A B$^+$-tree Based Approach to Index Transaction Time

Mario A. Nascimento

Relatório Técnico IC–97-09

Julho de 1997

A B$^+$-tree Based Approach to Index Transaction Time

Mario A. Nascimento[*]

Abstract

Transaction time of a record is the time interval when the record is stored in the database. In this paper we present an approach which provides efficient indexing of such kind of temporal data. The approach makes use of two standard B$^+$-trees with trivially modified node split policies – which yield a usage ratio of virtually 100% in each tree. We compare the proposed approach, which we name Two-Stage, to the Monotonic B$^+$-tree (by Elmasri et al). Our simulations show that the Two-Stage approach yields a structure up to 75% smaller than the Monotonic B$^+$-tree, and in all but one of the several investigated scenarios, the Two-Stage approach provided faster (or comparable) query processing time. Our main contribution, however, lies in the fact that the Two-Stage Approach does not require novel data structures but well-known B$^+$-trees. As such, and unlike all previous techniques for tackling this problem, it can be implemented using facilitites existing on most commercial DBMSs.

1 Introduction

"Transaction time of a database fact is the time when the fact is current in the database and may be retrieved" [J$^+$94]. Basically a temporal database that supports transaction time (so-called transaction time database) is able to keep all past history of the database. Once a tuple in a relation is updated the newer version replaces the older one, but the older is only logically deleted instead of physically deleted. In fact, in transaction time databases "... errors can sometimes be overridden (if they are in the current state), but they cannot be forgotten ..." [SA86]. We denote the transaction time of a data record R^k, by $T_t^k = [T_s^k, T_e^k]$. Usually the transaction-start-time (T_s^k) is set to the time when the transaction that wrote the data committed and the transaction-end-time (T_e^k) is left open. An open transaction-end-time is usually denoted by setting $T_e^k = NOW$ (where the symbol NOW stands for the current instant in time. For a thorough discussion on the semantics of this value refer to [C$^+$94]). When such data is to be updated, i.e., a version thereof is to be created, its transaction-end-time is set to the time point immediately before that of the transaction-start-time of the new version of the data. The new transaction-end-time is left open, and so on and so forth. Naturally, it is possible to "close" some data without

[*]Invited lecturer at the Institute of Computing, State University of Campinas (UNICAMP), and researcher at the Brazilian Agency for Agricultural Research (EMBRAPA). Permanent Address: CNPTIA – EMBRAPA, PO Box 6041, 13083-970 Campinas SP, Brazil. E-mail: mario@cnptia.embrapa.br. This research was developed as part of GEOTEC's effort, a PROTEM-CC project sponsored by the Brazilian National Council of Research (CNPq) Process 480007/97-7.

inserting a new version of it, that is, only (logically) delete it without versioning it. Such type of temporal data is common in many application domains.

Temporal databases in general (refer to [T$^+$93] for a good collection of papers) require the indexing of temporal data, besides the usual indexing of non-temporal data, for efficient query processing. Although more than 1,000 papers have been published in the area of temporal databases [Kli93, McK86, ÖS95, Soo91, TK96], relatively few have addressed the issue of indexing temporal data (refer to [ST94] for a survey). In this paper we address the problem of indexing transaction time. In what follows we briefly review the related approaches most mentioned in the literature.

The *Time-Split B-tree* [LS93] is unique in the sense that is the only data structure which indexes temporal and non-temporal data under a single framework. Recently, some work was done towards its parallelization [MKW96]. The *Snapshot Index* [TK95] may be optimal if a dynamic hashing function is used instead of a B$^+$-tree for handling updates. It uses an interesting strategy to keep track of closed time intervals. The *Append-Only Tree* [GS93] uses a framework which is similar to the B$^+$-tree, however, it indexes only the transaction-start-timeon the leaf node level. The transaction-end-time is kept inside additional buckets under the leaf nodes. This requires accessing beyond the leaf nodes level and affects the index update procedures. The *Monotonic B$^+$-tree* [EJK92] keeps track of both transaction-start and end time explicity in the tree, which is much like a B$^+$-tree. For all indexed points between the transaction-start and end time of records, pointers to those are maintained in an incremental manner (we review the Monotonic B$^+$-tree in more details in the Appendix). Unfortunately, keeping such pointers incrementally may still result in a large structure. In addition, the leaf nodes of the indexing tree are different from standard B$^+$-tree leaf nodes

All those access structure have their merits, but all lack one quite desirable property, which is feasibility. They all require deep modifications to existing DBMSs. In other words none of them can be used in conjunction with existing DBMSs. We, on the other hand, aim primarily at providing an index structure which can be used on top of facilities available in existing DBMSs. We do so by re-using the B$^+$-tree structure (refer to [EN94, Chapter 5] for an introduction) in an approach we name *Two-Stage* (2S for short). It is important to notice that, by re-using well-known B$^+$-trees, we also inherit all previous research on it regarding issues such as concurrency control and recoverability [JS93]. Note much has been done regarding those issue in most of the above mentioned access structures.

Given that we use B$^+$ trees and the monotonicity of time will allow us to explore the "append-only" nature of the indices we may derive the performance bounds for 2S as shown in Table 1 (the bounds for the other approaches were obtained from [ST94]). We also denote N as the number of updates performed, B the disk block size and A the answer size. D is proportional to the square of the largest indexed lifespan in the worst case and a constant in the expected case (see Section 2.2). A time slice query is assumed for computing the bounds for query processing time. We also assume the indices to be primary indices, in the sense that "the index controls the physical placement of data ... primary indices need not be on primary keys of relations" [ST94]. We discuss the bounds obtained for the 2S approach in more detail later in the paper.

Note that 2S's size is, like most other structures listed, linear on the number of updates, which is a desirable feature. Even though, it is not optimal, 2S's update time is good and it

Table 1: Comparing 2S to other transaction time oriented indices.

Structure	Storage	Update Time	Query Time
Two-Stage (2S)	$O(N/B)$	$O(\log_B N)$	$O(\log_B N + (A+D)/B)$
AP-tree [GS93]	$O(N/B)$	$O(N/B)$	$O(N/B)$
MBT Index [EJK92]	$O(N^2/B)$	$O(N/B)$	$O(\log_B N + A/B)$
Snapshot Index [TK95]	$O(N/B)$	$O(1)$	$O(\log_B N + A/B)$
TSBT [LS93]	$O(N/B)$	$O(\log_B N)$	$O(\log_B N + A)$

is achieved using a well known data structure (the B$^+$-tree) for which quite a lot of research has been devoted. Regarding query processing time, 2S depends on a variable no other structure does, namely the length of the largest indexed lifespan. We discuss this issue later in the paper. Nonetheless, if D is small and constant (in fact, we will show this assumption can be valid most of the time), 2S's query processing time is nearly optimal. Indeed, our performance analysis will show that 2S's performance is better (or close) to the MBT's, hence supporting our claim.

Valid time, is the time when a fact was true in the modeled reality [J$^+$94]. In several domains however, we can model valid time as transaction time. For instance, in an environment where satellite imagery or sample data is automatically collected at possibly, but not necessarily, regular time intervals could be interpreted as transaction time oriented. Many other papers have dealt with the indexing of "pure" valid time, however, due to the restricted space and our focus on transaction time we do not cover such topic.

The rest of the paper is organized as follows. Section 2 presents the main contribution of the paper, the 2S indexing strategy for transaction time data. We then show that using a standard B$^+$-tree, with trivially modified node split policies, we may ensure a node utilization ratio close to 100%. Comments on how to actually achieve this in an existing DBMS (ORACLE's in particular) are also drawn. In Section 3 we compare the performance of the proposed approach against that of the Monotonic B$^+$-tree, which despite its high storage requirement can be quite efficient in terms of query processing. It is interesting to note that, to our knowledge, no actual performance comparison has been made among the access structures mentioned above (not even when they were first proposed). The only comparison has been made in [ST94] and it was based on complexity analysis (e.g., Table 1) rather than actual implementations. In Section 4 we indicate how 2S can make use of optical disks in a rather simple manner. However, we do not focus on such an issue. Conclusions and directions for further research are offered in Section 5.

2 The Two-Stage Approach

Our approach has two stages, both functioning possibly concurrently. The idea is to be able to handle both open-ended and closed ranges. In what follows, we describe such stages briefly.

