Ann Math Artif Intell manuscript No.
(will be inserted by the editor)

The RABTree and RAB$^-$Tree: Lean Index Structures for Snapshot Access in Transaction-time Databases

Fabio Grandi

The final publication is available at Springer via http://dx.doi.org/10.1007/s10472-016-9509-7

Abstract In this work we introduce two lean temporal index structures to efficiently support snapshot access (i.e., timeslice queries) in a transaction-time database. The two proposed structures, the RABTree and its RAB$^-$Tree variant, are conceptually simple, easy to implement and efficient index solutions. In particular, the RABTree index guarantees optimal performances for transaction-time data which are naturally clustered according to their insertion time without redundancy. A theoretical and experimental evaluation of the two indexes, in comparison with their previously proposed competitors, is also provided.

Keywords Temporal database · Transaction time · Access methods · Index structures · B-Tree

Mathematics Subject Classification (2000) 68P05 · 68P20

1 Introduction

As an answer to advanced application requirements, temporal databases have been an active scientific field since the early 1980s [1–3]. In particular, several applications require to keep track of the full history of database states in order to be able to sometimes access past versions of database tables, for archiving, accountability or audit purposes. Transaction time [4], which represents when some fact is stored and considered current (i.e., up-to-date) in the database, is the time dimension to be adopted in this case. To this aim, tuples in a transaction-time relation can be timestamped with periods, say [Start, End), where Start is the transaction time when the tuple was inserted and End is the transaction time when the tuple was (possibly) archived, that is updated or deleted. Initially, tuples can be inserted

Fabio Grandi
Dept. of Computer Science and Engineering (DISI),
Alma Mater Studiorum - Università di Bologna
Viale Risorgimento 2, I-40136 Bologna BO, Italy
Tel.: +39-051-2093555
Fax: +39-051-2093953
E-mail: fabio.grandi@unibo.it

with an End value set to "UC" (Until Changed [5]), which denotes still current data. In order to "rollback" the database to its state as of time T, a timeslice query accessing the database snapshot that was current at time T has to be effected [4]. Such a snapshot is made of all the tuples having Start $\leq T <$ End. Support of transaction-time tables, though dubbed as "system-versioned tables", has also been included in the SQL:2011 standard [6] and is nowadays available in mainstream commercial DBMSs [3].

In order to support efficient snapshot access in transaction-time databases, several index structures have been proposed as surveyed in [7] and [8, Sec. 2]. These include the Time-Split B-Tree [9], the Multiversion B-Tree [10], the Append-Only Tree [11], the Time Index [12] and the Snapshot Index [8], which is the perfecting of the structure presented in [13] (we do not consider solutions based on maintenance of information concerning the history of updates in timestamped logs instead of storing explicit tuple versions as produced by updates). In particular, the proposal of the Snapshot Index seemed to settle things once and for all, as it was proven to have asymptotic optimal space requirements and I/O behavior for snapshot access and update. However, the Snapshot Index is a primary file organization, rather complex to implement, which improves the clustering of snapshots along transaction time at the price of introducing data duplication, which may even reach quite high levels. The balance between query efficiency and data redundancy can be tuned, within a limited range, by trimming a design parameter, the usefulness a, to best suit the requirements of a specific application.

In this work, as we did in its preliminary version [14], we take a different direction, renouncing the I/O theoretical optimality of the Snapshot Index to define a lean and more handy index structure, named RABTree, which could anyway demonstrate good performances although non guaranteed optimal in a wide range of settings. In particular, the RABTree is a lean index structure, which does not require invasive reorganizations of data, but exploits as much as possible the natural clustering of data on transaction time induced by their creation order, and does not introduce any redundancy at all. Such a feature could also make it interesting for adoption on solid-state storage as used in mobile devices. An even simpler index structure, leaner but less efficient than the RABTree, is the RAB$^-$Tree variant that will be also presented in this work.

The rest of the paper is organized as follows. In Sec. 2, the RABTree index, and its simplified RAB$^-$Tree variant, are presented. Section 3 is devoted to performance evaluation of the proposed indexes, both from a theoretical and from an experimental viewpoint. In Sec. 4, a comparison of the RABTree and RAB$^-$Tree with previously proposed solutions can be found. Conclusions are finally drawn in Sec. 5.

2 The RABTree Index

In this Section, we introduce a simple temporal index structure which can be used to speed up the execution of access to single snapshots (i.e., timeslice queries) of a transaction-time relation. The proposed index structure, which we will call RAB-Tree, as a shorthand for Right-Append B$^+$-Tree, is very lean, simple to implement and efficient to maintain. Further, we propose an extremely simple variant of the

Table 1 A Synopsis of the Symbols Used

n	number of tuples
N	number of snapshots
S	average snapshot size
$s(T)$	snapshot current at time T
b	disk block size
m	number of current tuples
h	height of RABTree
NL	number of index leaves
r	capacity of a leaf node
s	capacity of an intermediate node
AvgTLL	average TID list length
α	average TID list compression ratio
β	RAB$^-$Tree performance loss ratio

RABTree, called RAB$^-$Tree, which trades off higher query costs with a reduced space requirement.

The transaction-time relation displayed in the left part of Fig. 1 will be used to provide examples. It represents the results of the following sequence of transactions: at time T_0, a transaction creates the first snapshot by inserting the objects with keys from A to F; at time T_1, a transaction updates objects with keys A and C, deletes the object with key F and inserts new objects with keys from G to I; at time T_2, a transaction updates objects with keys C, G and H, deletes the objects with keys A and D and inserts new objects with keys J and K; finally, at time T_3, a transaction updates objects with keys C and J, deletes objects with keys G and K and inserts new objects with keys L and M. Notice that, in Table I, tuples are sorted on the Start attribute according to their original creation order (i.e., the new tuples produced by updates and insertions are simply appended at the end of the file containing the relation).

2.1 RABTree Organization

We can consider the RABTree index as similar to a traditional B$^+$-Tree [15,16] built on transaction time (more precisely, on the Start attribute), as shown in Fig. 2. Leaf index nodes contain an ordered array T of time values and an array of TID lists that can be used to access data tuples. Intermediate index nodes contain an ordered array T of time values and an array Ptr of pointers to children nodes. In both kinds of nodes, the two arrays have the same number of entries which is recorded by the variable Last (this is a first slight difference with respect to the B$^+$-Tree, where intermediate nodes contain one more pointer with respect to key values). The Boolean variable isLeaf can be used to discriminate between the two types of nodes and, in particular, is used during a search to detect when a leaf is reached.

The most relevant difference from a B$^+$-Tree index built on the Start attribute concerns the organization of the TID lists stored in the index leaves. In fact, in the leaf entry $T{:}L$ of a RABTree, the TID list $L{=}P_1, P_2, \ldots, P_\ell$, rather than being the list of pointers to the tuples having T as Start value, is defined as the list of all pointers to the tuples belonging to the snapshot $s(T)$ created at time T. This means that, if a tuple belongs to more than one temporal snapshot, its TID is present

TID	Key	Value	Start	End
P_1	A	10	T_0	T_1
P_2	B	20	T_0	UC
P_3	C	30	T_0	T_1
P_4	D	50	T_0	T_2
P_5	E	45	T_0	UC
P_6	F	25	T_0	T_1
P_7	A	15	T_1	T_2
P_8	C	20	T_1	T_2
P_9	G	30	T_1	T_2
P_{10}	H	10	T_1	T_2
P_{11}	I	15	T_1	UC
P_{12}	C	40	T_2	T_3
P_{13}	G	25	T_2	T_3
P_{14}	H	90	T_2	UC
P_{15}	J	10	T_2	T_3
P_{16}	K	30	T_2	T_3
P_{17}	C	15	T_3	UC
P_{18}	J	25	T_3	UC
P_{19}	L	10	T_3	UC
P_{20}	M	60	T_3	UC

TID	Key	Value	Start	End
P_1	A	10	T_0	T_1
P_2	C	30	T_0	T_1
P_3	F	25	T_0	T_1
P_4	D	50	T_0	T_2
P_5	B	20	T_0	UC
P_6	E	45	T_0	UC
P_7	A	15	T_1	T_2
P_8	C	20	T_1	T_2
P_9	G	30	T_1	T_2
P_{10}	H	10	T_1	T_2
P_{11}	I	15	T_1	UC
P_{12}	C	40	T_2	T_3
P_{13}	G	25	T_2	T_3
P_{14}	J	10	T_2	T_3
P_{15}	K	30	T_2	T_3
P_{16}	H	90	T_2	UC
P_{17}	C	15	T_3	UC
P_{18}	J	25	T_3	UC
P_{19}	L	10	T_3	UC
P_{20}	M	60	T_3	UC

Fig. 1 To the left, a sample transaction-time table. To the right, its "optimized" version (see text). The TID (Tuple IDentifier) column, which is not actually part of the table, shows pointers used for indexing.

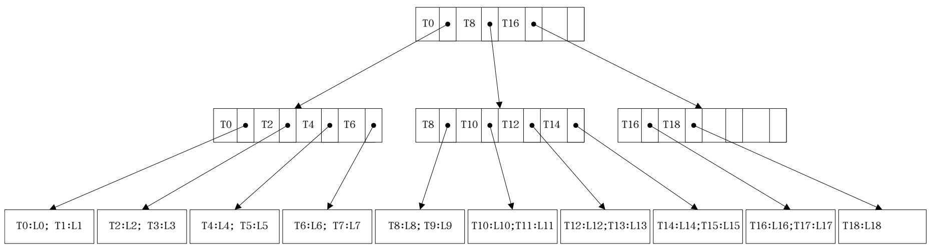

Fig. 2 A RABTree indexing 18 temporal snapshots. In leaf entry T_i:L_i, L_i is the list of pointers (TIDs) to the tuples belonging to the snapshot created at time T_i.

in all the index entries corresponding to the creation time of such snapshots. In order to access the snapshot $s(T)$, the function displayed as Alg. 1 can be called as Search(T, RootPtr), where RootPtr is a pointer to the index root node. Such a function returns the TID list that can be used to access the tuples belonging to $s(T)$. If all the snapshots have been created after T, the function returns a nil result. The root node, which is a leaf node when only a few snapshots have to be indexed, is an intermediate node when more than one index levels become necessary.

