Fast Concurrent Reads and Updates with PMAs

Dean De Leo
Centrum Wiskunde & Informatica
dleo@cwi.nl

Peter Boncz
Centrum Wiskunde & Informatica
boncz@cwi.nl

ABSTRACT

Fast navigation through graphs with $O(1)$ cost relies on compact storage of graphs in dense *arrays*, but is not efficiently updatable. In this paper we propose storage of updatable graphs in *Packed Memory Arrays* (PMAs), and tackle the problem of supporting *concurrent* updates and reads. So far, there has been no work on concurrently updating PMAs. We propose two novel techniques to perform concurrent scans and updates in the data structure and evaluate our implementation against other existing alternatives, showing that PMAs can in some cases be on par with data structures optimised for writes, while providing at least one order of magnitude higher throughput for reads.

ACM Reference Format:
Dean De Leo and Peter Boncz. 2019. Fast Concurrent Reads and Updates with PMAs. In *2nd Joint International Workshop on Graph Data Management Experiences Systems (GRADES) and Network Data Analytics (NDA) (GRADES-NDA'19), June 30-July 5 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3327964.3328497

1 INTRODUCTION

Our goal is to find a data structure for storing graphs that is efficient in a workload of multiple *concurrent* updates and reads; e.g., analytics on a constantly changing graph.

Systems for graph analytics typically do not support online updates (let alone concurrent updates), preferring batch update modes. However, there are plenty of graph analytics applications, like ride sharing, (security) dashboarding on social media, or network status monitoring, that actually require immediate and concurrent updates. In order to offer efficient $O(1)$ graph navigation, many systems store graphs in Compress Row Storage (CRS): essentially as an array V of vertices containing an offset (or memory pointer) to its first outgoing edge in the edge array E. This second array E holds destination vertex IDs (offsets into V), in which edge information starting from the same source vertex is contiguous. In recent years, it has been shown that solutions taking advantage of such representations, on a single machine, can even outperform distributed frameworks running on many machines [22, 25]. However, as these representations heavily leverage the compression and reduction in the memory footprint of the stored graph, these works also assumed dealing with static data, storing the graph in dense arrays.

Another challenge in graph processing is the poor spatial and temporal locality that edge traversals exhibit [21]. This problem can be alleviated using a space-filling curve such as Hilbert, popularised on the structurally similar problem of sparse matrix multiplications [13]. As such, graph storage benefits from a data structure that can keep data *ordered*, such that all vertex references in the edge array are stored ordered by their source and destination vertex IDs, and vertex IDs follow a special order (e.g. geographical locality, approximated by a space-filling curve).

The use of dense arrays is what makes CRS read-only; inserts require (on average) moving half the data, hence are $O(|V| + |E|)$, which is prohibitive in online workloads. A potential solution would be to store V and E in B$^+$-trees, so inserts would become $O(log(V) + log(E))$ – however, the downside of that is that the cost of navigation deteriorates from $O(1)$ to $O(log(V) + log(E))$ as well, and a single graph analytics query can make billions of navigation steps. B$^+$-trees suffer additionally from *aging*: as more inserts are happening and nodes are split; the data is still logically ordered; but physically the nodes will quickly get randomly scattered in memory or on disk. This means that the desired ordering is lost, and it also means that full data scans (quite common in analytics) become significantly more expensive, as sequentiality is lost and a scan becomes an avalanche of random accesses, introducing expensive memory or I/O latencies.

Here, we concentrate on an alternative approach, based on leaving some extra empty space in the original array, to accommodate potential future insertions. **Packed Memory Arrays**[1] (PMAs), by carefully controlling the amount of empty space in the distinct regions of the original array, guarantee the same theoretical performance of dense arrays for point lookups and data scans, while featuring a cost of $O(log_2^2 N)$ per update, in amortised sense. In practice, the size of the original array increases, due to the additional empty space, by a constant factor, but memory accesses in scans still remain sequential.

However, and this is where we focus our work, PMAs expose a primary obstacle for widespread usage: they lack concurrency. When regions of the array become too dense, a PMA is *rebalanced*, by relocating a percentage of its elements from its denser regions towards its sparser regions. Differently from B$^+$-trees, where single splits/merges are local operations that only involve a node and its neighbours, a rebalance can span over large sections of the array, potentially over the whole PMA. For this reason, known concurrency protocols employed for balanced trees do not apply directly on PMAs, leaving the achievements of concurrency and scalability as a major challenge. We only know of two previous works, further discussed on Section 5, Related Work, that dealt with the issue of concurrency on PMAs, but their solutions have merely a theoretical interest [6], or aimed towards a specific use case [29].

[1]In this paper, we refer to the terms packed memory array and sparse array interchangeably.

In this paper, we address the problem of concurrency on PMAs in a more general and practical manner. We propose a centralised service to rebalance the data structure when needed. The service can temporarily close parts of the data structure and rearrange its elements in parallel, following a master/worker paradigm. We also deal with the scenario of skewed updates, usually the worst-case workload for PMAs, by allowing a certain amount of asynchronicity and delaying some operations at later times. Our final goal is to accomplish a comparable throughput in updates with competitive tree balanced solutions, while always proving superior throughput in scans.

The paper is organised as follows. In Section 2, we cover the fundamental background on Packed Memory Arrays. In Section 3 we detail our design to achieve parallelism and concurrency on sparse arrays. In Section 4, we evaluate our implementation and compare it against some of best performing existing tree based data structures: ART [18, 19], the MassTree [23] and the Bw-Tree [20, 32]. We review related work in Section 5. We discuss the application of PMAs in dynamic graphs in Section 6 and conclude in Section 7.

