing structures. The conventional database and non-temporal indices are supplemented with the temporal database (i.e., temporal relations) and temporal indices. On top of the DBMS is placed a (software) temporal database interface. All DBMS queries (temporal and non-temporal) are sent through this interface. Non-temporal queries are passed directly to the DBMS without modification. Temporal queries are modified to access (or create or update) the appropriate temporal relations and indices. To the DBMS all relations and indices appear as standard ones. Determining whether a query is temporal or not cannot be made based upon the query format only. Indeed TSQL2 [Sno95] can be used for both temporal and non-temporal data, as it is upward compatible with SQL-92 [SKS95]. The distinction must be made upon the relation type, i.e., whether it is a temporal or non-temporal relation. The temporal interface is also required to collect query results and format those correctly for the user, e.g., coalescing data records. This approach is very similar to that used in TimeDB [Ste96].

We have two major objectives for our implementation. First, the underlying DBMS cannot be modified in any way. Second, it should utilize the indexing capabilities of the DBMS (where possible) to avoid having the temporal layer actually create and maintain the indices. This avoids any problems which might arise in terms of pointer manipulation and maintenance if we implemented the indexing in the temporal layer itself. Both of these objectives will lead to a relatively "small" temporal interface layer. That is, most of the work required will be performed by the DBMS itself.

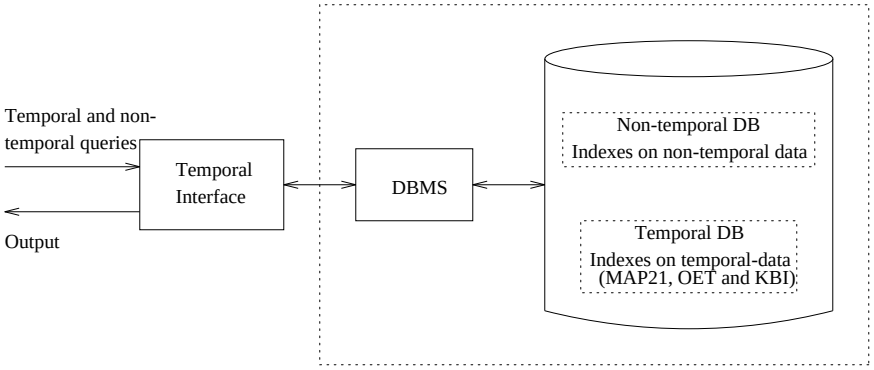


Figure 4: Incorporating a temporal interface to a standard DBMS.

4.1 Implementation of Indexes

To ensure that the DBMS can automatically maintain the indices we need to be able to issue a "CREATE TABLE" and "CREATE INDEX" for each relation/index pair required. This means that each MAP21 tree and each OET will be created as an index into a table. Given a set of temporal data, then, we divide this set into disjoint parts based on the MAP21 divisions. For example, when evaluating Figure 3 data, we see that (based upon Δ) data is divided into three disjoint sets. For this case we would have four table/index pairs: three for the MAP21 trees and one for the OET. What we are proposing, then, is that instead of creating one temporal table with multiple MAP21 indices, we will actually have multiple temporal tables each with its own MAP21 index. This allows us to meet our second objective above. The indices will be maintained by the relational DBMS itself. The temporal layer will have the knowledge that logically all of the MAP21/table and OET/table pairs fit together to form one user temporal relation. The underlying DBMS is not aware of this relationship. Figure 5 shows this proposed technique. The set of temporal data is partitioned based on the Δ value to be maintained in each MAP21 tree. The temporal table reflecting the temporal data is also partitioned. If there are n partitions reflecting the desire for n MAP21 indexes the data is actually partitioned into $n + 1$ disjoint subsets. The additional subset reflects the data with open ended ranges.

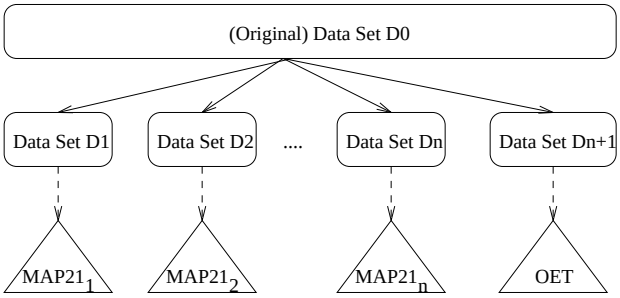


Figure 5: Proposed approach to implementing MAP21

As is done with TimeDB [Ste96]. we assume that the data dictionary information concerning the temporal data (including the partitioning and the Δ values) is maintained as regular DBMS tables. The temporal interface uses these tables to determine the partitioning used, to maintain Δ values, and to determine how to divide the temporal queries into traditional SQL queries.

MAP21 is a temporal index. If the user is expected to pose queries against non-temporal attributes often, another index, such as a B^+-tree or a hash table, should be built on those attributes. MAP21 does not replace a non-temporal index, but, along with the OET, complements it.

Consider, as an illustration only, Figure 6 where KBI stands for the non-temporal key-based index (which is considered a B^+-tree, but could be any other access structure as well). The set of pointers in MAP21 and the OET are obviously disjoint. On the other hand the pointers in the KBI point to the current version of each non-temporal key, which may or may not be open-ended. Therefore, each data record version is pointed to by a pointer from a MAP21 tree or (exclusively) the OET. The most current version of each data record is also pointed to by a pointer from the KBI. A query based solely on a non-temporal data would traverse the KBI only and process only the current versions of the data records. A query based on temporal and non-temporal data would traverse the MAP21 tree and/or OET and the KBI. A query based only on temporal data would traverse the MAP21 tree and/or the OET. Updating a record, i.e., versioning, is not complex either as all trees can be updated efficiently. Care must be taken when creating indices on user data. Because of the temporal time values, attributes which would represent key values in a snapshot database are no longer keys. This implies that if primary indexes on the snapshot keys are built they may be quite large and could contain many duplicates. Further research is required to determine the how to effectively integrate user indices with temporal ones.

Query processing algorithms, such as those presented in the previous sections, are implemented in the temporal interface. Given the temporal query language (such as TSQL2) such temporal layer will convert the query into a series of SQL statements directed to the DBMS. A major responsibility of the temporal interface will be in collecting the results from the various MAP21/OET trees (as is shown as set union operations in the the earlier algorithms). The temporal interface can either buffer the results, store them in temporary files, or actually create

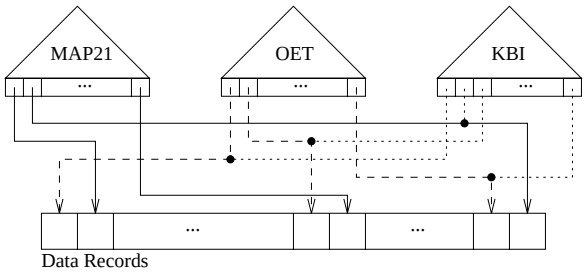


Figure 6: Combining all indexing trees.

temporal relations to store the results. The best option will be determined based on many factors such as the size of the result and the number of tables. Further study will explore the "best" approach to be used.

An interesting performance issue related to our proposed implementation is how data is clustered. In a traditional table/index approach, the data is clustered based on the primary key by which it is accessed. In our approach the data may also be clustered. How best to cluster should be based on how to efficiently provide access to the data. Since no temporal benchmark is available, how the data should be clustered is not really known. In normal DBMS environments (such as Oracle [Ora92]) the database user has the ability to cluster data as he/she desires using a "CREATE CLUSTER" command. This should still be allowed in a temporal environment which implies that the temporal interface layer must be able to recognize such statements and cluster the data accordingly. How to do this is an open research issue and requires much further examination. Nonetheless, in the following section we briefly explore how the temporal data can be easily clustered on the user key, while at the same time indexing the temporal data.

4.2 Example

In this section we briefly examine one approach to implement the simple example shown in Figure 2. In this case (as is shown) the data is partitioned into two parts: one for the MAP21 tree and one for the OET. Assuming that the user also wishes to frequently access via the user key (which is the Employee ID, denoted by `empid`) we cluster the temporal data accordingly. Below we show sample SQL code (using the format of SQL*Plus from Oracle [Ora92]) which might be generated by the temporal interface:

```
CREATE CLUSTER empid_cl (id char(2));
CREATE INDEX empid_cl_ind ON CLUSTER empid_cl;
CREATE TABLE employee_closed(
  empid char(2) not null,
  dept char(1),
  mapped_vsve number(2)
  ) CLUSTER empid_cl (empid);
CREATE INDEX employee_map21 ON employee_closed (mapped_vsve);
CREATE TABLE employee_open(
  empid char(2) not null,
  dept char(1),
  vs number(1)
  ) CLUSTER empid_cl (empid);
CREATE INDEX employee_oet ON employee_open (vs);
```

Notice that we did not create a primary index on the user key. However the cluster and its index serve the purpose of such an index. Keep in mind, though, that this contains duplicates. However, because of the clustering, all duplicate records with the same user key values will be stored close together. Using Oracle as an example DBMS this will create two tables, `employee_closed` and `employee_open`, stored together and clustered by empid. The data with open ended ranges is placed in the `employee_open` table and indexed by the OET,

`employee_oet`. Three B*-trees indices will be created[2]. The clustering index based on the key of `empid` will allow efficient access of data via `empid`. The MAP21 index, `employee_closed_map21`, indexes the table `employee_closed` by `vs` and `ve` combined.