A distinctive feature of the RABTree is that TID lists are stored in the index leaves in *compressed* form. For instance, the contents of the RABTree index leaves for the temporal relation displayed in the left part of Fig. 1 are stored in compressed

Algorithm 1 RABTree Search Function

```
function SEARCH(T:Time, P:NodePtr)
    Node:=GETNODE(P);    //read the node pointed by P
    if (T<Node.T[1]) then    //snapshot not present
        return nil
    end if
    i:=1;
    while (i < Node.Last && T $\geq$ Node.T[i+1]) do    //skip entries with too low T
        i++
    end while
    if (Node.isLeaf) then    //leaf reached by recursion
        return Node.TIDList[i]    //TID list found
    else
        SEARCH(T, Node.Ptr[i])    //recursively search one level down
    end if
end function
```

form as follows (where the TID P_i points to the tuple displayed as the i^{th} row in the figure, as shown in the TID column near the table):

$$T_0{:}P1\%5;\ T_1{:}P_2, P_4\%1, P_7\%4;\ T_2{:}P_2, P_5, P_{11}\%5;\ T_3{:}P_2, P_5, P_{11}, P_{14}, P_{17}\%3$$

Actually, in order to save storage space, ranges of consecutive tuples are indexed in a compact way: TID ranges are represented as $P\%n$, that means the tuple pointed by TID P and the n tuples which immediately follow. For instance, the TID list specification "$P_2, P_4\%1$" in the T_1 entry, which contains a singleton TID P2 (pointing to an "isolated" tuple) and a range specification $P_4\%1$, stands for the explicit TID list: P_2, P_4, P_5. For our convenience, we define the TID successor function $\oplus$ as follows: if P is the TID pointing to a tuple t in the relation, then $P \oplus k$ is the TID pointing to the k^{th} tuple coming after t in the relation (i.e., $P \oplus 1$ points to the tuple that immediately follows t and $P \oplus (k+1) = (P \oplus k) \oplus 1$. Hence, $P\%n = \{P, P \oplus 1, P \oplus 2, \ldots, P \oplus n\}$. Assuming the TID range counter k occupies less memory space than a TID, storing range specifications instead of a list of consecutive TIDs leads to space saving even for ranges composed of two TIDs only. Obviously, the saved space grows with the number of TIDs in the range. For instance, assuming a length of 4 bytes for TIDs, 2 bytes for TID range counters and 1 byte for separators (i.e., ":", ",", ";", "%"), the index entry "$T{:}P_1\%4, P_2, P_3, P_4\%3, P_5, P_6\%9;$" occupies 28 bytes less than its explicit expanded version, with a storage space saving around 40% (moreover, being the length of time values fixed, also the separator ":" can be spared).

Obviously, 2 bytes are sufficient to encode TID range counters only if a TID range is composed of at most 65,537 TIDs. In case it may be longer (e.g., as it happens for the exponential growth scenario examined in Sec. 3.3), different separators can be used to introduce variable-length TID range counters. For example, we can also use "\$$n$" or "&$n$" to introduce a 3-byte or 4-byte counter n, respectively. With a 3-byte or 4-byte counter we can represent a range containing up to 16,777,217 or 4,294,967,297 TIDs, respectively. For range counters less than 256, which may progressively become frequent in ageing snapshots, the use of 1-byte counters further improves compression. Management of variable-length TID range counters is a low-level detail and will not be considered in the algorithms that will be presented in the Section that follows, assuming all range specifications have the form "%n" indeed.

A simple *optimization*, which leads to a further space saving, consists in maintaining the transaction-time relation, which is naturally clustered with a primary order on the Start attribute, also sorted with a secondary order on the End attribute as shown in the right part of Fig. 1. This means that, when a transaction archives some tuples in a page, by changing their End attribute from UC to the current transaction time, the page that contains them must be rewritten with such tuples moved before the tuples in the same page which are still current (we can consider such tuple swap operation inexpensive as it can be done in main memory before the page is rewritten). Notice that, in this way, the tuples which are still current in each snapshot will always be found as the last ones of the snapshot itself. Hence, all the current tuples in a snapshot come out consecutive and, if there are at least two thereof, they are referenced in the RABTree leaves by the last item in a TID list specification which always has the form $P\%n$.

With this optimization, the contents of the RABTree index leaves for the temporal relation displayed in the right part of Fig. 1 become:

$$T_0{:}P_1\%5;\ T_1{:}P_4\%7;\ T_2{:}P_5\%1, P_{11}\%5;\ T_3{:}P_5\%1, P_{11}, P_{16}\%4$$

Notice that the reordering of tuples necessary to maintain the secondary order on End *does not affect the previous contents of the index leaves*, as it involves an exchange of tuples and, thus, of their TIDs, all belonging to the same TID range within an index entry (i.e., within all index entries corresponding to all snapshots to which the exchanged tuples belong either before and after the transaction). For example, if a transaction executed at time T_4 deletes the object with key E from the relation displayed in the right part of Fig. 1, the 6th tuple (E, 45, T_0, UC) is changed to (E, 45, T_0, T_4) to be archived and must be exchanged with the 5th tuple (B, 20, T_0, UC) to maintain the secondary order. However, although their position and, thus, their TIDs P_5 and P_6 are swapped, both tuples are still contained in the TID range $P_1\%5$ of T_0, $P_4\%7$ of T_1 and $P_5\%1$ of T_2 and T_3, where they already were before the execution of the transaction at T_4. The destinies of the two tuples only separate from T_4 onwards, as the new 6th tuple (with B) is still current and belongs to the new snapshot, whereas the new 5th tuple (with E) does not, as recorded by new index entry $T_4{:}P_6, P_{11}, P_{16}\%4$.

Besides its very simple organization shown in Fig. 2, the proposed index leaf compression and the optimized storage technique are the key ingredients of the practical usefulness of the RABTree. In fact, we will show in Sec. 3 that the resulting data clustering (primary order on Start and secondary order on End), which is quite inexpensive to maintain, gives rise to very high TID list compression ratios, which in turn makes the RABTree index a really lean and quite efficient index structure.

As an alternative to this compression scheme, we could also use PIDs (Page IDentifiers) instead of TIDs in the index leaves (from the query processing viewpoint, as long as complete snapshots are retrieved, there is actually no difference in using TIDs rather than PIDs). With this option, in the index entries, a TID range is substituted by a PID range when the tuple range spans more than one consecutive pages or by a singleton PID when the tuple range is all contained in a single page (improving, thus, the compression rate).

2.2 RAB$^-$Tree Organization

The RAB$^-$Tree variant differs from the RABTree structure previously described by the contents of the leaves (and the shorter height which originates from that). In particular, a RAB$^-$Tree index entry has the simple form $T{:}P$, where P is the TID pointing to the first tuple belonging to the snapshot $s(T)$. For instance, for the sample transaction-time relation displayed in Fig. 1 (either in its basic or optimized form), the contents of the RAB$^-$Tree index leaves is:

$$T_0{:}P_1;\ T_1{:}P_4;\ T_2{:}P_5;\ T_3{:}P_5$$

In practice, whereas a RABTree globally stores in the leaves a number of TIDs which is at least $\mathcal{O}(n)$, a RAB$^-$Tree stores only $\mathcal{O}(N)$ TIDs, where n is the number of tuples and N is the number of snapshots or modification transactions applied. Hence, for the same relation, the RAB$^-$Tree is usually much smaller (with less leaves and, thus, also less levels) than the corresponding RABTree. Since the RAB$^-$Tree entries contain a singleton TID and, thus, have all the same composition and length, no compression like the RABTree is required and also alternating time values and TIDs can be packed in the index leaves without the need for separators (i.e., either ":"s and ";"s can be spared).

During the processing of a timeslice query retrieving the snapshot $s(T)$, the RAB$^-$Tree is searched to locate the leaf containing the consecutive entries $T_i{:}P_i$; $T_{i+1}{:}P_{i+1}$ such that $T_i \leq T < T_{i+1}$. Then, the query is answered by sequentially accessing the range of consecutive disk pages starting from the one containing the tuple pointed by P_i until the last tuple with Start $< T_{i+1}$ is found (tuples with End $\leq T$ are discarded). The disadvantage of using the RAB$^-$Tree instead of the RABTree is that, although the first and the last pages accessed are the same, all the pages in the range are sequentially accessed. Indeed, the RABTree allows to skip the pages in the range that do not contain any qualifying tuple usually reducing, thus, the overall access cost.

2.3 RABTree (RAB$^-$Tree) Maintenance

Despite its structural similarity (see Fig. 2), the RABTree is not managed as a B$^+$-Tree as far as balancing algorithms are concerned, because it is more convenient to exploit its right-append property, owing to the fact that transaction time values are naturally inserted in ascending order. In fact, with respect to the traditional B$^+$-Tree organization, creation of a new snapshot involve the insertion of a T value always greater than the values already stored and, thus, an insertion algorithm that involves only the rightmost nodes at each level can be exploited. In this way, new entries are always added to the rightmost leaf node and index balancing (as well as optimal memory occupancy) is guaranteed by the procedure described in the following.

When a transaction executed at time T updates the indexed relation, a new index entry $T{:}L$ is appended to the last (i.e., rightmost) index leaf, if there is room. If the last leaf is completely full, a new leaf is allocated to contain the new entry and the transaction start T is also appended to the parent node of the former last leaf, that is to the rightmost node at the previous index level, together with a pointer to the newly created leaf. Notice that no entries are moved from the full leaf to the newly created one. Therefore, we cannot consider the event the full leaf undergoes as a classical split of the node but rather as the addition of a new node to accept its overflow entries. As in a standard B$^+$-Tree, the addition of the new node with the new time-pointer pair copied to the parent node can cause an overflow also there. If the last node at each of the levels is found full, the overflow propagates up to the root, causing the creation of a new root and the growth of the index tree structure by one level. After such an event, the newly created root contains two entries: the oldest time value in the index (i.e., the first values present in the old root) with a pointer to the old root and the latest time value in the index (i.e., the newly inserted value which triggered the overflow chain from the leaf to the root) with a pointer to the newly created sibling of the old root. After such an event, all the rightmost nodes at any level below the root will contain a single entry. The complete pseudocode for the insertion procedure is presented as Alg. 2. The array RPtr is a global data structure containing pointers to the rightmost nodes at each level of the RABTree (RPtr[1]=RootPtr and RPtr[h] is the pointer to the rightmost leaf, being h the height of the RABTree).

Such a function must be called, during the transaction creating a new snapshot, as InsertEntry(T, L, h), being T:L the new index entry that must be inserted in the rightmost leaf of the RABTree, where T is the current transaction time (which is also the Start value of the newly added tuples) and L is the list of TIDs pointing to the tuples belonging to the new current snapshot. The new TID list L must be computed in parallel to the update of the indexed temporal relation according to the pseudocode exemplified as Alg. 3. The base for the creation of the new index entry T:L is the last entry present in the rightmost index leaf before the update. The variable LWork is used by the algorithm to store the TID list of such an entry for further processing.

In particular, for each of the modified tuples, if its TID P is present in LWork as a singleton, it can simply be removed. On the contrary, if P belongs to a range specification $Q_0\%n = \{Q_0, Q_1, Q_2, \ldots, Q_n\}$ in LWork, first of all, it has to be determined which one of the Q_i is actually P. Then, in order to remove P, the range specification $Q_0\%n$ must be, in general, split and substituted by the range specifications $Q_0\%(i-1) = \{Q_0, Q_1, \ldots, Q_{i-1}\}$ and $Q_{i+1}\%(n-i-1) = \{Q_{i+1}, \ldots, Q_n\}$. Notice that, depending on the value of i, any of the two lists could rather be a singleton TID or one of the two can even be empty: the algorithm is designed to correctly deal also with these cases and always produce consistent TID list specifications. If a single tuple has been modified in the ForEach loop, PLast$\oplus$1 is added as a singleton TID to the new index entry. If NMod>1 tuples have been modified, new version tuples have been appended to the relation with TIDs: PLast$\oplus$1, PLast$\oplus$2,. . . , PLast$\oplus$NMod, which are added as a new TID range specification PLast$\oplus$1%(NMod$-$1) to the new index entry. Notice that NMod is also augmented by the number NIns of new tuples inserted by the transaction, in order to include the count of their TIDs in the new TID range. Finally, the last two (singleton TID or range) specifications in LWork, if consecutive, have to be merged into a single range.