2 PREREQUISITES

A packed memory array is an array containing elements mixed with *gaps*. The elements are stored according to a predetermined sorted order. The gaps are empty slots, placed to accommodate potential future insertions. Supported operations are insertions, deletions, point lookups and range scans. Point lookups and range scans are akin to standard dense arrays, except that encountered gaps are either skipped or ignored.

Given its capacity C, the array is logically split into C/B contiguous regions, named *segments*, of a certain size B. Insertions can be performed into a segment until it becomes full, that is, there are no more empty gaps. Analogously, elements can be removed from a segment by replacing the occupied slot with a gap, until the segment becomes less than half full. At that point, a *rebalance* operation is carried out. The intuition is to visit one or more neighbour segments and share their elements among them.

Formally, the segments involved in a rebalance are determined by a *calibrator tree* and a set of *density thresholds*. The calibrator tree is a logical binary tree. Its leaves are the segments of the PMA. The rest of the tree is built bottom-up by grouping the lower nodes two-by-two, as in Figure 1a. Each internal node represents a *window* W with a density (or fill factor) δ_W, defined as the sum of all cardinalities in the descendant segments, divided by their cumulative capacity. Furthermore, given the final height h of the calibrator tree and a set of constants $0 < \rho_1 < \rho_h \leq \tau_h < \tau_1 \leq 1$, the following lower ρ_k and upper τ_k density thresholds are defined for each node at height k, when $1 < k < h$:

$$\tau_k = \tau_h + (\tau_1 - \tau_h)\left(\frac{h-k}{h-1}\right)$$

$$\rho_k = \rho_h - (\rho_h - \rho_1)\left(\frac{h-k}{h-1}\right)$$

In general, the values for $\rho 1$, τ_1, ρ_h and τ_h are input parameters of the data structure. Here, we simply set them to $\rho_1 = 0.5$, $\tau_1 = 1$, $\rho_h = \tau_h = 0.75$, a choice that ensures that the overall PMA always has at least less than 50% of empty space [14]. To determine the segments involved in a rebalance, the calibrator tree is traversed bottom-up, starting from the invalidated segment, until a window, whose density is within its thresholds, is found. If so, all elements in the descendant segments of the window are redistributed. Otherwise, the array is resized to a new capacity $C' = 2N/(\rho_h + \tau_h)$. Finally, we note that the calibrator tree is not an explicit data structure, but it is implicitly examined only to drive the search of which segments should be part of a rebalance.

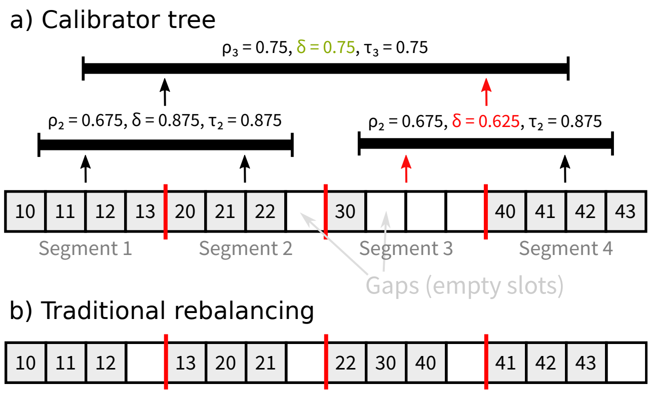

Figure 1: a) The calibrator tree of a sparse array of capacity $C = 12$, with 4 segments of size $s = 4$, and height $h = 3$. The internal nodes are labelled with their density thresholds ρ_k and τ_k and their current density δ. The red arrows depict the tree traversal performed when the third segment is rebalanced. b) The outcome of traditional rebalancing for the whole array.

Complexity. Assuming in the following the usage of the I/O model [1], we fix the capacity B of a segment to the block size of the model. Point lookups cost $O(log_2 N)$ when using a binary search over the array, but this can be reduced to $O(log_B N)$ by pairing the PMA with an auxiliary secondary index. Range scans feature $O(R/B)$ sequential accesses in the worst case, with R being the length of the range. Updates are $O(N/B)$ in the worst case scenario, with the worst case occurring when an update causes a resize or a complete rebalance of the PMA. Nevertheless, it can be still shown that, in amortised analysis, updates are $O(\frac{log_2^2 N}{B})$ in the worst case [3, 15] and, if the distribution of the keys is uniform, $O(log_B N)$ in the average case [5, 15].

Adaptive rebalancing. In a rebalance, there exist two policies to redistribute the elements among a set of elements. In *traditional rebalancing*, all segments receive the same amount of elements, as depicted in Figure 1b. However, with this policy, the PMA performs at its worst in presence of skew, where most insertions or deletions are contiguously executed over the same segments. In *adaptive rebalancing*, the data structure observes the pattern of updates and, while rebalancing, attempts to place more gaps where it predicts more future insertions will follow, and analogously, more elements where future deletions are expected. If the prediction turns out to be correct, adaptive rebalancing decreases the cost of updates to $O(log_B N)$ in presence of skew [7]. On the other hand, implementing

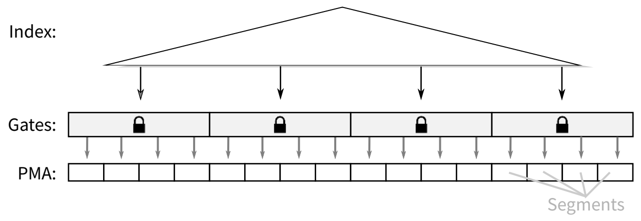

Figure 2: The three layers that compose the parallel sparse array.

adaptive rebalancing is more complex and causes some additional overhead w.r.t. traditional rebalancing.