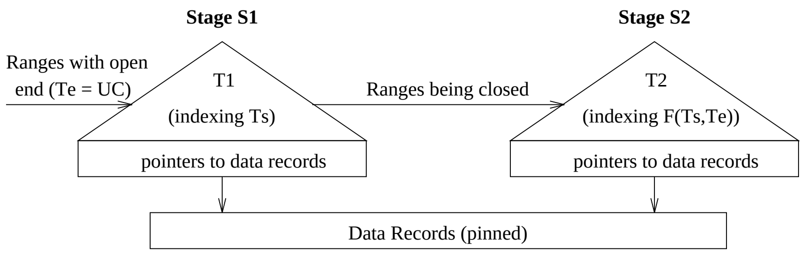


Figure 1: Illustration of the two stages on which we base our approach.

S1: In stage one all ranges which are open-ended (i.e., $\forall R^k \mid T_e^k = NOW$) are indexed by its transaction-start-time (T_s^k) under a B$^+$-tree, denoted by T1;

S2: In this stage all ranges that have been closed, suffer a simple transformation and are indexed under a second B$^+$-tree, denoted by T2.

Let us now briefly explain the rationale behind each stage. Initially, i.e., until the first transaction time range is closed, S2 does not function. S1 is responsible for managing, through T1, all incoming transaction-start-times. When one or more records are closed[1], those are deleted from T1 and moved over to T2. Before they are inserted into T2 though, these (closed) ranges go through a mapping function $F(T_s, T_e)$ which maps a range into a point. This mapping is such that it will maintain the incoming points ordered primarily by their original transaction-end-time and secondarily by their transaction-start-time. This will guarantee that any other range which is closed and input into T2 later on will not precede, with respect to this mapping value, any other range already input into T2. Thus, if the underlying B$^+$-tree uses a node split policy that instead of splitting nodes by "half" simply overflows to a newly created node, we may achieve a usage ratio close to 100%. The overall idea is illustrated on Figure 1. We detail how Stages S1 and S2 operate in the next Section.

2.1 Stage S1

Stage S1 is responsible for maintaining all those ranges which are still open. As such, it functions since the first update occurs, and needs to index only the transaction-start-time of the records. Recall that all open-ended ranges have all the same transaction-end-time, namely $T_e = NOW$. A simple B$^+$-tree suffices to handle the ranges in S1. The data type being indexed reveals an interesting property: the transaction-start-time grows monotonically. With this in mind we modify a little bit the standard B$^+$-tree node split policy in order to achieve a much better space usage. We explain this modification next.

Assume a standard B$^+$-trees of order n, i.e., a node holds n pointers and $n-1$ indexed data items. Once a value is to be inserted into a full node, a new node is created and the set of indexed values of the full node plus the new value is divided evenly among those two

[1]Note that all such records will have the same transaction-end-time but not necessarily the same transaction-start-time.

nodes. Figure 2(b) illustrates this procedure using the leaf node shown in Figure 2(a). Such splits may "propagate upwards" causing similar node splits in internal nodes of the tree. Such policy yield, in average, a 69% node utilization ratio [Yao78], i.e., almost a third of the slots in each node is not used.

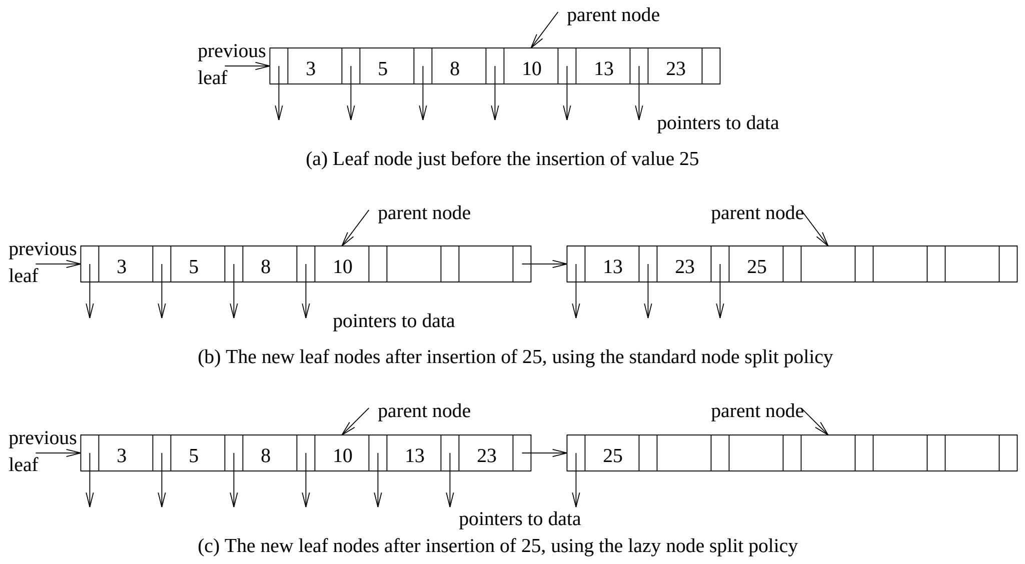

parent node
previous leaf
3
5
8
10
13
23
pointers to data
(a) Leaf node just before the insertion of value 25
parent node
parent node
previous leaf
3
5
8
10
13
23
25
pointers to data
(b) The new leaf nodes after insertion of 25, using the standard node split policy
parent node
parent node
previous leaf
3
5
8
10
13
23
25
pointers to data
(c) The new leaf nodes after insertion of 25, using the lazy node split policy

Figure 2: Illustrating the standard and lazy node split policy.

As noted above, the transaction-start-time, which is the data value being indexed, grows monotonically, i.e., any new data item being indexed is always greater than all other ones already indexed. Using the standard node split policy is a bad idea in such a case. Once a node is split it will never receive any value again, because all new incoming values are certainly greater the the last value indexed, which is, by construction, always the rightmost one on the rightmost leaf node. For instance, the left node in Figure 2(b) (and all others to its left) will remain nearly half empty. Therefore using the standard split node policy will yield an asymptotic node utilization of 50% when indexing monotonically growing data values. Note that such behavior also happens in the internal nodes, and therefore the ratio of 50% is valid to all nodes in general, internal and leaves.

To address this severe shortcoming we propose a modified node split policy, which we call "lazy". In the lazy node split policy, the full node, which is receiving the incoming value, remains full and a new node is created to host the incoming value. This leads to a much better node utilization. In fact, all nodes but the rightmost are full, which yields an asymptotic node utilization of 100%. Expanding this argument to the case of when internal nodes are split, we reach similar conclusion, i.e., in all levels of the tree all but the rightmost node will be close to 100% full. Note that even though update time remains logarithmic, as in the standard B$^+$-tree, it is nevertheless much simpler, as nodes do not split but rather overflow.

We must make clear though that we do not claim such lazy policy to be novel. In fact, it has been also proposed for the Monotonic B$^+$-tree [EJK92]. On the other hand, it is noteworthy pointing out that this strategy need not be actually implemented from scratch. For instance, ORACLE's DBMS [Ora92] does provide a directive `PCTFREE` to be used (optionally) when a `CREATE INDEX` command is issued. In fact, "`PCTFREE` is the percentage of usage to leave free for updates and insertions withing each of the index's data block"[2]. Thus, if we assume an index is created for T1 on the transaction-start-time using the option `PCTFREE 0` then we may indeed assume the lazy split policy explained above is actually being realized. Therefore the modified node split policy discussed above is feasible to achieve. Furthermore, the very nature of transaction-start-time (i.e., increasing monotonically) does facilitate maintaining the aimed high fill factor in T1.

Let us now discuss how deletions are processed in T1. Unlike insertion, deletion of indexed values can occur in any order. Once all data items indexed under a given value t are closed such value must be deleted from T1. This may yield some "holes" in tree nodes (internal and leaves). For simplicity we assume a lazy deletion policy as well (also called "free-at-empty" in [JS93]). In the lazy deletion policy nodes are not merged with others to maintain a minimum load, but have their values deleted until they are empty. When this happens the given (empty) node is simply deleted. This possibly violates the original minimum load requirement of the B$^+$-tree, but it has been shown [JS93] that such lazy deletion policy simplifies very much the deletion policy without sacrificing significantly search performance and space utilization.

Another possibility is that T1 may be periodically "compacted". Such compaction is not complicated to implement but implies significant overhead, as access to the tree is quite affected by the compacting process. Although choosing the best possibility is object of further research, we assume the lazy deletion policy is the one adopted.

Note that the lazy node split policy does not imply any change in the B$^+$-tree searching strategy, which remains logarithmic on the the number of indexed points. It is easy to see that the B$^+$-tree in question also remains efficient regarding space usage, which is linear on the number of ranges (N). Finally note that, overall, T1 is simpler to implement than a standard B$^+$-tree (should one decide to implement one from scratch).