Notice that such an algorithm is designed to correctly work either if the underlying transaction-time table is optimized or not. In case it is optimized, it assumes the TIDs of the archived tuples are from the final configuration, where the secondary clustering has already been enforced. Notice that, in such a case,

Algorithm 2 RABTree Insertion Function

```
function InsertEntry(T:Time, L:TIDList, Lev:int)
    Node:=GetNode(RPtr[Lev]);    //rightmost node at level Lev
    if (Node.isLeaf) then
        NMax:=r    //capacity of a leaf node
    else
        NMax:=s    //capacity of an intermediate node
    end if
    if (Node.Last<NMax) then    //room enough; insert the new entry here
        Node.Last++; Node.T[Node.Last]:=T;
        if (Node.isLeaf) then
            Node.TIDList[Node.Last]:=L
        else
            Node.Ptr[Node.Last]:=RPtr[Lev+1]    //pointer to the rightmost node below
        end if
        RewriteNode(RPtr[Lev], Node)    //rewrite in place
    else    //overflow; create a sibling of the node and insert the new entry there
        NewNode.Last:=1; NewNode.T[1]:=T;
        if (Node.isLeaf) then
            NewNode.TIDList[1]:=L
        else
            NewNode.Ptr[1]:=RPtr[Lev+1]    //pointer to the rightmost node below
        end if
        NewNode.isLeaf:=Node.isLeaf;    //same type of the sibling
        NewP:=WriteNewNode(NewNode);
        RPtr[Lev]:=NewP;    //the new node is now the rightmost
        if (Lev>1) then    //not the root; insert the new entry at the upper level too
            InsertEntry(T, L, Level-1)    //recursive call
        else    //create a new root
            NewRoot.isLeaf:=false; NewRoot.Last=2;
            NewRoot.T[1]:=Node.T[1]; NewRoot.Ptr[1]:=RootPtr;
            NewRoot.T[2]:=T; NewRoot.Ptr[2]:=NewP;
            $h$++;    //RPtr must take into account the increased height
            for ($i$:=$h$; $i$ >1; $i$ −−) do
                RPtr[$i$]:=RPtr[$i$-1]
            end for
            RPtr[1]:=WriteNewNode(NewRoot);
            RootPtr:=RPtr[1];    //update also the root pointer
        end if
    end if
end function
```

the algorithm could also be slightly improved if executed step by step in parallel to the archival of tuples in the relation.

In the case of a RAB$^-$Tree index, the update procedure is rather trivial: for each update transaction, it is sufficient to add a new entry $T{:}P$ to the rightmost leaf, where T is the current transaction time and P is the TID of the first (with respect to the Start order) tuple in the formerly current snapshot non archived by the transaction itself (i.e., thus, still current).

2.4 KT-Point, Range and History Query Support

The RABTree (RAB$^-$Tree) index is built on transaction time to support timeslice queries (snapshot access) and is of no use for data access by a non-temporal primary key. Obviously, it can be used to support a “thick” timeslice query, that

Algorithm 3 Index Leaf Update Procedure

```
Leaf := GetNode(RPtr[H]);    //rightmost leaf node
LWork := Leaf.TIDList[Leaf.Last];    //TID list of the last index entry
PLast := GetLastTID(LWork);    //TID of the last tuple in the relation
NMod := 0;    //counter of modified tuples
for all archived tuples do    //i.e., with End=T
    NMod++;
    P := TID of the updated tuple;
    if P in LWork then    //as a singleton TID
        LWork −= P
    else    //P belongs to the range Q%n in LWork
        Find the range Q%$n$ to which P belongs;
        Find $k \in [0..n]$ such that P=Q$\oplus k$;
        LWork −= {Q%$n$};
        if ($k$ >1) then
            LWork += {Q%($k-1$)}    //range
        else if $k$=1 then
            LWork += {Q}    //singleton
        end if
        if ($k < n-1$) then
            LWork += {Q$\oplus$($k+1$)%($n-k-1$)}    //range
        else if ($k = n-1$) then
            LWork += {Q$\oplus n$}    //singleton
        end if
    end if
end for
NMod += NIns;    //NIns = number of new inserted tuples
if (NMod=1) then
    LWork += {PLast$\oplus$1}    //singleton
else
    LWork += {PLast$\oplus$1%(NMod−1)}    //range
end if
Merge the last two specifications in LWork if consecutive;
InsertEntry($T$, LWork, $h$)    //append T:LWork to the rightmost index leaf
```

is a query accessing a range of consecutive snapshots $\{s(T) \mid T' \leq T \leq T''\}$ like a traditional B$^+$-Tree: once, starting from the root, the index entry corresponding to T' is found in a leaf to access the first snapshot $s(T')$, the index entries that immediately follow in the same leaf or in the successive leaves (with an extra pointer per leaf, leaf nodes can be connected as a linked list like in a standard B$^+$-Tree) can be used to access the intermediate snapshots until the entry corresponding to T'' is found and used to access the last snapshot $s(T'')$. If the results do not need to be grouped by snapshot, in order to save repeated block accesses, all the collected TID lists can be merged and duplicate TIDs eliminated before accessing data on secondary memory.

In order to efficiently support other types of queries, like KT-point queries (i.e., find the version of the data object identified by K as of time T) or history queries (i.e., find all the temporal versions of the data object identified by K), the RABTree (RAB$^-$Tree) index can be flanked by a companion B$^+$-Tree index built on the non-temporal primary key. In particular, we can consider two solutions that best fit different application requirements: a B$^+$-Tree built on the key indexing only current tuples (as in a traditional, non-temporal, database) or a B$^+$-Tree built on the key indexing all the tuples in the relation. In both cases, an off-the-shelf standard B$^+$-Tree can be used. In the former case, the resulting B$^+$-Tree is an unclustered unique index that not only supports point and range access by primary key to current tuples but also efficiently supports updates (cf. only current tuples can be updated or deleted; updating or deleting a data objects requires its current version to be archived). In the latter case, the resulting B$^+$-Tree is an unclustered non-unique index. For each key value stored in a leaf, the associated TID list points to all the temporal versions of the data object identified by such a key. Thus, such a list can be used to solve a history query (if the TID list is maintained ordered, versions are retrieved in ascending time order thanks to the data clustering). Such a list can also be intersected before access with a TID list found in the RABTree in order to retrieve the desired tuple only, when processing a KT-point query. Intersection of B$^+$-Tree and RABTree TID lists before accessing data can also be used to support KT-range queries and partial history queries. Updates are also supported as the pointer to the current version is the last one in the B$^+$-Tree TID list, unless the data object has been deleted (in order to ensure before access that the last version is still current we can check its TID also belongs to the last TID list in the rightmost leaf of the RABTree).

The RAB$^-$Tree is of less utility for non timeslice queries. Assume we want to access the version as of T of the object with key K and, on purpose, we find the entry $K : P_1, P_2, \ldots, P_n$ in a B$^+$-Tree indexing all the data tuples and the consecutive entries T_i:P_i; T_{i+1}:P_{i+1} ($T_i \leq T < T_{i+1}$) in the RAB$^-$Tree. Thanks to the RAB$^-$Tree we can only discard the first TIDs $P_1, \ldots, P_j$ such that $P_j < P_i$ (cf. P_i points to the first inserted tuple belonging to $s(T)$ and, thus, the first j tuples with key K were inserted before $s(T)$ was created) but then we have to possibly use all the TIDs that follow, starting from P_{j+1}, to read the pointed tuples until we find the one with Start$\leq T <$End. On the other hand, a B$^+$-Tree only indexing current data could still be useful for queries involving the current versions of data objects and updates.

Usefulness of the RABTree/RAB$^-$Tree in supporting other types of queries (e.g., temporal joins or the evaluation of temporal aggregates) will be explored in future work.

3 Performance Evaluation

In this Section, we present both theoretical and experimental results concerning the assessment of the performances of the RABTree and RAB$^-$Tree index structures.

3.1 Performance Figures

As far as space occupancy is concerned, at each level of the index from the root to the leaves, all the nodes but the rightmost one are always completely full, whereas the rightmost one can contain a number of entries ranging from 1 (right after its creation) to its maximum capacity (right before its overflow), which we define as r for leaf nodes and s for intermediate nodes. However, the impact of rightmost nodes on the global space utilization is not very relevant, as shown in the following. Such rightmost nodes are as many as the index levels, that is about $\log_s NL$, where NL is the number of leaf nodes (a more precise estimate can be found below). We can assume an average space utilization of 0.5 for such rightmost nodes, as there is one such node per level, whose utilization takes all values from single entry to full node with increments caused by the same number of index insertions (i.e., all the ones accommodated in the leaves of a subtree rooted on the node). Thus, since an index with NL leaves contains a total of about 2 NL nodes, the global space utilization for the whole index can be computed via a weighted average as $[1 \times (2NL - \log_s NL) + 0.5 \log_s NL]/(2NL)$, that is $1 - 0.25 \log_s NL/NL$, which approximates 1 as NL grows.

The maximum capacity, that equals the fan out, of an intermediate node can be computed as $s = \lfloor D/(\mathrm{L(time)} + \mathrm{L(ptr)}) \rfloor$, where D is the intermediate node dimension and L(time) and L(ptr) are the length of a time value and of a pointer, respectively. If a node size of one disk block is used, then $D = b$. The minimum height can be computed when all nodes at each level are completely full and, thus, we have: 1 root node at level 1, s nodes at level 2, s^2 at level 3, ..., s^{i-1} at level i. The maximum height can be computed when the root contains 2 entries only and all nodes at any other level are completely full but the rightmost one which contains 1 entry only and, thus, we have: 1 root node at level 1, 2 nodes at level 2, $s+1$ nodes at level 3, s^2+1 at level 4, ..., $s^{i-2}+1$ at level i. With the constraint that the level $h-1$ contains enough pointers to address all the NL leaves, we can obtain the precise boundaries: $1 + \lceil \log_s NL \rceil \leq h \leq 2 + \lfloor \log_s (NL-1) \rfloor$.

The number of leaves NL of a RABTree index can be easily computed from the leaf node size D' and L(time), once the average length AvgTLL of a TID list specification is known, as $NL = \lceil N(\mathrm{L(time)} + \mathrm{AvgTLL})/D' \rceil$, where N is the number of stored snapshots. In the normal case, we assume that more index entries can be packed together in a leaf (as in Fig. 2) and, thus, $NL < N$. If the average size of the index entries is limited by a constant, we can always obtain $NL < N$ by choosing a suitably big dimension D' for the leaf nodes. However, in the case in which the average index entry size is not limited by a constant (which may happen, e.g., if AvgTLL grows with time), since the index above the leaves is built on the snapshot creation times and can at most address N distinct leaf blocks, the only viable solution is to employ variable-size leaf blocks: when a new index entry has to be inserted, new pages are added to the rightmost leaf until the whole entry can be accommodated. A pathological case, which can be an issue for space costs, occurs when AvgTLL grows at a higher rate than the file size n and will be discussed in Sec. 3.2. Without TID list compression, the number of leaves of the RABTree would be $\lceil N\big(\mathrm{L(time)} + S(\mathrm{L(TID)} + \mathrm{L(sep)})\big)/D' \rceil$, where $S = \sum_{i=1}^{N} |s(T_i)|/N$ is the average snapshot size. Hence, we can introduce a parameter $\alpha < 1$ that measures the average degree of compression that is obtained with the techniques presented in Sec. 2.1 and write, in general, AvgTLL= $\alpha\ S(\mathrm{L(TID)} + \mathrm{L(sep)})$. The compression ratio α depends on the number of versions spanned by each tuple and on the interleaving of tuples belonging to different snapshots in the same blocks. More insight on the values normally assumed by the α parameter will be given in the experimental evaluation section (Sec. 3.3) that follows. For a RAB$^-$Tree index, the number of leaves can be computed as $NL = \lceil N(\mathrm{L(time)}+\mathrm{L(TID)})/D' \rceil$. Notice also that the whole capacity of leaf blocks is exploited in a RABTree or RAB$^-$Tree, as all leaves but the last one are completely full (whereas all leaves have a 0.69 average occupancy in a B$^+$-Tree).