Memory rewiring. Rebalances can take advantage of memory rewiring [28], a technique where the mapping between the virtual and physical pages is explicitly adjusted to swiftly move or copy large contiguous sections of data. Typically, the positions where elements need to be relocated overlap with the positions occupied by other elements. Without memory rewiring, all the elements need to be firstly moved into a temporary area and, afterwards, copied again to their designed positions in the array. This involves two copies per element. With memory rewiring, instead, the elements can be copied only once, directly to their final positions, but in a temporary buffer. Eventually, the virtual pages of the buffer are swapped with the virtual pages of the array's window, so that the old physical pages of the buffer are now part of the array, whereas the previous physical pages of the array become a new buffer for a future rebalance. On Linux, memory rewiring is performed through invocations of the *mmap* system call, and due to its overhead, it is know to work better on huge pages of 2MB [28].

3 OVERVIEW

We describe our proposal to achieve parallelism in sparse arrays. Our starting point is straightforward: split the array in logical contiguous chunks and protect each chunk with a traditional read-write latch[2]. Multiple readers can operate on the same chunk by acquiring the latch in shared mode. Instead, writers need to acquire the related latch in exclusive mode before making any change to a chunk. While this scheme is simple, sparse arrays pose a challenge on how to rearrange the array during rebalances or resizes, as multiple chunks at the same time may need to be altered.

Figure 2 depicts the layers of our parallel sparse array. Besides a latch, we associate some additional metadata to each logical chunk, in a data structure named *gate* and outlined in Section 3.1. To improve the cost of point lookups and updates, we also propose a secondary index to the sparse array, described in Section 3.2. In Sections 3.3 and 3.4 we tackle the main issue of sparse arrays, how to parallelise rebalances and resizes. Finally, our usage of latches can be inefficient in presence of skew in updates, when multiple writers may end up clogging the same latch. We mitigate this scenario by executing some updates asynchronously, as described in Section 3.5.

In the following we are going to introduce different kinds of service or auxiliary threads. We refer as *clients* all the non-service threads, be it readers or writers, capable of performing any update or scan in the sparse array.

[2]In this paper, we always refer to locks and latches as synonyms.

3.1 Gate

The sparse array is split in contiguous logical chunks. All chunks have the same size, set to be a multiple of the segment size. We associate to each chunk an auxiliary data structure, named gate. All gates are contiguously stored in a dense array, following the same order of the chunks in the underlying sparse array. When the sparse array is resized, also the array of gates is rebuilt. A gate contains: a) the read-write latch that clients need to acquire before accessing the related chunk; b) the minimum key of each segment in the related chunk, to aid point lookups inside a chunk; c) a pair of *fence keys* and d) a few further fields eventually described in Section 3.5.

The fence keys represent the minimum and maximum keys that can be stored in the chunk associated to the gate. The minimum fence key for the first gate is $-\infty$ and the maximum fence key for the last gate is $+\infty$. In all the other cases, the maximum fence key of a gate G_i is equal to the predecessor value of the minimum fence key of the gate G_{i+1} that follows G_i. Fence keys can only be altered during rebalances or resizes that span multiple gates. When these occur, a minimum fence key is set to the minimum key stored in the associated chunk, whereas a maximum fence keys is updated accordingly.

3.2 Static index

To achieve the bound $O(log_B N)$ per lookup, rather than $O(log_2 N)$ achieved by binary search on the array, a simple secondary index is needed. We exploit the peculiarity of sparse arrays to create a custom *static* B$^+$tree. The elements indexed are the single gates, with the minimum fence keys playing the role of the separator keys. The index is static because the number of separator keys does not change, unless the whole sparse array is resized. However, the actual value of a separator key can change; this normally occurs during rebalances. Separator keys have fixed length. The nodes of the tree do not contain explicit pointers, but instead are laid out contiguously in a dense array, sorted per level. To traverse the tree, it uses a simple strategy based on pointer arithmetic. Similarly to the array of gates, when the sparse array is resized, the whole index is rebuilt from scratch.

We handle concurrent access to the index by multiple threads as follows. To alter the value of a separator key, the latch in the gate related to that specific key must be owned in exclusive mode. Because the index is static, it is not necessary to traverse the tree to find the position of the separator key to alter, but this can be inferred directly again by means of pointer arithmetic. In other words, altering the separator key associated to a gate can be achieved in $O(1)$.

There is no mechanism to protect the access of concurrent threads while traversing the tree. That is, a reader occasionally may read an invalid value for a separator key, in the process of being updated by another thread. As a consequence, the reader can eventually reach an incorrect gate, unrelated to the search key. For this reason, a thread, while accessing a gate, verifies whether the search key belongs to the interval bound by its fence keys. If not, it can either a) restart the search from the top of the index, or b) visit one by one the neighbours of the gate selected, until it reaches the correct gate, with our implementation always following this latter choice. Note that, unless a resize occurred, it is always guaranteed that a reader reaches *some* existing gate, it cannot end up outside the gate array. We will deal with the scenario of resizes in Section 3.4.

3.3 Rebalances

We identify two different kinds of rebalances: *local* and *global*. A *local* rebalance takes place when the window to adjust is fully contained in the chunk associated to a gate. A rebalance, in general, can only be caused by a writer, after an insertion or a deletion in a segment invalidated its density thresholds. Since the writer already owns the associated latch in exclusive mode, it can start computing the window to rebalance without suffering the interference of other threads. If the window is fully part of the associated gate, the same writer can complete the rebalance by using the sequential algorithm. Otherwise, we have a *global* rebalance, that is, a rebalance that spans over multiple gates. We aim to avoid the possibility of a writer acquiring multiple latches for a rebalance, as this can cause a deadlock with competing writers blocked on each other's latch. In our design, both readers and writers can only hold a single latch at a time.