Data Set at time 3

ID	Ts	Te
A1	0	*NOW*
B1	2	*NOW*
C1	0	*NOW*
D1	3	*NOW*
E1	2	*NOW*
F1	2	*NOW*

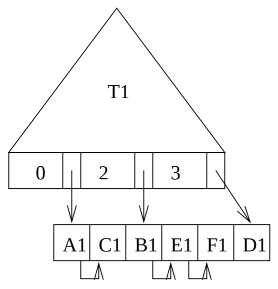


Figure 3: T1 indexing the initial data set.

For the sake of illustration, consider the data set shown in Figure 3, where Xi is the i-th version of surrogate X. Up to time 3, only new and still open-ended versions were input, and thus T1 is like the one shown. The details of leaf nodes boundaries and internal nodes

[2]Quoted from ORACLE's SQL*Plus: Release 3.2.2.0.0 on-line help.

are omitted for simplicity.

Searching T1 is straightforward. We assume that *NOW* is greater than any other currently indexed time point. As such, to find all open-ended ranges that intersect with a given query range $Q = [Q_s, Q_e]$, one needs to search from the initial indexed value (possibly 0) until Q_e, as obviously any range starting after Q_e cannot intersect with Q and all open-ended ranges starting before Q_e do intersect with Q as it is assumed that all open-ended ranges are valid until *NOW* which is farthest in the future than any Q_e.

Summarizing, we can achieve, as far as T1 is concerned, the performance bounds in Table 1. Namely: T1's size is linear on the number of updates (transaction-start-times). It has logarithmic update time (at most the rightmost branch of the tree is updated) and has a very good query processing time, basically acessing only leaf nodes (disk blocks) relevant to the queries.

2.2 Stage S2

This is probably the main stage, as it will maintain all closed versions within T2, and must provide an efficient way to process queries which involve those.

First let us discuss how S2 works. Suppose that at a given time t, some number of versions are closed. S2 then receives from S1 a set of ranges which also posseses an interesting property: all the incoming ranges have the same transaction-end-time, and such transaction-end-time is greater than all other transaction-end-times already indexed under T2, and is smaller than all other transaction-end-times that will possibly be input in the future. Thus, in a sense, T2 holds several groups of ranges (grouped by transaction-end-time) and we must use this to our advantage. If we are able to maintain a relative order among those groups, then we can guarantee that the index, will grow only to the right, in an "append only" manner. This is important as we can make better use of space. We discuss this shortly.

The chief question we need to answer is how to index the closed ranges, in order to keep the grouping mentioned above. Unlike in S1, we now have two values (T_s and T_e) for each record. We propose to do this using an approach similar to the one used in [ND97] to index valid time, which is to use a function which maps ranges to points. The function originally used in that paper preserved the original lexicographical order of the ranges, i.e., the ranges were ordered by transaction-start-time and secondarily by transaction-end-time. We modify the mapping function such that the ranges are primarily ordered by transaction-end-time and secondarily by transaction-start-time. Recall that each of the groups mentioned above, have the same transaction-end-time. Thus, such order reflects the desired grouping on the leaf nodes of the indexing tree. Formally, we define the mapping function in question as follows:

Definition 1 $\phi(T^k) = \phi(T_s^k, T_e^k) = T_e^k 10^\alpha + T_s^k$

where α is the maximum number of digits needed to represent any time value.

Any transaction time range $T^k = [T_s^k, T_e^k]$ can thus be indexed using the value provided by $\phi(T^k)$ – which is a point instead of a range – into T2. Obtaining the original range from $\phi(T^k)$ is clearly straigthforward.

We need now to discuss how insertions and the searching are processed. Note however, that due to the very nature of transaction time databases, data is not ever deleted, thus deletions never occur in T2.

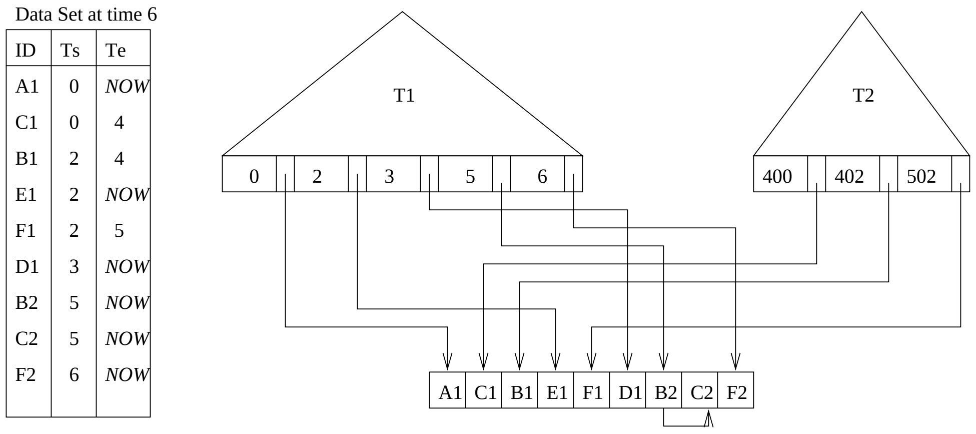
When a closed range is inserted into T2 (which is a B$^+$-tree) it must be first mapped into a point via the function $\phi(.)$ above. We could use a standard B$^+$-tree insertion procedure, but given the order imposed by the mapping above we can achieve a much better space utilization by using the very same approach one used for T1, provide we can do some pre-processing in main memory. At any given point in time there may be a number of open-ended ranges being closed with the same transaction-end-time but not necessarily same transaction-start-time. Let us assume this number to be not very large, i.e., we assume it is much smaller than the total number of indexed closed ranges. If we sort those ranges based on their transaction-start-times, then we can input them, in the sorted order, into T2 using the very same approach we described for T1. The assumption of having a not very large number of ranges being closed at a given time allows us to assume that the pre-processing of the ranges (i.e., the sorting) can be done in main memory. Note that should such assumption be not feasible then a small number of I/Os may suffice to perform such sorting using secondary storage. At any rate we believe that such number of I/Os is bound to be much smaller than the total number of leaves, hence we consider update time to be $O(\log_B N)$ instead of $O(N/B)$.

Again, this strategy can be actually realized as explained in Section 2.1, by simply setting the `PCTFREE 0` option when a `CREATE INDEX` command is issued (assuming ORACLE [Ora92] to be the underlying DBMS, of course).

Notice that we need not be concerned about deletion of data in T2. By the very definition of temporal databases, particularly transaction time databases, data is not ever deleted. Deleting a tuple means actually closing its transaction time range, thus functioning as a logical deletion rather than a physical deletion.

For a better idea of how both T1 and T2 cooperate in order to index both open-ended and closed ranges consider again the data set shown in Figure 3. Suppose that at time 5 new versions of B1, C1 and F1 are input, which implies that B1, C1 and F1 are not valid any longer and therefore must be closed. The new version, respectively B2, C2 and F2, are input into T1. The transaction time ranges for B1 and C1 now become $T_t^{B1} = [2, 4]$ and $T_t^{C1} = T_t^{F1} = [0, 4]$. Figure 4 shows the resulting trees, again internal nodes and details of the leaf nodes are omitted for simplicity.

We now discuss how T2 is searched. Only the new ordering among the ranges, implicitly given by the relative positions of the indexed points, is not enough to provide an efficient search. Consider the query range $Q = [Q_s, Q_e]$. Any range with a transaction-end-time greater than Q_s could intersect with Q, depending obviously on its transaction-start-time. However as the index is primarily ordered by the ranges' transaction-end-time (the groups we mentioned earlier) we would need to search all the groups with transaction-end-time greater than or equal to Q_s and retrieve the respective transaction-start-time and only then decide whether a range belongs or not to the query answer. This is obviously inneficient as in the worst case could cause the read of all the leaf nodes ! Let us assume though that we know an upper bound for the length of the indexed ranges, and let us call it Δ, that is:



Data Set at time 6

ID	Ts	Te
A1	0	*NOW*
C1	0	4
B1	2	4
E1	2	*NOW*
F1	2	5
D1	3	*NOW*
B2	5	*NOW*
C2	5	*NOW*
F2	6	*NOW*

Figure 4: T1 and T2 indexing the data set at time 6 ($\alpha = 2$).

Definition 2 *Given a set of ranges* V^k $\Delta = \max_k\{T_e^k - T_s^k\}$.

