Packed Memory Arrays – Rewired

Dean De Leo
CWI
The Netherlands
dleo@cwi.nl

Peter Boncz
CWI
The Netherlands
boncz@cwi.nl

***Abstract*—The physical memory layout of a tree-based index structure deteriorates over time as it sustains more updates; such that sequential scans on the physical level become non-sequential, and therefore slower. Packed Memory Arrays (PMAs) prevent this by managing all data in a sequential sparse array. PMAs have been studied mostly theoretically but suffer from practical problems, as we show in this paper. We study and fix these problems, resulting in an improved data structure: the Rewired Memory Array (RMA). We compare RMA with the main previous PMA implementations as well as state-of-the-art tree index structures and show on a wide variety of data and query distributions that RMA can reach competitive update and point lookup performance, while always providing superior scan performance – close to dense column scans.**

I. Introduction

Columnar formats have become the staple storage for analytical data processing systems, but increasingly analytically strong systems are being expected to also deliver at least decent throughput while this data is being updated and queried with short-running queries. Such workloads are *mixed* and sometimes also called Hybrid Transactional-Analytical Processing (HTAP). The need to provide sub-linear update and lookup performance can be fulfilled by some variant of the B$^+$ tree index. Following [24], we use the term B$^+$ tree when the node capacity is optimised for disk access, while using (a, b)-tree for trees optimised for CPU cache-line access [4].

In columnar engines, (a, b)-trees do not quite fit naturally, though. Column stores are optimised to crunch data through sequential (range-)scans, whereas the "optimality" of (a, b)-trees stems from the I/O model [2], [24]. If the node capacity B is chosen to fit a cache-line, then (a, b)-trees have complexity $\Theta(R/B)$ for a scan of R elements. However, these are actually $\Theta(R/B)$ random jumps, which are much more expensive than sequential jumps [13]. We find scans on (a, b)-trees to be 3x slower than dense column scans. Our goal is a data structure that provides much faster column scans.

Specifically, as tree data structures sustain continuous update (insert/delete/modify) operations, the logical order that they maintain on their data, cannot be maintained on the physical level (disk block, or cache line), since new tree nodes (typically allocated at the physical end) are inserted in the logical *middle*, and/or blocks are deleted and merged, creating physical holes.[1] To mitigate this, columnar engines store sorted data in a hybrid manner [16], [21]. A large part of the data is statically stored in dense columns, sorted by the search key. Instead, updates are performed in a secondary data structure, the "*delta*", which can even be modeled as an (a, b)-tree. To retrieve the actual tuples, an operator needs to scan both the static section and the delta, merging in the updates from the latter. Due to this merge effort that slows down each read query, this solution only works well if the delta is small.

In this paper we investigate an alternative design, based on packed memory arrays (PMA) [2]. The columns become "sparse" rather than "dense". Among the stored elements there are empty gaps, to accommodate potential future updates. Therefore, updates can be directly performed in place. Scans are now truly sequential, and updates can come at logarithmic data complexity, but are slower than on (a, b)-trees.

Despite their theoretical properties, straight implementations of sparse arrays do not match properly tuned (a, b)-trees in practice. First, in Figure 1a we point out that our starting baseline implementation is faster than the implementations published so far[3]. In this experiment we insert, following a uniform distribution, $1G$ of 8 byte integer key/value pairs in an empty container, and then perform random contiguous scans of 1% of the final data structure. Then, we compare the same data structure over a custom implementation of (a, b)-trees, similar to STX-Tree [11]. In (a, b)-trees, we can vary the maximum capacity B of the leaves, to improve either scans or updates and point look-ups. However, already in this experiment, we note that both insertions *and* scans perform better on a (a, b)-tree with well-chosen node size ($B = 256$ elements $= 4kB$). One contribution of this paper is hence to point out that PMAs so far provide no practical value, something not asserted in publications describing them.

There are several factors that hinder the performance of sparse arrays. For scans, a substantial CPU *branch-misprediction* penalty is paid by checking whether each slot of the array is filled or not. For updates, the data structure needs to be occasionally *rebalanced*, increasing the average cost of these operations. Moreover, during a rebalancing, elements in the array may be moved, also causing an update to the separator keys of the *index* that PMAs keep on the side in order

[1] A fast-forward to Figure 13a shows this significantly affects performance already after 1-2% of tuples are updated.

[2] In this paper we consider the terms *sparse array* and *packed memory array* (PMA) as synonyms.

[3] The source code of PM14 [25], KLS17 [20] and SLH17 [30] is publicly available online, released by their respective authors. The source code of DRF12 [14] has been provided by its authors upon request. We briefly discuss these applications in Section VI, Related work.

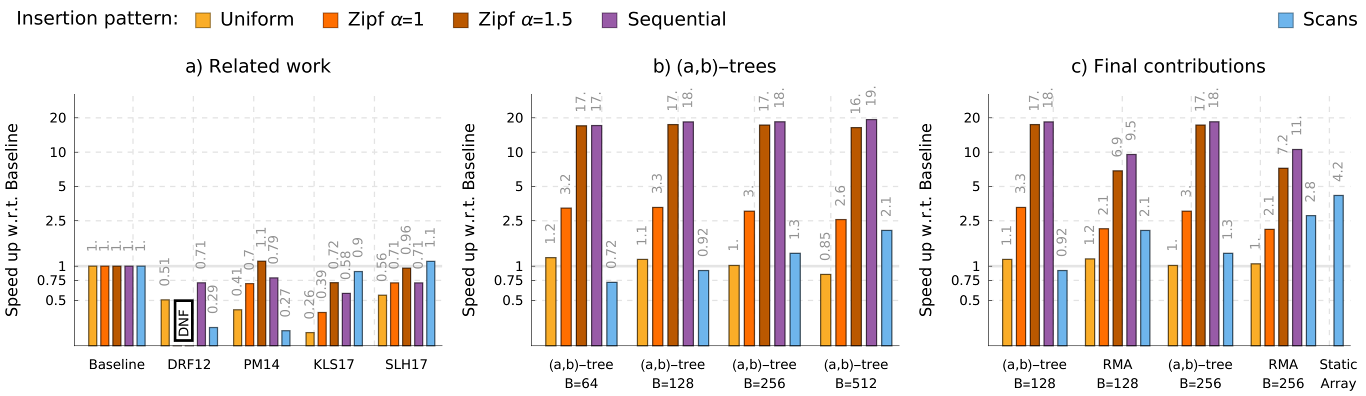


Fig. 1. a) Our PMA baseline is competitive with state of the art PMAs. b) (a, b)-trees are superior to state of the art PMAs. c) Our final data structure RMA is a practical alternative to optimised (a, b)-trees for HTAP applications where scan performance is the primary concern: RMA scans are 2x faster while update performance is much closer to (a, b)-trees than state of the art PMAs.

to avoid pure binary search. In presence of skew, sparse arrays hit their worst-case scenario and more internal reorganisations are carried over during updates, while update skew for (a, b)-trees is their best case, as it creates most locality.

Summary of contributions. We propose the Rewired Memory Array (RMA), an improved version of sparse arrays that fixes its major flaws and adds the new technique of memory rewiring [29] into its rebalancing mechanism. The RMA can be tuned by an extra parameter, the segment size B, similar to the leaf capacity of (a, b)-trees. The RMA provides scan performance closer to dense arrays, and an update performance competitive to (a, b)-trees, at corresponding segment/leaf capacity. Figure 1c depicts our experimental results of RMA, compared to our initial PMA baseline. Our contributions are:

- Through the introduction or adoption of a number of novel techniques, which are clustering, fixed size segments, *static index* [12] and memory rewiring [29], we overcome the base hindrance of range scans while reducing the latency of internal rebalances.
- We refine a feature, *adaptive rebalancing*, first proposed in the *Adaptive PMA* (APMA) [10]. Adaptive rebalancing reduces the number of rebalances occurring in presence of update skew. Still, we uncover that the adaptive rebalancing strategy of APMA can be detrimental, and propose a new algorithm, resolving its limitations.
- We present a new bulk loading algorithm. It is particularly suitable for the *streaming scenario* [30], [31], where the cardinality of the array is kept constant, and updates, featuring the same amount of insertions and deletions, are executed in batches at regular intervals.

This paper is organised as follows: In Section II, we summarise PMAs and their properties. In Section III, we describe our contributions: clustering, fixed segment sizes, the static index structure, the usage of memory rewiring for rebalances, while leaving the treatment of adaptive rebalancing to Section IV. In Section V, we evaluate our RMA and compare it to (a, b)-trees, ART [22] and static arrays. We review related work in Section VI, and conclude in Section VII.

II. Preliminaries

Traditional PMA. A packed memory array is an array where elements are stored according to a sorted order, interleaved with empty slots or *gaps*. The gaps serve the purpose of providing extra room to insert new elements in arbitrary positions of the array, without the need to shift long sequences of existing elements to maintain the sorted order.