We handle global rebalances by introducing a third entity, named *rebalancer*. The rebalancer is a service, it consists of multiple threads set up in the master/worker paradigm. There is only one rebalancer per sparse array, one master thread and multiple worker threads per rebalancer. When a writer understands that a global rebalance is needed, it transfers the ownership of the held latch to the rebalancer, requesting to perform the rebalance on its behalf. The master thread then computes the final window to rebalance. It proceeds both backwards and forwards from the original gate, acquiring all the necessary latches in exclusive mode along the way. Once the window to rebalance has been identified, the master splits it in a set of logical partitions and stores them in a job queue.

On demand, the workers fetch the partitions from the job queue. Each worker runs the sequential rebalancing algorithm on its own partition, almost independently. Some amount of coordination is required, as the elements from a single partition could be moved into a different partition. To avoid overwriting the elements of a partition that has not been processed yet, the workers still rely on memory rewiring, initially storing the elements in a separate buffer and, eventually, swapping the logical address of the buffer with the address of the chunk in the sparse array. The technique is as described in Section 2, with the addition that a given worker may need to delay the rewiring step according to the progress of the other workers. In the process of rebalancing, the fence keys of the involved gates are also properly updated by the workers.

Once the rebalance of a certain window completes, the master releases all the associated latches and wakes up the threads waiting on the gates. As elements in the window could have been moved into different chunks, resumed threads verify whether their selected gate is still correct by comparing the fence keys. If not, they follow the same process described when traversing the index, i.e. they iterate over the neighbour gates, until reaching the correct final gate.

3.4 Resizes

PMA resizing occurs when rebalancing is not sufficient to bring the sparse array into threshold, and is due to the policy of altering the capacity of the array, a rare event. Upon resizing, we recreate the sparse array, the gates and the static index. There exists only a single entry pointer for each of these data structures. After a resize, the memory associated to the old sparse array can be immediately released by the rebalancer. Nevertheless, some clients can still be visiting the gates or traversing the index. We handle this case with a protocol based on *epochs* and *centralised garbage collection*.

All active client threads have an associated epoch. This can be the timestamp read from the CPU counter when their current logical operation started. To free a pointer, the rebalancer adds it to a global list (the garbage) together with the current timestamp. A background thread, the *garbage collector*, runs periodically. It computes the minimum epoch min_{epoch} among all clients and inspects the garbage list, freeing all pointers with an epoch prior to min_{epoch}. Moreover, the rebalancer sets a specific flag in each gate after a resize, so that clients can still detect an invalid gate even after having crossed its outdated entry point. In this case, a client restarts its operation after having entered in a new epoch.

3.5 Asynchronous updates

Our concurrency protocol works adequately when writers operate on different gates. But, in presence of high skew, multiple writers continuously compete to exclusively access the same gate, drastically reducing the overall throughput. We alleviate this scenario by only allowing a single writer to be active on a gate, operating in a form of the *local combining* paradigm. In this approach, we attach to each writer w a queue Q_w. When a second writer w_2 detects it needs to access a gate where a writer w_1 is already active, it forwards its update to w_1 by appending the operation in Q_{w_1}. This behaviour adds a form of asynchronicity as updates can be handled later by different threads.

We implement this scheme by extending the gate with an additional pointer field p_Q. Initially p_Q is unset. When a writer w accesses an inactive gate, it sets p_Q to its queue Q_w. It then proceeds executing its original operation. Other threads, that in the meanwhile request the same gate, detect that p_Q is already set and are able to forward their updates into Q_w. When w completes its original operation, it continues processing all other updates in Q_w, until the queue becomes empty. At that point, it resets the pointer p_Q and leaves the gate.

We considered two strategies to process the updates in the queue Q_w: *one by one* and *batch processing*. In *one by one*, the writer performs one update at a time, following the order in the queue. The advantage of this scheme is that it can still leverage adaptive rebalancing as the order of updates is respected. On the other hand, if a gate is involved in a global rebalance, the updates in Q_w can now start to refer to different and multiple gates. In this situation, the writer leaves the gate held and processes the remaining updates in Q_w without accepting new elements from other threads, until Q_w becomes empty.

In *batch processing*, we merge the updates from Q_w and the existing elements in the gate in two passes. In the first pass, we execute all deletions from Q_w, to temporarily decrease the density of the segments and provide more space for the insertions. In the second pass, we compute the window W in the calibrator tree so that all insertions fit and, then, rebalance the segments in W merging the insertions from Q_w. If the window W is larger than the gate size, the batch is processed by the rebalancer. In this scheme, rebalances are executed according to the traditional policy. Furthermore, to ease the pressure on the rebalancer, we define a minimum span of time t_{delay} that must elapse before a global rebalance can be invoked again for a gate since it has been lastly rebalanced. When a writer w cannot immediately invoke the rebalancer due to t_{delay}, it transfers the ownership of its queue Q_w to the rebalancer leaving p_Q still set to Q_W, and allocates a new private queue to man other gates.

The two schemes deal with skew in contrasting directions. One by one relies on adaptive rebalancing to decrease the total number of rebalances that affect the PMA. Batch processing does not leverage adaptive rebalancing because the global order of updates is altered, e.g. deletions occur before insertions, while single batches can also be postponed at a later time. Batch processing, instead, attempts to skip altogether the small rebalances in the deeper windows of the calibrator tree, by directly inserting multiple elements in a larger window.