Now, in order to process the intersection query we need only to search through the transaction-start-times from the groups of transaction-end-times beginning at $Q_s, Q_s+1, ...$ until $Q_e + \Delta$. This is so because any range ending before Q_s cannot intersect with Q, nor can any range ending after $Q_e + \Delta$, as this would contradict our assumption about Δ. That is, the range farther in the past that can intersect with Q is $[Q_s - \Delta, Q_s]$ and the one farther in the future is $[Q_e, Q_e + \Delta]$. Therefore mapping those ranges, it implies that we must search the leaves in T2 from value $\phi([Q_s - \Delta, Q_s])$ up to value $\phi([Q_e, Q_e + \Delta])$.

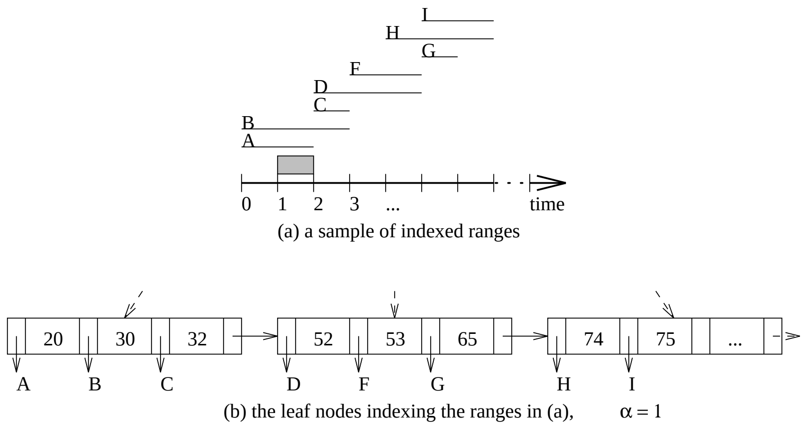

I
H
G
F
D
C
B
A
0 1 2 3 ...
time
(a) a sample of indexed ranges
20
30
32
52
53
65
74
75
...
A B C D F G H I
(b) the leaf nodes indexing the ranges in (a), α = 1

Figure 5: Searching T2.

Let us illustrate the rationale above through an example. Consider the ranges in Figure 5(a), where they are (for illustration purposes) already sorted primarily by transaction-end-time and secondarily by transaction-start-time. Figure 5(b) shows the leaf nodes of T2 (a B$^+$-tree with $n = 4$) indexing those ranges – note the full node utilization yielded by the lazy-propagated split node policy. Let the query range be $Q = [1, 2]$. If we cannot make use of Δ one must search through all the index. However, if we know that in this case we have $\Delta = 3$, it is obvious that no range ending before $Q_s = 1$ or after $Q_e = 5$ will intersect Q. Thus it suffices to search from the indexed point 0 (mapped range [0,0]) until the indexed point 52 (mapped range [2, 5]). In Figure 5(c) that would retrieve items A, B, C and D, which do have transaction time ranges that intersect with [1, 2].

Notice that we might read indexed points which correspond to ranges that will not belong to the query's answer. For example, if the Δ value associated to the ranges in Figure 5(a) were 4 instead of 3, we would have to search from the indexed point 0 (mapped range [0,0]) until the indexed point 62 (mapped range [2, 6]). This would make us read the indexing point 53 (mapped range [3, 5]) which does not belong to the answer (i.e., it obviously does not intersect with [1, 2]). In what follows we present upper bounds (worst and expected cases) for the number of such "false-hits".

The range $[Q_s - \Delta, Q_s]$ is the only range starting at Q_s investigate and it does intersect with $Q = [Q_s, Q_e]$ so that implies no false-hits. In the linear scan of the leaves however, we may have indeed some false-hits. Any point T in time may possibly be the starting point of $\Delta + 1$ ranges, namely: $[T, T]$, $[T, T+1]$, $[T, T+2]$, ..., $[T, T+\Delta]$. The next indexed point after $[Q_s - \Delta, Q_s]$ is (if existing) $[Q_s - \Delta + 1, Q_s - \Delta + 1]$. Of all those ranges starting at $Q_s - \Delta + 1$ only those ending at Q_s and $Q_s + 1$ do intersect with Q, which means that $\Delta - 1$ ranges do not and are thus false-hits. Similarly, the ranges starting at $Q_s - \Delta + 2$ imply $\Delta - 2$ false-hits, those starting at $Q_s - \Delta + 3$ imply $\Delta - 3$ false-hits, and so on and so forth until the one starting at Q_s (and others that will follow it in the linear scan) which will imply no false-hits. Therefore the number of false-hits is given by: $(\Delta - 1) + (\Delta - 2) + ...2 + 1$ which sums up to $\Delta(\Delta - 1)/2$. Hence, in the worst case the number of false-hits is bounded by $O(\Delta^2)$.

Fortunately, the expected case is much better. Suppose we have N indexed ranges and the length of the modeled time window is T_{max}. Then, in the average, we have N/T_{max} ranges starting at any point in time instead of Δ. For many (if not most) applications and specially those with a fine transaction time granularity, we expect $N/T_{max} < 1$, whereas on the other hand it would be to reasonable to expect $\Delta > 1$. Hence, if we assume $T_{max} > N$ in the long term, then the number of false-hits can be upper bounded by a constant, $O((N/T_{max})^2)$, which is unarguably much better.

The Δ value of the indexed ranges can be easily maintained in some sort of dictionary by the DBMS. For example, every time a range is closed, its length is checked against the current Δ which is then updated if needed. Furthermore, recall that transaction time databases do not allow correction nor deletion of data. Hence, Δ is dynamic and always correct. If we further assume that the value of Δ is relatively not very large, using the information about Δ will insure efficient query processing.

There is the problem that a single range with a large Δ may render the searching process longer than it actually need be. Similarly to the approach adopted in [ND97], we could make use of several trees, each indexing disjoint subsets of all ranges within specific range lengths. Such trees could be searched (and updated) in parallel improving overall performance considerably. We do not treat such case here, but we are certain that it can be dealt with, and this is subject of further research. For now, we assume that there is no range with a highly skewed length.

Summarizing, T2 uses clearly linear space. Under the assumption that the sorting (by transaction-start-time) of the ranges being migrated from T1 into T2 can be done in main memory (or using at most a constant number of I/Os) the T2's update time is $O(\log_B N)$. Finally, the time to process a query in T2 is $O(\log_B N + (A + D)/B)$, where D is upper bounded by O(1) in the expected case or by $O(\Delta^2)$ in the worst case.

At least two other mapping-based approaches have been proposed to allow using standard B^+-trees to index ranges, namely DOT [FR91] and the B^+-tree-based Time Polygon Index [G^+96] (which we refer to as B-TP, for short). DOT has been originally designed to cope with spatial data, and as such does not seem well equiped to handle the case of open-ended ranges. Also it aims at preserving the distance among the indexed itens, not the ordering. In order to take advantage of transaction time characteristics though we believe the ordering is a much attractive feature to be preserved. The B-TP's major drawback 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 a particular mapping is appropriate, but it 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]. In addition it was originally designed to index valid time, not transaction time, thus it does not explore the features of the latter. Nonetheless we believe that the B-TP could be specialized to accomplish that, and this should be object of future research.

3 Performance Analysis

In this section we investigate the performance of the proposed 2S approach. We compare its storage requirements and query processing time, with respect to intersection type of queries, to the Monotonic B^+-tree (MBT). We chose the MBT because, despite its inefficient use of storage it is indeed quite efficient for query processing (refer to Table 1), and yet it is quite simple to simulate. We assume the reader is familiar with the MBT, nonetheless, for the sake of completeness, we present a brief review of the MBT in the Appendix.

For simplicity, we will use the number of disk blocks used at leaf node level (the leaves themselves and the incremental buckets in MBT's case) and the number of disk blocks read during query processing as the indicators for the structures' size and query processing time. We believe this is reasonable for two reasons: first because the trees, by construction, should be very wide and not too deep, therefore the number of internal nodes should be much smaller than the number of leaves; secondly, the tree traversals are done in logarithmic time, whereas the leaves are traversed linearly. We use the notation (and default values) in Table 2. We have used 8 and for 4 bytes for the sizes of a pointer to a record (or set thereof) and the size of the data type representing a time value. It is important to recall that the size (in bytes) of the data type indexed under T2 is twice as big as those under T1 and the MBT (due to the mapping used). This was taken into account when we performed the simulation that follows.

Table 2: Parameters used in the performance analysis.