The insert, delete, search and range-scan operations are implemented as follows. Searching an element in the array can be realised by exploiting the sorted order with a binary search. A range scan involves sequentially iterating over the array between the two endpoints of the range. Here, empty slots are simply ignored. To insert a new element, the algorithm first searches its target slot in the array. However, if the position is already occupied by another element e, the algorithm will first shift e and all adjacent elements of e towards the nearest gap in the array. Finally, to delete an element in the array, the algorithm simply marks its slot as empty.

The key to efficiently support these operations is to readjust *locally* the data structure once portions become too sparse or too dense. The underlying array, of capacity C, is logically split in $O(C/log_2C)$ *segments* of size $O(log_2C)$. When inserting (resp. deleting) an element, the algorithm also checks the number of gaps g present in the related segment. If g is too small (resp. too big) *w.r.t.* a given predefined threshold, the algorithm starts to inspect the adjacent segments until it detects an amount of gaps within a range of certain specified thresholds. At this point a *rebalancing* operation takes place: all elements in the *window* W, i.e. the considered sequence of adjacent segments, are *evenly* spread in W. As consequence, all segments $s \in W$ feature the same number of gaps and the same density. However, if the whole underlying array cannot satisfy the required thresholds, then the data structure is resized, altering the capacity of the array.

Both the predefined thresholds and the order in which the segments are visited play a fundamental role in the complexity of rebalancing. A binary tree, named *calibrator tree* [17], is logically built bottom-up starting from the segments, which act

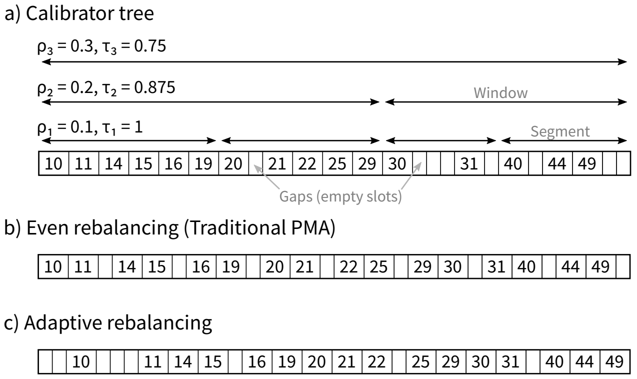


Fig. 2. a) The calibrator tree of a sparse array of capacity 24, with 4 segments of size $s = 6$, and height of the tree $h = 3$. The arrows above the array represent the nodes of the calibrator tree, together with their associated lower ρ_l and higher τ_l thresholds. b) The outcome of even rebalancing the elements in the whole array. c) The outcome of adaptive rebalancing, when the whole array is considered and the algorithm detects elements are inserted only at the start of the array.

as the leaves of the tree (see Figure 2a). Grouping the adjacent leaves two by two, we form the inner nodes at the second level of the tree. We recursively repeat the above process up to the root. Furthermore, for each level l in the calibrator tree, a lower ρ_l and a higher τ_l *density threshold* are defined, following an arithmetic series. The leaf-to-root path identifies the segments to inspect at each step of rebalancing, while the thresholds bind the amount of gaps that each window can contain, or, equivalently, their minimum and maximum densities. Formally, the density $\delta(W)$ of a window W is the ratio between the number of stored elements $card(W)$ and the capacity $cap(W)$.

Complexity. The tightest bounds for the data structure are based on amortised analysis. In the RAM model, given C the capacity of the underlying array, updates feature $O(log_2^2 C)$ in worst case amortised analysis per update operation [4], [18], and, if the keys follow a uniform distribution, $O(log_2 C)$ in average amortised analysis [7], [18]. In fact, because the number of elements N stored at any time is proportional to the capacity C of the array, i.e. $N \propto C$, the actual complexity is bounded by $O(log_2^2 N)$ in the amortised worst case, per update operation. In the I/O model, the complexity straightforwardly turns to $O(log_2^2 N)/B$ amortised worst case per update [3], [4]. Lookups cost $O(log_2 N)$ due to binary search, but it can be enhanced to $O(log_B N)$ in the I/O model with the use of an external index [3], [4], [10]. Range scans are bounded by $O(R/B)$ sequential accesses in worst case complexity, where R is the number of elements in the range.

Adaptive PMA. The Adaptive PMA (APMA) refines the rebalancing strategy of the Traditional PMA (TPMA), improving its behaviour for sequential and, more generally, *hammer insertions*. Hammering [10] refers to continuous and frequent insertions occurring in the same regions of the array. It triggers the worst case scenario in TPMA. The intuition of the adaptive strategy is that, during a rebalancing, if an interval is subject to hammering, the elements contained are not spread evenly. Rather, as many gaps as allowed by the thresholds of the calibrator tree are displaced in the area where hammering occurred, while most elements are moved to the rest of the window being rebalanced.

If the same segments continue to be hammered, then the complexity in the RAM model (resp. I/O model) becomes $O(log_2 N)$ (resp. $O(log_2 N/B)$) in amortised worst case, per insertion. However, if the prediction turns out to be wrong, and insertions occur in different sections of the array, this strategy still guarantees an upper bound of $O(log_2^2 N)$ (resp. $O(log_2^2(N)/B)$) per insertion in amortised worst case complexity. Figures 2b and 2c depict the outcome of traditional and adaptive rebalancing, assuming that the APMA algorithm detects insertions only occurring at the start of the array.

While APMA improves theoretically the behaviour of TPMA, it also carries some drawbacks in practice. First, it does not support deletions. Second, its rebalancing algorithm employs a complicate scoring heuristics to assert the amount of "hammering" of each segment. Third, and foremost, the algorithm can cause a *ping-pong* effect, where the adaptive strategy can, in practical scenarios, even become detrimental compared to traditional rebalancing.

For instance, consider the scenario where the latest insertions in the sample array of Figure 2a were the elements 14, 15 and 16. On a rebalance, if APMA detects this pattern, it tries to place as many gaps as possible in the first segment, expecting future insertions to continue in the same area of the array. Again, this could lead to the same outcome depicted in Figure 2c. However, if the same pattern continues, the next elements to be inserted are 17 and 18, now placed in an area even denser than what achieved with the traditional rebalancing.

Density thresholds. The density thresholds are an input parameter, set by the designer of the implementation. Typically, only the four extremes ρ_1, ρ_h, τ_h and τ_1 are explicitly set, while the rest of the thresholds are properly adjusted, based on the current height h of the calibrator tree. There is an order to respect in their choice: $0 \leq \rho_1 < \rho_h \leq \tau_h < \tau_1 \leq 1$. The value of ρ_1 determines the minimum potential *fill factor* N/C of the data structure.

There are two main strategies to set the thresholds, depending on the approach used to handle resizes. In the first approach, the capacity of the array is doubled (or halved) when a resize is executed. For consistency [10], this approach imposes a further constraint: $2 \cdot \rho_h \leq \tau_h$. For this strategy, common thresholds in existing applications [10], [14], [30] are $\rho_1 \sim 0.1$, $\rho_h \sim 0.3$, $\tau_h \sim 0.75$ and $\tau_1 \geq 0.9$. In the second approach, the capacity of the array is set to $\frac{2 \cdot N}{\tau_h + \rho_h}$ on resize. Notably, for this strategy, [17] proposes a choice of $\rho_h = \tau_h = 0.75$ and $\rho_1 \geq 0.5$. Therefore, the density of the array is kept close to 75%, while the minimum fill factor is at least 50%.

The choice of the thresholds involves a trade-off. The first approach favours updates. The greater the difference $|\tau_l - \rho_l|$, the looser are the density constraints to respect, and the fewer resizes and rebalances will occur. On the other hand, the

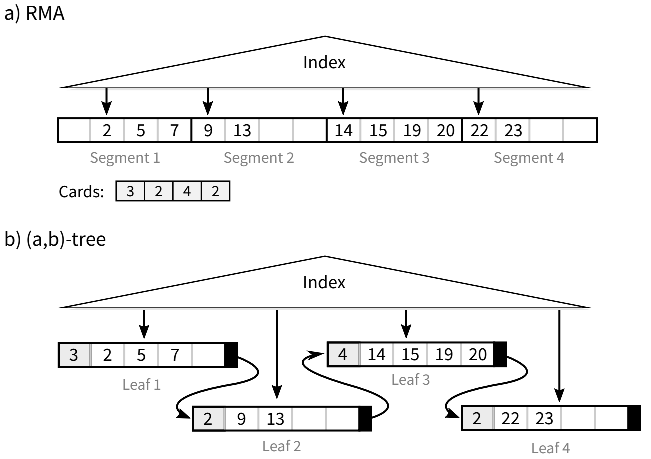


Fig. 3. a) Sample layout of an RMA with 4 segments of size 4. b) Equivalent layout of the leaves of an (a,b)-tree with node size 4.

second strategy favours scans. The greater the values of ρ_l and τ_l, the greater is the average fill factor of the array, and the smaller the memory footprint of the sparse array. For these reasons, hereafter, we name the former strategy *update oriented*, and the latter *scan oriented*.