4 EVALUATION

We compare our design in several workloads against some of the existing best performing tree based implementations:

- Masstree [23]: a hybrid trie / B$^+$ tree optimised for writes. It achieves high scalability for updates through, among other features, the usage of optimistic concurrency control, small sized nodes and unsorted elements stored inside the nodes. At the same time, it penalises range scans as small leaves (256 bytes) cause more random memory jumps while introducing additional overhead due to version checks and unsorted elements. We evaluated the implementation published by the authors [24].
- Bw-Tree [20]: a lock-free B$^+$ tree variant. Updates never modify existing nodes but attach their alterations in the form of local delta records. Reads need to replay the changes of a local delta before processing a node. This data structure should ensure high concurrency due to its lock-free protocol. We evaluated the implementation from the OpenBw-Tree [32] published by its authors [33].
- ART [18] / B$^+$ tree: ART is a trie index, which has been shown [32] to excel in updates and point lookups. We adopted ART as a secondary index where the elements are ultimately stored in the leaves of a custom B$^+$ Tree. We developed the data structure by adjusting the source code of ART with optimistic lock coupling [19] from [27]. In the B^+ Tree, the leaves have a fixed size of $4Kb$ (plus some metadata). We issue explicit prefetch instructions for leaf traversals in scans, while concurrency consistency is ensured through conventional lock coupling [30].

We consider four different update patterns based on the uniform and Zipfian distributions. In the Zipfian distribution, we set the range of the keys to $\beta = 2^{27}$ and vary the amount of skew through the Zipf factor from $\alpha = 1$ (mild skew) up to $\alpha = 2$ (high skew). We run all the experiments using 16 threads, with part of the threads only performing the updates, and the rest of them constantly scanning all elements in the data structure according to their sorted order. The elements consist of 8 byte key/value integer pairs.

We configured the PMA as follows. Segments have a fixed capacity of $B = 128$ elements. The granularity of a gate is 8 segments. The thresholds for the calibrator tree are those described in Section 2, but we relax the lower threshold to $\rho_1 = 0$ and downsize the array when the number of elements in the PMA is less than 50% of its capacity. As consequence, we note that there can exist, in theory, some scans of R elements not matching the upper bound of $O(R/B)$. Memory rewiring is executed over huge pages of 2MB. The number of workers in the rebalancer are 8, meant to match the number of cores in the underlying machine. The threads for the workers are arranged in a thread pool. The plots refer to the asynchronous version of the PMA with batch processing and t_{delay} set to 100 milliseconds.

We executed the experiments on a set of equal dual socket Intel Xeon E5-2650 @ 2Ghz machines running Linux Fedora 28. Each machine's node has 8 cores, 2 SMT threads per core, and 128 GB of memory. We pinned all threads and memory allocations to the same node, to isolate from NUMA effects. The source code has been written in C++17 and compiled with Clang 7 passing the optimisation flags `-O3 -mtune=native -march=native`. Our implementation of concurrent and parallel PMA extends the sequential implementation of [9]. Similarly to [32], our competitor indexes rely on the library `tcmalloc 2.7` for their memory allocations. Each experiment has been repeated 5 times. The reported results refer to the median, unless stated otherwise.

4.1 Updates

Figure 3 depicts the throughput for updates and scans in the examined data structures. The plots a-c) show the average throughput, starting from an empty data structure, to insert $1G$ elements and concurrently scan the whole data present. The plots d-f) also take into account deletions: starting from the data structure already storing $1G$ elements, they show the average throughput to repeatedly execute $16M$ insertions, roughly 1.5% of the initial size, followed by $16M$ deletions. We partitioned the 16 available threads in three manners: a,d) all threads only execute the insertions and the deletions; b,e) 12 threads perform the updates while 4 threads constantly execute the scans; c,f) half of the threads are dedicated to the updates and half of the threads to the scans.

All tree-based designs tend to be superior in the updates and inferior in the scans. Both the MassTree and the BwTree employ techniques optimised to minimise write conflicts at the detriment of scans. In particular, the MassTree is at least 60% faster in terms of single insertions or mixed updates, and in presence of skew, up to 7x faster. However, it is always at least one magnitude order slower in terms of scans.

B$^+$ trees are a closer match to PMAs. In our setting, the B$^+$ tree paired with ART provides a throughput of 20% more than the