Notation	Used for	Values (default in **bold**)
B	Size of the disk block (in bytes)	1,024; 2,048; 4,096 and **8,192**
N	Number of records indexed	5,000; 10,000; **50,000** and 100,000
L_r	Average length ranges	250, 500; **1,000** and 2,500
T_{max}	Time value of NOW	5,000; **10,000**; 20,000 and 50,000
P_o	Ratio open-ended/closed ranges	0%, **25%**, 50% and 75%
L_q	Average length of a query range	250, 500; **1,000** and 2,500

We investigate how the sizes of the structures behave as a function of five variables: B, N, L_r, T_{max} and P_o. We have used the values shown in Table 2. We vary the parameters one at a time, while keeping the others fixed at default values. All values for the generated ranges and queries used a uniform distribution.

Our ultimate goal in this section is to help the user to be able to identify different scenarios and in which ones each structure is the "best" choice.

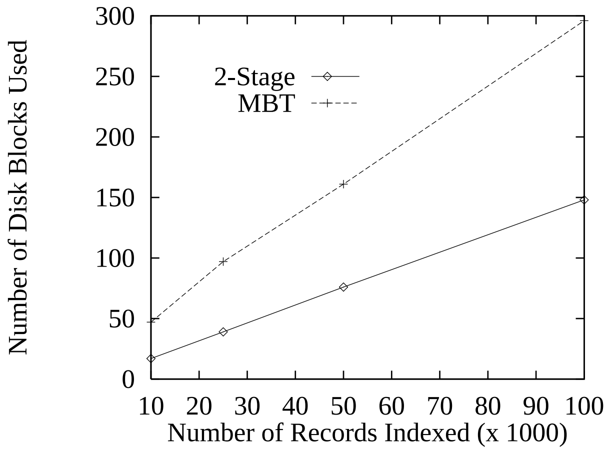
3.1 Space Requirements

To investigate the sizes of the indexing structures, as discussed earlier we compute only the number of leaf nodes disregarding the internal nodes. In the case of the MBT we must also compute the number of disk blocks used for the incremental buckets as it is a integral part of the overall structure. It is obvious that varying L_q does not affect the index sizes and therefore we disregard this parameter in the simulations presented in this section. Let us now discuss the results depicted in Figure 6.

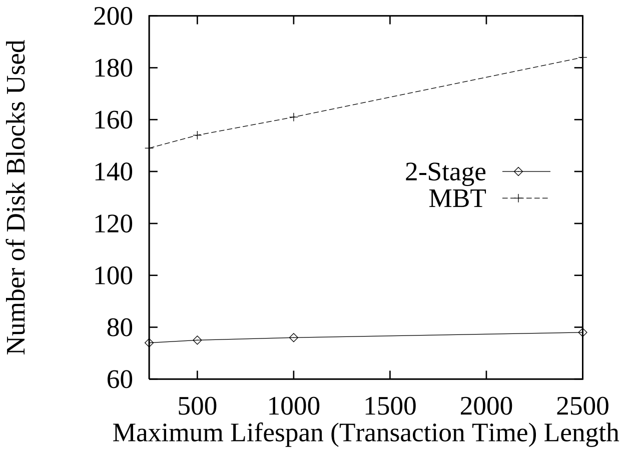
Varying N – Figure 6(a) shows that MBT's size grows a bit faster with N than 2S's. Unlike in the 2S approach the MBT may replicate pointers to tuples. In fact, a pointer to a tuple will appear in as many SCs as it spans over and thus each additional record will cause at least two (one in some SP bucket and another in a SM bucket), but most likely several pointers to be added to the underlying MBT. In 2S every additional record will contribute with a single entry being added (if needed) under T1 or T2. The MBT was, in average 106% bigger than the 2S.

Varying L_r – While Δ is an important factor for 2s's query performance one can see from Figure 6(b) that this is not the case in terms of storage. However, the MBT grows a little faster as a greater lifespan (i.e., Δ) yields more replication in the SC buckets. The MBT used in average 113% more storage space than 2S.

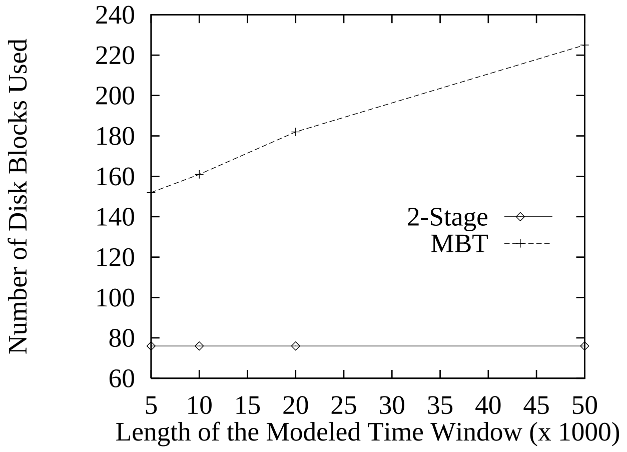
Varying T_{max} – The MBT is affected by the length of the total time frame being modelled, while 2S is totally insensitive to it (see Figure 6(c)). This can be explained as follows. We are generating a fixed number of pairs (T_s, T_e), in fact, N pairs. The shorter the range



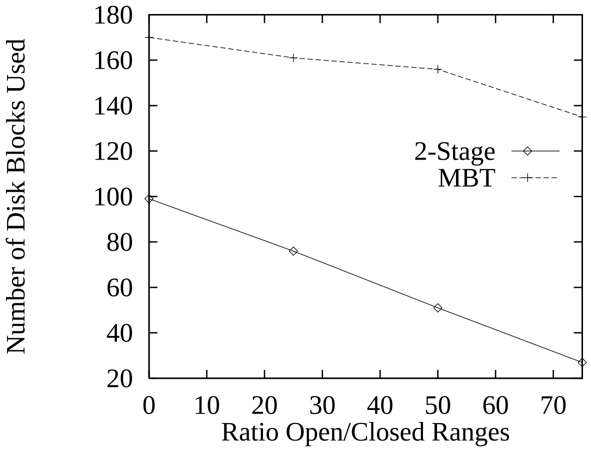
(a) Varying N.



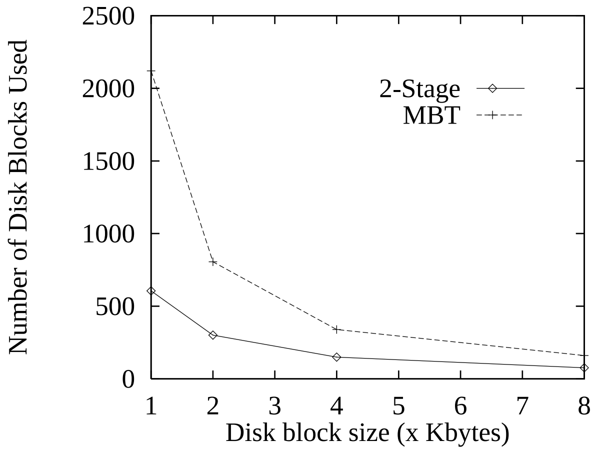
(b) Varying Δ.



(c) Varying T_{max}.



(d) Varying P_o.



(e) Varying B.

Figure 6: Index sizes.

$[0, NOW]$, the less distinct time values can be generated, and thus less leaf nodes are needed. However, as we enlarge $[0, NOW]$ more distinct time values can exist and thus there is demand for more leaf nodes. The more leaf nodes we have the more likely it is that pointers to the data records will be replicated in their respective SCs. On the contrary, 2S does not depend on T_{max} at all. This is a quite important feature of 2S, as in a temporal database time, and thus the range $[0, NOW]$, is always growing. The MBT required from 100% up to nearly 200% more space than the 2S.

Varying P_o – By inspecting Figure 6(d) we can see that with the increase of P_o, both structures decreases in size, with 2S decreasing faster. This is simple to explain. The larger the P_o, the larger T1 and the smaller T2 (recall we are keeping the total number of records constant). The data type under T1 is smaller than the one under T2, thus the gain in storage. As for the the MBT, increasing P_o forces the generation of more ranges towards the "right end" of the time line (i.e., closest to NOW), this ultimately leads to less distinct time values being generated (i.e., many transaction-start-time and transaction-end-time will coincide) and thus less space being occupied. However, the MBT's size decrease much slower, again due to the SC buckets. In fact, with P_o set to zero the MBT is about 90% larger whereas when P_o is set to 75% it is about 400% larger than the 2S.