III. Overview

The RMA evolves from the PMA in several features. In this section, we detail its layout, its static index, the usage of memory rewiring for rebalancing, bulk loading and a few optimisations. Whereas, we will finally describe adaptive rebalancing in the next section.

Segments. Compared to the PMA, there are two major departures in our treatment of segments. First, in the RMA, segments characterise the layout of the elements inside the sparse array. Instead, in the PMA, segments only play a role in rebalances, to define the smallest interval where to validate the density thresholds ρ_1 and τ_1. The second distinction is that we bind the segment size to the block size $O(B)$ of the I/O model, rather than to $O(log_2 N)$[4]. Hence, the segment size is fixed and does not depend on the current cardinality or capacity of the data structure.

Our RMA clusters together long sequences of elements alternated with long sequences of gaps. In contrast, the classic PMA layout is a sequence of elements intermixed with gaps. This organisation turns out to be detrimental for scans, due to branch mispredictions: at every slot, a scan needs to check whether it is empty and should be ignored. To overcome this, we keep track of the current cardinality of each segment in a side array, named *cards*. We pack all elements in one end of the segment, and the gaps to the other end. As we want to maximise the sequence of consecutive elements, we alternate to which extreme to pack the elements, storing them towards the right end for odd indexed segments, and towards the left end for even numbered segments[5]. As we know the cardinality of each segment and it is dense, testing for gaps is not required. Rather, a scan performs one tight loop over a dense sequence of values for each two segments.

Our choice of the segment size is driven by tailoring the data structure to the I/O model. For any practical value of N, it turns out that $O(log_2 N)$ is less than $O(B)$. In the PMA, any insertion or deletion involves a rebalance of a window of at least $O(log_2 N)$. Theoretically, as an update already must touch a portion of the array large $O(B)$, rebalances on the first $O(log_2(\frac{B}{log_2 N}))$ levels of the calibrator tree yield no effects, they are merely overhead. Our solution is to fix the segment size to $O(B)$. For insertions, we fill a segment until it is full, and only then, we trigger a rebalance. Consequently, the upper threshold τ_1 is 1.

In theory, this change does not alter the underlying complexity. It only reduces the height of the calibrator tree by an additive constant $O(log_2 \frac{N}{B}) = O(log_2 N)$. In practice, it hints that the segment size needs to be tuned according to the ideal block size B of the underlying architecture, analogously to the node size of (a, b)-trees. This is significant as, in our approach, elements are clustered inside a segment. The conventional choice of $log_2 C$, employed by most existing PMA implementations examined, would instead generate segments too small and far from ideal for both scans and updates.

Comparison with (a,b)-trees. RMAs deeply resemble (a, b)-trees. Indeed, the segments that compose the array are analogous to the leaves of an (a, b)-tree. Roughly, the same leaf layout and size employed in an (a, b)-tree, can be adopted for the segments in the RMA. Figure 3 compares the layout of both an RMA and an (a, b)-tree with the same elements and equal node/segment size. Conceptually, the major functional difference is how they are reorganised when the segments/leaves become either overfilled or underfilled. (a, b)-trees employ node splits and merges, operations *local* to a leaf and one of its neighbours. RMAs, and sparse arrays in general, require *rebalances*, a *global* operation, they may affect multiple, potentially all, segments of the array.

Figure 4 summarises the complexity of the operations of (a, b)-trees and sparse arrays. RMAs match the same complexity of APMA. In general, (a, b)-trees have an edge in updates, while sparse arrays in scans. The reason the cost per update, in skewed scenarios, is constant in (a, b)-trees is both because node splits/merges actually exhibit a penalty of $O(1)$ in amortised sense [24], and because the root-the-leaf path is consistently cached. In this scenario, even with adaptive rebalancing, the tightest upper bound for RMAs can only be $O(log_2 N/B)$.

	(a,b)-tree	TPMA	APMA / RMA
Updates, uniform	$O(log_B N)$ worst case	$O(log_B N)$ amortised avg. case	
Updates, sequential	$O(1)$ amortised	$O(log_B^2 N)$ amortised	$O(\frac{log_2 N}{B})$ amortised
Point lookups	$O(log_B N)$	$O(log_B N)$	
Range scans, O(R) elts	$O(\frac{R}{B})$, random access	$O(\frac{R}{B})$, sequential access	

Fig. 4. Summary of the theoretical complexity of (a, b)-trees and sparse

[4] Note that, as described in Section II, it holds $O(log_2 N) = O(log_2 C)$, due to $N \propto C$.

[5] The first segment starts with index 1.

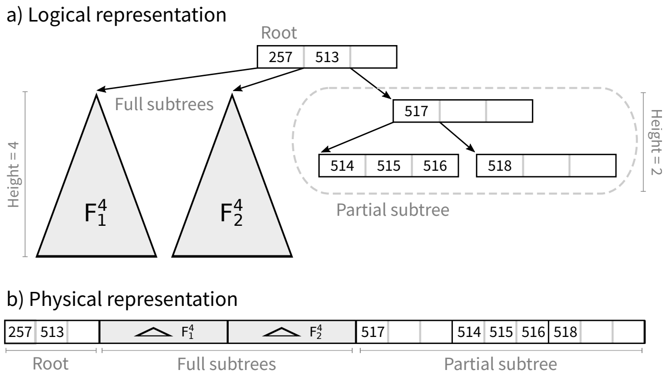


Fig. 5. The representation of an index of size 518 and fanout $f = 4$. The height of the tree is 5. F_i^4 is the i-th child of the root, a full subtree of height 4. The third child of the root is a partial subtree of height 2. The content of the slots are the separator keys for the related segments, e.g. the value 257 in the root is the separator key for segment 257 of the underlying RMA.

Point lookups have the same cost. Indeed, the same index used for the leaves of (a, b)-trees can be adopted for the segments of sparse arrays. Notwithstanding, rebalances in PMAs can shift the keys over large spans of the array, causing an additional maintenance burden to the index. To ease this overhead, our design is based on a different kind of index.

Index. RMAs maintain a *static* index to improve point lookups and updates. It is static because, once built at a RMA resizing, it contains a fixed a number of entries. Still, the single entries can be altered, which happens during RMA rebalances. The index does not contain explicit pointers to traverse the nodes. Only the separator keys are stored, packed together in a contiguous array. Node traversals are performed by computing the offset of nodes from the current position in the array.

Figure 5 depicts the logical and physical representation of the index. As the inner nodes of B^+-trees, nodes in the index have a fixed capacity and a maximum fanout f. The root of the index forms a *partial subtree* P^h of height h. The root node contains r separator keys, with $1 \le r \le f - 1$, and $r + 1$ children. The leftmost r children are *full subtrees* F^{h-1} of height $h - 1$, while the rightmost child is a, possibly empty, partial subtree $P^{\bar{h}}$ of some height $\bar{h} : 0 \le \bar{h} < h$. A full subtree F^h is a subtree of height h, where all nodes are filled with $f - 1$ separator keys. Thus, it stores exactly $f^h - 1$ separator keys in total. The order of the separator keys and the logic for the node traversal is analogous to B^+-trees.

The indexed elements are implicitly the segments in the underlying RMA. Similarly to B^+-trees, the separator keys are the minimums in each segment. When the underlying RMA is resized, the index is recreated with the new number of segments. Compared to (a, b)-trees, node traversals are cheaper, as the memory footprint of the index is smaller, and the memory distances between parents and children shorter.

Rebalancing. Our design makes use of memory rewiring to reduce the execution time of rebalances. The common approach to rebalance elements inside an interval operates in two passes. In the first pass, proceeding backwards, all elements are moved and compacted either towards the right end of the interval or in auxiliary additional storage. In the second pass, proceeding forwards, the elements are copied to their final positions in the array. Our solution still follows this scheme if the interval to rebalance is smaller than a virtual page. Otherwise, we directly move and directly spread the elements from the pages of the array to rebalance p_{array} into a set of unused physical pages p_{buffer}. Eventually, we swap one by one the virtual addresses between p_{array} and p_{buffer}. The advantage of this technique is that it only performs a single copy per element, rather than two.

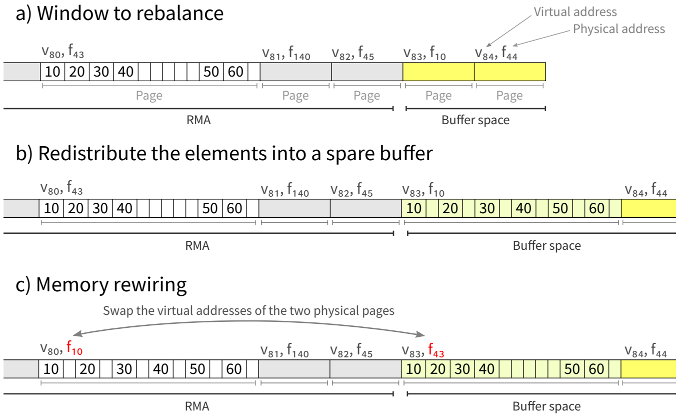


Fig. 6. The rebalancing procedure with the use of memory rewiring. The pictures depict the pages of both the RMA and the buffer area, together with their virtual v_n and physical f_m addresses. a) The input window to rebalance, it occupies exactly one page. b) The elements of the input window are redistributed in a spare page of the buffer space (ignoring clustering for simplicity). c) The virtual addresses of the buffer page and of the page rebalanced are swapped.