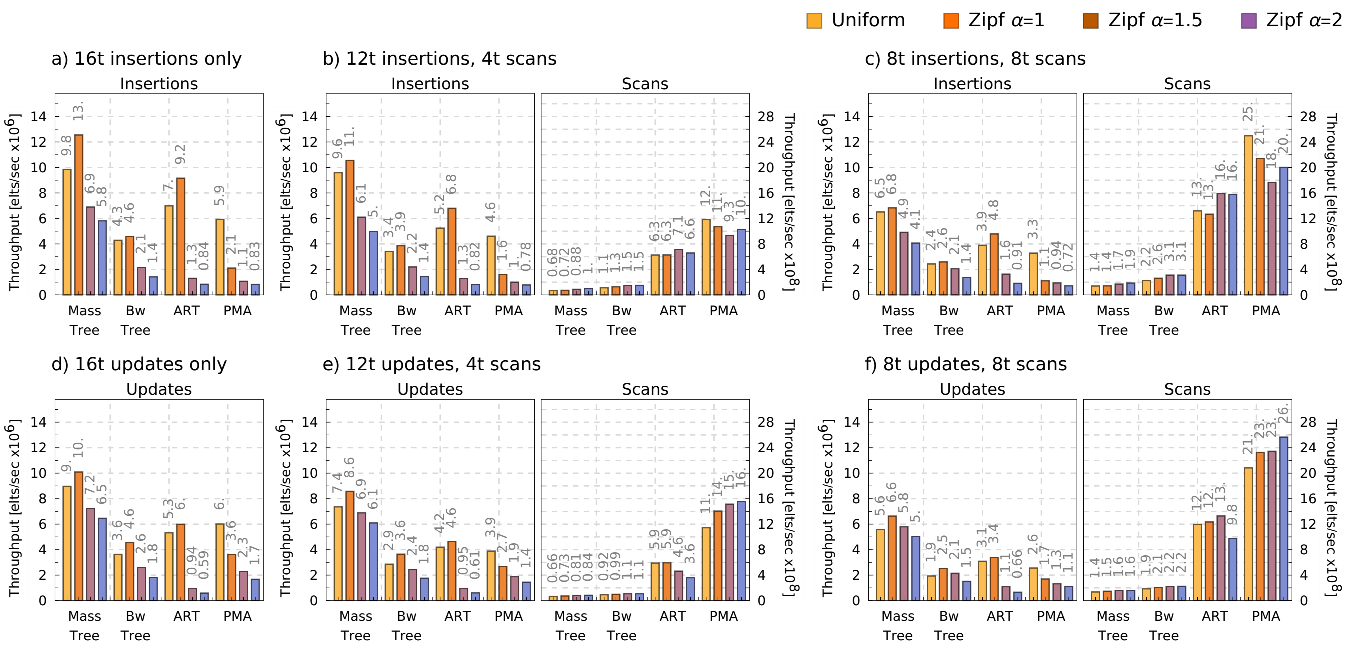

Figure 3: Average throughput, per element, to insert and to scan the elements in the data structures.

PMA for the updates, and, in some cases, up to 4.5x higher. The throughput of ART / B$^+$ tree slightly decreases when considering both insertions and deletions, as the size of the data structure is, on average, larger than the case with only insertions, while deletions are generally a more complex and slower operation on trees. On the contrary, PMAs favour this scenario since the size of the array stays constant and resizes do not happen frequently. In presence of skew, ART copes more effectively with moderate extents. In the distribution with higher Zipf factors, the leaves of the tree undergo significant conflicts among the writers. In this case, the PMA gains advantage by asynchronously processing updates, diminishing latch contention.

Scans are the main strength of the PMA. In the more stable scenario where both insertions and deletions are performed, scans on the PMA can be twice faster than the combined solution with ART and the B$^+$ Tree. To compensate that, we can increase the fixed capacity of a leaf in ART. For instance, setting it to $B = 512$ elements $\sim 8Kb$, the difference reduces to a merely 10% - 20% in favour of the PMA. By increasing the leaf capacity, we trade the performance in scans with the update throughput, now being superior in the PMA for the uniform distribution by about %5 - 20%, and with more mixed results in presence of skew. We note that a similar option exists for the PMA. Doubling the segment size from 128 to 256, the PMA gains 15% more throughput in scans and, likewise, a decrease by 15% in the uniform distribution for updates. Still, in presence of skew, the throughput in the updates increases, roughly by 20%, due to the lesser number of rebalances issued with the larger segment size.

4.2 Asynchronous updates

We examine the contribution of the techniques described in Section 3.5 to handle skew. Figure 4 shows the speed up gained by *one by one* and *batch processing* compared to the baseline implementation without them. For batch processing, we tested different intervals for t_{delay}, the minimum amount of time that must pass between consecutive global rebalances. The experiment is analogous to the one of the previous section: starting from an empty sparse array, we insert 1G elements with 16, 12 and 8 threads, while the rest of the available threads continuously scan the data structure. The plots only report the insertion throughput, as the results for scans are comparable.

Both one by one and batch processing raise the throughput in presence of skew. One by one reduces the contention in the gates and relies on adaptive rebalancing to diminish the total amount of global rebalances. Depending on the amount of skew and the parallelism degree, the speed up can be 2x ~ 4x. Batch processing, without any delay set, is ineffective as it loses the benefit of adaptive rebalancing, whereas the global rebalancer is invoked too frequently with small batches. Still, already setting $t_{\text{delay}} = 100ms$, batch processing outperforms one by one by 10% ~ 75%. Further increasing t_{delay} improves the throughput even beyond, but single updates can be processed later.

All implementations yield similar outcomes in the case of the uniform distribution. Every thread likely accesses a different gate and endures minimal contentions. Moreover, adaptive rebalancing only aims to enhance the behaviour of skewed updates. It still resorts to the traditional rebalancing when there is no skew. In our experiments, one by one introduces a 10% of overhead, we believe it is due to the additional complexity of our implementation w.r.t. to the simpler baseline. Batch processing generally attains advantage by dealing with large batches. In the uniform distribution, all queues are widespread over distant gates and their sizes are too modest when processed to achieve significant speed ups.

