VCSR: Mutable CSR Graph Format Using Vertex-Centric Packed Memory Array

Abdullah Al Raqibul Islam[1], Dong Dai[1], and Dazhao Cheng[2]

[1]Computer Science Department, University of North Carolina at Charlotte, {aislam6, ddai}@uncc.edu
[2]School of Computer Science, Wuhan University, Wuhan, China, {dcheng@whu.edu.cn}

Abstract—The compressed sparse row (CSR) is a widely used graph storage format due to its compact memory layout and high performance on graph analytic tasks. However, the compact design also limits itself from supporting many graph applications that operate on dynamic or temporal graphs as updates on these graph will need to rebuild the entire CSR structure, leading to high costs. Extending CSR to support efficient graph mutations without losing its high performance on graph analysis then becomes critical to these applications. Existing mutable CSR extensions leverage packed memory array (PMA) to store edge list to enable graph mutations. But, such a naive way has fundamental limitations in handling imbalanced graphs, which many real-world graphs belong to. To address such issues, we propose VCSR, a new mutable CSR storage format that leverages the packed memory array (PMA) via a new *vertex-centric* strategy to efficiently support temporal graphs. Our evaluation results show that compared with the state-of-the-art mutable CSR extensions, VCSR can achieve 1.41x-3.81x better performance in graph insertions and 1.22x-2.05x better performance in running typical graph analytic algorithms. In addition, VCSR can achieve similar performance as the original immutable CSR in running graph analytic tasks, making it a promising storage format for temporal graphs.

I. Introduction

Sparse graphs are widely used in many real-world applications, such as social networks, financial analysis, biological science, and even storage systems [1, 2, 3, 4, 5]. To efficiently store these graphs in memory, researchers have proposed various in-memory graph storage formats, such as edge list (EL), adjacency list (AL), ellpack (ELL) [6], diagonal format (DIA) [7], and compressed sparse row (CSR) [8], etc. Among these options, compressed sparse row or CSR is often the first choice in practice for its compact memory layout and high performance on graph analytic tasks [9]. Specifically, CSR stores graph by packing edges into an edge array, where edges of the same vertex will be grouped together. The vertices in CSR are sequentially stored in a separate vertex array. Each vertex stores a number indicating the start index of its edges in the edge array. Such a compact design gains high memory and cache usage efficiency, hence offering extreme performance to graph analytic tasks.

The CSR storage format, however, can not be easily mutated once built. Inserting a new edge or vertex will trigger the edge or vertex arrays to be rebuilt, which is extremely time-consuming for large graphs. This limits its usage for dynamic or temporal graphs, where new edges and vertices are continuously inserted. In fact, dynamic graphs are increasingly seen in today's applications. For example, in social networks, new users and connections are added constantly [10]; in sensor networks where devices and their communications are modeled as graph vertices and edges, new communications and devices are continuously updated [11]. If using CSR in these scenarios, we have to wait until a new CSR is rebuilt before running the graph analysis, leading to long latency or outdated results.

Several recent studies, PCSR [12] and GPMA [13], extended CSR to support mutations by using the Packed Memory Array (PMA) to store the CSR edge list. Here, PMA is an array data structure that supports dynamic insertions [14]. It partitions the entire array into fixed-size sections, each of which contains some empty space (gaps) reserved for future insertions. If one section gets full, nearby sections will evenly redistribute their gaps together (an action called re-balancing). In this way, new insertions can be done locally within nearby sections instead of shifting the whole array. The amortized complexity for inserting into N elements array is much smaller ($O(\log^2 N)$) than the normal array ($O(N)$). Leveraging PMA, mutable CSR essentially stores its edges in different sections. Each contains gaps for future edge insertions. More details about its implementation can be seen in Section II.