Specifically, memory rewiring is a technique to explicitly control the mapping between virtual (logic) addresses and their associated physical pages [29]. Besides the space used by the RMA, we maintain a set of spare buffers, allocated on demand. On rebalance, the elements are redistributed into the buffer space (see Figure 6b). Then, using memory rewiring, the pages from the buffer become part of the array, while the old physical pages of the array become spare buffers to be reused in a future rebalance (see Figure 6c). Actually, the single pages of the array can be rewired as soon as their elements have been redistributed, making them immediately available for reuse.

In most cases, we treat resizes as a special case of a rebalance. Conceptually, a resize is a rebalance where the elements are redistributed from an interval having the current capacity of the array to an interval of the new capacity. The idea is to reserve a large virtual memory area, say 2^{37} bytes, for the RMA when it is firstly created. We arrange the physical pages for the spare buffers immediately after the storage currently used by the RMA (as in Figure 6a). When the RMA needs to be expanded, we absorb the existing spare buffers in the RMA and only request additional physical memory to reach the final capacity. If the RMA needs to be shrunk, we absorb the freed pages at the end of the RMA to the buffer area. To limit the amount of physical memory dedicated to the buffer space, we employ this scheme only when expanding the RMA or when the number of physical pages in the spare buffers is not greater than the number of physical pages used by the RMA. Otherwise, we perform a resize in standard manner, by creating a new RMA of the needed capacity and copying the elements from the old array to the new one. Note that, in both approaches, only one copy per element is performed. The benefit of memory rewiring for resizes is to alleviate the overhead in acquiring new zeroed physical pages from the operating system [29].

Bulk loading. As in B^+ trees, there are no algorithms generally improving the theoretical complexity of batch updates. In practice, the only savings that can be achieved are thanks to avoid rebalancing multiple times the same segments in a single batch. To this purpose, a *top down* scheme has been previously presented in [14]. Assuming the elements in a batch have been sorted beforehand, the key idea is to traverse the calibrator tree, starting from the root, and recursively propagate the input sequence to the children. However, if the thresholds of a given node cannot be satisfied, the algorithm will trigger a rebalance, merging the input sequence with the existing elements in the current window. Nevertheless, this scheme has a drawback: by starting from the top of the tree, where the densities are tighter, the algorithm may also cause unnecessary rebalances that would not have been issued if this procedure was not utilised in the first place.

We propose a *bottom up* strategy. First, the input sequence S of insertions is sorted. Then, the algorithm operates in three passes. In the first pass, it scans S and only alters the final cardinality of each segment, where the elements are going to be inserted. In the second pass, it scans the touched segments, checking whether the thresholds are respected. If not, it identifies the intervals that need to be rebalanced. Finally, it performs a third pass on the touched segments. If a segment has not been marked for rebalancing, it simply inserts the related elements from S. Otherwise, it rebalances the window identified in the second step, and similarly to the *top down* scheme, merges the existing elements with the related run from the sorted sequence S. Analogously to [14], the method can be extended to batches containing both insertions and deletions, by performing an initial pass where all deletions are performed, but rebalances are disabled.

Key-value split. We store the keys and their associated values in two separate arrays. This causes no harm to the hardware prefetcher in scans, while it improves point lookups, and, by extension, updates, as less memory space needs to be traversed to find a specific key.

Scan-oriented thresholds. We refine this strategy with the thresholds $\rho_1 = 0$, $\rho_h = \tau_h = 0.75$, $\tau_1 = 1$. We also add a special rule, forcing a resize if, after a deletion, the fill factor of the sparse array becomes less than 50%. The aim is to still achieve a minimum potential fill factor of 50%, while allowing the adaptive rebalancing to underfill specific regions of the array where deemed appropriate.

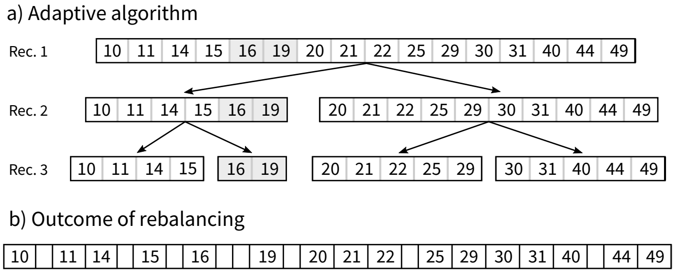


Fig. 7. a) Sample execution of the adaptive algorithm for the array of Figure 2a. b) Final layout of the sparse array after rebalancing.

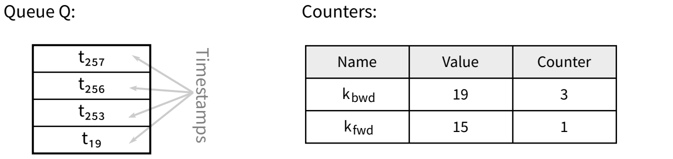
IV. Adaptive rebalancing

RMAs employ adaptive rebalancing to alleviate the number of future rebalances in presence of hammering. The adaptive strategy is split in two distinct parts. In the first part, upon each insertion, some additional metadata is collected in a custom data structure named *Detector*. The second part is the actual rebalancing mechanism, where the collected metadata is examined to influence how to redistribute the elements in the array. The core algorithm, named *adaptive algorithm*, is recursive. It starts from the top of the current calibrator subtree and, at each step, it determines the number of elements to distribute on its children.

This section is organised as follows. To aid the intuition, next we show an example of how the adaptive algorithm operates. We then describe the first part of the adaptive strategy: what data is collected upon insertions and how the Detector is implemented. The next paragraphs delve into the second phase: the actual rebalancing and its steps. So far, the description assumes that only insertions can be executed, at the end we present the extensions required to support deletions.

Example. Figure 7a depicts a sample execution of the adaptive algorithm for the same array of Figure 2a. Again assuming that the last insertions in the array were 14, 15 and 16, the rebalancing procedure predicts that the new insertions will continue this sequence. It creates a *marked interval* I, highlighted in gray in Figure 7a, with the pair [16, 19]. A marked interval states that new insertions are expected in the represented range, in this case between 16 and its successor, 19. The goal of the adaptive algorithm is to push I towards the least dense region of the array.

At the first recursion level, the run consists of all elements in the topmost node of the calibrator tree. The algorithm transfers the elements before I to its left child, and all elements after I to its right child. Then, it transfers I to the child with less elements, the left child in this case. In the end, the left child receives the first 6 elements of the run and the right child the remaining 10. At the second recursion level, the process is repeated for the nodes at height 2. In the first node, it transfers the first 4 elements before I to its left child, and 0 to its right child. Then, it moves I to its right child. In the second node of height 2, there are no marked intervals, and the elements are equally distributed between its children. At the third recursion level, the algorithm reaches the leaves/segments



Name	Value	Counter
k_{bwd}	19	3
k_{fwd}	15	1

Fig. 8. Sample metadata associated to a segment, as stored in the Detector.

of the calibrator tree and the recursion stops. The output of the algorithm are the target cardinalities of the segments: [4, 2, 5, 5].

Eventually, the elements are rearranged in the RMA according to these cardinalities, producing the array of Figure 7b. Note that, the layout used to arrange the elements inside a segment is orthogonal to the adaptive algorithm. In our RMA design, elements are actually clustered towards the boundaries of the segments, as described in Section III.

Detector. The Detector contains metadata to identify the intervals subject to recent updates. In our design, the granularity of a marked interval can either be the whole content of a single segment or, as showed in the example of Figure 7, a pair of two consecutive elements. For each segment of the array, we associate a supplementary data structure, sketched in Figure 8, consisting of a fixed length queue Q and two keys, k_{bwd} and k_{fwd}, together with an associated counter per key.

Algorithm 1 sketches the code to update the segment's metadata, invoked after each insertion. It first records in Q the timestamp of the current operation. The timestamp can be obtained by either some discrete global counter or the CPU timestamp counter. Moreover, it checks whether the successor $succ_k$ or the predecessor $pred_k$ of the key k being inserted matches k_{bwd} or k_{fwd}, respectively. In case one of the two keys match, the associated counter is incremented by 1, up to a maximum threshold SC. Otherwise, both counters are decremented by 1. When a counter reaches 0, the associated values of k_{bwd} and k_{fwd} are replaced with $succ_k$ and $pred_k$.