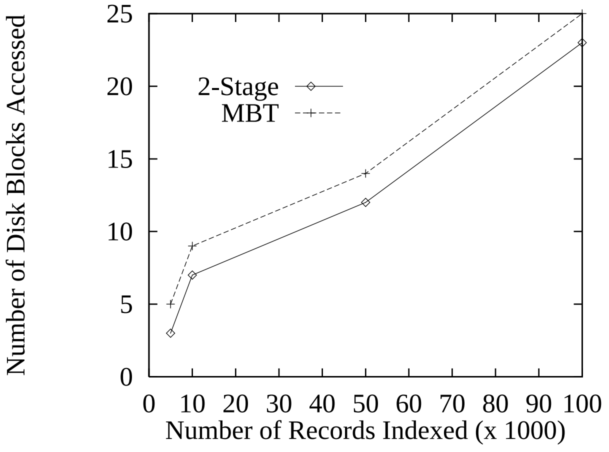
Varying B – It is natural to observe that, given a fixed number of indexing values, the smaller the leaf nodes, the more nodes are need to store such values. This is reflected in Figure 6(e). However, recall that in the MBT each leaf node maintains an SC bucket. It then follows then that the smaller the leaves, the more SC buckets will exist and the higher the degree of replication on MBT. Thus, enlarging the nodes, benefits MBT much more than 2S. Nonetheless 2S is still the smaller structure in all cases investigated. The MBT was up to 249% larger than the MBT. We should point out that the reason we chose the default value of B equal to 8 Kbytes was exactly to try to diminish the effect of the SC buckets and therefore favor the MBT.

3.2 Query Processing Time

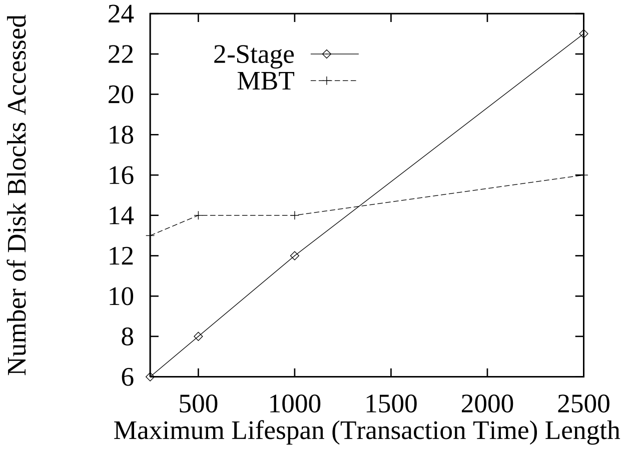
We assume that query processing time is basically driven by the number of I/Os performed, as performing one I/O is several orders of magnitude slower than executing a CPU instruction. We do not compute the number of I/Os due to the actual retrieval of the data records, but only the I/Os needed to obtain the pointers which are used to retrieve them.

We vary the same parameters used in the previous section with the addition of L_q. Unlike before, it is expected that query processing time depends on the length of the queried range. We analyze the results obtained, shown in Figure 7, next.

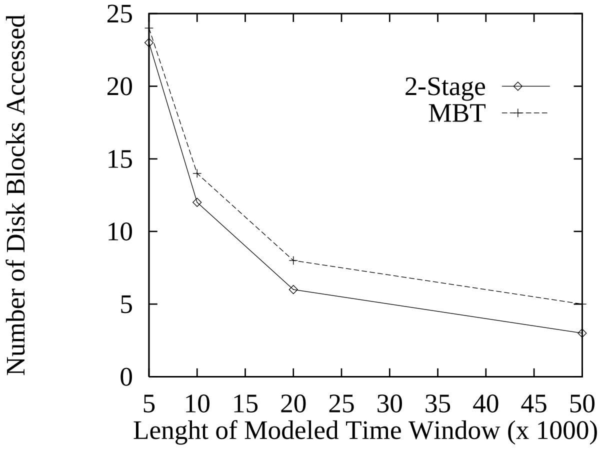
Varying N – Figure 7(c) shows us that both structure are equally affected by the increase in N. 2S was sligthly faster than the MBT with the advantage remaining constant throughout our experiments. MBT's size grow much faster with this variable (see Figure 6(a)) because all buckets's size increase proportionally. When we investigate query processing time we just take into account a much smaller range of the whole index which



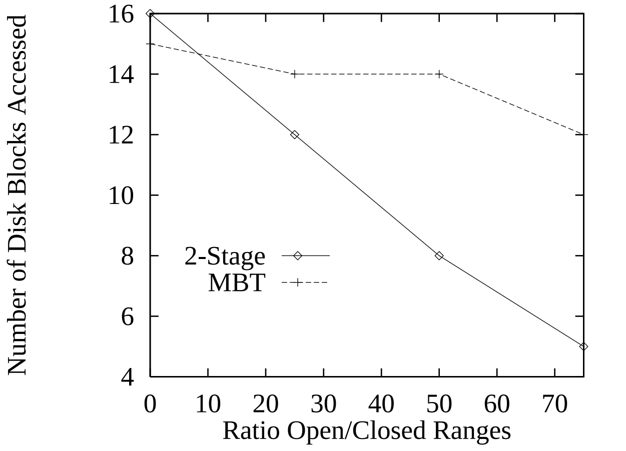
(a) Varying N.



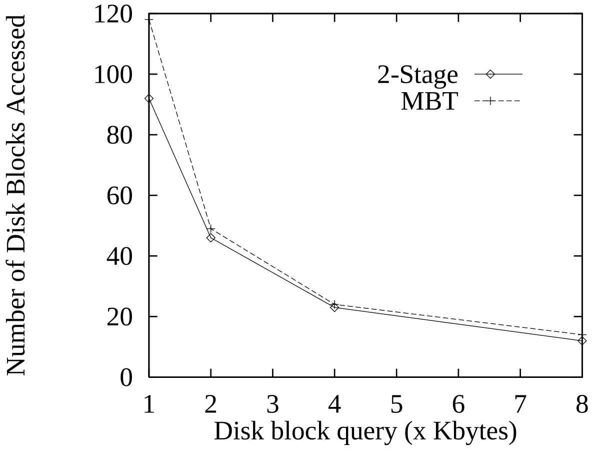
(b) Varying Δ.



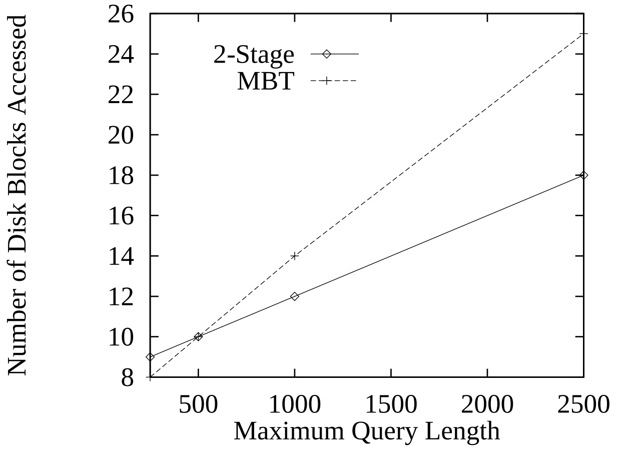
(c) Varying T_{max}.



(d) Varying P_o.



(e) Varying B.



(f) Varying L_q.

Figure 7: Query Processing Times (average number of I/Os).

does not grow as fast. On the other hand 2S's performance degenerates close to linearly with the increase in its size.

Varying Δ – Figure 7(b) shows the biggest shortcoming of 2S. As the largest indexed lifespan gets larger, the query processing time gets proportionaly slower. This happens because under 2S the search has to scan the index for ranges starting as far as Δ time points before the actual query start time until the query end time, whereas the MBT must scan only from the query start time until the query end time. As we keep all other parameters fixed, the MBT shows that it is not much insensitive to the Δ factor, unlike 2S. In our experiments 2S ranged from being more than 100% faster to being 43% slower in the case Δ was up to half as big as T_{max}. Even though we do believe that a temporal database is of more value when managing highly dynamic data, and therefore relatively short lifespans (when 2S is actually faster then the MBT), it is important to reveal that a large Δ may hurt 2S's performance considerably.

Varying T_{max} – For this case in particular (see Figure 7(c)) both access structure delivered virtually the same performance. Which is important given that in a temporal database T_{max} is ever increasing. Even though one may think that MBT's processing time should degenerate as its size, that is not the case. Incresing T_{max} do increase the number of indexed points but as N and L_r are maintained fixed, each incremental bucket actually becomes smaller and thus the increase in performance.