Although existing studies have shown the feasibility of implementing mutable CSR using packed memory array (PMA), their naive way of directly storing graph edges as array elements in packed memory array is problematic to handle real-world graphs. Specifically, PMA is designed based on the assumption that array insertions are distributed evenly across the whole array. However, many real-world graphs follow the power-law distribution, which has a small number of extremely large vertices and a large number of small vertices [15]. This imbalanced nature of the graph is relevant and impacts how the graph grows and how new edges will be inserted. For instance, some vertices may receive more neighboring edge insertions. Without proper handling, these edge insertions may hit a hot area of PMA, leading to more costly re-balancing operations and worse performance.

Through checking how imbalanced graphs grow and how their edges are inserted in real-world temporal graphs, we actually observed that the current degree of a vertex is a strong indicator on the number of its future edge insertions. Detailed experimental results about such an observation are reported in Section III. Based on it, we propose a new approach to

take the imbalanced graph structures into consideration. The key idea is to leverage the current vertex degree information, which is already available in CSR vertex array, as the main indicator for gap reservation during maintaining the PMA data structure. Based on the current degree, we will proportionally redistribute the gaps among vertices. The benefits are twofold. First, vertices and sections with more degrees will obtain more gaps, and hence are able to tolerate more future insertions without shifting nearby vertices, leading to better performance in graph insertions. Second, maintaining a smaller gap for low-degree vertex also improves the read performance because the cache can be better utilized.

To efficiently obtain vertex degree information and leverage it in PMA, we introduce VCSR, a new vertex-centric packed memory array strategy to build mutable CSR[1]. Specifically, VCSR partitions the PMA edge array into multiple sections based on a fixed number of vertices in each section instead of a fixed number of edges, which is widely used in traditional PMA-based CSR extensions. More details can be seen in Section IV. In this way, all the edges of the same vertex will always be in the same section and the gaps can be assigned in a per-vertex way. The 'vertex-centric' strategy preserves the degree information in sections, so that we can easily leverage it. We conducted extensive experiments to demonstrate the performance advantages of VCSR. The results show that VCSR achieves up to 3.81x better performance on edge insertions and 2.05x better performance on graph analytic tasks comparing with the state-of-the-art PMA-based mutable CSR extensions, making it a promising storage format for temporal graphs.

The remainder of this paper is organized as follows: In §II we discuss several typical graph storage formats and their fits to temporal graphs. We introduce the concept of Packed Memory Array and the existing PMA-based CSR extension. In §III, we show the motivations of VCSR, i.e., the strong correlation between the degree of a vertex and its future edge insertions. We also discuss the limitation of degree information and how VCSR addresses that. In §IV we present VCSR and its key components in details. We present the extensive experimental results in §V. We compare with related work in §VI, conclude this paper and discuss the future work in §VII.

II. BACKGROUND AND CHALLENGES

A. Graph Storage Formats and Dynamic Graphs

There are an extensive number of graph storage formats in the literature. Fig. 1 shows an example graph and three typical storage formats: edge list (EL), adjacency list (AL), and Compressed Sparse Row (CSR).

The edge list (EL) is essentially a sequence of edge collections. New edges can be easily appended at the end of the list. It achieves the best performance if the applications only scan the whole edge list for analysis [16]. But, it is severely limited on vertex-based operations, such as single-source breadth-first search (BFS), as locating all edges takes time.

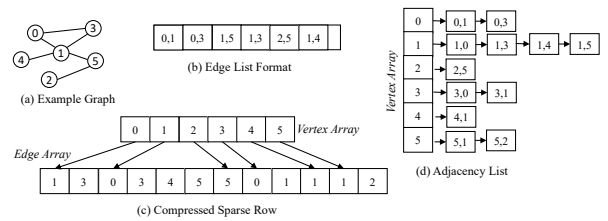[1]Source available at https://github.com/DIR-LAB/VCSR

Fig. 1: Several typical graph storage formats.

The adjacency list (AL) and its variations (e.g., blocked adjacency list [17]) manage the neighbors of each vertex in separate per-vertex linked lists. Edge insertions are easily supported as a new edge can be appended to the corresponding edge list. Vertex-based graph analysis is also easy to implement [18, 11]. The major issue of the adjacency list is its high memory overheads due to the pointers and poor cache performance due to pointer chasing.