The purpose of the queue is to record the times of the last insertions in a segment. Intuitively, by comparing the timestamps of all segments in a window to rebalance, the algorithm infers if recent insertions have occurred only in a limited set of segments. The additional counters allow, instead, to identify sequential insertion patterns. For instance, in Figure 8, the value 3 associated to k_{bwd} implies that, at least 3 times in the last insertions, the successor of the key just inserted was 19. Therefore, the algorithm may guess that, in future, new insertions might likely be in the range [predecessor$(19), 19]$.

Rebalancing procedure. The adaptive algorithm integrates and augments the traditional rebalancing procedure of TPMA. As in TPMA, the first step of the the whole procedure is to find the window W to rebalance. This operation is accomplished by traversing and validating the density thresholds of the calibrator tree. The second step, named *preprocessing phase*, consists of producing a set of marked intervals, exploiting the information in the Detector. If no marked intervals are created, the rebalancing procedure proceeds as in TPMA, by even spreading all the elements in W. Otherwise, the third step is the actual adaptive algorithm. Traversing the calibrator tree top-down, the algorithm determines the amount of elements to place on each child, transferring the marked intervals in the nodes with the least cardinality. The output of this step are the target cardinalities of all segments in W.

Algorithm 1 Update segment metadata

```
Input:
    $k$                          ▷ The key just inserted
    $pred_k$                     ▷ The predecessor of $k$ in the array
    $succ_k$                     ▷ The successor of $k$ in the array

$Q$.APPEND($Read\_CPU\_TimeStamp$)

if $succ_k = k_{bwd}$.value then
    $k_{bwd}$.counter ← MIN($k_{bwd}$.counter $+ 1, SC$)
else if $pred_k = k_{fwd}$.value then
    $k_{fwd}$.counter ← MIN($k_{fwd}$.counter $+ 1, SC$)
else
    $k_{bwd}$.counter ← MAX($k_{bwd}$.counter $- 1, 0$)
    $k_{fwd}$.counter ← MAX($k_{fwd}$.counter $- 1, 0$)
    if $k_{bwd}$.counter $= 0$ then
        $k_{bwd}$.value ← $succ_k$
    if $k_{fwd}$.counter $= 0$ then
        $k_{fwd}$.value ← $pred_k$
```

No elements of the array are physically copied during the execution of the adaptive algorithm. Only at its end, when the target cardinalities of all segments in W have been determined, the elements are redistributed in the array. We define by $pos(k)$ the position of the key k in the sorted sequence of all keys present in W. At all stages, the algorithm represents an interval I as a pair $\langle s, l\rangle$, where $s = pos(k_{\text{first}})$ and k_{first} is the first key that belongs to I, and $l = |I|$ is the size of the interval. To *transfer* elements between nodes in the calibrator tree, the algorithm actually assigns their representing intervals.

Preprocessing phase. It works as follows. First, it computes the 99.9 percentile p of all timestamps in the metadata for the segments in the window W being rebalanced. Second, it marks all segments such that 75% of their timestamps are greater than p. Third, for any marked segment, it either emits a marked interval of size 2 with k_{bwd} or k_{fwd} if their associated counter is greater than a given threshold θ_{SC}, or, otherwise, a marked interval representing all elements in the segment. The final output of this phase is a sequence of marked intervals.

Adaptive algorithm. The adaptive algorithm is a top-down traversal of the calibrator subtree, rooted at W. For each node u, it keeps track of the current interval R of elements assigned, and of the marked intervals ν inside R. In the first iteration, $u = W$, $R = \langle 1, card(W)\rangle$, and ν is the sequence of marked intervals computed in the preprocessing phase. In the last iteration, the base case, u is a segment and its target cardinality is $|R|$. Algorithm 2 sketches the pseudo code.

The core of the algorithm starts at line 7. The objective function attempts to redistribute the same amount of marked intervals to each child. If this cannot be achieved, e.g. with $|\nu| = 1$, the remaining marked intervals are moved to the child with the least cardinality. Figure 9 shows the outcome of redistribution with different sets of ν. In the pseudo-code

Algorithm 2 Adaptive algorithm

```
Input:
    $u$                              ▷ The current window/node in the calibrator tree
    $R$                              ▷ The sequence of elements allocated to $u$
    $\nu$                            ▷ The list of marked intervals present in $R$
Output:
    $T$                              ▷ The target cardinalities of all segments $s \in u$

 1: if $|u| = 1$ then                ▷ Base case, the node consists of only one segment
 2:     $T[1] \leftarrow |R|$
 3: else if $|u| = 2 \wedge \nu$ is "too big" then       ▷ Split the marked interval
 4:     $T[1] \leftarrow |R|/2$
 5:     $T[2] \leftarrow |R|/2$
 6: else                  ▷ Determine the amount of elements to transfer to the left child
 7:     $R_{\text{left}} \leftarrow$ OBJECTIVE FUNCTION$(R, \nu)$
 8:     Let $\bar{C}, \bar{\rho}, \bar{\tau}$ the capacity and the density thresholds of u's childrens
 9:     $min =$ MAX$(|R| - \bar{\tau} \cdot \bar{C}, \bar{\rho} \cdot \bar{C})$        ▷ Minimum cardinality allowed
10:     $max =$ MIN$(|R| - \bar{\rho} \cdot \bar{C}, \bar{\tau} \cdot \bar{C})$        ▷ Maximum cardinality allowed
11:     if $|R_{\text{left}}| < min$ then
12:         Add to $R_{\text{left}}$ at least $min - |R_{\text{left}}|$ elements
13:     else if $|R_{\text{left}}| > max$ then
14:         Remove from $R_{\text{left}}$ at least $|R_{\text{left}}| - max$ elements
15:     $R_{\text{right}} \leftarrow R \setminus R_{\text{left}}$
16:     $T \leftarrow$ ADAPTIVE ALGORITHM(left child of u, $R_{\text{left}}$, $\nu \cap R_{\text{left}}$)  ||
            ADAPTIVE ALGORITHM(right child of u, $R_{\text{right}}$, $\nu \cap R_{\text{right}}$)
```

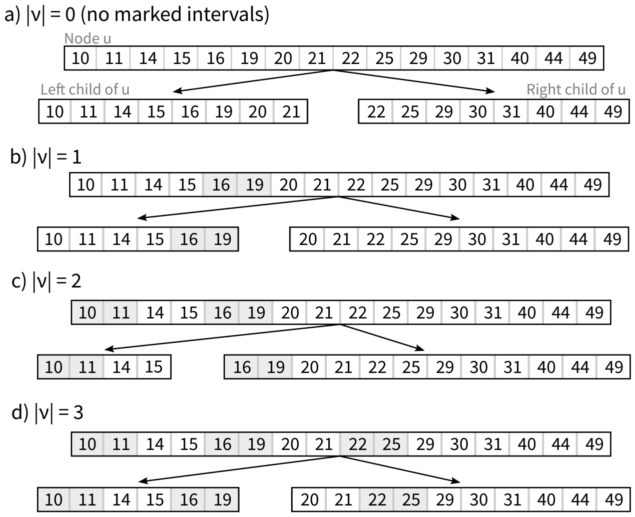


Fig. 9. Sample redistribution of the elements from a node u to its children, with different number of marked intervals $|\nu|$. The marked intervals ν are highlighted in gray.

(line 7), the function only determines the interval R_{left} for the left child, the interval for the right child can be simply derived as $R_{\text{right}} = R \setminus R_{\text{left}}$.

Before proceeding to the next recursion level, the target cardinalities of the nodes might need to be sanitised. Intuitively, if the density $\delta(v)$ of a child v becomes less than its lower threshold ρ_v, it borrows as many elements as necessary from its sibling, so that $\delta(v) \geq \rho_v$. Similarly, if $\delta_v > \tau_v$, elements are transferred to its sibling so that the upper density threshold is respected. The lines at 9 - 14 express the above logic, but in terms of the minimum and the maximum cardinality of the left child, rather than the densities of both children. It is this sanity check that guarantees the data structure matches the worst case amortised complexity $O(\frac{log_2^2 N}{B})$ per insertion.

Deletions. To support deletions, we introduce a few extensions to the components described so far. Intuitively, in presence of skew for deletions, we want to push the elements where new deletions are predicted to follow, into denser areas of the array. We deal with the conflicting aims of insertions and deletions through a simple scoring system. The idea is to assign a score (or *weight*) of $+1$ to each marked interval if it is due to frequent insertions, and -1 if it is due to frequent deletions. The objective function of the adaptive algorithm becomes to find a partitioning of the marked intervals ν, so that both children receive the same cumulative score and, roughly, the same amount of marked intervals $|\nu|/2$. Furthermore, in the Detector, we associate a third counter sc to each segment, incremented upon each insertion and decremented upon each deletion, up to a given maximum $|sc| < SC$. Finally, in the preprocessing phase, a segment can be marked only if also $|sc| \geq \theta_{SC}$, assigning a score of +1 to the marked interval if sc is positive, and -1 otherwise.