5 Performance Analysis

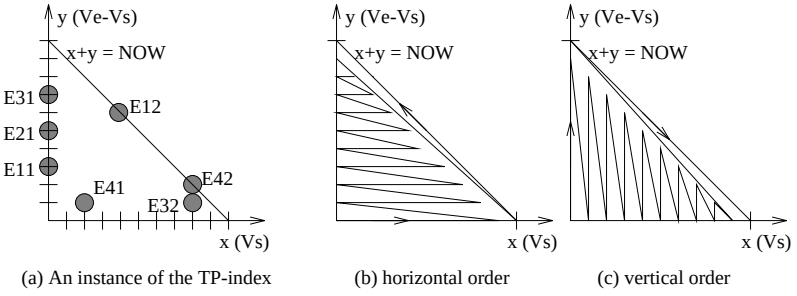
In this section we compare the performance yielded by MAP21 against those by Time Index and by R*-tree We examine the space (i.e., number of disk blocks to store the structure) and the time (i.e., number of disk blocks read) needed to process the queries discussed earlier. MAP21 and the R*-tree were actually implemented. The Time Index was partially implemented on top of a B+-tree, with its incremental buckets being simulated (we discuss such simulation in Appendix A.1).

We chose the Time Index because it is a conceptually simple structure, which despite its inefficient use of storage may yield good average query processing time. It has also been used as a reference structure in other published research [SOL94, K+94, AT95, G+96]. Furthermore, the TP-index and the Interval B-tree have rather unique internal data structures, and we believe that the use of simple data structures, such as the B+-trees, is a very desirable important feature of temporal indices, if they aim at being of practical use.

The B+-tree based TP Index [G+96], recently published, deserves a deeper discussion. As the TP-index [SOL94], it maps a range $V = [V_s, V_e]$ into a two-dimensional point $(V_s, V_e - V_s)$. Instead of mapping a query on a two-dimensional search problem, the authors propose to use an ordering among the spatial points (which is basically a mapping, although very different from MAP21's), and this ordering is then reflected into those points in the leaf nodes of a B+-tree. Figure 7(a) shows the B+-tree based TP-index indexing the same data set of Figure 2 and Figures 7(b) and (c) show two possible mappings, respectively denoted as horizontal and vertical. For example, the data items in Figure 7(a) would have the order {E41, E32, E11, E21, E31, E42, E12 } under the horizontal ordering and the order {E11, E21, E31, E41, E32, E12, E42} under the vertical ordering, which are obviously quite distinct.

Data Set

ID	Vs	Ve
E11	0	3
E12	4	now
E21	0	5
E31	0	7
E32	8	9
E41	2	3
E42	8	now



(a) An instance of the TP-index (b) horizontal order (c) vertical order

Figure 7: Illustrating the B+-tree based TP-index

.

The major problem in this approach is that "... different indexes (constructed using different ordering relations) may be used to support the various types of queries" [G+96]. That is, for some queries the horizontal mapping is appropriate, but it may may not be for others. This does not mean an adequate mapping cannot be always found, but it does mean that one may need to maintain (concurrently) several indices, each representing a different mapping, which may not be very desirable. Furthermore, "... not all temporal queries may be mapped to a simple range query, it may be necessary for the spatial search to be decomposed into a number of interval queries" [G+96]. For all those reasons we believe MAP21 is a superior temporal access structure. The authors compared, only by analytical means though, their approach to Time Index. As with MAP21, they found their technique to require less space and the number of pages accessed was somewhat smaller. However, their comparison was based on one query: "Retrieve all persons who arrived during the interval $[T_s, T_e]$" and one single ordering (V-ordering). Their comparison did not examine different orderings or more general queries. In addition, the paper does not present

[2]Oracle uses a variation of the B+-tree called a B*-tree. This probably creates a more efficient way to implement MAP21 as the fill factor of each node is greater. Nevertheless, using B*-trees instead of B+-trees is irrelevant as far as MAP21's rationale is concerned.

a methodology to chose the underlying order based on the queries. Thus, to avoid taking the risk of not using the best ordering for the queries we investigated, we elected not to compare their approach to ours. Finally it is not clear how (or whether it would be possible) to parallelize access to the B^+-tree based TP-index.

Due to the fact that we do not impose any requirement on the order the ranges are input, nor is the MAP21 specialized for such requirement, we do not compare it to those indexing structures based on transaction time (Time Split B-tree) or monotonically growing valid (start) time (Monotonic B^+-tree, Append-Only tree and Snapshot Index).

The R-tree [Gut84] is probably the most well known data structure for spatial indexing, and although we are dealing with ranges in the temporal domain, they can be understood as segments in the one dimensional space. Such segments can be seen as degenerate minimum bounding rectangles (MBRs), and indexed in a one-dimensional R-tree. However, more recent work upon R-trees have designed more efficient R-tree "derivatives". Such as the R^+-tree [SRF87] and the R^*-tree [BKSS90]. (We review them briefly in Appendix A.2). It has been well argued in the literature that R^*-trees perform better than R-trees and R^+-trees [BKSS90, TP95, KSCL95]. We thus chose to use the R^*-tree

There are other techniques for indexing spatial data. Among those we can cite those based on mapping N-dimensional objects to points in the N-dimensional space, such as the Z-ordering [Ore86] and the DOT approach [FR91]. The Z-ordering transforms a spatial object, not necessarily a rectangle, to one or more, possibly disjoint, segments in the one dimensional space. Similarly to the R^+-tree, this results in the possible replication of pointers to objects, which is a feature we would rather avoid. A good feature of the Z-ordering is that it can manage, besides MBRs, other "non-regular" structures. Unfortunately though, such a feature is of no use in our application domain. Unlike the MAP21 mapping, which preserves the lexicographical ordering among the ranges, DOT preserves the distance among them by using a fractal curve for the range mapping. To index one dimensional ranges, DOT transform them into two-dimensional points (step called 1st-transformation) and then, via fractal functions, maps such points again into one dimensional space (called 2nd transformation). In [FR91] DOT is shown to be up to 50% faster than the linear R-tree [Gut84]. In this paper we compare MAP21 to the R^*-tree, which has been shown to be up to 400% faster than the linear R-tree [BKSS90]. We thus conjuncture that if we verify MAP21 to be at least comparable to the R^*-tree, then it is quite safe to conclude that MAP21 outperforms the R-tree as well, and this is accomplished by a much simpler approach than DOT's. Finally, it seems that the DOT technique cannot be easily paralellized, unlike MAP21. The same observation holds true for the Z-ordering approach.

5.1 Assumptions

In most of the experiments conducted next we assume that there are no open-ended valid time ranges, even though both the MAP21 and the R^*-tree can support them. The reason for such an assumption is that indexing open-ended ranges would not help to compare the performance of the investigated structures. This is discussed in more detail later (in Section 5.4.5). Nevertheless, for the sake of completeness, we do investigate MAP21's performance with respect to the presence of open-ended ranges (even though not in a comparative way).

In order to conduct our experiments to evaluate MAP21's performance we adopted the parameters shown in Table 1. The reasoning behind the choices for the ranges length is that we believe that TDBs are to keep track of records which are updated often, therefore large values to those ranges would not reflect that. In Section 5.2 we present the size of the structures, while in Section 5.3 we investigate the query processing time (by means of the number of I/Os needed to process several types of queries). In Section 5.4 we vary some of the parameters in Table 1, e.g., distribution of the ranges length (where we employ an exponential distribution instead of a uniform).

The range and query lengths were generated using a uniform distribution with the maximum generated range lengths being those shown in Table 1 (the mean being, naturally, half of those). We first generate the range length, and then a starting point within the range $[0, V_e^{max}]$ is generated (also using a uniform distribution) until the generated range fits in $[0, V_e^{max}]$. A set of 250 queries was also generated using the same scheme and we report the average number of nodes accessed, i.e., the number of I/Os yielded.

We ran tests using a single and two MAP21 trees, for the last one a parallel implementation was simulated by neglecting any CPU overhead and using the maximum number of I/Os implied by any of the trees as the overall number of I/Os needed to process a query. We denote those by MAP21, 2-MAP21 and 2P-MAP21 respectively. Even though a larger number of trees could be used we will show that as few as two parallel MAP21 trees suffices to deliver the best overall performance.

Table 1: Parameters used in the experiments conducted.

Parameter	Value	Comments/Used for
Disk block size	1024 bytes	Equal to a node in the trees
V_e^{max}	2000 time units	Maximum possible time value ($\neq NOW$)
VSL, VSQ	Uniform(0, 1% of V_e^{max})	Very Short Lifespan and Queries
SL, SQ	Uniform(0, 5% of V_e^{max})	Short Lifespan and Queries
ML, MQ	Uniform(0, 10% of V_e^{max})	Medium Lifespan and Queries
LL, LQ	Uniform(0, 20% of V_e^{max})	Long Lifespan and Queries
VLL, VLQ	Uniform(0, 50% of V_e^{max})	Very Long Lifespan and Queries
N_r	10000	Number of ranges
Δ	VSL, SL, ML, LL, VLL	Upper bound for range lengths

Note: We use the term lifespan and valid time range length interchangeably

5.2 Index Size

We first compared the size (in number of nodes, i.e., disk blocks being used) of the indices. MAP21 indexes a range via the mapping function $\Phi(.)$ defined in Equation 1. The size (in bytes) of the data type indexed is twice as big as the size of each end point of the input range and this was taken into account. When multiple MAP21 trees are employed, the size reported was the combined total size of all trees. We do not include disk blocks used to hold the actual data records, but only the indexing structures themselves.