The compressed sparse row (CSR) stores all edges in an edge array similar to EL. But It further groups the edges from the same vertex together in the edge array and uses a vertex array to record the starting index of each vertex. In this way, CSR supports both efficient per-vertex queries and fast edge iterations. It presents good cache behaviors because most of the vertex and edge accesses are sequential and compact. However, CSR has a key shortage: it can not support graph updates. Inserting an edge will need to rebuild the edge array, which is extremely time-consuming for huge graphs. Recent studies tried to extend it using packed memory array.

B. PMA and Existing PMA-based CSR

The Packed Memory Array (PMA) [19] is essentially a sorted array where elements are interleaved with empty slots or gaps to accommodate fast updates, as Fig. 2 shows. Here, the blank cells indicate the empty space or gaps. These gaps will provide extra room for future insertions without the need to move long sequences of existing elements. PMA dynamically maintains the placement of the gaps during insertions. It will re-balance the array and re-distribute the gaps whenever sections of the array become too sparse or too dense.

PMA utilizes a binary tree structure to self-balance the gaps. Given an array of N elements, PMA partitions the whole memory space into fixed-size sections, and each has $O(\log N)$ elements. There will be $O(N/\log N)$ sections. PMA builds a binary tree structure by considering these sections as the leaf nodes of the tree. Each internal node of the tree indicates the whole range of its children. The tree structure is shown in Fig. 2. The height of the tree is then $O(\log (N/\log N))$. For any section located at height i (leaf height is 0), PMA designs a way to assign the lower and upper bound density thresholds for the section as ρ_i and τ_i. Once an insertion/deletion causes the density of a section to fall out of the range defined by (ρ_i, τ_i), PMA tries to adjust the density by re-allocating all elements stored in the section's neighbors (by checking its parent). The adjustment process is invoked recursively towards the tree root and will only be terminated if all sections' densities fall back into the range defined by PMA's density thresholds.
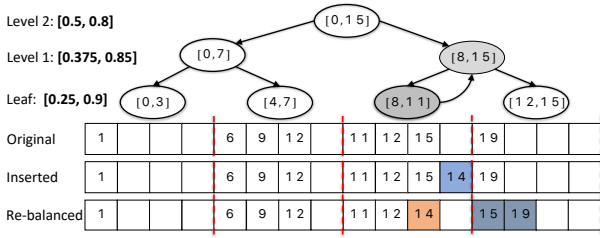
2

Fig. 2: PMA data structure and one insertion example.

Fig. 2 shows an example of PMA structure and its insertion procedure. The total size of the array is 16 ([0, 15]). Each section then contains $\log(16) = 4$ slots. There will be $16/4 = 4$ sections. These sections are organized as a binary tree with height $\log(16/\log(16)) = 2$. Each tree node manages a range of sections identified as an interval (starting and ending position in the array) labeled in the tree node. All values stored in PMA are displayed in the array. For each level i, we show its (ρ_i, τ_i) parameters in bold, such as [0.25, 0.9] for the leaf nodes. The density ratio must be between these two parameters.

To insert an element, i.e., 14, into PMA, we first look for its position in the array via binary search. We locate it should be put in-between 12 and 15. Since there is no space for it, we first place it at the end of the section and will reorder them within the section later. However, the insertion itself causes the density of the section to become 1.0, which exceeds the threshold 0.9 and triggers a re-balance. Then, PMA finds the nearest sections that can accommodate the insertion without violating the thresholds. In this case, it will recursively check whether such insertion will cause its parent (i.e., [8, 15]) to exceed its threshold along the tree. Once find it, the gaps in the parent range will be re-balanced. For example, elements 15 and 19 in Fig. 2 will be put into the sibling section.
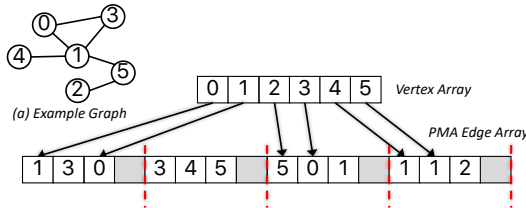