In asymptotic terms, the NL formulae above translate into space requirements which are $\mathcal{O}(\alpha SN/b)$ for the RABTree and $\mathcal{O}(N/b)$ for the RAB$^-$Tree. If the snapshot size remains limited by a constant, then $S = \mathcal{O}(n/N)$ and, thus, the

RABTree presents a guaranteed $\mathcal{O}(\alpha n/b)$ space cost. If the snapshot size grows with time, its average size S could be at most $\mathcal{O}(n)$ but its growth can still be compensated by the compression of the TID lists yielding $\alpha S = \mathcal{O}(n/N)$ and, thus, a space cost still no higher than $\mathcal{O}(n/b)$. In Sec. 3.2, we will characterize the theoretical worst case for the growth of NL, which is $\mathcal{O}(N^2/b)$. However, this is not sufficient to yield a more than linear growth of the index leaves in terms of the file size n, as it also depends on the growth rate of n in terms of N (e.g., if also $n = \mathcal{O}(N^2)$ we still have an $\mathcal{O}(n/b)$ space cost). Space requirements of a RAB$^-$Tree are $\mathcal{O}(N/b)$. The RAB$^-$Tree leaves represent a small fraction of the space occupied by the indexed relation, fraction that decreases with N if the file size grows more than linearly in N.

The height h represents the index access cost for the execution of a timeslice query (assuming for the RABTree that the associated TID list specification is contained in a single leaf), which is $h = \mathcal{O}(\log_b N)$ either for the RABTree and for the RAB$^-$Tree. The number of data accesses required to retrieve the snapshot $s(T)$ is less or equal to $|s(T)|$ for the RABTree (it is less when some of the tuples of $s(T)$ are in the same block). This yields a number of I/Os which is $\mathcal{O}(\log_b N + |s(T)|)$ for answering a timeslice query, even if the TID list associated with T spans several blocks: in such a case the second term $|s(T)|$ also absorbs the cost paid for accessing the index leaves. With the RAB$^-$Tree, we may expect a number of I/Os which is $\mathcal{O}(\log_b N + \beta|s(T)|)$, where β is some number greater than 1 that takes into account the interleaving of tuples belonging to different snapshots within the same blocks (i.e., it is an inverse measure of snapshot clustering). Some experimental evaluation of β will be reported in the section 3.3 that follows. However, in the worst case, which happens when a tuple inserted in the first block is never deleted or updated, so that it always belongs to all the snapshots, the RAB$^-$Tree requires a full scan of the relation (i.e., $\mathcal{O}(\log_b N + n/b)$ I/Os) to retrieve the current snapshot, and about $\mathcal{O}(\log_b N + T/N \cdot n/b)$ I/Os to retrieve the snapshot $s(T)$. Since the most frequently used snapshot is certainly the current one, which is accessed by all modification transactions and by a large fraction of read-only queries (indeed, "rollback" queries are very likely seldom operations, mainly effected for accountability or audit purposes), the full scan required by the RAB$^-$Tree in the worst case could be a problem. Thus, also in order to overcome such drawback and provide fast access to the current snapshot (also with the RABTree index), we can build a standard B$^+$-Tree on the key to index the current tuples only as suggested in Sec. 2.4.

As far as update costs are concerned, a transaction creating a new current snapshot requires reading and rewriting of the rightmost index leaf only, unless the leaf is full and the creation of new index nodes at higher levels is triggered. In general, we can estimate the average update cost for a RABTree as follows, by considering the transition from a completely full index with height h to a completely full index with height $h + 1$. In the initial state of such transition, the RABTree has $1 + s + s^2 + \ldots + s^{h-2} = (s^{h-1} - 1)/(s - 1)$ intermediate nodes and s^{h-1} leaves, whereas in the final state has $1 + s + s^2 + \ldots + s^{h-1} = (s^h - 1)/(s - 1)$ intermediate nodes and s^h leaves. Hence, during the whole transition, s^{h-1} new intermediate nodes and $s^{h-1}(s - 1)$ new leaf nodes are created. Each intermediate node is written a first time to insert the first entry when the node is created and then read and rewritten to insert each of the $s - 1$ remaining entries, with a total cost of $2s - 1$ I/Os to get full. Similarly, if r is the average number of entries that can be inserted in a leaf node, each leaf node needs $2r-1$ I/Os to get filled. This yields a total cost of $s^{h-1}(2s-1)+s^{h-1}(s-1)(2r-1) = s^{h-1}[2s-1+(s-1)(2r-1)]$ for the whole transition. The new entries involved in the transitions are the ones needed to fill up all the new index leaves, that is r for each leaf yielding a total of $s^{h-1}(s-1)r$ new entries. Thus, the average cost for the insertion of a single entry can be evaluated over the transition as the ratio between the total cost and the number of inserted entries, that is $(2s-1)/[(s-1)r]+(2r-1)/r$. For the RAB$^-$Tree, if L(ptr)=L(TID) and $D = D'$ are chosen, then intermediate and leaf nodes have the same capacity and, thus, the average update cost becomes $(2s-1)/(s-1)$. In asymptotic terms, the average update cost is $\mathcal{O}(1)$ both for the RABTree and for the RAB$^-$Tree. For the modification of data, the tuples to be updated or deleted must be retrieved in the current snapshot and rewritten, in order to be archived. If the B$^+$-Tree indexing current tuples by the key is available, locating each of the qualifying tuples will cost $\mathcal{O}(\log_b m)$, where m is the size of the current snapshot ($m = \mathcal{O}(n)$ in the worst case). Otherwise, all the tuples qualifying for the transaction can be found by accessing once the whole current version of the relation with the RABTree or RAB$^-$Tree index, at the cost of a timeslice query. Newly inserted tuples and the new versions of updated tuples must be appended at the end of the data file.

Finally, the cost of constructing a RABTree/RAB$^-$Tree from scratch (bulk loading) is $\mathcal{O}(n/b)$, since all the blocks making up the file to be indexed have to be read once (and some of them rewritten in place to enforce the secondary order on End attribute in the case of the RABTree).

3.2 Worst-case Space Cost Analysis

In order to characterize the worst case for space occupied by index leaves, we start with the proposition that follows that states an upper bound on the number of items in a TID list.

Proposition 1 *In a RABTree with compressed index leaves indexing a transaction-time relation with optimized storage (primary order on Start and secondary order on End), the number of items in the TID list L of the leaf entry T:L is bounded by the number of snapshots stored at time T.*

Proof Let $\bar{T} < T$ be the transaction time of creation of any of the snapshots but the last one in the relation and consider the tuples with Start=$\bar{T}$, that is the new versions of the tuples updated plus the fresh new tuples inserted by the transaction executed at time $\bar{T}$. At time T, some of such tuples might have been archived (i.e., deleted or updated) by some transaction executed between $\bar{T}$ and T and, thus, have the End attribute set to some transaction time between $\bar{T}$ and T, whereas the rest of the tuples are still current. The still current tuples have End=UC, belong to $s(T)$ and, thus, must be indexed by the index entry T:L. Thanks to the secondary ordering on End inherent to the optimized storage technique, such tuples are all consecutive (and placed after the other tuples with Start=$\bar{T}$ and End<UC) and, thus, can be indexed by a single item $P\%n$ of the index entry T:L (or by a singleton pointer if there is only one such tuple). In other words, in T:L there cannot exist more than one index entries pointing to tuples with the same Start value without violating the secondary order constraint.

In the worst case for the length of the index entry $T{:}L$, $s(T)$ is composed of still current tuples inserted by every transaction executed before T (i.e., $s(T)$ shares tuples with every preceding snapshots). Since we just proved that to each group of still current tuples inserted by such a transaction correspond at most one entry in in the TID list $T{:}L$, the total number of items (TID ranges or singleton pointers) in $T{:}L$ is limited by M, where M is the number of transactions executed before T. $\square$

Notice that the fresh new tuples inserted by the transaction executed at time T are all current at time T. Therefore, if $M > 1$, the TID list entry corresponding to such tuples can be merged in $T{:}L$ to the TID list entry corresponding to the last transaction executed before T (i.e., the last considered in the proof above), as it is done at the end of Alg. 3, and, thus, does not contribute to the count of items in $T{:}L$ even in the worst case. Hence, if $M > 1$, the items in $T{:}L$ are actually at most $M - 1$, where M is the number of snapshots stored at time T. As a consequence, the worst case for the *NL* growth in terms of N is $\mathcal{O}(N^2/b)$.

Notice that, if the snapshot size $s(T)$ remains always bounded by a constant $\bar{S}$ (like in the stationary growth scenario presented in Sec. 3.3), also the number of entries in any $T{:}L$ index entry is bounded by $\bar{S}$ (in the worst case, it contains all singleton pointers). Hence, both the file size and the number of leaves in the RABTree index linearly grow with the number of snapshots N.

The worst case occurs when the TID list growth matches the upper bound stated in Prop. 1 in the presence of the minimal growth of the file size that can cause it. It is not difficult to see that this happens with a pathological relation like the one in Fig. 3 (which is a special variant of the linear growth scenario presented in Sec. 3.3). Starting from an initial snapshot $s(T_0)$ containing two tuples, each transaction adds two new tuples and deletes one of the tuples inserted by the immediately preceding transaction. Such a modification pattern causes one tuple belonging to $s(T)$ to remain current forever and, thus, also belonging to each snapshot that follows $s(T)$. As a consequence, the snapshot size grows by one tuple per transaction while the overall file size growth remains linear. On the other hand, the RABTree size grows quadratically. In fact, after N transactions have been executed, we have $|s(T_{N-1})| = N + 2$ and $n = 2N$ and the contents of the index leaves is as follows:

$$T_0{:}P1\%1;\ T_1{:}P_2\%2;\ T_2{:}P_2, P_4\%2;\ T_3{:}P_2, P_4, P_6\%2;\ \ldots$$
$$\ldots, T_{N-2}{:}P_2, P_4, \ldots, P_{2N-4}, P_{2N-2}\%2;\ T_{N-1}{:}P_2, \ldots, P_{2N-2}, P_{2N}\%2$$

Hence, the total number of items in the TID lists grows as $1 + \sum_{M=1}^{N-1} M = 1 + (N-1)N/2$ (N thereof are range specifications and the rest are singletons). The worst case average length can be easily computed as AvgTLL= $[(N-1)/2 + 1/N][$ L(TID) + L(sep) $]+$ L(range) $= (n/4 - 1/2 + 2/n)[$ L(TID) + L(sep) $]+$ L(range). Therefore, $NL = \mathcal{O}(n^2/b)$.