V. Evaluation

We evaluate our design and the effects of the segment size, the adaptive rebalancing, the density thresholds and batch updates. We conclude with a summary of the cumulative contributions of the main features described in the paper. As competitors, we consider both a standard, with separator keys, and a trie-indexed (a, b)-tree. We refer to the latter as ART: it is still actually an (a, b)-tree, but the leaves are this time indexed by ART [22][6], a form of trie. In both implementations, in the scans, we issue memory prefetch instructions to the accesses of each next leaf.

In all experiments, the elements loaded consists of 8 byte key/value integer pairs. The maximum capacity of the internal nodes, both for the static index of the RMA and for the standard (a, b)-trees, is fixed to 64 separator keys, an optimum determined by a series of micro-benchmarks. The leaf and segment capacities, except for the first experiment, are fixed to $B = 128$ elements. Therefore, each leaf of an (a, b)-tree takes roughly $2kB$ of memory (plus some metadata). Both keys and values are stored sorted inside the segments/leaves.

The experiments have been conducted on dual socket cpus Intel Xeon E5-2650 @ 2GHz, with 256 GB of memory in total. The code has been written in C++ and compiled with Clang v6.0. The code is sequential, both the CPU and the memory node are pinned at the start of an execution. Memory rewiring is performed on huge pages of 2MB. Each experiment has been repeated 15 times. The reported results refer to the median, unless stated otherwise.

Node and segment size. Figure 10 mimics the same experiment showed in the introduction. This time ART is compared with our version of the RMA. The experiment starts by inserting $1G = 2^{30}$ elements according to a uniform distribution. Figure 10a depicts the average throughput, per insertion, while the cardinality of the data structure increases. Figure 10b shows the average throughput to perform $1M$ random point look-ups of existing keys. Figure 10c reports the average

[6] Our implementation is based on the publicly available sequential source code of ART [1].

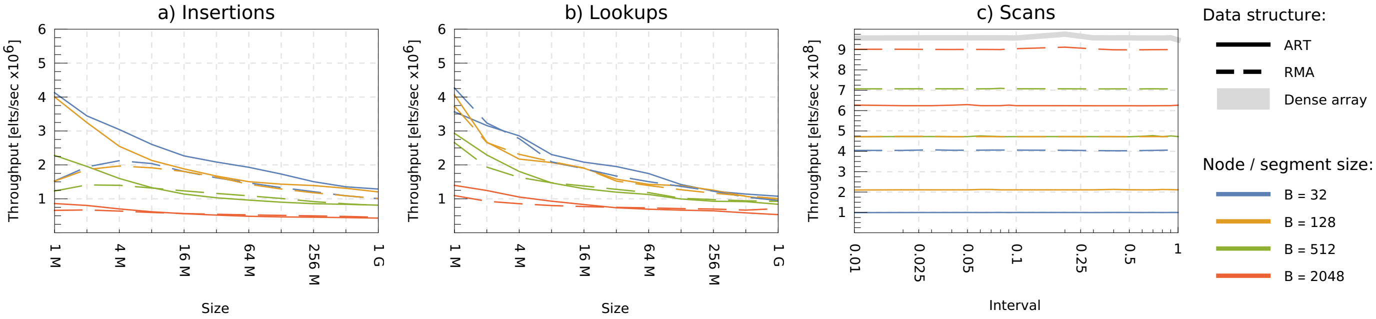


Fig. 10. Average throughput for a) insertions, b) point look ups and c) scans at varying node sizes B for the leaves of an (a,b)-tree, indexed through ART, and the segment sizes B of the RMA.

throughput, per element, to sum the values in a contiguous region, from an interval of 0.1%, up to all elements, when the data structures contain $1G$ elements.

At corresponding node and segment sizes B, while the throughput of point look-ups is comparable, ART leads in the insertion throughput for smaller values of B, but scans are always significantly slower. For instance, with $B = 128$, ART is 20% faster for insertions, whereas with $B = 512$, the difference disappears. On the other hand, for a fixed segment size of the RMA, ART can achieve the same throughput for scans, by increasing its leaf size. For instance, ART with $B = 512$ matches the same throughput in scans of the RMA with $B = 128$. However, due to the larger leaf size, insertions are now also 25% slower. In general, by both increasing the node and the segment size, the difference in scans becomes less prominent, from $4x$ with $B = 32$, up to 40% with $B = 2048$. At the same time, by increasing B, also the throughput for insertions decreases in absolute terms. Finally, note that with $B = 2048$, the RMA almost matches the scan throughput of static dense arrays.

We believe that our choice of ART represents a strong competitor. Compared to our custom implementation of a standard (a, b)-tree, employing separator keys, at equal leaf capacity, ART is always faster or alike in terms of insertions and look-ups, while obtaining the same performance for scans. Indeed, the leaves of both ART and the standard (a, b)-tree feature the same layout. The actual relative difference in performance depends on the leaf size. For $B = 128$, in this experiment, ART is 20% faster in insertions and roughly 12% in point look-ups. However, with $B = 2048$, there is no practical difference anymore. For reference, in comparison to the STX-Tree [11], again at the equal leaf capacity of $B = 128$, our tuned ART implementation, in this experiment, is roughly $25\% \sim 30\%$ faster in insertions, about 12% faster in point look-ups and 30% faster in scans.

Adaptive rebalancing. Figure 11 compares the average update throughput of RMA with and without adaptive rebalancing (AR) enabled, in presence of skew. In the experiments, we insert and delete the elements according to both the uniform and the Zipfian distribution of range $\beta = 2^{27}$, while varying the amount of skew through the Zipf factor α. Figure 11a refers to the scenario of only inserting 1G elements in an empty data structure. Figure 11b represents a mixed workload where, starting with $1G$ elements, sequences of $\gamma = 1024$ contiguous insertions are interleaved by γ contiguous deletions. The distributions are initialised with different seeds for insertions and deletions. Consequently, insertions and deletions "hammer" different portions of the array. We also include the results of ART for completeness.

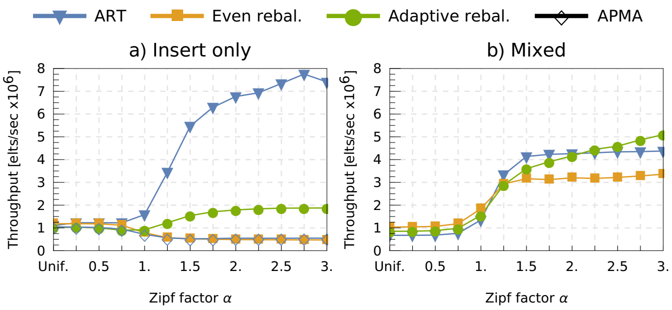


Fig. 11. Average throughput, per element, at different Zipf factors α, when a) inserting 1G elements in an empty data structure, b) fixing the cardinality of the data structure to 1G, repeatably executing 2^{10} insertions followed by an equal number of deletions.

The difference in performance emerges as the skew in updates increases. With moderate skew, AR exhibits an overhead of about 20% w.r.t. the traditional policy of even rebalancing. This is due to the cost of updating the metadata in the Detector, noticeable in our set up of fine grained elements of 16 bytes. At $\alpha \geq 1.5$, with rebalances notably more frequent, adaptive rebalancing allows to achieve a 3x - 4x throughput in the insert only scenario, and 1.2x - 1.5x in the mixed workload.

In absolute terms, the mixed workload is where the RMA shines. The deletions performed compensate for the space filled by the carried insertions. In comparison, in the insert only scenario, ART can still be up to 4x faster than RMA with adaptive rebalancing in presence of significant skew, whereas it generally yields similar performance in the mixed workload. Nevertheless, although not shown here due to lack of space, we note that, for the mixed workload, by increasing the window γ, in the RMA the average throughput of updates proportionally decreases, eventually reaching the same performance of the "insert only" workload. However, bulk loading can become a better alternative in this context.