Fig. 3: PMA-based CSR extension, PCSR [12]

Based on the PMA data structure, the PMA-based CSR extensions (e.g., PCSR) are straightforward: directly use PMA to store the CSR edge array. We show how it works in Fig. 3. Here the *vertex array* is kept the same as in the original CSR format, but the *edge array* becomes a PMA array, which is partitioned into fixed-size sections. Each section has $O(\log|E'|)$ edges. Here $|E'|$ estimates the total number of edges. In this particular example, each section contains four slots. The empty boxes indicate the gaps where future edges can be stored. As we can see, the edges of vertex v_1 span across two sections. To insert a new edge (v_i, v_j), we can use source vertex (v_i) and the *vertex array* to locate the starting

index in the *edge array* and then store the edge at the end of its neighboring edges. PMA will automatically re-balance the gaps while new edges are continuously inserted.

C. Issues of PCSR

Although PCSR is able to extend CSR to support graph mutations, its naive way of simply treating graph edges as array elements is indeed sub-optimal. There are mainly two reasons. First, many real-world graphs follow the power-law distribution and have an uneven graph structure. This may lead to imbalanced graph insertions on certain ranges of the edge array. But PCSR reserves the same number of empty spaces or gaps in each section, which may trigger more frequent re-balancing in the hot areas and hence is more time-consuming. Second, in PCSR, as each section has a fixed number of edges, the high-degree vertex may have edges spanning multiple sections. Depending on how the edges from the same vertex would be sorted, some of these inner sections may never receive edge insertion but still reserve empty space, waiting for the gaps to be filled. Also, because edges of a vertex can span across multiple sections, its vertex degree information can not be leveraged to re-distribute the gaps for a section.

III. KEY OBSERVATIONS

In this study, we propose to leverage vertex degree information to adaptively distribute the gaps to address the issues of PCSR. Such a strategy is based on one key observation: the current degree of a vertex is a strong indicator of the number of its future edge insertions in real-world temporal graphs. In this section, we show detailed experimental results about it.

Intuitively, a real-world temporal power-law graph evolves while maintaining its power-law property. For instance, the Twitter network structure would still be exhibiting a power-law last year, last week, and today. Such a property suggests the graphs grow in certain patterns. To show the pattern, we picked four real-world temporal graphs (*sx-stackoverflow* [20], *enron* [21], *sx-mathoverflow* [20] and *fb-wall* [22]) from the SNAP graph dataset [23] and analyzed how they grow. We also picked a static power-law graph, i.e., *com-Amazon* from SNAP and shuffled its edges as the edge insertion order to analyze its growth.