3.3 Experimental Evaluation

In this Section, we present some results concerning the experimental evaluation of the RABTree and RAB$^-$Tree index structures. In our experimental settings, we considered three rather different evolution scenarios for a transaction-time relation.

TID	**Key**	**Start**	**End**
P_1	K_1	T_0	UC
P_2	K_2	T_0	UC

(at time T_0)

TID	**Key**	**Start**	**End**
P_1	K_1	T_0	T_1
P_2	K_2	T_0	UC
P_3	K_3	T_1	UC
P_4	K_4	T_1	UC

(at time T_1)

TID	**Key**	**Start**	**End**
P_1	K_1	T_0	T_1
P_2	K_2	T_0	UC
P_3	K_3	T_1	T_2
P_4	K_4	T_1	UC
P_5	K_5	T_2	UC
P_6	K_6	T_2	UC

(at time T_2)

....

TID	**Key**	**Start**	**End**
P_1	K_1	T_0	T_1
P_2	K_2	T_0	UC
P_3	K_3	T_1	T_2
P_4	K_4	T_1	UC
P_5	K_5	T_2	T_3
P_6	K_6	T_2	UC
P_7	K_7	T_3	T_4
P_8	K_8	T_3	UC
...	...	...	...
P_{2N-1}	K_{2N-1}	T_{N-2}	T_{N-1}
P_{2N}	K_{2N}	T_{N-2}	UC
P_{2N+1}	K_{2N+1}	T_{N-1}	UC
P_{2N+2}	K_{2N+2}	T_{N-1}	UC

(at time T_{N-1})

Fig. 3 The pathological relation giving rise to worst-case growth of the RABTree.

In the first one, named **stationary growth** scenario, starting from an initial state $s(T_0)$ with size S_0, only updates are applied to tuples in the relation and/or deletions are exactly balanced by insertions, such that the size of any successive snapshot remains constant (i.e., $|s(T_i)| = S_0$ for each $T_i \geq T_0$). The temporal relation grows linearly with respect to time owing to the addition of new snapshots. In the second one, named **linear growth** scenario, starting from the same initial state $s(T_0)$, each modification transaction applies a constant number (on average) of insertions, deletions and updates, with a positive balance between insertions and deletions, to the tuples making up the current snapshot. As a result, the size of any successive snapshot grows linearly (i.e., $|s(T_i)| = S_0 + K(T_i - T_0)$, for each $T_i \geq T_0$, where K is proportional to the average number of insertions minus deletions) and the temporal relation size undergoes a polynomial (quadratic) growth with respect to time. In the third scenario, named **exponential growth** scenario, starting from the initial state, a constant (on average) rate of deletions, insertions and updates, with a positive balance between insertions and deletions, is applied to the tuples making up the current snapshot (i.e., the newly inserted tuples, the deleted and the updated ones are a fixed fraction of the current tuples) so that the size of the current snapshot exponentially increases with time due to a positive feedback effect (i.e., $|s(T_i)| = S_0\, H^{(T_i - T_0)}$, for each $T_i \geq T_0$, where H depends on the average rate of insertions minus deletions). The temporal relation size also grows exponentially with respect to time.

In particular, starting from an initial state containing 1,000 tuples, by executing 2,500 transactions each of which updates an average 50% of current tuples (i.e., each current tuple has a probability 0.5 of being updated), we end up with a total of about 1,170,000 tuples in the stationary growth scenario. In the linear growth scenario, starting from the same initial state, we executed 7,000 trans-

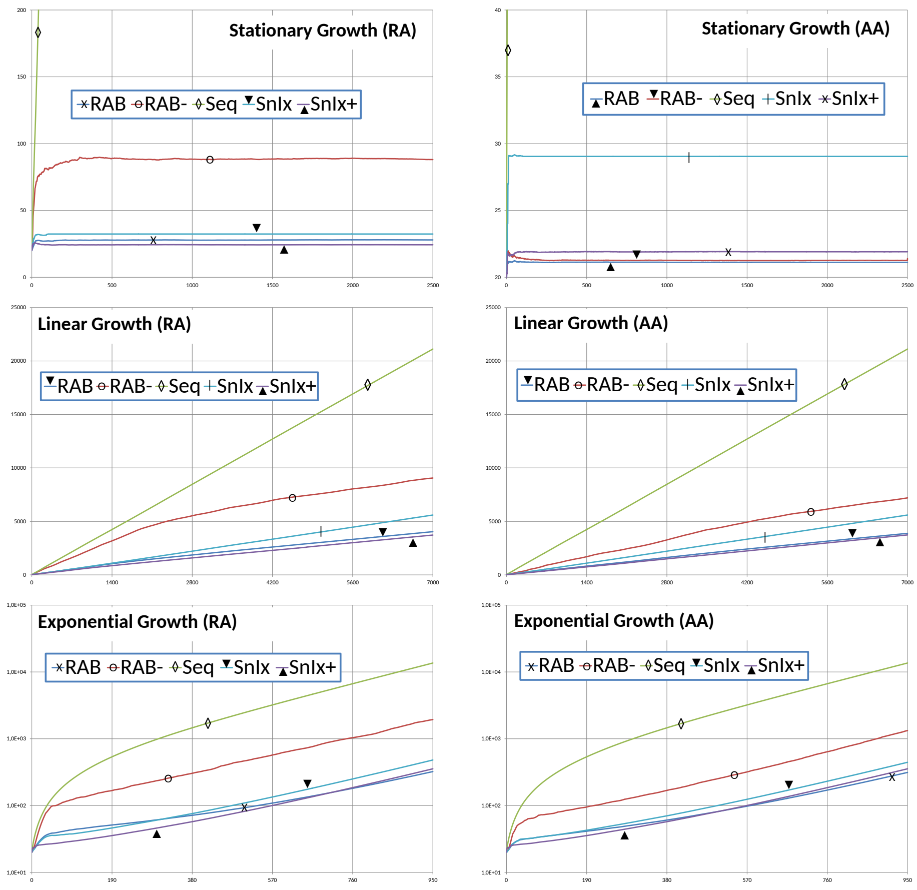

Fig. 4 Average snapshot access costs in different settings.

actions each of which applies a random number of deletions, insertions and updates uniformly distributed between 0 and a fixed bound. In practice, the bounds for deletions/insertions/updates were chosen as 100/200/400, respectively, so that each transaction on average executes 50/100/200 thereof, ending up with a relation containing about 2,100,000 tuples after the execution of 7000 transactions. Finally, in the exponential growth scenario, starting from the same initial state and executing 950 transactions each of which deletes a 1%, inserts a 2% and updates a 20% of current tuples, we end up with about 3,100,000 tuples.

Moreover, for each evolution scenario, we considered two variants according to the way the tuples to be archived (i.e., updated or deleted) by a transaction are selected. In the **random archival (RA)** variant, tuples to be archived are randomly selected in the current snapshot, that is every current tuple has the same probability to be archived. In the **ageing archival (AA)** variant, tuples to be archived are chosen according to their ageing, that is the probability of current tuples to be archived is proportional to their permanence time within the system. This corresponds to assume that the occurrences of modifications to a data object

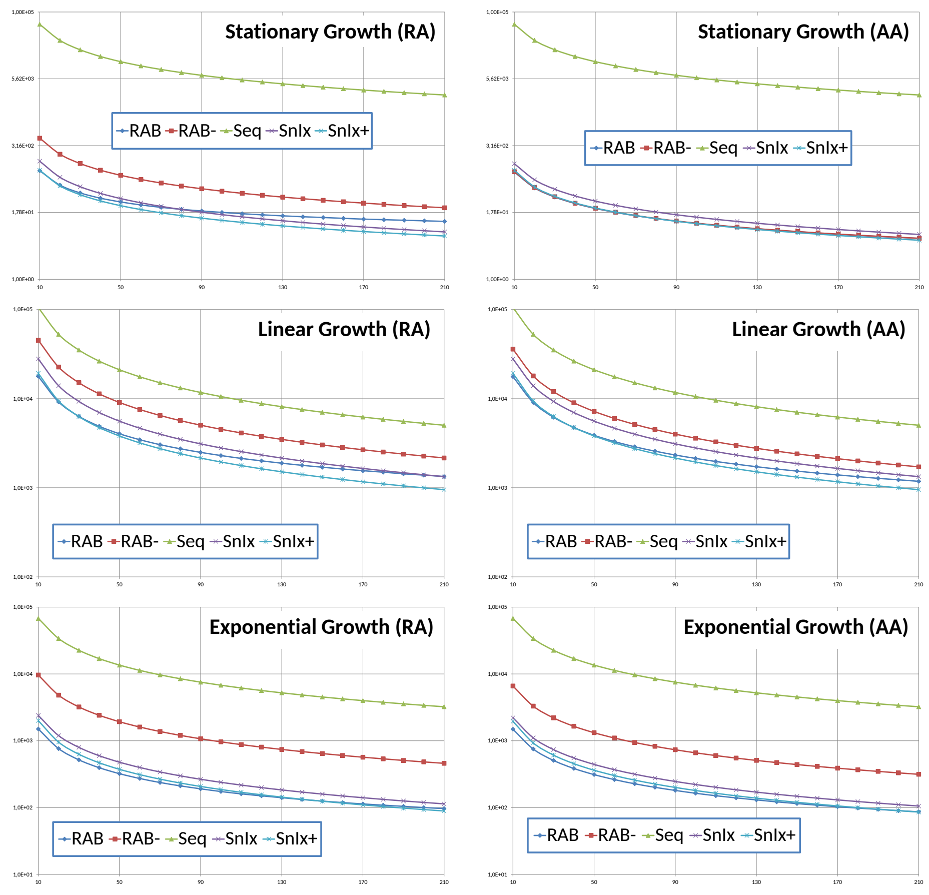

Fig. 5 Performance dependence on the block size in different settings.

make up a Poisson process, where the probability of an occurrence in a time interval is proportional to the length of the time interval. Obviously, the less conservative AA variant improves the clustering of the snapshots by reducing the number of long-lived tuples and the length of the interval between the insertion times of the first and of the last tuple belonging to each snapshot, which corresponds to the range of blocks spanned by the snapshot. RA and AA are variants of a growth scenario in the sense that the number of tuples deleted/inserted/updated by each transaction is the same, so that the file grows in the same manner; basically the same tuples are differently reshuffled, giving rise to different degrees of snapshot clustering.