We also re-implemented the rebalancing algorithm of [10],

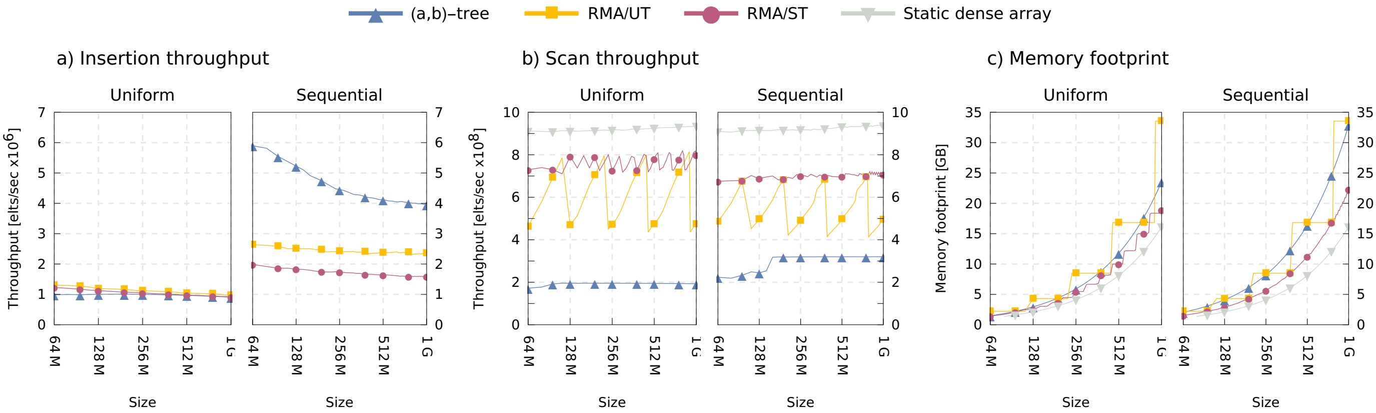


Fig. 12. Average throughput for a) insertions and b) scans with the Update Thresholds (UT) and Scan Thresholds (ST), at equal segment sizes. Elements are inserted following to a uniform distribution and the sequential pattern. Plot c) shows the memory footprint, in GB, used by the whole data structure.

marked in the graph as APMA[7]. In Figure 11a, there are only marginal differences w.r.t. even rebalancing. The cause is mainly the *ping-pong* effect, which hinders the benefits of its rebalancing algorithm. Note that, we can generally reproduce the experiment results of [10]. For instance, in the sequential pattern, where elements are only appended at the end of the array, APMA is 3.8x faster than even rebalancing. Still, their experiments only exhibit insertions at predetermined positions of the array, without taking into account a sorted order, and the above effect does not appear. Furthermore, even in the sequential pattern, our adaptive rebalancing algorithm is still 25% faster than APMA. Finally, as [10] does not support deletions, it has not been evaluated for the mixed workload.

Density thresholds. Figure 12 compares the Update-oriented Thresholds (UT) and the Scan-oriented Thresholds (ST). In this experiment, starting from an empty data structure, we insert $1G$ elements, using the uniform distribution and the sequential pattern. Moreover, at different stages, we measure the respective throughput of full scans and the memory footprint of the whole data structure. Although not reported, we similarly evaluated the Zipfian distribution, obtaining intermediate results with respect to Figure 12. The ST are those determined in Section III. The UT are derived with $\rho_1 = 0.08$, $\rho_h = 0.3$, $\tau_h = 0.75$ and $\tau_1 = 1$, mimicking the configuration of previous work [10], [14], [30]. These are also the density thresholds employed in the rest of the experiments.

In general, compared to the ST, the UT provide $10\% \sim 40\%$ speed up in insertions, but are, on average, 20% slower in scans. For updates, the difference becomes smaller for the uniform distribution, and more marked for the sequential scenario. For scans, the UT achieve the same peak performance of the ST, registered in proximity of a resize. Just after a resize, the performance immediately drops by 40%, as consequence of the array being sparser and, ultimately, producing a *zig-zag* pattern in the graphs. The memory footprint of the RMA, with the ST, is about 1.4x bigger than static dense arrays. With the UT, the difference varies, up to 2x the optimal space of dense arrays.

(a, b)-trees favour the sequential scenario. Nevertheless, while the peak throughput for insertions is expected, the outcome of scans being 60% faster in the sequential scenario than with the uniform distribution, represents an artifact of our benchmarks. In this scenario, sibling nodes are allocated closely in the memory space, significantly reducing the distance of memory jumps in scans. In reality, this characteristic rapidly vanishes once updates start to alter the leaves in the middle of the (a, b)-tree. Figure 13a shows the average throughput of scans, where, after loading an empty (a, b)-tree with a sorted batch of $1G$ elements, sequences of $1M$ random insertions are repeatably followed by the same amount of deletions. Already after altering 5% of the element in the (a, b)-tree the throughput of scans decreases by 25%. In general, the performance of (a, b)-trees deteriorate with their usage, resembling a form of “aging”.

Lastly, for reasons of space, we only summarise the actual costs of rebalances of the RMA. In our experiments, the maximum latency we measured for an insertion was within 10 seconds, occurring only once per experiment, when the RMA doubled its capacity from $16Gb$ to $32Gb$. Still, the 99th percentile for the latency of insertions in all our experiments was under $3\mu s$. Rebalances are responsible between 2% (uniform) and 50% (highest skew) of the cost of insertions. The execution of the adaptive algorithm, together with its preprocessing phase, accounts, on average, for about 10% of the overall cost of all rebalances.

Bulk loading. Figure 13b reports the insertion throughput for batch loads. In the experiment, we compare the bottom-up approach, described in Section III, with the top-down implementation of [14]. The experiment first fills the data structure inserting 512M elements following a uniform distribution. Then, it further loads 512M elements in batches of 1M, corresponding to 1% - 2% of the whole data structure. For a fair comparison with [14], we considered both the case with memory rewiring enabled (+RWR) and disabled (-RWR).

The insertion rate for bulk loads adequately copes with skew. Both schemes, bottom-up and top-down, have a robust

[7] The original source code of APMA was never openly released by [10].

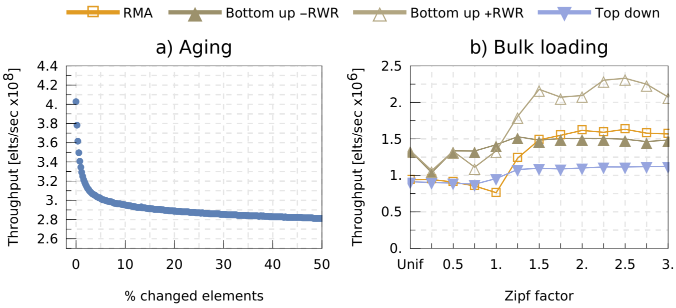


Fig. 13. a) Average throughput, per element, to scan the elements of an (a, b)-tree of cardinality 1G, in relation to the amount of updates sustained. b) Average throughput, per element, to load 512M elements in batches of 1M, starting from the data structure containing 512M elements.

average latency. Still, even without memory rewiring, the bottom-up approach is about 50% faster for moderate skew and 30% for high skew than the top-down scheme. With memory rewiring enabled, instead, bulk loading is, on average, 40% faster compared to single insertions. In general, because each batch is sorted before being loaded, bulk loading takes advantage of a certain degree of locality when performing the insertions, while providing an alternative mechanism to adaptive rebalancing to deal with skew.

Contributions. Figure 14 inspects the cumulative impact of our contributed features in the same experiment described in the introduction. The first two features, clustering and fixed size segments, improve the performance of scans, achieving a 2x speed-up w.r.t. our baseline. The rightmost features, instead, deal with the cost of point-lookups and updates, finally reaching a 9x speed-up in the sequential pattern.

In clustering, elements are split in two separate arrays for the keys/values and packed towards the boundaries of the segments. It improves the cost of scans, by avoiding the checks on empty gaps. It also hampers the cost of rebalances, as the separator keys on the index needs to be altered to match to the minimum of each segment/cluster.

Fixed-size segments replace the variable-length segments from $O(log_2 N)$, a remnant of the RAM model, to a fixed capacity $O(B)$, set according to the I/O model. Although it also transforms the data structure from cache oblivious to cache aware, fixed-length segments create even larger contiguous chunks, favouring scans, while avoiding the rebalances on the lowest levels of the original calibrator tree.

The static index improves both look-ups and updates. The main advantage is that altering the value of a given entry in the index is $O(1)$. The benefit is particularly noticeable in presence of skew, as more rebalances are performed, and more updates to the index are carried. Memory rewiring further improves the cost of updates by about 20%. It saves the cost of multiple copies per element in rebalances, and exhibits a lower overhead in acquiring the memory from the O.S. in resizes. Finally, while adaptive rebalancing brings a certain amount of overhead, about 20% in the uniform distribution, it makes the data structure much more robust in presence of skew, transforming the worst-case of TPMA in a best-case.

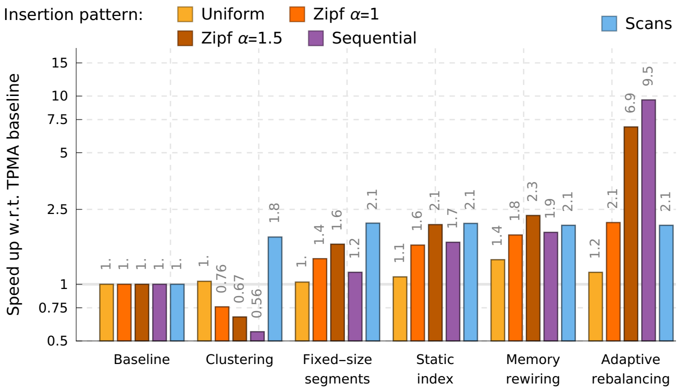


Fig. 14. Contributions of the design features of RMA's performance. The values on the bars represent the cumulative speed-up of each feature relative to the TPMA baseline.