To show how vertices with different degrees grow, we first inserted the first *X%* of the graph as the base graph and recorded the degree of every vertex in the base graph, $d_{base}(v)$. Then, we inserted the rest of the graph and counted how many new edges $e_{new}(v)$ were inserted for each vertex v. This will record the relationship between the current degree of a vertex and the number of its future edge insertions. Since there may be multiple vertices having the same degree in the base graph, we aggregated them together and averaged their $e_{new}(v)$ as the expectation of new edge insertions for vertices with the same base vertex degree. Note that if a vertex never shows in the base graph, its base degree will be 0. The later edge insertions to it will still be counted and contribute the "degree 0" bucket. We tested *X* from 10% to 50% to examine the stability of the observation. Due to the space limit, we plot the results of 10% and 30% in Fig. 4.
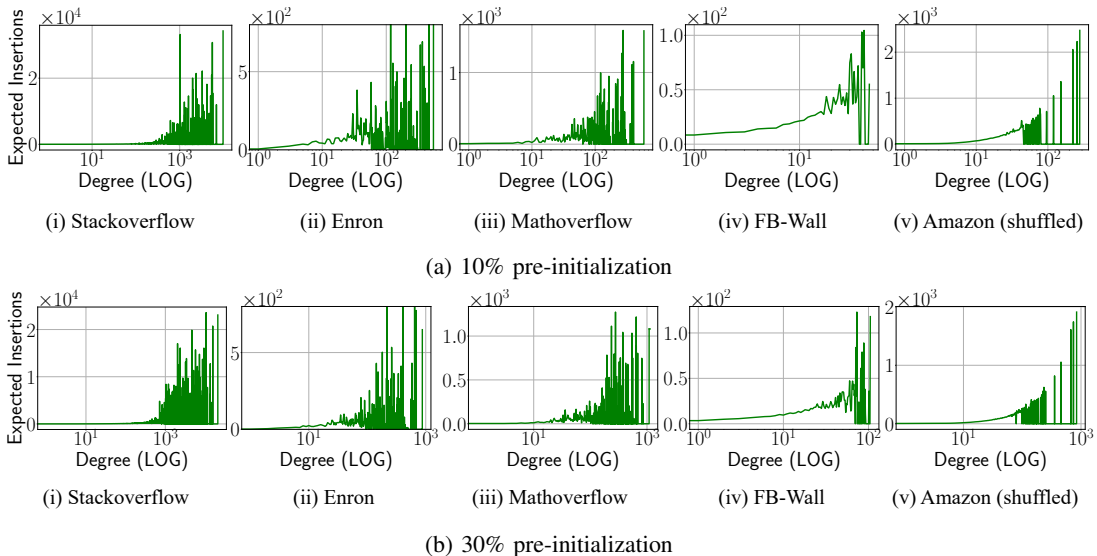
(i) Stackoverflow   (ii) Enron   (iii) Mathoverflow   (iv) FB-Wall   (v) Amazon (shuffled)

(a) 10% pre-initialization

(i) Stackoverflow   (ii) Enron   (iii) Mathoverflow   (iv) FB-Wall   (v) Amazon (shuffled)

(b) 30% pre-initialization

Fig. 4: The relationship between vertex' base degree in both the 10% and 30% pre-initialization cases and their expected new edge insertions. *x-axis is the degree; y-axis is the expected insertions; both are in log-scale*

The results in Fig. 4 clearly show a strong correlation between the current degree of a vertex and the number of its future edge insertions. More importantly, such a property preserves across different real-world temporal graphs and the randomly shuffled static graph. The value of *X%* has barely impacts on the trend as well. For randomly shuffled graph *com-Amazon*, the trend is extremely regular. This is expected as randomly shuffling all the edges makes insertions to a vertex scattered evenly. But for real-world naturally growing graphs, the trend is still clear as seen in the figures. Such an observation serves as the foundation of VCSR design.

A. Limits of Degree Information

Even though Fig. 4 shows prevailing results across graphs, we do understand that the degree information may not be this useful in certain scenarios. For example, if users always insert all the edges of each vertex in a batch and never add new edges to it later, then the degree information could not indicate any future insertion anymore. Previous research [24] suggests that tracking a short history of recent insertions to adaptively redistribute gaps will help handle such extreme workloads. We implement such a feature in VCSR as an option for users to enable.

Although it is not hard to add the 'recent insertion history' feature into VCSR to support such extreme workloads, we strongly argue that such a feature should not be enabled by default and should only be used when users explicitly know their graph insertion workloads are extreme. The reason is, tracking recent insertions introduces extra memory accesses due to maintaining the counters for each insertion. These amplified memory accesses diminish the benefits of a better gap distribution (if there was any). To control the overheads, we can only trace a limited number of insertions. For instance, previous research tracks $\log N$ inserted locations (i.e., 30 for 1 billion elements) [24]. Such a limited number of tracked

locations can barely capture any accurate pattern or offer any useful information better than vertex degree except in extreme insertion patterns.