3.3.1 Query Performance

Query costs are compared with a reference sequential scan cost which must be paid to execute a timeslice query in the absence of indexes. Notice that the sequential scan to retrieve the snapshot current at a time $T <$ now does not need to always access the whole file, due to its natural clustering on the Start attribute. The scan must always start from the first page, as the initial state may still contain tuples which were current at time T (i.e., having End $< T$), but can be stopped immediately before the first tuple belonging to a snapshot created after T is found (i.e., the first one with Start $> T$). If the RAB$^-$Tree is used, the sequential scan can be started from the first page containing a tuple belonging to the snapshot $s(T)$. Access costs to the snapshot $s(T)$ with the RABTree and RAB$^-$Tree indexes are also matched against the performance provided by the Snapshot Index in a trade-off implementation with $a = 0.5$, which should ensure good query behavior with a redundancy less than 100% (see also the discussion in Sec. 4). In particular, two implementations of the Snapshot Index are used. The former, denoted as SnIx in the Figures, is the basic one diffusely analyzed in [8], for which, in order to retrieve the snapshot $s(T)$, at most $2|s(T)|/b$ (and $1.62|s(T)|/b$ on average, for $a = 0.5$) disk blocks have to be accessed (also blocks actually non containing qualifying tuples must be accessed, as they are used to stop the search). The latter, denoted as SnIx+ in the Figures, exploits the optimization suggested in [8, p. 250], for which the number of disk blocks to be accessed reduces to the number of blocks all containing qualifying tuples plus one. On the other hand, only disk blocks containing qualifying tuples are accessed via the RABTree index.

In our first set of presented experiments, we used (as in [8]) a blocking factor $b = 50$, which corresponds to a mid-size disk page. The influence of the block size (or of the tuple size) on performance will then be studied in a second batch of experiments.

Graphs in Fig. 4 plot the average I/O costs (block reads) for accessing a single snapshot versus the total number of snapshots in the relation at a given transaction time T, which equals the number of transactions executed until T. The average is taken over all the available snapshots (i.e., from $s(T_0)$ to $s(T)$), representing the expected value of the cost paid at time T to access a randomly selected snapshot. Graphs to the left and to the right display query costs in the RR and RA archival variants, respectively (sequential scan costs are not influenced by the archival variant). Logarithmic scales are used for the exponential growth scenario.

In all the tested scenarios, except for a short initial transient in the exponential growth (RA) case, the RABTree index outperforms the basic Snapshot Index. Moreover, in the exponential growth scenario, after an initial transient, the RAB-Tree index also outperforms the optimized Snapshot Index. In order to provide lower access costs than the RABTree also in this case, the usefulness parameter of the Snapshot Index should be set to a value greater than 0.9, which means a data redundancy higher than 800%. This superior performance of the RABTree index is due to the optimized storage technique that improves the clustering of snapshots. In fact, the query costs plotted in [14, Fig. 2] that were actually measured on temporal relations built without exploiting the optimized storage technique, always showed a winning (basic) Snapshot Index. In the stationary growth scenario, access costs via one of the indexes become rather constant after a short transient and, thus, remain very low over the full range of executed transactions, while the sequential access cost linearly grows going out soon of the diagrams (reaching 11,786 for $T = 2500$). With the AA growth variant, the Snapshot Index is outperformed by the RAB$^-$Tree index too (but the difference is not so significant, owing to the small numbers anyway involved in the stationary growth scenario).

Graphs in Fig. 5 show the dependence of query costs on the block size, from a small size $b = 10$ to a big size $b = 210$, in all the growth scenarios. The graph ordinate is the average I/O cost paid for accessing a snapshot in the full grown file. Logarithmic scales are used to make all curves fit in the same graph.

In all the growth scenarios, the RABTree index, basic Snapshot Index and optimized Snapshot Index performances are always rather close to each other. There is no qualitative difference from the previously studied case with $b = 50$, except for the following cases. In the stationary scenario, as b increases, the RABTree index gets outperformed by the the optimized Snapshot Index in the AA variant and also by the basic Snapshot Index in the RA variant. However, the differences are not so significant since the costs involved are very low. In the linear growth scenario, the RABTree index behaves better than the Snapshot Index when the block size is very small (i.e., when $b \leq 20$ and $b \leq 30$ for the RR and RA variants, respectively). For $b \leq 70$, cost differences between the RABTree and the Snapshot index are under 10% anyway. In the exponential growth scenario, the optimized Snapshot Index becomes the fastest access structure in the presence of a big page size (i.e., for $b \geq 150$ and $b \geq 200$ for the RR and RA variants, respectively). In favor of the RABTree index, we can observe that it reduces the I/O costs with respect to the Snapshot Index, in the scenario with the highest growth rate, when it is most desirable, that is for small b values where the I/O costs are higher. The only other remarkable difference concerns the fact that, with a larger page size, the performance of the RAB$^-$Tree is closer to the performance of the RABTree, confirming the preliminary results provided in [14]. This behavior can be explained with the probability that a page in the range scanned via the RAB$^-$Tree does not contain any qualifying tuple, which decreases with the page size (e.g., two consecutive pages of size b, one which contains useful tuples and one which does not, can be merged by doubling the page size becoming a single page containing useful tuples).

Fig. 6 allows us to better appreciate the differences in the behavior of RABTree and RAB$^-$Tree. The graphs show, for all the evolution scenarios, the β ratio between the average snapshot access costs with the RAB$^-$Tree and with the RABTree. In particular, when the number of snapshots grows, it can be observed that after the initial transient the β ratio remains almost constant in the stationary growth (approximately, $\beta = 3.16$ and and $\beta = 1.1$ for RA and AA, respectively), it slowly linearly decreases in the linear growth scenario (the minimum values reached in our setting are around $\beta = 2.25$ and $\beta = 1.86$ for RA and AA, respectively) and slowly linearly grows in the exponential growth scenario (the maximum values reached in our setting are around $\beta = 6.11$ and $\beta = 4.24$ for RA and AA, respectively). All the graphs confirm that the performance of the RAB$^-$Tree gets closer to the performance of the RABTree in the AA archival variant, thanks to the better clustering of snapshots. A reduction of β with the growth of the page size can also be evidenced from the data plotted in Fig. 5 (and from [14, Fig. 3]). In practice, the experiments show that the simpler RAB$^-$Tree could be a viable access solution, if we can bear I/O costs $2\div3$ times higher, in the stationary or linear growth scenarios, as its performance scales very well with the growth of the file. Use of the RAB$^-$Tree should be discouraged in the exponential growth scenario indeed, as its performance with respect to the RABTree progressively degrades with the growth of the file.

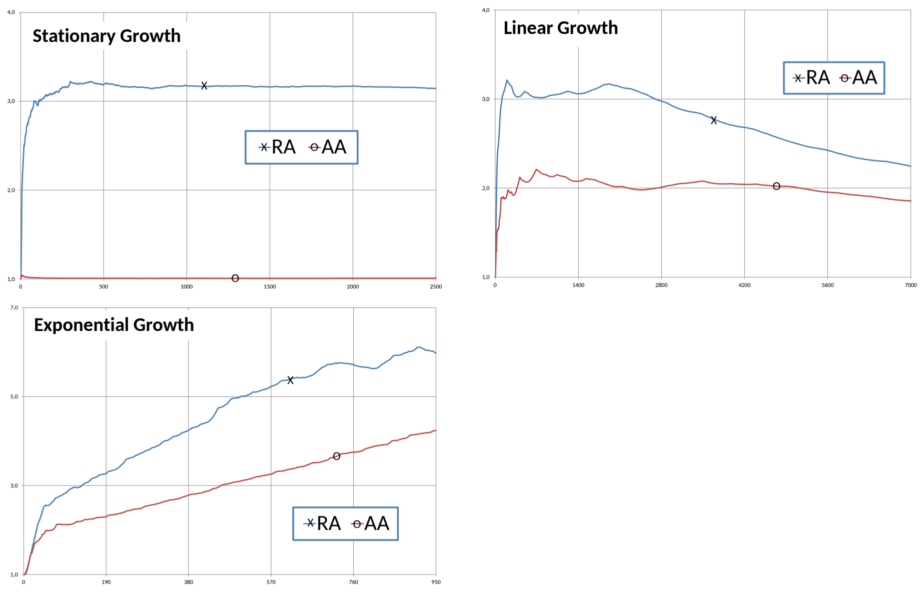

Fig. 6 The RAB$^-$Tree performance loss ratio β in different settings.

3.3.2 Space Requirements

In this Subsection we experimentally study the space requirements of the RABTree in different settings, also in comparison with its main competitor in this respect: the Time Index, which has a similar query performance but employs a different compression techniques in index leaves and enjoys a bad reputation of $\mathcal{O}(n^2/b)$ space costs [7].

Graphs in Fig. 7 plot the total number of TID list items, equal to the number of TIDs explicitly stored, globally present in the index leaves versus the file size n. TIx10 and TIx100 represent the space costs of two Time Index structures having different leaf size, that is 10 and 100 entries per index leaf, respectively (whereas the RABTree compression is not influenced by the leaf size, the effectiveness of the Time Index compression method increases with the size of the leaf block). The growth scenarios are the same as described at the beginning of this Section.

Since the Time Index size is independent of the archival variant (it only depends on the number of deleted/inserted/updated tuples per transaction, which is the same for the RA/AA variants), we organized the Figure as follows. The graphs to the left show the space costs of the Time Index in comparison with the RABTree ones (the RABTree curves are close to the x-axis and the differences between RA and RR variants are visually indistinguishable at this scale). The graphs to the right evidence, at a suitable scale, the influence of the archival variant on the RABTree size.

In all the analyzed growth scenarios, the RABTree clearly outperforms the Time Index by orders of magnitude, in both the RR and RA growth variants and regardless of the Time Index leaf size. In the stationary growth scenario, the space cost is linear for all the tested structures (as ensured by the constant size

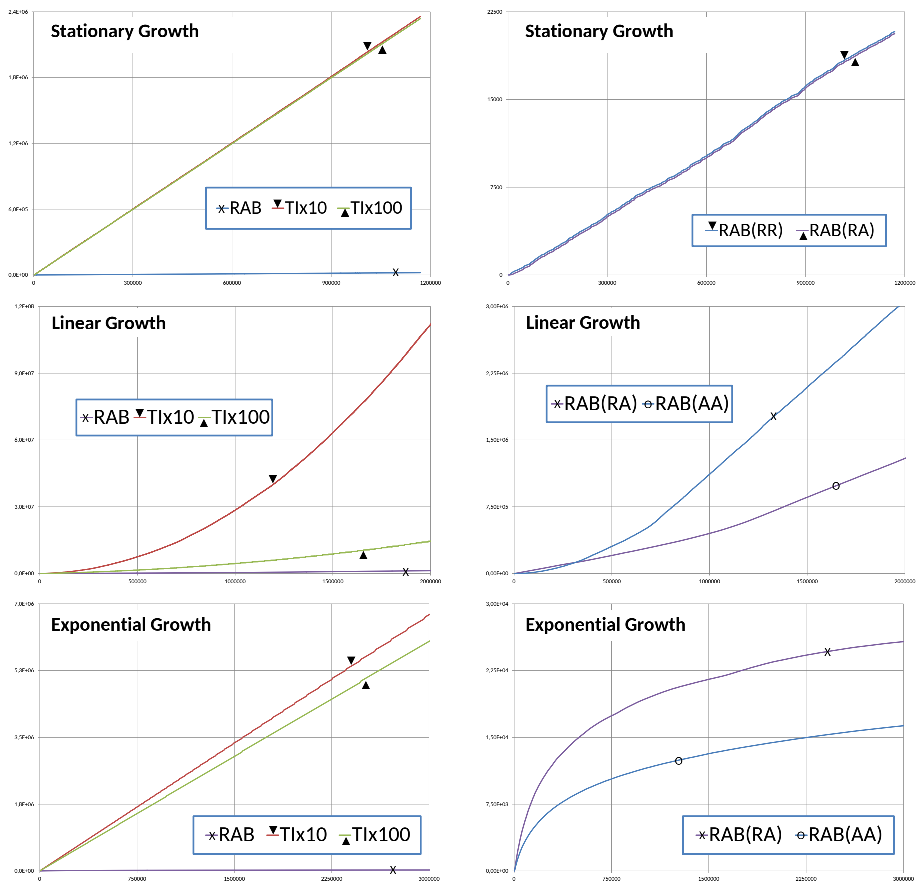

Fig. 7 Space requirements for the index leaves in different settings.

of the snapshots), but with a much lower slope for the RABTree. In the linear growth scenario, after an initial transient, the growth of the RABTree becomes linear, whereas the Time Index growth is quadratic. In the exponential growth scenario, the growth of the RABTree is sublinear (actually $\mathcal{O}(\sqrt{n}/b)$), whereas the Time Index growth is linear. Except for the stationary growth scenario, the ageing archival gives rise to a slower growth of the RABTree size, thanks to the improved clustering of snapshots that enables a better compression of the TID lists.