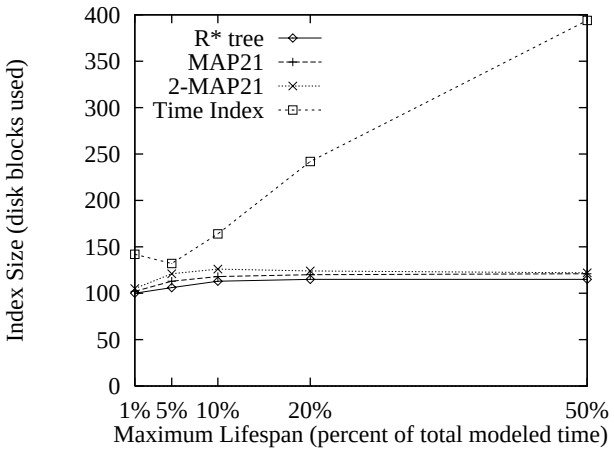


Figure 8: Size of the indexing trees (one tree node = one disk block).

Figure 8 shows the results obtained. The MAP21 tree, i.e., the B^+-tree underneath it, consumed less than 10% more storage than the R^*-tree. Note that as the average lifespan increases, the amount of replication in Time Index's incremental buckets also increases, and hence the overall size of that index grows rather sharply.

Using multiple trees implied very small, if any, storage overhead, i.e., the sum of the multiple tree sizes is approximately the same size as the single tree. We observed, though, that the size of the trees depends on the ordering in the input data. In fact, due to the "plain" B^+-tree framework the worst case happens when the ranges are input in lexicographical order. In that case, once a node splits it will remain nearly half empty, because no indexing value will be input in any other leaf but the rightmost one.

5.3 Query Processing Time

To investigate the query processing time, we first ran experiments by fixing the lifespan length as very short, medium and very large, and queried each of those data sets for intersections, inclusions and containments using all

five sizes of queries. The cases where the lifespans were short and long are not shown for brevity, but did follow the trend suggested by the results reported.

5.3.1 Intersection Query

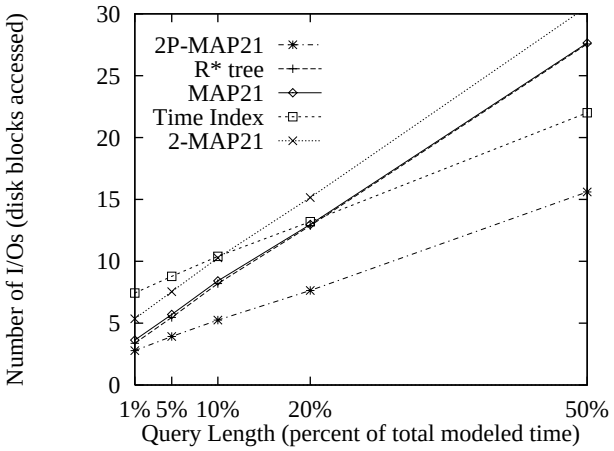


Figure 9: Intersection query performance when indexing very short lifespans

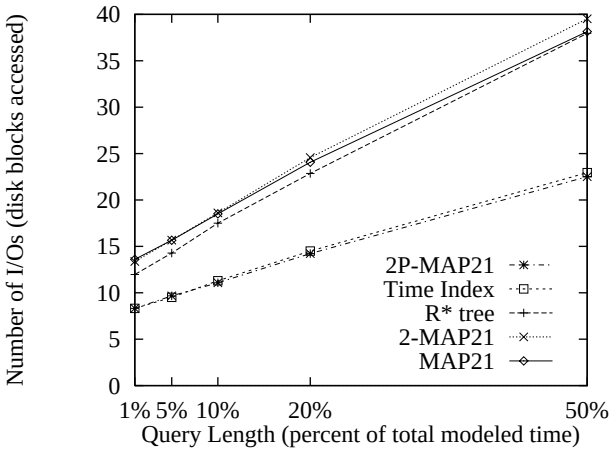


Figure 10: Intersection query performance when indexing medium lifespans

Results for the intersection query are presented in Figures 9, 10 and 11. As the number of leaves scanned in the Time Index depends chiefly on the query length, its performance degrades slower (as the lifespan increases) than the other structures, indeed it yields the least number of I/Os for very large lifespan. It is important to remember though, that when indexing very long lifespans Time Index is nearly 300% larger than the R*-tree and MAP21 (using one or two trees). When indexing very short lifespans two non-parallel MAP21 trees yield worse performance than using a single one. This happens because the "local" Δ of those trees are relatively close to each other and thus there is little gain in the local Δs. On the other hand, when indexing very large lifespans, the local Δs are farther apart, and thus the gain in using smaller local Δs can be better appreciated. Two parallel MAP21 trees are always faster than the R*-tree and much smaller than the Time Index.

Point and Extremely Short Queries For the sake of completeness, we also investigate the performance of all access structures with respect to point and (what we call) extremely short queries. For simplicity we have set the lifespan fixed at the medium size and varied the query size from point queries (i.e., zero length) to queries as large

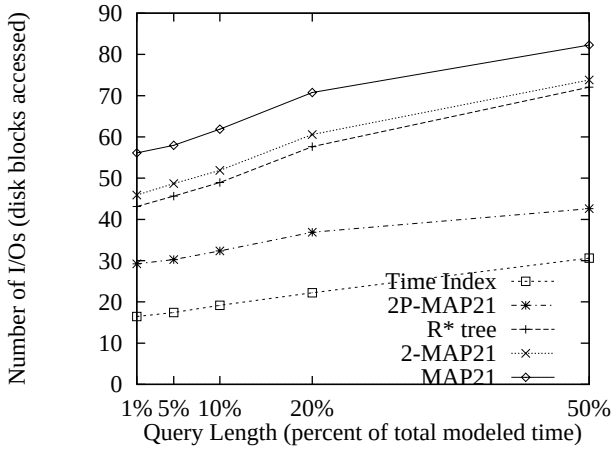


Figure 11: Intersection query performance when indexing very long lifespans

as only 0.5% V_e^{max}. The results are shown in Figure 12. We can observe that the results are similar to those in Figure 10, when the query length is very short. Again, we can verify that using multiple parallel trees is the best overall approach, i.e., while Time Index presents virtually the same query processing time, MAP21 is always smaller.

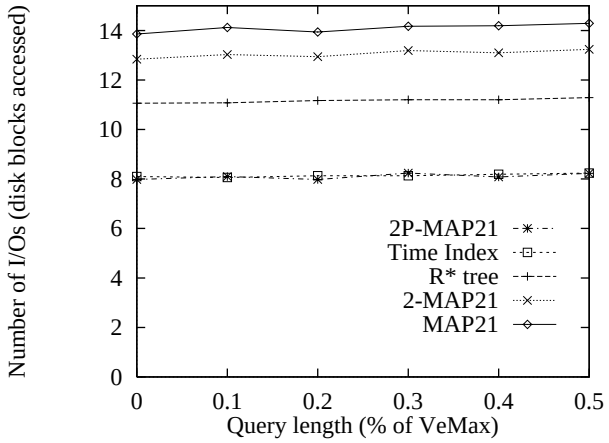


Figure 12: Intersection query performance when querying points and extremely short ranges (indexing medium lifespans).

5.3.2 Inclusion Query

Figures 13, 14 and 15 show the experiments carried out using the inclusion query. It is easy to see that the R*-trees are not very efficient to process inclusion queries. As discussed in Appendix A.2, this is due to the fact that the R*-tree is traversed in the very same way when processing an intersection query or an inclusion query, and, in a sense, inclusion queries are much less demanding than intersection queries. Therefore, even the single tree based MAP21 is better than the R*-tree in all cases. In fact, as the indexed lifespans increase, so does the gap between MAP21's and R*-tree's performance.

In general, as the query length increases using multiple MAP21 trees is worthwhile (i.e., better than using a single tree) only if they are implemented in parallel. This can be explained as follows. If the query length is large enough, most, if not all (due to the uniform distribution), indexed ranges in the "initial" trees (i.e., those indexing smaller ranges) will be part of the answer, hence all such leaf nodes will be scanned. In a single tree the same ranges would have a more clustered ordering which would imply only one smaller linear scan. See, for instance,

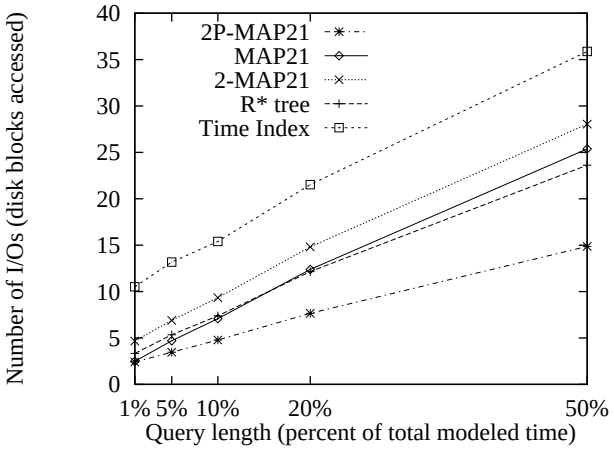


Figure 13: Inclusion query performance when indexing very short lifespans

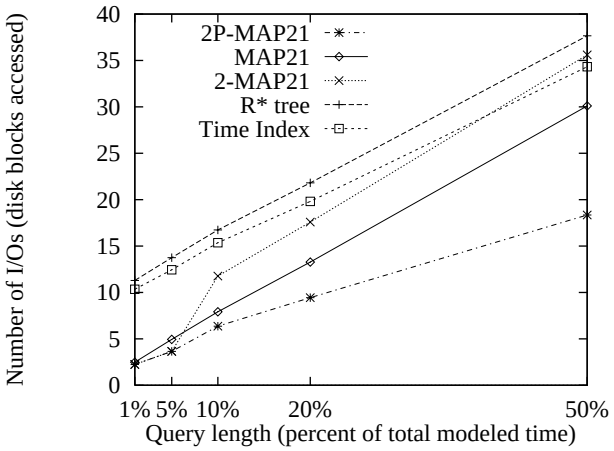


Figure 14: Inclusion query performance when indexing medium lifespans

the case when very long lifespans are indexed (Figure 15). When two MAP21 trees are used, non-parallel MAP21 trees are viable only for long sized (or smaller) queries. We conclude, as a rule of thumb, that using non-parallel MAP21 trees is worthwhile only when the query size is smaller then the average indexed lifespans, on the other hand operating MAP21 in parallel is always the best option.