Graph	Build Time (s)		Mem. Acc. (M)		% of Adap.
	PCSR	APT-CSR	PCSR	APT-CSR	*rebalance*
Amazon	1.44	1.67	44.40	44.39	0.05
Stackoverflow	31.87	39.03	9219.67	9053.07	3.11

TABLE I: Comparison of PCSR and APT-CSR. *Here, the Build Time is showing the time to insert the whole graph (in seconds). Mem. Acc. is showing the recorded memory accesses (in millions)*

To demonstrate such, we revised the PCSR code base [25] with the adaptive mechanisms (namely APT-CSR) by adding workloads tracer and adaptive re-balancing and then compared its performance with the original PCSR on 1) overall execution time; 2) total memory accesses; 3) the percentage of adaptive re-balancing. We show the results on real-world temporal graph *sx-stackoverflow* and randomly shuffled static graph *com-Amazon*. From Table I, we have several observations. First, tracking history (APT-CSR) actually leads to worse performance in graph building. This is due to the extra cost of maintaining the workloads trackers. Second, we observe that APT-CSR does introduce some adaptive re-balancing leveraging the tracked history. However, the percentage is extremely low (0.05% and 3.11%) compared with the total re-balancing. This shows it does not capture the workload pattern for such not-extreme workloads. These two key observations motive us to base VCSR on vertex degree information instead of recent insertion history to better support mutable CSR for real-world temporal graphs.

IV. VCSR DESIGN AND IMPLEMENTATION

Fig. 5 shows the structure of the VCSR storage format generated from an example graph. Similar to CSR and PCSR formats (shown in Fig. 3), VCSR includes vertex array and

edge array to store the graphs. We describe these components below in more detail.
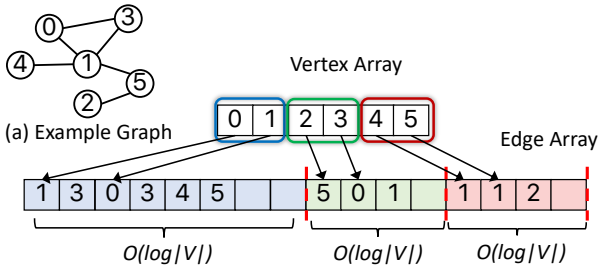

Fig. 5: VCSR storage format.

First, the *vertex array* is exactly the same as its CSR counterpart. It stores all the vertices sequentially based on their Ids. Each vertex contains two key information items: the degree of a vertex and the starting index of its edge in the *edge array*. In Fig. 5, we demonstrate the starting index of each vertex as arrows. The initial size of the vertex array is configurable via MAX_VERTEX parameter. If the number of vertices reaches the limits, we will double the vertex array space and copy data to the new vertex array. Such a resizing operation could be time-consuming. But, we expect it will not often happen as most graph mutations are on the edges. In the future, we plan to investigate splitting the vertex array into multiple segments to avoid this costly resize operation, similar to [26].

Second, the *edge array* stores the graph edges. It is initially small but grows as new edges are inserted. Each of the *edge array* elements contains three key information items: (i) the destination vertex id, (ii) graph property data such as the weight of the edge, and (iii) additional versioning metadata needed for multi-version and snapshot. Note that, to support long-running iterative graph analytics on dynamic graphs, VCSR supports multi-version and snapshot mechanisms similar to those in [18, 11]. In VCSR, edges will have monotonously increasing version numbers (a rolling over 32-bit timestamp). The edges of a vertex are stored in order based on their version numbers. Each analytic task will be implicitly assigned a snapshot Id (based on task submission time). Once the tasks start to run, any new data with higher version numbers will be ignored to guarantee data consistency.

Similar to the edge array in PCSR (shown in Fig. 3), the *edge array* in VCSR contains empty slots in-between array elements for storing future edge insertions. However, in VCSR, these empty slots will no longer be evenly distributed among different sections in the array. Instead, VCSR partitions the edge list by ensuring each section will store the edges of the same number of vertices ($O(\log |V|)$). Here $|V|$ is the total number of vertices. For example, in Fig. 5, each VCSR section will store edges of $O(\log |6|)$ or two vertices. This will lead to different-sized sections if counted based on edges. For instance, in Fig. 5, because vertex v_1 has more edges, its section is bigger than others.