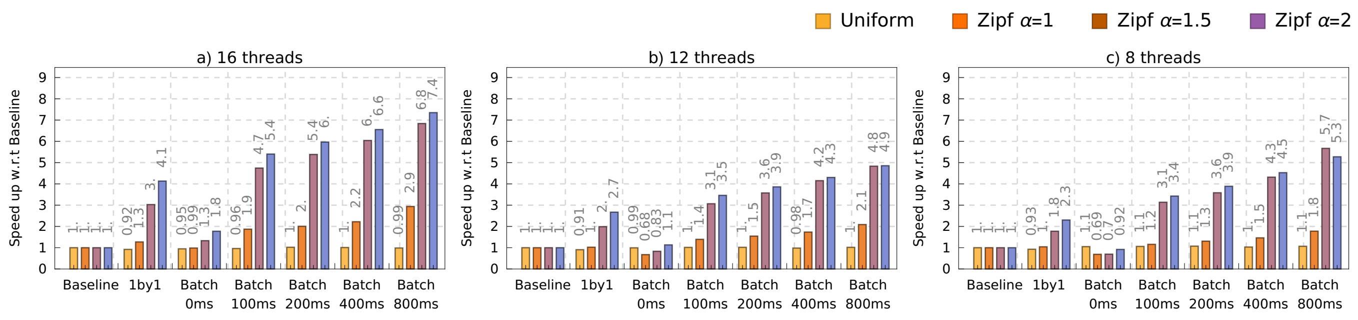

Figure 4: Relative speed up of asynchronous updates compared to the synchronous PMA. Below the label for batch processing is the value set for $t_{\mathbf{delay}}$.

5 RELATED WORK

Packed Memory Arrays were firstly introduced by [15][3]. Since then, several variants, often of theoretical interest, have been proposed [4, 6–8, 14]. PMAs, in particular, are often one of the internal data structures to build *Cache-Oblivious B-Trees (COBT)* [2, 3]. Both our treatment and implementation are based on the variant *Rewired Memory Array (RMA)* [9]. Some practical applications showcasing the usage of PMAs have been published so far [11, 16, 29, 31]. Still, their implementations have been found simpler and less performing than standard B$^+$-trees or the RMA [9]. The usage of the PMA for a dynamic CRS graph representation has been previously presented by [34], but their implementation is also basic and sequential.

We are aware of only two prior works discussing the problem of concurrency in sparse arrays: [6] and [29]. Reference [6] is a theoretical paper. It describes two protocols to achieve concurrency, studying their complexity and correctness. Nevertheless, the proposed protocols assume further restrictions on the PMA, e.g. keys and values are limited by the machine word size. The protocols also provide no parallelism for rebalances. Reference [29] focuses on the specific scenario of graph analysis on GPUs. The protocol relies on a form of batch updating to achieve concurrency. Again, there are several limitations in the proposed scheme. The major one is that updates cannot be interleaved with scans. That is, at a given time, the whole PMA must operate either in read or in update mode. Moreover, within the same batch, updates can issue rebalances of the same window multiple times. Finally, individual rebalances are still sequential.

Some of the techniques used in our design can be found in previous research. Epoch-based garbage collection is a common method, also adopted by our competitors [19, 23, 32], to ensure concurrent threads do not visit freed memory. Fence keys are an established technique employed in the B-Trees found in relational DBMSes, although aimed at different purposes [12]. Local combining is an existing synchronisation paradigm, firstly presented for a synchronous context by [10]. Static indexes were first described by [26]. A similar protocol for concurrent access to a static index was sketched by [6]. Besides based on a different layout, the index from [6] requires the usage of atomic keys, a restriction lifted in our design, which also handles concurrent resizes.

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

6 UPDATING GRAPHS IN PMAS

Our journey in the study of sparse arrays began with the quest for a data structure suitable for dynamic graph processing. For a dynamic CRS representation, we envision the usage of PMAs by storing all edges in a sparse array E, while the vertices are maintained in a different data structure V. The V could be stored in either a dense array [34] or a hash table to still achieve $O(1)$ direct access per vertex, or by indeed a second sparse array. When the first outgoing edge from some vertex v moves (i.e., it is inserted in front, the first gets deleted, or is moved in a rebalance), the offset $V[v]$ should be updated as a side-effect to that change in E. Therefore, this update should happen under a limited form of transactionality that keeps V and E consistent at all times, essentially integrating maintenance of V in the PMA update code used for E. We omitted details of that algorithm extension. A further extension to our locking protocol is to let any query access to a vertex $v \in V$ be governed by the gate of E that corresponds to the first edge of v. Rebalances in E thus potentially have to update V, but these updates are protected by the very same locks that protect E, maintaining the transactional isolation property and guaranteeing consistent behavior.

We recognize that this paper has concentrated mostly on the algorithmic side of PMAs, and much of the details of actually storing and updating graphs in PMAs have been left out, for reasons of space. Yet, we think that the application of PMAs to dynamic graphs is quite clear, essentially replacing dense arrays used in graph storage schemes like CRS by sparse arrays. The ability to thus offer $O(1)$ navigation cost, in an an efficiently updatable data structure we think makes for an attractive proposition.

7 CONCLUSION

In this paper, we have contributed ideas that enable PMAs to perform concurrent updates, as well as ideas to significantly increase the performance of updates using (slightly) asynchronous processing. We think the resulting data structure is very interesting for storing dynamic graphs, since PMAs naturally fit in the array layout (and navigation) of the CRS format. In an extension and follow-up of this work we will go more in-depth on the details of storing and updating graph data with PMAs and evaluating their performance versus the competing trees and tries in an end-to-end benchmark such as the LDBC Social Network Benchmark [17].

The source code of our implementation is available online at `http://github.com/cwida/rma_concurrent`.

REFERENCES

[1] Alok Aggarwal and Jeffrey Vitter. 1988. The Input/Output Complexity of Sorting and Related Problems. *CACM* 31, 9 (1988), 1116–1127.

[2] M. Bender, E. Demaine, and M. Farach-Colton. 2000. Cache-oblivious B-trees. In *FOCS*. IEEE, Washington DC, USA, 399–.