Varying P_o – When no open-ended ranges exist all of 2S's effort is spent on T2, which indexes a larger data type and is subject to the Δ constraint. As more open-ended ranges are considered, more of 2S's effort is devoted to T1 which is smaller and, in a sense, faster to process. Figure 7(d) shows that 2S may be a little as 6% slower when no open-ended ranges are considered and as much as 140% faster when the majority of ranges are open-ended.

Varying B – Figure 7(e) reveals an fairly natural result. Both 2S and MBT gain by using larger disk blocks. MBT may yield query processing time up to 28% is small blocks are used – this is a direct consequence of its much larger size (see Figure 6(e)).

Varying L_q – Even though both structures must perform a larger index scan as the query length gets larger, Figure 7(f) shows that the MBT suffers more. For the 2S a larger scan implies in proportionally more leaves being read. For the MBT however, not only more leaves are read but also the incremental buckets associated to them, hence the faster degeneration in MBT's performance. The MBT was sligthly faster for small queries and about 40% slower for the larger ones.

3.3 Remarks

We conclude this section by noting that if one considers a "default" scenario, i.e., where all investigated parameters have the default values we defined in Table 2 we would have the 2S approach consuming 112% less space and requiring 16% less I/Os to process queries. Even though one may argue that time is more costly than space, it should not be ignored that temporal databases grow indefinitely, and as such it is important to keep the indices size under control. In addition, one may note that except for the case where the Δ is large 2S delivered a performance, at least comparable to the MBT's. Nevertheless, for several scenarios 2S is bound to provide faster query processing time than the MBT.

One must not forget that 2S, unlike all other approaches originally proposed to index transaction time, can be easily implemented using facilities available in most (if not all) commercial DBMSs. These facts, allied to the fact that 2S is very compact, lead us to believe 2S to be a promising approach.

4 Using Tertiary Storage

In this section we indicate how 2S can easily make use of tertiary storage. This was also a concern with respect to the MBT [EJK92]. In particular we consider media of the WORM type, (e.g., optical disks). Use of cheaper, albeit slower, media storage may needed for temporal databases, which, by their very nature do not allow deletion of data, and thus may become very large quite fast. In an environment where updates happen very often that may be even a bigger problem. In what follows we describe how 2S, particularly T2, can make use of tertiary storage. T1 suffers deletions, i.e., open-ended ranges are eventually closed, hence it cannot take much advantage of tertiary storage.

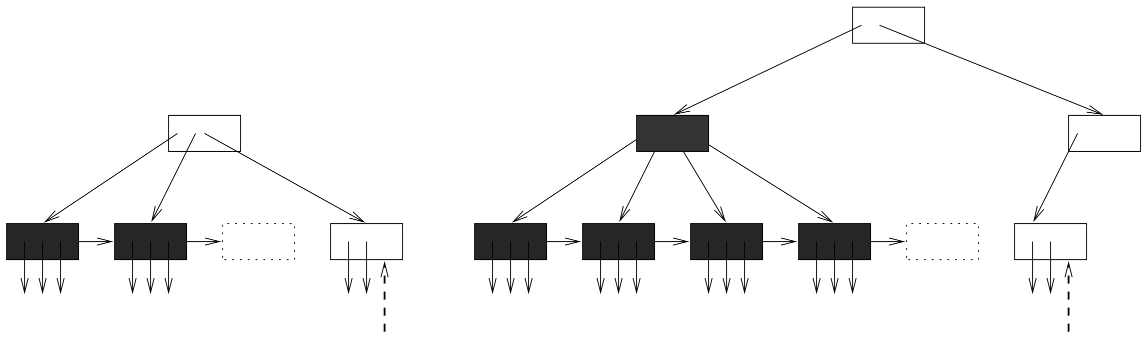


Figure 8: Illustration of the migration procedure.

Initially 2S functions using only magnetic disk as described earlier. Once a leaf node (in T2) becomes full we can migrate it to tertiary storage as we can be sure that no indexed value smaller than those in that leave can be input into that tree. One potential problem would be that each leaf points to it rigth neighbor and as such it could have to wait until this neighbor migrates as well. However this would create a "domino effect" that would end up preventing the migration of leaves. We resolve this by pre-assigning the location of the next leave to be migrated on the optical disk (denoted by the dashed "disk block" in the Figure 8). This will allow the migration of a leave as soon as it becomes full. Also given that the tree nodes have constant size this is actually simple to achieve. For internal nodes we need to make sure that it is not only full but also that all the nodes it points to have also been migrated. Otherwise we could not update the corresponding pointers in the internal node. Finally, a linear scan on the leaf nodes can be jeopardized as beginning from the leftmost node all others but the rightmost can be reached. Therefore once a linear scan reaches what seems to be the last leaf node, it must search the actual last leaf, which still resides in magnetic disk. For that we must keep the minimal overhead of one extra pointer (represented in the Figure by the dashed arrow pointing upwards).

For an illustrative example consider Figure 8 where the lighter (darker) rectangles represent disks blocks which are resident in the magnetic (optical) disk. In the leftmost tree the leaves which are full have already been migrated to the optical disk, their parent node however cannot be as the rightmost leave still resides in magnetic disk and eventually its address will change (when it is migrated) and its parent must change its pointer address accordingly. After some time, we may have the situation depicted in the rightmost tree. Note that the internal node which points only to migrated leaves has been also migrated.

Even though we have not performed any simulations using tertiary storage, we would expect to have all curves in Figure 7 shifted up. Tertiary storage is slower and most of the query processing time is spent on scanning the leaves which reside mostly on the slower media.

5 Conclusions

This paper presented 2S, an approach based on two B$^+$-trees which can be used to index transaction time ranges. We have shown that we may achieve virtually 100% node utilization without much difficulties. One of 2S's great appeals is that it is straightforward to implement, unlike all other existing proposals to tackle the same problem. We have also shown that 2S can cope with tertiary storage, e.g., optical disks, rather easily.

When comparing 2S against the Monotonic B$^+$-tree (MBT) we have reached two main conclusions: (1) 2S always yields an structure smaller than the MBT, regardless of the underlying scenario; and (2) 2S is particularly sensitive to large lifespans, otherwise it provides good query processing time, that is, at least comparable to MBT's. One should take into account the fact that the MBT is regarded as fairly efficient for processing queries (which is its trade-off to the use of a large storage).

As mentioned in Section 2 we believe that 2S's inneficiency when processing data sets with large lifepans (i.e., large Δ) can be overcome by using multiple (possible parallel) trees. This is object of future research.

Acknowledgements

The author thanks valuable feedback received from Christian S. Jensen and Michael H. Böhlen while visiting Aalborg University. The author also wishes to acknowledge Margaret H. Dunham for earlier discussions on the subject.

References

[C+94] J. Clifford et al. On the semantics of "NOW" in temporal databases. Technical Report R-94-2047, Dept. of Mathematics and Computer Science, Aalborg University, November 1994. (to appear in ACM Transactions on Database Systems).

[EJK92] R. Elmasri, M. Jaseemuddin, and V. Kouramajian. Partitioning of Time Index for optical disks. In *Proceedings of the 8th Intl. Conf. on Data Engineering*, pages 574–583, Phoenix, AZ, 1992.

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

[EWK93] R. Elmasri, G.T.J. Wuu, and V. Kouramajian. The Time Index and the Monotonic B^+-tree. In [T+93], chapter 18. 1993.

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

[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.

[J+94] C.S. Jensen et al. 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 current data structure algorithms. *ACM Transactions on Database Systems*, 18(1):51–101, March 1993.

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

[LS93] D. Lomet and B. Salzberg. Transaction time databases. In [T+93], chapter 16, pages 388–417. 1993.

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

[MKW96] P. Muth, A. Kraiss, and G. Weikum. LoT: Dynamic declustering of TSB-tree nodes for parallel access to temporal data. In *Proceedings of the 5th International Conference on Extending Database Technology*, pages 553–572, Avignon, France, March 1996. (Published as LNCS Vol. 1057).

[ND97] M.A. Nascimento and M. H. Dunham. Indexing valid time databases via B^+-trees – the MAP21 approach. Technical Report CSE-97-08, School of Engineering and Applied Sciences, Southern Methodist University, 1997. Available at URL http://www.cnptia.embrapa.br/~mario/Papers/tr-97-cse-08.ps.

[Ora92] Oracle. *SQL*Plus User's Guide and Reference, Vol. 3.2.* Oracle Corp., 1992.

[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.

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

[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. (To appear in ACM Computing Surveys).