5.3.3 Containment Query

Recall that the containment query is most meaningful when there are indexed ranges with a lifespan larger than the query length. This is particularly important in the case where multiple trees are used, because if a tree indexes only lifespans smaller than the query then it need not be traversed. Nevertheless, neither the Time Index nor the R*-tree seem to be able to take advantage of such fact. In order to not bias the conclusion towards MAP21 we run tests using very long lifespans only, hence we are assured (due to the uniform distribution of the ranges length) that it is very likely that some indexed ranges contain the queried range.

The results are shown in Figure 16. It is quite natural to think that the larger the query the less ranges will be selected in the answer. Hence it would be desirable that the indexing structure would also require less I/Os as the query size increases. That is exactly what MAP21 and the R*-tree do, with MAP21 requiring on average 30% more I/Os. The Time Index is far more efficient than MAP21 and the R*-tree for small queries. However, instead of decreasing the number of I/Os needed as the query size increases, the number of I/Os actually increases proportionally to it. This is due, as discussed before for the intersection query, to the fact that the length of the linear scan in the Time Index tree is dictated mainly by the query size, regardless of the answer size. Notice that

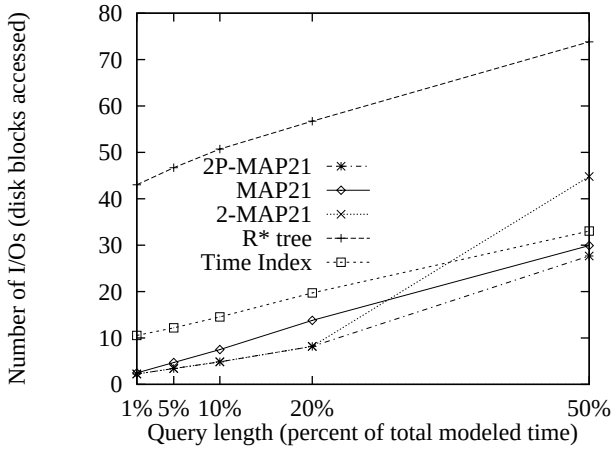


Figure 15: Inclusion query performance when indexing very long lifespans

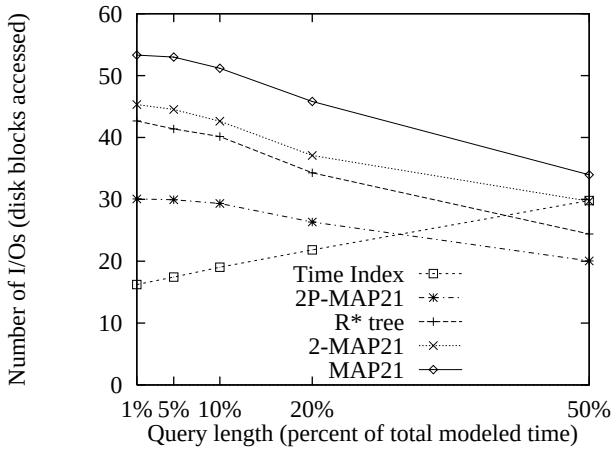


Figure 16: Containment query performance when indexing very long lifespans

an average query size larger than a very long query, would deem Time Index to be worse than both the R*-tree and MAP21.

By analysing Figure 16 we verified that multiple MAP21 trees (even if not implemented in parallel) were always faster than the single tree counterpart. This happens because not all trees are traversed for all queries (as was the case for the inclusion query). In fact for larger query sizes, the initial trees are not traversed at all, because they cannot contain a large range. Hence the sharper drop in the MAP21's performance as the query gets larger. The non-parallel MAP21 are not as efficient as the Time Index for smaller queries, although they are for larger queries. Nonetheless, the parallel MAP21 trees are always better than the R*-trees and better than the Time Index for lifespans larger than what we defined as medium. One should not forget that the high efficiency of Time Index is obtained at the expense of a large storage, where as MAP21, using single or multiple trees is always much smaller than the Time Index.

5.4 Varying Other Parameters

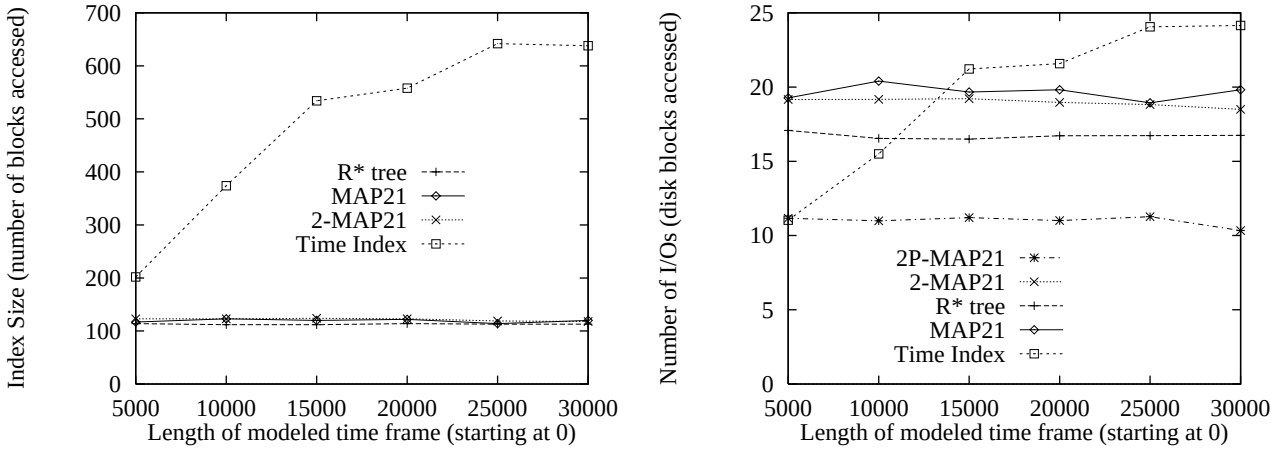
In the results just discussed we have used the parameters in Table 1, i.e., we have basically set some constants and varied the length of the indexed lifespan and queries.

For the sake of completeness, in this section we vary the disk block, V_e^{max}, N_r and the distribution of the lifespan ranges, one at a time. We fixed the maximum lifespan and maximum query length in the medium range, i.e., both are set to 10% of the current V_e^{max}. We conclude this section by investigating MAP21's performance with respect to the ratio of open-ended ranges.

Finally, due to limited space we show only the experiments we ran using the intersection query as, in a sense, this is the most demanding one. The qualitative behavior of the other types of queries was not significantly different.

5.4.1 Varying the Modeled Time Frame Size – V_e^{max}

In our previous runs we used a fixed $V_e^{max} = 2,000$ time units. As Time Index indexes both end points of the ranges, the number of indexing points which could be actually indexed was upper bounded by V_e^{max}, i.e., 2,000. On the other hand, MAP21 maps each distinct range to a point and thus it might index (using the same data set) a much higher number of distinct points. The same is valid for the data under the R*-tree. Therefore we experimented using different values for V_e^{max}, from 5,000 up to 30,000 in increments of 5,000. Note that the maximum lifespan size and the maximum query length are proportional (10%) to the current V_e^{max}, therefore these values increase with V_e^{max}.



(a) Index sizes.

(b) Query performance.

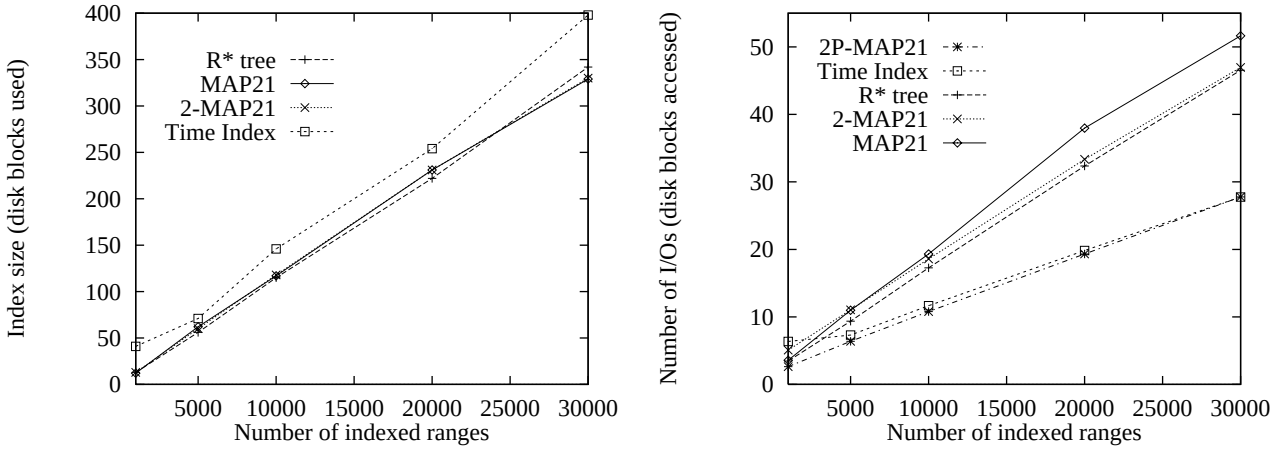
Figure 17: Varying V_e^{max}.