In order to evaluate the efficiency of the compression methods proposed for the RABTree and Time Index, Tab. 2 reports the values of the TID list percentage compression ratios (i.e., $100(1-\alpha)$) that can be measured at the end of our experiments in the various growth scenarios. Although at a first sight it looks that also the Time Index has a good compression performance in the linear growth scenario, it should be taken into account that the compression ratio provided by the Time Index is constant, as it only depends on the leaf size, whereas the compression ratio provided by the RABTree increases with the number of created snapshots and asymptotically tends to 1.

Table 2 Average TID Lists Compression Ratios

		RAB	TIx10	TIx100
Stationary Growth	RA	98.75%	99.80%	99.84%
	AA	99.99%		
Linear Growth	RA	99.47%	89.72%	98.70%
	AA	99.59%		
Exponential Growth	RA	99.72%	50.77%	55.73%
	AA	99.82%		

As a result, as shown in Tab. 3, the overall space occupied by the RABTree index leaves usually represent a small fraction of the space occupied by the data file. This is true also for the linear growth scenario, provided that the tuple size is not too small (i.e., at least 100).

Notice that if we start from the final states reached in the linear and exponential growth scenarios and reverse the sign of the balance between insertion and deletion rates, we can also consider the corresponding shrinking scenarios. The shrinking scenarios that can be obtained in such a way are mirror images of the growth ones and lead back to their initial state $s(T_0)$. In particular, the access cost to the current snapshots and the length of the rightmost index entries that can be measured during the shrinking experiments are, in reverse order, the same (on average) as seen in the corresponding growth scenarios.

In conclusion, in all the effected experiments, it could be appreciated how the RABTree query performance is generally not so distant form the gold standard represented by the theoretically optimal Snapshot Index behavior, and can even be better in some cases. Moreover, it can be also seen how the adoption of the simpler RAB$^-$Tree could be in some cases acceptable, being anyway able to reduce the costs of a sequential scan by more than 50%.

As to storage space, the compression method proposed for the RABTree, together with the optimized storage technique, has shown much more efficient than the one of the Time Index in all the analyzed settings. In particular, it allows to keep the space costs of the RABTree at most linear in n, also in the linear growth scenario representing the (realistic) worst case, where the Time Index grows quadratically indeed. We underline the fact that update patterns like that discussed in Sec. 3.2 and giving rise to a quadratic storage cost for the RABTree are pathological and absolutely implausible. In a realistic workload, we can generally consider that the order in which tuples are archived follows an archival model intermediate between the RA and AA models, with RA as a worst case. Yet assuming all the current tuples in the relation of Fig. 3 have the same probability to be deleted, the worst-case quadratic growth of the TID list size is avoided (i.e., the example of Sec. 3.2 becomes a particular case of the RA linear growth scenario).

Finally notice that, although the data sets used in our experiments, made of millions of tuples, can be considered rather small for today's big data standards, all the experiments sharply evidence a clear steady-state trend. We also effected some exploratory experiments with bigger data sets, larger by up to two orders of magnitude, which have produced results qualitatively quite similar to those illustrated in the figures of this Section but with larger numbers on the axes.

Table 3 Ratios Between RABTree Leaves and File Size

	Stationary Growth		Linear Growth		Exponential Growth	
tuple size	RA	AA	RA	AA	RA	AA
50	0.28%	0.058%	30.64%	7.52%	0.13%	0.084%
100	0.14%	0.029%	15.32%	3.76%	0.065%	0.042%
200	0.71%	0.015%	7.66%	1.88%	0.032%	0.021%
300	0.047%	0.0097%	5.11%	1.25%	0.022%	0.014%

4 Comparison with Other Indexes

The main touchstone for comparison with the RABTree and RAB$^-$Tree is the Snapshot Index by Tsotras and Kangelaris [8], which is an I/O optimal solution. It is a quite complex primary file organization, whereas the RABTree and RAB$^-$Tree index (like a B$^+$-Tree) are indeed index structures, which can be created and dropped at any point of the lifetime of a transaction-time relation and without changing its contents. Only in case the secondary ordering on End necessary to embody the "optimized" storage is not already present, the groups of data tuples with the same Start value must be sorted and rewritten during the file scan necessary to build the index. After the RABTree index has been built, the secondary order can be maintained for free with a careful execution of the data delete/update operations during the creation of a new snapshot. On the other hand, the Snapshot Index requires the implementation of an access forest, composed of a doubly-linked list L and an array AT to keep track of the blocks containing tuples current at a given time T, and is based on the adoption of a usefulness parameter a to constrain the contents of data blocks in order to cluster tuples belonging to the same snapshots and allow paginated access to them. The usefulness a measures the fraction of tuples in a block that are current in some time interval (representing a kind of usefulness period) after the insertion of the last tuple in the block. When a block ends its usefulness period, copies of its still current tuples are reinserted in the block currently receiving new tuples. The price to pay, in order to obtain clustering of tuples making up a snapshot, is the data duplication due to this controlled copying procedure. The array AT, implemented as a multilevel paginated index (which is, in practice, equivalent to a RABTree or RAB$^-$Tree index, which only differ from it by the contents of the leaves), provides temporal access to the access forest.

Thanks to the introduction of the usefulness concept and the deployment of the related copying procedure, the Snapshot Index is an access structure that provides theoretically optimal asymptotic performance. In fact, the Snapshot Index requires a $\mathcal{O}(n/b)$ storage space and guarantees that the time complexity of executing a timeslice query is $\mathcal{O}(\log_b n + |s(T)|/b)$. Besides the theoretical bounds above, the constants in the $\mathcal{O}(.)$ notation are also interesting, because of the role they play in actual storage and access costs to pay. The constant in the space requirement is $1/(1-a)$, while the constant in front of the main query time component is $2/a$ (but can be made to asymptotically tend to $1/a$ with the optimization strategy outlined in [8, p. 250]).

In practice, by adjusting the a parameter, the designer can trade between data duplication and query performance, trying to fit as best as possible application requirements. For example, with a low value of $a = 0.1$, a small space overhead (around 11%) is added to data storage but query costs may increase by a factor of 20, which could neutralize the benefit of having tuples belonging to the query answer clustered in disk blocks and, thus, transferred together (the factor 20 obviously cancels out with a block size of 20, but also virtually reduces by 20 times the size of bigger disk blocks). On the contrary, with a high value of $a = 0.9$, query costs can be at most as double as the theoretical optimal cost, but the storage space required to obtain such a performance blows out to 10 times as strictly necessary. A storage space overhead of about 900% would be frankly unacceptable for many applications, in particular when very large collections of data have to be managed. Even with a compromise mean value of $a = 0.5$, the multiplying factor of the query costs is 4 and a 100% storage space overhead is required to support such a performance. The useful size is also reduced by one unit by the presence, in each block, of a support data structure called SR record used to implement the list L. Nevertheless, the experimental results presented in [8, Sec. 4] show that, even without the further optimization, the multiplying factor of query costs (QR, as in [8, Fig. 7]) ranges from about 2 when $a = 0.1$ to 1.5 when $a = 0.6$ (we would not consider the Snapshot Index usable with $a > 0.6$, due to the very large data duplication introduced), with a value of about 1.62 for $a = 0.5$. This is a consequence of the fact that, in the adopted experimental setting, many blocks contain a fraction of tuples in their usefulness period greater than the minimum a. On the other hand, the expected degree of redundancy $1/(1 - a)$ is confirmed by experimental results (SR, as in [8, Fig. 6]) with good approximation for any $a \leq 0.6$.

The Snapshot Index also presents optimal update performance, assuming current tuples can be accessed by key by means of a dynamic perfect hashing scheme, which is a quite complex addressing method (requiring the periodic recomputation of a perfect hashing function), which provides for $\mathcal{O}(1)$ insertion and deletion cost (in the expected amortized sense, due to the use of hashing), and $\mathcal{O}(1)$ worst case lookup time, and requires linear space. Notice also that the $\mathcal{O}(1)$ update bound cannot be guaranteed against pathological worst cases (e.g., due to a bad choice of the hashing function, with respect to the actual data distribution).

The Time-Split B-Tree (TSBTree) [9] is a B-tree-like primary file organization, which clusters data both on the key and on transaction time. Block splits on the time coordinate unavoidably give rise to data duplication, which can reach very high degrees depending on the data distribution and percentage of updates applied. With the adaptive split policy presented in [17], redundancy can be controlled within a user-defined range, yet at the detriment of index selectivity on time and, thus, on timeslice query performance. Since tuples belonging to the same snapshots can be spread among many blocks owing to the concurrent splitting on the key, a snapshot access cannot be paginated and, therefore, has an $\mathcal{O}(\log_b n + |s(T)|)$ I/O cost. Since actual costs are amplified by a factor which grows with data duplication, the performance of the TSBTree for snapshot access is no better than that of the RABTree. The same suboptimal asymptotic query performance (non paginated snapshot access) is granted by the Snapshot index predecessor described in [13], which is also a rather complex primary file organization.

The Multiversion B-Tree (MVBTree) [10] is also a primary file organization, which provides a fully paginated solution to key-range timeslice queries. Hence, it allows to execute a snapshot access with $\mathcal{O}(\log_b n + |s(T)|/b)$ optimal I/O cost, yet at the price of a high necessary duplication (which is about 10 times as with the TSB-Tree). It could be said that, with the aim of maintaining the same asymptotic query performance, the Snapshot Index tries to modulate a MVBTree-like duplication by means of the a parameter. On the contrary, the RABTree proposed in this paper takes another direction (and its RAB^{-}Tree variant goes farther in the same direction), by giving up theoretical optimality of query performance to privilege indeed a low-impact solution, by means of a lean index organization which does not require data reorganization and does not introduce duplication at all. Nevertheless, as shown by the experiments reported in Section 3.3, the performance of the RABTree index in the tested settings is very close to, and sometimes even better than, the performance of the Snapshot Index. Even the more modest query performance of the RAB^{-}Tree index could be acceptable in some application scenarios, being anyway able to reduce by 50%÷60% the I/O costs paid in the absence of indexes.