[T^+93] A. Tansel et al., editors. *Temporal Databases: Theory, Design and Implementation.* Benjamin/Cummings, Redwood City, CA, 1993.

[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.

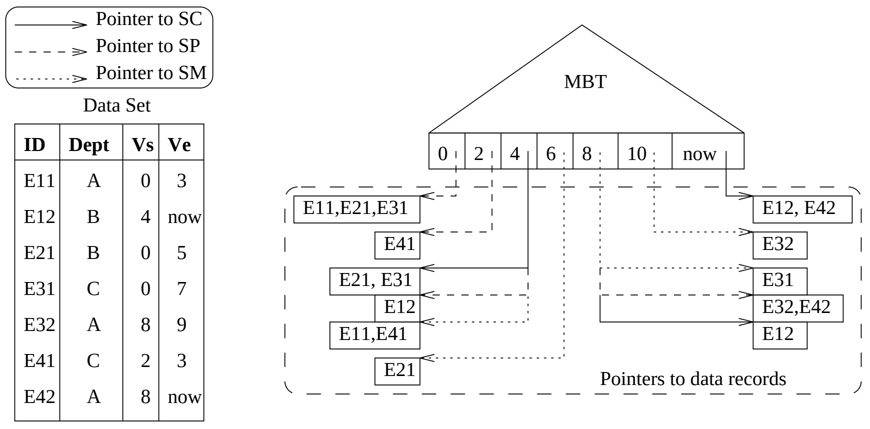
[TK96] V.J. Tsotras and A. Kumar. Temporal database bibliography update. *ACM SIGMOD Record*, 25(1):41–51, March 1996.

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

A Appendix – Review of the Monotonic B^+-tree (MBT)

The MBT [EJK92] is a specialization of the Time Index [EWK93] for the case where the valid time grows monotonically, i.e., it has the same behavior as of transaction time, in the sense that no retroactive nor predictive updates are supported. We therefore consider it to index transaction time ranges. Similarly to our approach it aims to use a B^+-tree as its framework, although, unlike our approach, the structure of the leaf nodes are quite modified. It also achieves close to 100% node utilization.

The internal nodes of the MBT are like those of a B^+-tree. The leaf nodes however are rather different. The MBT (as the Time Index) makes use of incremental buckets, SP, SM and SC. Associated to every indexed point there is a SP and a SM bucket. A SP bucket contains pointers to all the records (or their "ids") that were inserted at that point in time. Similary SM holds pointers to those there were (logically) deleted. Every leaf node has an SC bucket that contains the records the were valid in the last indexing point of the previous leaf. Figure 9 shows the MBT indexing a data set adapted from [EWK93].



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 9: An example Monotonic B^+-tree.

The major drawback of the MBT, is that the longer the lifespan of a record, i.e., the longer it is recorded in the database, the larger the number of indexing points it will span over. Although the insertion and deletion of such records will appear in only one SP and SM it is replicated through as many SCs as many leaf nodes it spans over. The smaller the node size, the larger the number of leaves and thus of SCs and thus the worse this problem becomes.

Using the MBT to process queries is rather simple. For instance, to find all ranges that intersect a query range $Q = [Q_s, Q_e]$, one needs to calculate all records which are "alive" at Q_s and add all records found in all sets SP in the leaf entries between $[Q_s, Q_e]$. The set of records alive at Q_s is computed by using the SC of the leaf containing Q_s and subtracting all those records in the SMs between the leading entry of this leaf and Q_s.

For the the simulations in this paper we have assumed that every leaf node has associated to it (and physically near, as illustrated in Figure 9) at least one disk block with the incremental buckets. The other option, clustering all such buckets in a single continuous set of blocks could imply in less (but not likely much less) storage but would make query processing time worst as for every leaf node traversed the associated buck would imply in the retrieval of (non-clustered) blocks, thus making it not as efficient as the option we chose.

Relatórios Técnicos – 1996

96-01 **Construção de Interfaces Homem-Computador: Uma Proposta Revisada de Disciplina de Graduação,** *Fábio Nogueira Lucena and Hans K.E. Liesenberg*

96Abs **DCC-IMECC-UNICAMP Technical Reports 1992–1996 Abstracts,** *C. L. Lucchesi and P. J. de Rezende and J.Stolfi*

96-02 **Automatic visualization of two-dimensional cellular complexes,** *Rober Marcone Rosi and Jorge Stolfi*

96-03 **Cartas Náuticas Eletrônicas: Operações e Estruturas de Dados,** *Cleomar M. Marques de Oliveira e Neucimar J. Leite*

96-04 **On the edge-colouring of split graphs,** *Celina M. H. de Figueiredo, João Meidanis and Célia Picinin de Mello*

96-05 **Estudo Comparativo de Métodos para Avaliação de Interfaces Homem-Computador,** *S'ılvio Chan e Heloisa Vieira da Rocha*

96-06 **User Interface Issues in Geographic Information Systems,** *Juliano Lopes de Oliveira and Claudia Bauzer Medeiros*

96-07 **Conjunto fonte máximo em grafos de comparabilidade,** *Marcos Fernando Andrielli e Célia Picinin de Mello*

96-08 **96-08 The Effectiveness of Multi-Level Policing Mechanisms in ATM Traffic Control,** *J.A. Silvester, N. L. S. Fonseca, G. S. Mayor e S. P. S. Sobral*

96-09 **Sequential and Parallel Experimental Results with Bipartite Matching Algorithms,** *João Carlos Setubal*

96-10 **96-10 A CPU for Educational Applications Designed with VHDL and FPGA,** *Nelson V. Augusto, Mario L. Côrtes and Paulo C. Centoducatte*

96-11 **Network Design for the Provision of Distributed Home Theatre Services,** *Nelson L. S. Fonseca, Cristiane M. R. Franco, Frank Schaffa*

96-12 **Modelling the Output Process of an ATM Multiplexer with Correlated Priorities,** *Nelson L. S. Fonseca e John A. silvester*

96-13 **Algoritmos de afinamento tridimensional: exemplos de técnicas de otimização,** *F. N. Bezerra and N. J. Leite*

96-14 **Ensino de Estruturas de Dados e seus Algoritmos através de Implementação com Animações,** *Pedro J. de Rezende e Islene C. Garcia*

96-15 **Sinergia em Desenho de Grafos Usando Springs e Pequenas Heurísticas,** *H. A. D. do Nascimento, C. F. X. de Mendonça N., P. S. de Souza*

96-16 **A Temporal Extension to the Parsimonious Covering Theory,** *Jacques Wainer and Alexandre de Melo Rezende*

96-17 **Workcase-centric workflow model,** *Jacques Wainer and Paulo Barthelmess*

96-18 **Integrating heuristics and spatial databases: a case study,** *Cid Carvalho de Souza, Claudia Bauzer Medeiros, Ricardo S. Pereira*

96-19 **Uma Abordagem de Programação Inteira para o Problema da Triangulação de Custo Mínimo,** *A. P. Nunes e C. C. de Souza*

96-20 **Contracting and Moving Agents in Distributed Applications Based on a Service-Oriented Architecture,** *B.Schulze and E.R.M.Madeira*

Relatórios Técnicos – 1997

97-01 **Um Ambiente Distribuído de Visualização com Suporte para Geometria Projetiva Orientada,** *Pedro J. de Rezende e César N. Gon*

97-02 **Approximate Models for the Output Process of an ATM Multiplexer with Markov Modulated Input,** *Nelson L. S. Fonseca and John A. Silvester*

97-03 **Controle de Concorrência no Cm,** *Célio Norbiato Targa, Mauro da Silva Oliveira Filho, Celso Gonçalves Jr, Rogério Drummond*

97-04 **Compilação Condicional em Cm,** *Sheila P. Maceira, Alexandre Prado Teles, Rogério Drummond*

97-05 **Linear: Linearizador de Estruturas Complexas,** *Rogério Drummond, Carlos Hoyos*

97-06 **LegoShell: Linguagem Visual de Programação Distribuída,** *Rogério Drummond, Celso Gonçalves Jr.*

97-07 **Desenvolvendo Aplicações Distribuídas em Cm,** *Celso Gonçalves Jr., Alexandre Prado Teles, Rogério Drummond*

97-08 **Fast interval branch-and-bound methods for unconstrained global optimization,** *Luiz Henrique de Figueiredo, Ronald Van Iwaarden, Jorge Stolfi*

Instituto de Computação
Universidade Estadual de Campinas
13083-970 – Campinas – SP
BRASIL

`reltec@dcc.unicamp.br`