In terms of size (see Figure 17(a)), neither MAP21 nor the R*-tree are affected by the increase in V_e^{max}, basically because the size of these structures is a function of N_r much more than of V_e^{max}. Time Index, however, is affected by the variance in V_e^{max}. The greater it is, the more distinct indexed points are generated and thus the more populated the index. Consequently the more incremental buckets are also needed. Figure 17(b) shows the results in terms of I/Os for query processing. As we expected, the Time Index is the best alternative for small values of V_e^{max}. As it increases, the structure becomes larger (as discussed above), and thus the queries are more expensive to process. As before, using two parallel MAP21 trees offers the best overall performance.

5.4.2 Varying the Number of Indexed Ranges – N_r

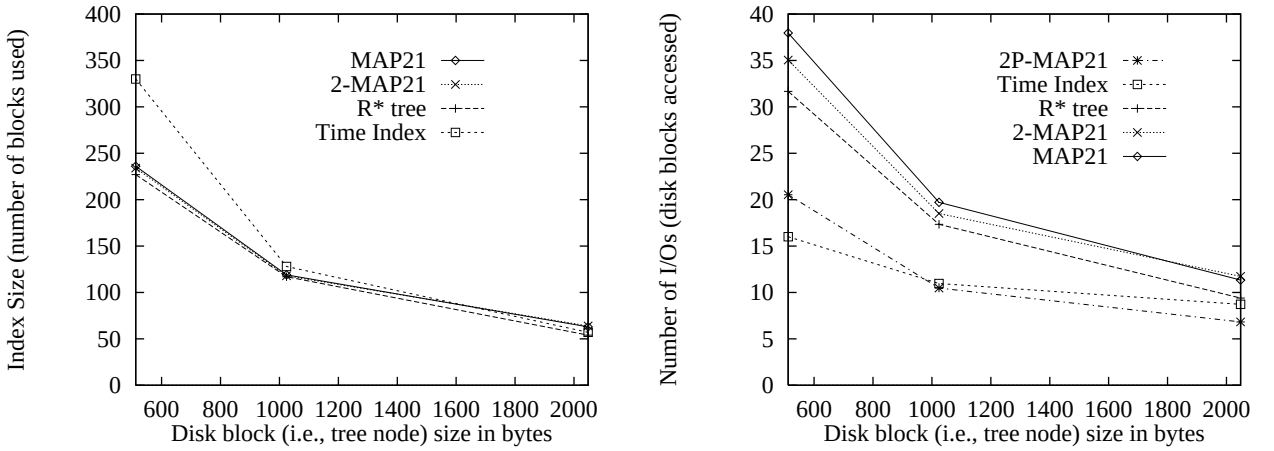
We also run experiments varying N_r = 1,000; 5,000; 10,000; 20,000 and 30,000. The size of the indexing structures are shown in Figure 18(a), while the query performance is shown in Figure 18(b).

None of the structures' size seemed to be particularly sensitive to the increase in N_r, all increased practically linearly with N_r. In terms of query performance however, we have some more interesting results. For rather small values of N_r Time Index delivers the worst performance whereas it previously presented the best performance. This is so because that for small values of N_r the length of the linear scan on the Time Index search is chiefly dictated by the query length, whereas for both MAP21 and R*-tree it is dictated by the number of indexed ranges. After $N_r = 5,000$, both MAP21's and R*-tree's performance decrease faster, due to the fact that both will have more leaf nodes to search as N_r increases. 2P-MAP21, on the other hand delivers performance comparable to Time Index, while consuming less storage space.



(a) Index sizes. (b) Query performance.

Figure 18: Varying N_r.



(a) Index sizes. (b) Query performance.

Figure 19: Varying page size.

5.4.3 Varying the Page Size

We finally experimented with a varying disk block size. In addition to the size of 1024 bytes used earlier, we also use 512, and 2048 bytes. Figure 19(a) shows how the size of the indexing structures vary with the block size. As expected, the bigger the block size the smaller the number of blocks needed for all structures. However, Figure 19(b) also serves to confirm an interesting result. As discussed in Appendix A.1 that every leaf node in the Time Index maintains an SC bucket in the leading entry with pointer to all records valid at that indexed point in time. The bigger the leaves, the more values it holds and thus a smaller number of SC buckets is needed. Basically, a long lived record will be replicated in less SC buckets. For this reason, we observe that the drop in the number of nodes used for the Time Index is much sharper than for the other structures[3]. Notice that for block sizes greater than 1024 Time Index's curve shape follows the other structures. This happens because we are simulating its incremental buckets. As the simulation assumes a block fully used, for larger block sizes one disk block is probably enough to hold all incremental buckets, and thus the Time Index "degenerates" to a B^+-tree with one additional disk block.

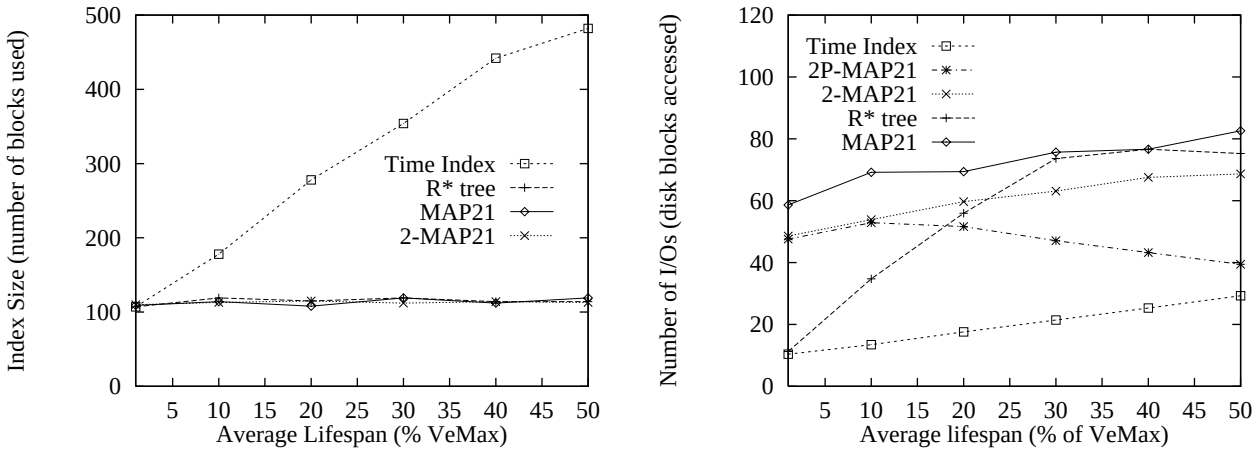
The results regarding query processing are shown in Figure 19(b). For both MAP21 and the R^*-tree larger nodes allow more data to be indexed in a given node, hence the proportional reduction on the searching effort.

[3]This also serves to confirm that our simulation of the Time Index incremental buckets is based upon an accurate model.

The Time Index though, does not benefit as much in terms of query processing time. One reason is that there is a minimum number of nodes it needs to read, namely to traverse down the tree, one leaf node and the associated incremental buckets, which are hosted in at the very least one disk block. We believe that the small slope in Time Index's curve is due to the fact that it is approaching such point.

5.4.4 Using an Exponential Distribution for the Lifespan Lengths

In all previous experiments in this section we have used a fixed value for the Δ, namely 10% of V_e^{max}, and used an underlying uniform distribution. This let us determine a good upper bound for Δ. In what follows we use an exponential distribution for the lifespans, where the average of the generated values is pV_e^{max}. The values generated by such a distribution can lie, in theory in the interval $[0, +\infty]$ but for simplicity we force them to lie in $[0, V_e^{max}]$. Therefore, unlike when the uniform distribution was used, we cannot infer a good upper bound for the ranges length, i.e., a tight value for Δ. We thus aim to investigate how much MAP21 is dependent on the choice of Δ. We have again fixed $N_r = 10,000$, $V_e^{max} = 2,000$, and the page size at 1,024 bytes and we use only the medium sized intersection query. The p value used in the generation of the range lengths varied between 1% and 50% (again, per our assumption that most records should have a relatively short lifespan). The results obtained are detailed next.



(a) Index sizes. (b) Query performance.

Figure 20: Varying the average lifepan size under an exponential distribution.

The sizes of the structures is shown in Figure 20(a). We again note a steep increase in Time Index's size as the average lifespan grows. Even though one cannot clearly see from the figure, it is interesting to note that when using two MAP21 trees, and small p values, one of the trees had only one node (obviously the root). This happened because that tree was responsible to hold only large ranges, and due to the low average value, very few of those where generated. As p increased such long ranges were more numerous and thus the trees tended to have a mode equivalent load. Nonetheless, comparing to previous results, the sizes of the investigated structures was not affected by the different distribution in lifespan sizes.