The Append-Only Tree (APTree) [11] is conceptually very similar to the RAB^{-}Tree: it is a right append-only B^{+}-Tree-like index built on time and relying on the natural clustering of data induced by their creation time. The only difference is that in the index entry $T{:}P$, P points to the page containing the last inserted tuples with Start $\leq T$. While the APTree has been proposed to speed-up temporal joins, it is of little help for snapshot access: when accessing $s(T)$, the entry $T{:}P$ found in the APTree leaves allows us to exclude from the search the tuples in the pages following the one pointed by P, but all the preceding ones have to be searched for tuples belonging to $s(T)$, giving rise to an $\mathcal{O}(\log_b n + n/b)$ query cost. In practice, whereas the RAB^{-}Tree limits the range of pages that must be sequentially scanned to find qualifying tuples from both sides, the APTree limits that range from the upper side only. Space and update costs are the same as the RAB^{-}Tree.

Another proposed structure very similar to the RABTree is the Time Index [12]. Actually, the basic idea underlying the RABTree is the same as the Time Index, as they both derive from the preliminary structure presented in [12, Fig. 2], where pointers to tuples making up snapshots are stored in the index leaves. Then, the Time Index stores the full TID list only for the first entry of each leaf, whereas the TID lists of the other entries in the leaf keep track of the incremental changes only with respect to the first entry, by means of pointers to the tuples added or deleted. In practice, we can say that the Time Index and the RABTree index differ by the TID list compression technique used in the leaves. Hence, the detailed analysis developed for the RABTree in Sec. 3 mostly applies also to the Time Index (and the Monotonic B^{+}-Tree [18]): the I/O cost for accessing data tuples is the same as the RABTree index but query answering via the Time Index also requires extra I/O time to fetch a bigger index leaf and extra CPU time to process leaf contents to reconstruct explicit TID lists. Moreover, the Time Index compression technique is partial, being relative to the first leaf entry that is stored uncompressed, whereas the RABTree compression technique is uniformly applied to every leaf entry. Therefore, the storage space required by the Time Index leaves is guaranteed $\mathcal{O}(n/b)$ only if the snapshot size is bounded, as it happens for the stationary growth scenario where the snapshot size is constant. On the other hand, if the snapshot size grows even by a single tuple per transaction, the storage space required by the Time Index leaves can easily meet the $\mathcal{O}(n^2/b)$ worst case, as it happens in our linear growth scenario, where the space occupied by the Time Index leaves becomes soon several times larger than the size of the data file itself (59 and

7.5 times for TIx10 and TIx100, respectively). Moreover, if most of the current tuples are modified by each transaction, the Time Index TID list compression method becomes definitely ineffective, whereas the RABTree behaves at best with TID list compression approaching 100%. For these reasons, the Time Index cannot be considered a lean index structure. Also the implementation variants proposed for the Time Index in [19,20], which allow to slightly reduce the TID duplication in index leaves, are not exempt from a quadratic storage space growth.

Notice that, as suggested in [7, Sec. 5.1.2] for the Time Index, also the RABTree index could be transformed into a primary file organization by storing the data tuples directly in the index leaves (giving up compression and making multiple copies of the tuples shared by multiple snapshots), yielding a $\mathcal{O}(\log_b n + |s(T)|/b)$ optimal I/O cost for snapshot access, but at the price of a high duplication (which can be indeed reduced by the compression technique used by the Time Index but has anyway a worst case $O(n^2/b)$ space requirement).

As far as update costs are concerned, the Time Index and the APTree require in general $\mathcal{O}(n/b)$ I/Os as the stand-alone RAB$^-$Tree, whereas the TSBTree requires $\mathcal{O}(\log_b n)$ I/Os. The MVBTree behaves exactly as the RABTree with the companion B$^+$-Tree built on current data, requiring $\mathcal{O}(\log_b m)$ I/Os, where m is the total number of current tuples (m is usually much less than n, but it can be $\mathcal{O}(n)$ in the worst case), whereas the stand-alone RABTree requires $\mathcal{O}(m)$ I/Os. The Snapshot Index and its predecessor are optimal also in this respect, by requiring a constant time (in expected amortized sense, due to the use of hashing, provided that pathological situations could be avoided).

Owing to their lean nature and to the fact that they do not require any duplication of the indexed data, the RABTree and RAB$^-$Tree outperform all previously proposed index structures as far as space requirements are concerned, thanks to the high compression ratios that can be obtained (cf. Tab. 2 and Fig. 7). Asymptotically, excluding the pathological worst case discussed in Sec. 3.2, the memory space occupied by the index together with the indexed relation is $\mathcal{O}(n/b)$ either for the RABTree/RAB$^-$Tree and for all the other index structures considered in this Section but the Time Index. However, thanks to the good space performance evidenced in Sec. 3.3 (cf. Tab. 3), we could expect the multiplying constant before n/b in actual space costs to be usually less than 1.2 (i.e., 1 for the data without redundancy plus less than 20% for the index) for the RABTree index and even less for the RAB$^-$Tree index, whereas it is surely greater than 2 for the other index solutions.

5 Conclusion

In this work, we presented the RABTree index, and its RAB$^-$Tree variant, which are lean and theoretically I/O-suboptimal access structures for supporting timeslice queries (i.e., rollbacks) in transaction-time databases. These are simple to implement and to maintain index structures, which exploit as much as possible the natural clustering of data resulting from their insertion order. Other index structures require restructuring of the data storage to force a better clustering and, thus, improve query performance at the price of the introduction of a high redundancy degree (which is about 100% for the Snapshot Index with $a = 0.5$). In order to provide a privileged access to the current snapshot (which is likely used, say, by 99% of the queries and also impacts on update costs), the RABTree or RAB$^-$Tree index can be accompanied by a conventional B$^+$-Tree built on the key and indexing current tuples only. Other kinds of queries, like KT-point and history queries, can be supported by flanking the RABTree with a companion B$^+$-Tree indexing by the key all the tuples in the relation.

The theoretical analysis and the experimental results reported in Sec. 3 are rather encouraging, as they confirm that the RABTree/RAB$^-$Tree performance, although not guaranteed optimal, could be very good in a wide range of practical settings. Moreover, the RABTree becomes the optimal solution if an index must be adopted, when reorganization of the indexed data cannot be applied or redundancy cannot be tolerated and only the natural clustering induced by the insertion order of data can be exploited.

Last but not least, it is worth noting that the fact that the RABTree (or RAB$^-$Tree) index does not require any data duplication and is a lean structure with quite limited memory requirements, makes it also interesting for adoption on solid-state storage devices widely equipping portable or mobile computing systems, where storage space is more an issue than query performance with respect to classical disk memories, owing to their faster access time but limited and costly capacity.

On the other hand, in our future work, we will also consider the introduction of a controlled data redundancy similar to the one proposed for the Snapshot Index in the management of a temporal relation indexed by a RABTree index, in order to further improve its query performance.

References

1. D. Etzion, S. Jajodia, and S. Sripada (Eds.), *Temporal Databases - Research and Practice.* LNCS vol. 1399, Springer-Verlag, Berlin, Germany (1998).
2. C.S. Jensen and R.T. Snodgrass, "Temporal database". In: L. Liu and M.T. Özsu (Eds.), *Encyclopedia of Database Systems*, Springer-Verlag, Berlin, Germany, pp. 2957–2960 (2009).
3. F. Grandi, "Temporal databases". In: M. Khosrow-Pour (Ed.), *Encyclopedia of Information Science and Technology, 3rd ed.*, IGI Global, Hershey, PA, pp. 1914–1922 (2014).
4. C.S. Jensen, C.E. Dyreson (Eds.), M.H. Böhlen, J. Clifford, R. Elmasri, S.K. Gadia, F. Grandi, P. Hayes, S. Jajodia, W. Käfer, N. Kline, N. Lorentzos, Y. Mitsopoulos, A. Montanari, D. Nonen, E. Peressi, B. Pernici, J.F. Roddick, N.L. Sarda, M.R. Scalas, A. Segev, R.T. Snodgrass, M.D. Soo, A. Tansel, P. Tiberio and G. Wiederhold, "The Consensus Glossary of Temporal Database Concepts - February 1998 Version". In: [1], pp. 367–405 (1998).
5. J. Clifford, C.E. Dyreson, T. Isakowitz, C.S. Jensen and R.T. Snodgrass, "On the semantics of "now" in databases", *ACM Trans. Database Syst.* **22**(2) 171–214 (1997).
6. K. Kulkarni and J.-K. Michels, "Temporal features in SQL:2011", *ACM SIGMOD Rec.* **41**(3) 34–43 (2011).
7. B. Salzberg and V. J. Tsotras, "Comparison of access methods for time-evolving data", *ACM Comput. Surv.* **31**(2) 158–221 (1999).
8. V. J. Tsotras and N. Kangerlaris, "The Snapshot Index: An I/O-optimal access method for timeslice queries," *Inf. Syst.* **20**(3) 237–260 (1995).
9. D. Lomet, and B. Salzberg, "The performance of a multiversion access method". In: *Proc. 1990 ACM SIGMOD Int. Conf. on Management of Data*, Atlantic City, NJ, pp. 353–363 (1990).
10. B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer, "An asymptotically optimal multiversion B-tree", *The VLDB J.* **5**(4) 264–275 (1996).
11. H. Gunadhi, and A. Segev, "Efficient indexing methods for temporal relations", *IEEE Trans. Knowl. Data Eng.* **5**(3) 496–509 (1993).

12. R. Elmasri, G. T. J. Wuu and Y.-J. Kim, "The Time Index: An access structure for temporal data". In: *Proc. 16th Int. Conf. on Very Large Data Bases*, Brisbane, Australia, pp. 1–12 (1990).
13. V. J. Tsotras, B. Gopinath and G.W. Hart, "Efficient management of time-evolving databases", *IEEE Trans. on Knowl. Data Eng.* **7**(4) 591–608 (1995).
14. F. Grandi, "Lean Index Structures for Snapshot Access in Transaction-Time Databases". In: *Proc. 21st Int. Symposium on Temporal Representation and Reasoning*, Verona, Italy, pp. 91–100 (2014).
15. D. Comer, "The ubiquitous B-Tree", *ACM Comput. Surv.* **11**(2) 123–137 (1979).
16. R. Elmasri and S. B. Navathe, *Fundamentals of Database Systems, 6th ed.*, Addison-Wesley Boston, MA, pp. 652-660 (2011).
17. L. Amadesi and F. Grandi, "An adaptive split policy for the Time-Split B-Tree", *Data Knowl. Eng.* **29**(1) 1–15 (1999).
18. R. Elmasri, G. T. Wuu and V. Kouramajian, "The Time Index and the Monotonic B$^+$-Tree". In: A. U. Tansel, J. Clifford, S. K. Gadia, A. Segev, and R. T. Snodgrass (Eds.), *Temporal Databases: Theory, Design, and Implementation*, Benjamin/Cummings, San Francisco, CA, pp. 433–456 (1993).
19. R. Elmasri, Y.-J. Kim and G. T. J. Wuu, "Efficient Implementation Techniques For the Time Index". In: *Proc. 7th Int. Conf. on Data Engineering*, Kobe, Japan, pp. 102–111 (1991).
20. V. Kouramajian, I. Kamel, R. Elmasri and S. Waheed, "The Time Index+: An Incremental Access Structure for Temporal Databases". In: *Proc. 3rd Int. Conf. on Information and Knowledge Management*, Gaithersburg, MD, pp. 296–303 (1994).