VI. Related work

The *Traditional PMA*[8] has been originally proposed and characterised for the RAM model by [18]. The data structure has been further simplified, refined and extended to the I/O model by [3], [4]. The *Adaptive PMA* has been invented by [10] to cope with hammer insertions. Reference [17] remarks that the calibrator tree for rebalancing is actually unnecessary, leading to a form of circular array. The recent paper [8] deals with the problem of *dearmortisation*, i.e. reducing the (non amortised) worst case complexity of updates, and surveys some of the earlier schemes proposed.

PMAs have found some traction in cache-oblivious data structures, in particular Cache-Oblivious (CO) B-Trees [3]–[6], [9], [12]. It is theoretically possible to create a B-Tree achieving the lower bound $O(log_B N)$, in amortised sense, for look-ups and updates, and $O(max\{\frac{R}{log_2 N}, \frac{R}{B}\})$ for scans of R consecutive elements, without explicitly tuning for a particular block size B. Although there exist many variants, the main scheme is similar and relies on three layers. The first layer is an index, in the form of a binary tree in the van Embde Boas layout [3], [27]. This index references a second layer, a sorted PMA of size $O(N/log_2 N)$. The items in the PMA are pointers to memory chunks, which compose the third layer of the tree. These chunks resemble the leaves of a traditional B^+ tree. A chunk has a capacity $O(log_2 N)$ and is the place where elements, keys and payload are stored. CO B-Trees have been studied mainly theoretically. Some results for simpler designs that do not match these bounds are [6], [12], [19].

The index employed in the RMA relies on the technique of *pointer elimination*, firstly presented for B^+-trees in CSS-trees [28]. A similar procedure, referred as static and implicit index, was independently proposed by [5], [12] for PMAs. In particular, the index employed in the RMA was strongly influenced by [12]. Nevertheless, in [12], the index embeds the whole PMA, following a binary tree in the van Emde Boas layout. In our design, the PMA and the index are different components, conceptually resembling the structure

[8] Originally named as *hierarchical sparse table*.

of a B^+ tree. Furthermore, the index is a dense array, where the intermediate nodes have a large fan-out.

There have been only few practical published applications that rely on sparse arrays. In [14], a sparse array is employed to maintain the elements sorted according to the Z-order for the problem of neighbour search. The overall size of the array remains constant. Rather, the elements can be moved in batch of updates. A similar approach has been considered in the recent paper [31], where sparse arrays are considered to store and process temporal streams of tweets. In [30], the PMA is utilised to store a graph on GPU following the compressed storage row (CSR) representation. The paper discusses and evaluates a few alternative protocols for concurrent batched updates. In [20], single attributes of relational tables are materialised and maintained sorted into sparse arrays, for the computation of inequality joins. In [23], sparse arrays are evaluated to compute shortest paths of dynamic road networks. All the above applications are based on Traditional PMAs, while updates in their evaluation use a uniform key distribution.

VII. Conclusions

The purpose of the RMA is to provide fast column-oriented scans, while being close to (a, b)-trees in terms of updates. PMAs are interesting in that they maintain data in physical sequential order under updates, but we observed that all existing PMAs do not reach the performance of properly tuned (a, b)-trees in either update or scans. We then proposed, refined and evaluated several features to overcome the underlying penalties: fixed size segments, clustering, the static index, memory rewiring and adaptive rebalancing. In our experiments, at equal node/segment capacities, our RMA always has a strong lead over (a, b)-trees in terms of scans, and even matches their performance for updates in uniform or low skew distributions. For higher skew, due to adaptive rebalancing, the RMA still retains robust behaviour.

While we focused on the in-memory scenario, we note that RMAs can be adapted to out-of-memory scenarios. This can be achieved either by a variation of memory rewiring (the R in RMA), or memory-mapped files.

Currently, we recognise two constraints in the adoption of sparse arrays: concurrency and the amortised bounds in their performance, rather than absolute bounds. To deal with the occasional peaks in the latency of single updates (a feature common to LSM designs [26]), caused by resizes or rebalances, we envision two possible complementary solutions. A first possibility is the usage of system transactions [15], where parts of the data structure are *asynchronously* rearranged. A second possibility is to combine multiple RMAs, with a fixed maximum capacity, as the leaves of a large B^+-tree. General concurrency, instead, has not been thoroughly evaluated yet in sparse arrays. The global nature of rebalances, the contiguous allocation of elements, the shift towards scan performance, rather than updates, are peculiar characteristics of sparse arrays that provide an interesting agenda of future research, now justified by our good scan and update performance.

Our code for the RMA will be released in open source at https://github.com/cwida/rma/.

References

[1] ART synchronized, 2016. Available online at: github.com/flode/ARTSynchronized.

[2] A. Aggarwal and J. Vitter. The input/output complexity of sorting and related problems. *CACM*, 31(9):1116–1127, 1988.

[3] M. Bender, E. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. In *FOCS*, 2000.

[4] M. Bender, E. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. *SIAM J. Computing*, 35(2):341–358, 2005.

[5] M. Bender, Z. Duan, J. Iacono, and J. Wu. A locality-preserving cache-oblivious dynamic dictionary. *J. of Algorithms*, 53(2):115 – 136, 2004.

[6] M. Bender, M. Farach-Colton, and B. Kuszmaul. Cache-oblivious string B-trees. In *ACM PODS*, 2006.

[7] M. Bender, M. Farach-Colton, and M. Mosteiro. Insertion sort is O(N log N). *Theor. Comp. Sys.*, 39(3):391–397, 2006.

[8] M. Bender, J. Fineman, S. Gilbert, T. Kopelowitz, and P. Montes. File maintenance: When in doubt, change the layout! In *ACM-SIAM SODA*, 2017.

[9] M. Bender, J. Fineman, S. Gilbert, and B. Kuszmaul. Concurrent cache-oblivious B-trees. In *ACM SPAA*, 2005.

[10] M. Bender and H. Hu. An adaptive packed-memory array. *TODS*, 32(4), 2007.

[11] T. Bingmann. STX B+ tree C++ template classes v0.9, 2013. Available online at: github.com/bingmann/stx-btree.

[12] G. S. Brodal, R. Fagerberg, and R. Jacob. Cache oblivious search trees via binary trees of small height. In *ACM-SIAM SODA*, 2002.

[13] U. Drepper. What every programmer should know about memory. Technical report, Red Hat, Inc., 2007.

[14] M. Durand, B. Raffin, and F. Faure. A packed memory array to keep moving particles sorted. In *VRIPHYS*, 2012.

[15] G. Graefe. Modern B-tree techniques. *Foundations and Trends in Databases*, 3(4):203–402, 2011.

[16] S. Héman, M. Zukowski, N. Nes, L. Sidirourgos, and P. Boncz. Positional update handling in column stores. In *ACM SIGMOD*, 2010.

[17] A. Itai and I. Katriel. Canonical density control. *Inf. Process. Lett.*, 104(6):200–204, 2007.

[18] A. Itai, A. Konheim, and M. Rodeh. A sparse table implementation of priority queues. In *Proc. Colloquium on Automata, Languages and Programming*, pages 417–431, 1981.

[19] Z. Kasheff. Cache-oblivious dynamic search trees. Master's thesis, MIT, 2004.

[20] Z. Khayyat, W. Lucia, M. Singh, M. Ouzzani, P. Papotti, J.-A. Quiané-Ruiz, N. Tang, and P. Kalnis. Fast and scalable inequality joins. *VLDB Journal*, 26(1):125–150, 2017.

[21] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear. The Vertica analytic database: C-store 7 years later. *PVLDB*, 5(12):1790–1801, 2012.

[22] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *ICDE*, 2013.

[23] G. Mali, P. Michail, A. Paraskevopoulos, and C. Zaroliagis. A new dynamic graph structure for large-scale transportation networks. In *Algorithms and Complexity*, pages 312–323. Springer, 2013.

[24] K. Mehlhorn and P. Sanders. *Algorithms and Data Structures: The Basic Toolbox*. Springer, 2008.

[25] P. Montes. Packed-memory array, 2014. Available online at: github.com/pabmont/pma.

[26] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[27] H. Prokop. Cache-oblivious algorithms. Master's thesis, MIT, 1999.

[28] J. Rao and K. Ross. Cache conscious indexing for decision-support in main memory. *PVLDB*, pages 78–89, 1999.

[29] F. M. Schuhknecht, J. Dittrich, and A. Sharma. RUMA has it: Rewired user-space memory access is possible! *PVLDB*, 9(10):768–779, 2016.

[30] M. Sha, Y. Li, B. He, and K.-L. Tan. Accelerating dynamic graph analytics on gpus. *PVLDB*, 11(1):107–120, 2017.

[31] J. Toss, C. Pahins, B. Raffin, and J. Comba. Packed-memory quadtree: A cache-oblivious data structure for visual exploration of streaming spatiotemporal big data. *Computers & Graphics*, 76:117–128, 2018.