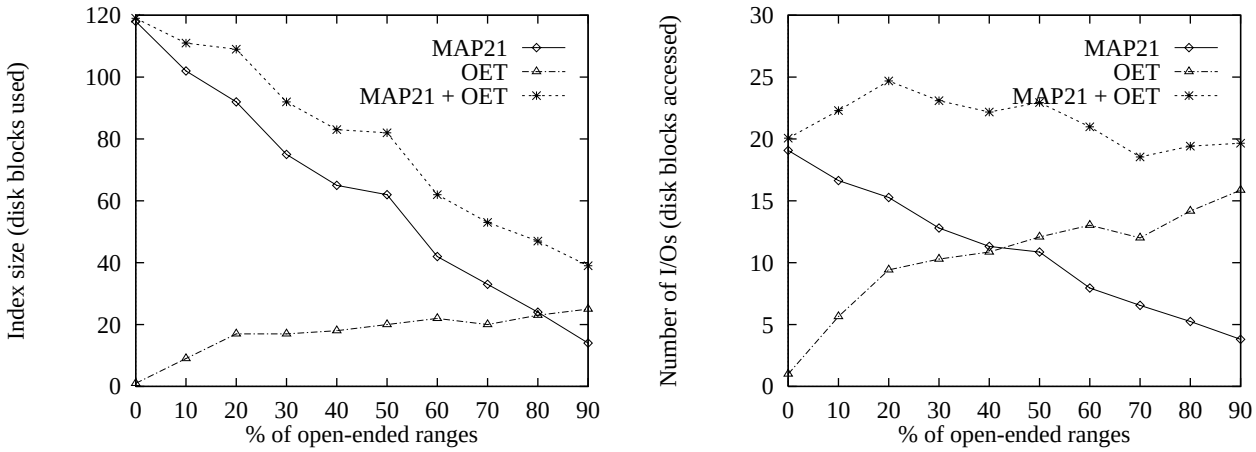
A quite different conclusion can be made regarding query processing time. Figure 20(b) shows how MAP21 is affected by a Δ which is not very tight. This is easier to see by looking at the curves yielded by using two parallel MAP21 trees. When p is small, there is virtually no difference whether or not one uses such trees in parallel. This happens because one tree contain virtually all the ranges, and this single tree dominates the searching. As p increases it becomes more advantageous to search both trees in parallel. The larger the average the higher the likelihood that larger ranges are generated, and thus the better load balancing between the trees. This ultimately implies a larger difference between using the two trees in parallel or not. The figure shows this trend well. It is interesting to note though that the R*-tree's curve had the highest growth rate. This can be explained by the fact that as p increases, the number of larger ranges grows relatively faster, and also the ratio of overlaps, which leads to this performance degeneration. The overall conclusion about not knowing a good Δ when using MAP21 is that it may indeed hurt its performance, and may render the use of parallel MAP21 trees not very effective. However,

the larger the average lifespan, the more effective is the use of parallel MAP21 trees. Finally, the Time Index had the best performance, but at the expense of a much larger storage size (see Figure 20(a)).

5.4.5 Using the OET with MAP21 to Handle Open-Ended Ranges

When there are open-ended ranges, i.e., ranges where $V_e = NOW$, we propose to use another B$^+$-tree, called OET, to handle them. As a matter of fact the same approach could be used for both the Time Index and the R*-tree. Under the Time Index, the open-ended ranges would cause a large number of replicated pointers to the associated data records. If one consider that an open-ended range is basically valid (and thus replicated) from its valid start time until NOW, and that NOW is always the largest indexed value, it is easy to see how fast Time Index size, namely the number of SC buckets, will increase. The faster such number increases the faster its overall performance degenerates as well. The R*-tree suffers from a different shortcoming, equally bad for its performance though. It is natural to think that open-ended ranges tend to become larger as long as they are kept open. Therefore those ranges increase the amount of overlap among all the indexed ranges. As argued in the literature, the larger the overlap ratio, the more degenerated the R-tree performance, and the R*-tree, although more resilient, is not an exception to the rule. Therefore the conclusion that an OET-like approach would benefit both the Time Index and the R*-tree.

Nonetheless it is interesting to observe the behavior of the indexing structure with respect to the ratio of open-ended ranges. Thus, in this section we investigate only MAP21's behavior given that we do not wish to modify either the original Time Index nor the R*-tree.



(a) Index sizes.

(b) Query performance.

Figure 21: Varying the ratio of open-ended ranges.

We have then fixed N_r, V_e^{max} as before (see Table 1) and the distribution of the closed ranges follow a uniform distribution with lengths up to medium size. The query size is also fixed at medium. Finally we used only one MAP21 tree, although we could have used several ones (in parallel or not). We vary the ratio of open-ended ranges from 0% up to 90% of the total number of ranges. Observe that using 100% of the ranges as open-ended does not make too much sense as (1) the overall approach reduces to a single B$^+$-tree (the OET), and (2) our basic concern is to propose an efficient approach to index ranges and not only points. To make the analysis simple we assume that the time point NOW is equal to V_e^{max}, we also assume that the query range lies strictly in the interval $[0, V_e^{max}]$.

Figure 21(a) shows the size of both the MAP21 tree and the OET. The overall size (i.e., MAP21's plus OET's) is also shown. Recall that the OET indexes only one time point (V_s), which is half as big as the mapped range indexed under the MAP21 tree. This explains why the sharper drop in MAP21's size as it becomes less populated. This also explains why the OET size grows much smoother. Overall, the larger the ratio of open-ended ranges, the smaller the space consumed by the MAP21 approach.

The presence of open-ended ranges does not significantly affect MAP21's performance, that is the conclusion we can draw from Figure 21(b). We can observe that for small ratios of open-ended ranges the performance is

driven by the MAP21 tree. As that ratio increases the effort in searching the OET becomes the dominant factor. The overall performance though (which can be derived by "summing up" both curves) does not vary in a significant way. Notice that this implicitly assumes that both trees are searched sequentially. If the MAP21 tree and the OET are hosted under different disks then they may be searched in parallel and the overall performance would follow the shape of an wide open "V", with its minimum given by a open-ended ranges ratio close to 50%.

6 Summary and Future Directions

We have presented MAP21, an indexing approach (rather than a new and specialized data structure) for temporal ranges, which can be implemented using simple B^+-trees. Hence, one of the appeals of such an approach is its ease of implementation using available DBMS facilities, such as embedded SQL and the fact that most DBMSs offer some sort of B^+-tree implementation. The basic assumption made is that an exact upper bound for the length of the temporal ranges indexed is kept. As data is never physically deleted from a VTDB we believe that such assumption is reasonable. The larger such upper bound the greater the number of I/Os needed to process some types of queries. To address this we have proposed the use of multiple trees, where the data set is partitioned based upon the length of the valid time ranges. This can also be used to deal with the case where the ranges do not have a uniformly distributed length.

MAP21 is about 1/3 larger than than the R^*-tree, and much smaller than the Time Index. Unlike the Time Index and like the R^*-tree, its storage does not vary as a function of the average lifespan, but rather as a function of the number of indexed ranges.

In terms of query processing time, We have seen that using as few as two parallel MAP21 trees was virtually always the best choice. When it was not, it lost to the Time Index, which is a much larger structure. We have observed that it is always a good option to partition the data among several MAP21 trees, even if not implemented in parallel. An exception, which demands a bit more care is the case of inclusion queries. We have also shown MAP21 to be resilient to several factors, such as the length of modeled time window, distribution of the lifespan sizes (when using parallel trees) and ratio of open-ended ranges. We thus conclude that MAP21 is an atractive and feasible alternative to index temporal data.

Future research regarding MAP21 will be towards two mainstreams. First, we plan on investigating how to build the Temporal Interface mentioned in Section 4. Second, we will extend MAP21 to use it for indexing data in higher dimensions. In such case, one could use two (or more) MAP21 trees to index spatial data. In fact, in our preliminary studies [ND96] we used two MAP21 trees to index the projections (in both axes) of two-dimensional MBRs, and we verified that such framework outperforms the classical R-tree [Gut84]. Likewise, one might use two (or more) MAP21 trees to index each temporal dimension in a bitemporal database [KTF95, NDE96]. In such case, however, we may specialize the MAP21 framework to more efficiently index transaction time, which has very distinct characteristics when compared to valid time. We are currently verifying this issue in particular.

Acknowledgements

We wish to thank several people, specially: V. Kouramajian, R. Elmasri, R.T. Snodgrass, C.S. Jensen, A. Steiner, G. Yap and the anonymous reviewers. Their feedback (as well as those from many colleagues) helped us improve this paper. We acknowledge the use of Jan Janninck's implementation of the B^+-tree [Jan95] as MAP21's framework. The R^*-tree implementation was kindly provided by Yannis Theodoridis. Mario A. Nascimento performed this research while on leave from EMBRAPA at Southern Methodist University and supported by the Brazilian National Research Council (CNPq, Process 260088/92.7). Margaret H. Dunham is with the Southern Methodist University (mhd@seas.smu.edu).