[3] Michael Bender, Erik Demaine, and Martin Farach-Colton. 2005. Cache-Oblivious B-Trees. *SIAM J. Computing* 35, 2 (2005), 341–358.

[4] Michael Bender, Ziyang Duan, John Iacono, and Jing Wu. 2004. A locality-preserving cache-oblivious dynamic dictionary. *J. of Algorithms* 53, 2 (2004), 115 – 136.

[5] Michael Bender, Martin Farach-Colton, and Miguel Mosteiro. 2006. Insertion Sort is O(N log N). *Theor. Comp. Sys.* 39, 3 (2006), 391–397.

[6] M. Bender, J. Fineman, S. Gilbert, and B. Kuszmaul. 2005. Concurrent Cache-oblivious B-trees. In *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '05)*. ACM, New York, NY, USA, 228–237.

[7] Michael Bender and Haodong Hu. 2007. An Adaptive Packed-memory Array. *TODS* 32, 4 (2007).

[8] Gerth Stølting Brodal, Rolf Fagerberg, and Riko Jacob. 2002. Cache Oblivious Search Trees via Binary Trees of Small Height. In *ACM-SIAM SODA*.

[9] Dean De Leo and Peter Boncz. 2019. Packed Memory Arrays – Rewired. In *Proceedings of the 2019 IEEE International Conference on Data Engineering (ICDE 2019)*. 830–841.

[10] Dana Drachsler-Cohen and Erez Petrank. 2014. LCD: Local Combining on Demand. In *Principles of Distributed Systems*. Springer International Publishing, 355–371.

[11] Marie Durand, Bruno Raffin, and François Faure. 2012. A Packed Memory Array to Keep Moving Particles Sorted. In *VRIPHYS*.

[12] Goetz Graefe. 2011. Modern B-Tree Techniques. *Foundations and Trends in Databases* 3, 4 (2011), 203–402.

[13] Gundolf Haase, Manfred Liebmann, and Gernot Plank. 2007. A Hilbert-order Multiplication Scheme for Unstructured Sparse Matrices. *Int. J. Parallel Emerg. Distrib. Syst.* 22, 4 (Jan. 2007), 213–220. https://doi.org/10.1080/17445760601122084

[14] Alon Itai and Irit Katriel. 2007. Canonical Density Control. *Inf. Process. Lett.* 104, 6 (2007), 200–204.

[15] Alon Itai, Alan Konheim, and Michael Rodeh. 1981. A Sparse Table Implementation of Priority Queues. In *Proc. Colloquium on Automata, Languages and Programming*. 417–431.

[16] Zuhair Khayyat, William Lucia, Meghna Singh, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, and Panos Kalnis. 2017. Fast and Scalable Inequality Joins. *VLDB Journal* 26, 1 (2017), 125–150.

[17] LDBC Council. 2015. The LDBC Social Network Benchmark. Available online at: github.com/ldbc/ldbc_snb_docs.

[18] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The Adaptive Radix Tree: ARTful Indexing for Main-memory Databases. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*. 38–49.

[19] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. The ART of Practical Synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware (DaMoN '16)*. ACM, New York, NY, USA, Article 3, 8 pages.

[20] Justin Levandoski, David Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for New Hardware Platforms. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*. 302–313.

[21] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. 2007. Challenges in Parallel Graph Processing. *Parallel Processing Letters* 17 (2007), 5–20.

[22] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. 2017. Mosaic: Processing a Trillion-Edge Graph on a Single Machine. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*. ACM, New York, NY, USA, 527–543.

[23] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache Craftiness for Fast Multicore Key-value Storage. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys '12)*. ACM, New York, NY, USA, 183–196.

[24] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Masstree source code. Available online at: github.com/kohler/masstree-beta.

[25] Frank McSherry, Michael Isard, and Derek G. Murray. 2015. Scalability! But at What Cost?. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems (HOTOS'15)*. USENIX Association, Berkeley, CA, USA, 14–14.

[26] J. Rao and K. Ross. 1999. Cache Conscious Indexing for Decision-Support in Main Memory. *PVLDB* (1999), 78–89.

[27] Florian Scheibner. 2016. ART source code. Available online at: github.com/flode/ARTSynchronized.

[28] Felix Martin Schuhknecht, Jens Dittrich, and Ankur Sharma. 2016. RUMA Has It: Rewired User-space Memory Access is Possible! *PVLDB* 9, 10 (2016), 768–779.

[29] Mo Sha, Yuchen Li, Bingsheng He, and Kian-Lee Tan. 2017. Accelerating Dynamic Graph Analytics on GPUs. *Proc. VLDB Endow.* 11, 1 (Sept. 2017), 107–120.

[30] Abraham Silberschatz, Henry Korth, and Sundararajarao Sudarshan. 2010. *Database System Concepts* (6 ed.). McGraw-Hill.

[31] Julio Toss, Cicero Pahins, Bruno Raffin, and João Comba. 2018. Packed-Memory Quadtree: A cache-oblivious data structure for visual exploration of streaming spatiotemporal big data. *Computers & Graphics* 76 (2018), 117–128.

[32] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David Andersen. 2018. Building a Bw-Tree Takes More Than Just Buzz Words. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. ACM, New York, NY, USA, 473–488.

[33] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David Andersen. 2018. OpenBw-Tree source code. Available online at: https://github.com/wangziqi2016/index-microbench.

[34] B. Wheatman and H. Xu. 2018. Packed Compressed Sparse Row: A Dynamic Graph Representation. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. 1–7.