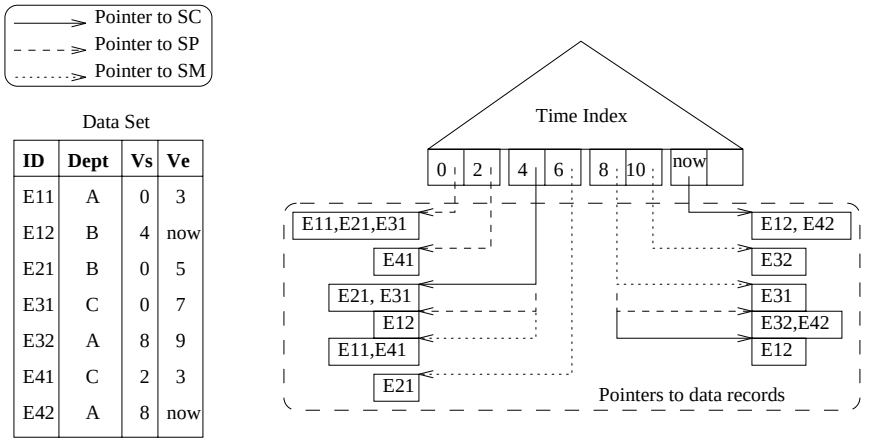
A Appendix – An Introduction to the Time Index and the R-trees

In this section we briefly review the Time Index and the R^*-tree. We focus on those issues related to this paper. For further details refer to the original papers on the Time Index [EWK90] and on the R^*-tree [BKSS90].

A.1 Time Index

The basic idea on the Time Index is to extend a B$^+$-tree to index temporal ranges. This is done by indexing the end points of each record's lifespan (i.e., valid time range). That is, each record EIJ valid at each time point $T_k \in [V_s, V_e)$, will appear at the leaf node entry indexing T_k. It is easy to see that as long as the object is valid its "ID" is replicated, what consumes considerable space. To alliviate this problem the authors proposed the use of incremental buckets. Such buckets are for those records valid in the previous node's last entry and still valid in the first entry of the current node (called SC bucket); and for those records that start or end being valid at a given entry (called SP and SM, respectively). In such incremental approach, only the leading node of a leaf would hold all IDs that are valid at that point in time, the subsequent nodes of that leaf would just record the IDs that are inserted or deleted[4].

Such incremental approach is shown in Figure 22. For good descriptions of the algorithms used to process several types of queries using the Time Index framework described above we refer the reader to [Kou94].



ID	Dept	Vs	Ve
E11	A	0	3
E12	B	4	now
E21	B	0	5
E31	C	0	7
E32	A	8	9
E41	C	2	3
E42	A	8	now

Figure 22: An example Time Index [EWK93].

To the best of our knowledge, no research has been done regarding the parallelization of the Time Index. Recently, some work has been done regarding the Monotonic B$^+$-tree [KEC94], but this takes advantage of monotonically increasign valid start time, which is not always the case with VTDBs in general.

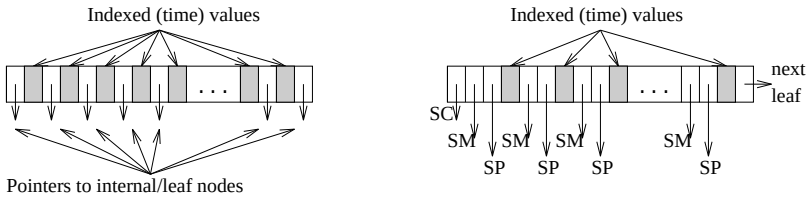
Simulating Time Index's Incremental Buckets – To implement the Time Index we have used the same B$^+$-tree source code used for MAP21, while the incremental buckets were simulated. We now discuss how such simulation was implemented.

For each range the Time Index will index both end values, i.e., given N_r ranges, the Time Index will hold $2N_r$ values, where some may be repeated (but indexed only once). Let L_i be the indexed lifespan length of record i and let $\overline{L}$ be the average length of the indexed ranges. The average number of versions valid at any given time is: $\frac{\sum_{i=1}^{N_r} L_i}{V_e^{max}} = \frac{N_r \sum_{i=1}^{N_r} \frac{L_i}{N_r}}{V_e^{max}} = \frac{N_r \overline{L}}{V_e^{max}}$.

While an internal node in the Time Index is exactly the same as an internal/leaf node in a B$^+$-tree (see Figure 23(a)) the same is not true for the leaf nodes (depicted in Figure 23(b)) due to the incremental buckets. Assuming that a pointer consumes S_p bytes, and an indexing value consumes S_v bytes, we can derive that a leaf node (i.e., a disk block of size S_d) in the Time Index can hold N_v indexing points, where $N_v = (S_d - 2S_p)/(S_v + 2S_p)$. Hence, if the leaf nodes are in average $\ln 2$ full [Yao78], we need $N_l = 2N_r/(N_v \ln 2)$ leaf nodes to index all $2N_r$ values.

Therefore every leaf node has an SC bucket of size: $\frac{N_r \overline{L}}{V_e^{max}} S_p$ i.e., one pointer to each version valid at the indexing point in the leading entry of that node. The number of disk blocks consumed by such bucket is $\frac{N_r \overline{L}}{V_e^{max}} \frac{S_p}{S_d}$ and the total space needed by all SC buckets is obtained by multiplying this equation by N_l.

[4]Insertion (deletion) meaning that such an ID is starting (ending) to be valid at that point in time



(a) Internal nodes in the Time Index (internal/leaf nodes in a B+-tree)

(b) Leaf nodes in the Time Index

Figure 23: Internal and leaf nodes in the Time Index.

The number of disk blocks consumed by the SP (SM) buckets is found similarly. Given V_e^{max} possible time points and $2N_r$ end values of the N_r ranges, we have on average N_r/V_e^{max} ranges starting (ending) at any of the V_e^{max} possible indexable time points. Hence, we have that each SP (SM) bucket consumes $\frac{N_r}{V_e^{max}} \frac{S_p}{S_d}$ space in a disk block.

Note that, when compared to a "standard" B^+-tree leaf node, a leaf node in the Time Index indexes a smaller number of values. This is due to the overhead space consumed for the incremental buckets (clearly seen in Figure 23(b)). By assuming a "standard" B^+-tree leaf node for the Time Index and simulating the incremental buckets we are not impairing the Time Index performance. In fact, we are improving its performance, by allowing more entries per leaf, and thus less leaf nodes are to be traversed in Time Index's query processing. Therefore all results shown in this study provide in fact optimistic values for the Time Index's size and query processing time.

A.2 R-Trees

The R-tree was first developed by Guttman [Gut84] and is almost certainly the most well-known indexing structure for spatial data. The main assumption is that the objects to be indexed can be modeled by means of the smallest rectangle, called Minimum Bounding Rectangles (MBRs), that contain them. Although such MBRs are N-dimensional, we, for simplicity, refer to them as if they were two dimensional rectangles.

The R-tree [Gut84] is a hierachical data structure, designed to index minimum bounding rectangles (MBRs) and which resembles the B^+-tree in many aspects. It is paginated, balanced, has the leaf nodes pointing to the actual data records, and non leaf nodes pointing to either leaf nodes or represent a super-MBR which englobes other super-MBRs or MBRs. Figure 24 (adapted from [SRF87]) shows an example of MBRs (C, D, E, ..., J) and super-MBRs (A and B) and the resulting R-tree.

The R^+-tree [SRF87] and the R^*-tree [BKSS90] are R-tree derivatives which address the R-tree's original problems in dealing with a large degree of overlap among the indexed MBRs.

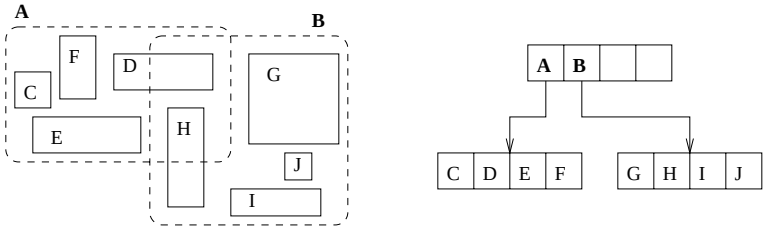


Figure 24: A sample of MBRs and the resulting R-tree (adapted from [SRF87]).

The authors of the R^+-tree propose that MBRs need not be indexed as a whole unit, but rather may be "clipped" in such a way that no super-MBR has any overlap with any other super-MBR in the internal nodes of the structure. This may enhance the query processing time significantly but causes the overall index size to increase.

When designing the R^*-tree, the authors noticed that simply deleting a the heuristics employed by the R-tree made poor decisions which led to poor performance in the long term. The authors of the R^*-tree then proposed the concept of forced re-insertion, sometimes also referred to as deferred spliting. Simply putting, the main idea is to, whenever a node split is to occur, to delete some of the MBRs in the node about to be split and re-insert

them. This will avoid the split while ensuring (by the virtue of the heuristics employed to select the MBRs) good properties of the R-tree, hence improving performance considerably [BKSS90]. Notice that the R-tree and R*-tree are essentially the same structure, unlike the R$^+$-tree. The basic difference is in the node split policy. It is worthwhile mentioning that the CPU cost of the update in the R*-tree is higher than that of the R-tree, which is paid-off with the increase in the query processing performance. As argued earlier (Section 5) is recognized to be the R-tree derivative to present the best overall performance.

There is one shortcoming regarding the R-tree family of structures, which is worthwhile discussing. The way the tree is constructed, makes it very intuitive to search for MBRs that either overlap or contain a reference MBR. Indeed, to answer a query where one needs all MBRs that overlap (intersect with) a reference MBR (which we denote by R) the procedure is the following: Start from the root traversing each sub-tree with a root (super-MBR) which intersects with R until the leaf node is reached (possibly several ones) is reached. Then each entry in that leaf is compared to R and returned or not as part of the response. (Notice that possibly several leaf nodes may be reached, and some may not contribute to the answer at all.) However, if one is interested in the MBRs which are contained within a reference MBR, the R-tree (or the R*-tree) performance degrades to that of an overlap query. The reason is that any super-MBR that overlaps the reference may contain another super-MBR (or MBR if it is in a leaf node) that is contained within that very same reference. Therefore, the search for the MBRs contained within a reference is identical to the search of the MBRs that overlap that reference. The only difference is made at the leaf nodes level, when only those actually contained within the reference are returned as part of the response. Notice that if the reference MBR is small, the search may be very unproductive, as several useless sub-trees will eventually be searched. So, the R*-tree (as well as the R-tree) suffers from one drawback MAP21 also does, namely reading useless data pointers. In the processing of a query not all indexed MBRs in a leaf node actually belong to the answer and thus must be filtered out. Note that in the inclusion query, this becomes even worse, it may read whole sub-trees (all the way down to the leaf nodes) that do not contribute to the answer.

Some work has been done regarding the parallelization of the R-tree [KF92]. However, as R-trees (or derivatives) are not widely available in most commercial DBMSs (there seem to exist exceptions though, such as CA-OpenIngres[5]) we believe that Parallel R-trees cannot be claimed to be a practical approach, unlike MAP21. To our knowledge no research has been done on parallelizing the R*-tree or the R$^+$-tree.

References

[AT95] C-H. Ang and K-P. Tan. The interval B-tree. *Information Processing Letters*, 53(2):85–89, January 1995.

[BKSS90] N. Beckmann, H.P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conferencee on Management of Data*, pages 322–331, Atlantic City, NJ, June 1990.

[CDI+94] J. Clifford, C. Dyreson, T. Isakowitz, C.S. Jensen, and R.T. Snodgrass. On the semantics of "NOW" in temporal databases. Technical Report R-94-2047, Dept. of Mathematics and Computer Science, Aalborg University, November 1994.

[EN94] R. Elmasri and S.B. Navathe. *Fundamentals of Database Systems*. Benjamin/Cummings, Redwood City, CA, 2nd edition, 1994.

[EWK90] R. Elmasri, G.T.J. Wuu, and Y.-J. Kim. The Time Index: An access structure for temporal data. In *Proceedings of the 16th Very Large Databases Conference (VLDB'90)*, pages 1–12, Brisbane, Australia, 1990.

[EWK93] R. Elmasri, G.T.J. Wuu, and V. Kouramajian. The Time Index and the Monotonic B$^+$-tree. In A. Tansel et al., editors, *Temporal Databases: Theory, Design and Implementation*, chapter 18, pages 433–456. Benjamin/Cummings, Redwood City, CA, 1993.

[FR91] C. Faloutsos and Y. Rong. DOT: A spatial access method using fractals. In *Proceedings of the 7th IEEE International Conference on Data Engineering*, pages 152–159, Kobe, Japan, April 1991.

[5] As per URL http://www.cai.com/products/addbm/oidir/oidir.htm (as of December 19, 1996).

[G+96] C.H. Goh et al. Indexing temporal data using existing B+-trees. *Data and Knowledged Engineering*, 18:147–165, 1996.

[GS93] H. Gunadhi and A. Segev. Efficient indexing methods for temporal relations. *IEEE Transactions on Knowledge and Data Engineering*, 5(3):496–509, June 1993.

[Gut84] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conferencee on Management of Data*, pages 47–57, Jun 1984.

[Jan95] J. Janninck. Implementing deletions in B+-trees. *ACM SIGMOD Record*, 24(1):6–8, March 1995.

[JCG+94] C.S. Jensen, J. Clifford, S.K. Gadia, A. Segev, and R.T. Snodgrass. A consensus glossary of temporal database concepts. *ACM SIGMOD Record*, 23(1):52–64, Jan 1994.

[JS93] T. Jonhson and D. Shasha. The performance of concurrent data structure algorithms. *Transactions on Database Systems*, 18(1):51–101, March 1993.

[K+94] V. Kouramajian et al. The Time Index+: An incremental access structure for temporal databases. In *Proceedings of Third International Conference on Knowledge and Management (CIKM'94)*, pages 296–303, Gaithersburg, MD, November 1994.

[KEC94] V. Kouramajian, R. Elmasri, and A. Chaudhry. Declustering techniques for parallelizing temporal access structures. In *Proceedings of the 10th IEEE International Conference on Data Engineering*, pages 232–242, Houston, TX, February 1994.

[KF92] I. Kamel and C. Faloutsos. Parallel R-trees. In *Proceedings of the 1992 ACM SIGMOD International Conferencee on Management of Data*, pages 195–204, San Diego, CA, June 1992.

[Kli93] N. Kline. An update of the temporal database bibliography. *ACM SIGMOD Record*, 22(4):66–80, December 1993.

[Kou94] V. Kouramajian. *Temporal Databases: Access Structures, Search Methods, Migration Strategies, and Declustering Techniques*. PhD thesis, University of Texas at Arlington, Arlington, TX, 1994.

[KSCL95] M.S. Kim, Y.S. Shin, M.J. Cho, and K.J. Li. A comparative study of spatial access methods. In *Proceedings of the 3rd ACM International Workshop on Advances in Geographic Information Systems (ACM-GIS'95)*, pages 29–36, Baltimore, MD, December 1995.

[KTF95] A. Kumar, V.J. Tsotras, and C. Faloutsos. Access methods for bi-temporal databases. In *Proceedings of the International Workshop on Temporal Databases*, Workshop in Computing, pages 235–254, Zurich, Switzerland, September 1995. Springer and British Computer Society.

[LS93] D. Lomet and B. Salzberg. Transaction time databases. In A. Tansel et al., editors, *Temporal Databases: Theory, Design and Implementation*, chapter 16, pages 388–417. Benjamin/Cummings, Redwood City, CA, 1993.

[McK86] E. McKenzie. Bibliography: Temporal databases. *ACM SIGMOD Record*, 15(4):40–52, December 1986.

[Nas96] M.A. Nascimento. *Efficient Indexing of Temporal Databases Via B+-trees*. PhD thesis, Southern Methodist University, Dallas, TX, August 1996. Available at URL http://www.dcc.unicamp.br/~mario/Papers/dissertation.ps.

[ND96] M.A. Nascimento and M.H. Dunham. Using B+-trees as a practical alternative to the classical R-tree. In *Proceedings of the 12th Brazilian Symposium on Databases (SBBD'96)*, pages 187–200, São Carlos, Brazil, October 1996. Available at URL http://www.dcc.unicamp.br/~mario/Papers/tr-96-cse-05.ps.

[NDE96] M.A. Nascimento, M.H. Dunham, and R. Elmasri. M-IVTT: An index for bitemporal databases. In *Proceedings of the 7th International Conference on Databases and Expert Systems Applications (DEXA'96)*, pages 779–790, Zurich, Switzerland, September 1996. Lecture Notes in Computer Science, Vol. 1134.

[Ora92] Oracle. *Oracle 7 Server - SQL Language Reference Manual*. Oracle Corp., 1992.

[Ore86] J. Orestein. Spatial query processing in an object-oriented database system. In *Proceedings of the 1986 ACM SIGMOD International Conferencee on Management of Data*, pages 326–336, Washington, DC, May 1986.

[ÖS95] G. Özsoyoğlu and R.T. Snodgrass. Temporal and real-time databases: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 7(4):513–532, August 1995.

[SA86] R.T. Snodgrass and I. Ahn. Temporal databases. *IEEE Computer*, 19(9):35–42, September 1986.

[SKS95] M.D. Soo, N. Kline, and R.T. Snodgrass. SQL-92 compatibility issues. In R.T. Snodgrass, editor, *The TSQL2 Temporal Query Language*, chapter 26, pages 501–504. Kluwer Academic, Boston, MA, 1995.

[Sno95] R.T. Snodgrass, editor. *The TSQL2 Temporal Query Language*. Kluwer Academic, Boston, MA, 1995.

[SOL94] H. Shen, B.C. Ooi, and H. Lu. The TP-Index: A dynamic and efficient indexing mechanism for temporal databases. In *Proceedings of the 10th IEEE International Conference on Data Engineering*, pages 274–281, Houston, TX, February 1994.

[Soo91] M.D. Soo. Bibliography on temporal databases. *ACM SIGMOD Record*, 20(1):14–23, March 1991.

[SRF87] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R^+-tree: A dynamic index for multidimensional objects. In *Proceedings of the 13th Very Large Databases Conference (VLDB'87)*, pages 507–518, Brighton, England, September 1987.

[ST94] B. Salzberg and V.J. Tsotras. A comparison of access methods for time evolving data. Technical Report NU-CCS-94-21, College of Computer Science, Northeastern University, 1994. (Also published as Technical Report CATT-TR-94-81 at Polytechnic University).

[Ste96] A. Steiner. TimeDB home page. URL: http://www.inf.ethz.ch/personal/steiner/TimeDB.html, 1996.

[TK95] V.J. Tsotras and N. Kangelaris. The snapshot index, an I/O optimal access method for timeslice queries. *Information Systems*, 3(20):237–260, 1995.

[TK96] V.J. Tsotras and A. Kumar. Temporal database bibliography update. *ACM SIGMOD Record*, 25(1):41–51, March 1996.

[TP95] Y. Theodoridis and D. Papadias. Range queries involving spatial relations: A performance analysis. In *Proceedings of the 2nd International Conference on Spatial Information Theory (COSIT'95)*, Semmering, Austria, September 1995.

[Yao78] A. Yao. 2-3 trees. *Acta Informatica*, 2(9):159–170, 1978.