We name this new edge array partition strategy 'vertex-centric' as it counts vertices instead of edges to partition the *edge array*. Its benefits are two-fold. First, it guarantees edges of the same vertex will always be stored in the same section. This allows us to easily trace the vertex degree information of each section to determine its appropriate empty space. Second, it avoids the scenario where a high-degree vertex spans its edge across multiple sections. This may happen in PCSR. When it happens, the middle sections may never receive any edge insertion due to increasing version numbers, but still need to reserve empty slots, which are wasteful. We will describe how the empty slots can be adaptively re-balanced in the next section.

A. Graph Operations in VCSR

In this subsection, we introduce how various graph operations are executed in VCSR.

1) Initialization: The VCSR graph can be initialized from scratch or using existing base graph data. It takes one parameter from users during the initialization: MAX_VERTEX. Based on the maximal number of vertices, it allocates the *vertex array*. It allocates the *edge array* with an estimated size of edges, which could be very small initially. If there are existing base graph data, we insert them and build the PMA-tree on top of the *edge array*, similar to Fig. 2 shows. In this regard, we calculate the capacity and number of actual edges in each PMA-tree node and store this information for future usage.

2) Edge Insertion/Deletion: Every edge comes in a tuple of source vertex-id (src), destination vertex-id (dst), property data (e.g., *weight*), and an implicit timestamp. Based on the degree of the source vertex src, these edges will be inserted in the *edge array*.

Algorithm 1 Insert an edge in the Edge Array

1: ⋄ *Insert an edge from u to v with weight w.*
2: $insert\_location \leftarrow start_u + degree_u$
3: **if** $insert\_location \geq start_{u+1}$ **then**　　　▷ not enough space
4:　　shift nearby sections to make space
5: **end if**
6: insert edge (u, v, w) at insert_location

7: **if** $density(section_u) \geq density_{thres}$ **then**
8:　　$do\_rebalance(section_u)$;
9: **end if**

Algorithm 1 shows the details of inserting an edge $u \rightarrow v$ in VCSR. We will first locate where the new edge should be inserted. In VCSR, we associate the current timestamp if not provided with each edge insertion and store edges of the same vertex sequentially based on their timestamp. So, the upcoming location for any edge should be where its start index plus its degree ($start_u + degree_u$).

If that location is empty, we simply place the current edge there (as shown in line 6). Otherwise, the location will be overlapped with the starting index of the next vertex ($start_{u+1}$). Then, we will need to make space for it by moving the edges of the neighboring vertices (in line 4). To do so, we lookup in the logical tree to find an appropriate search

range with enough space. Then, we shift the neighboring edge-lists one step forward or backward (depending on where we get the gap) and insert the current edge in the newly created space. As there are empty spaces reserved in each section, such a shift should not move many edges. After inserting the edge, we will check whether this insertion makes the current section too dense (in line 7). If yes, we initiate a re-balance (in line 8). The re-balance procedure is the key to maintaining the adaptive number of gaps among sections. We will discuss it in a later section.

Similar to other graph storage engines [11], edge deletions in VCSR are done in a lazy way. Specifically, for deleted edges, we will place a tombstone to it by changing its *src* field to be an invalid negative number and return without reclaiming the space. These marked edges will not be used in reading and will be asynchronously removed by a maintaining thread.

3) Uneven Re-balancing: As described earlier, after every edge insertion, we check whether the current section reaches its density thresholds (ρ_i, τ_i). If so, we will search the logic tree from leaf to root for a valid range of sections where the density is still in their given density boundary. If we cannot find any such section range before reaching the tree root, we will resize (double) the whole VCSR edge list. If we do find a range of sections that will still fall between their density thresholds after the insertion, we calculate all the available gaps in these sections and redistribute them among all the sections. So far, everything is done similarly as the original packed memory array or PCSR. However, instead of distributing gaps evenly, we follow an uneven strategy to redistribute the empty space.

Specifically, assume all the available gaps in these sections are $gaps_{avail}$. It is easy to calculate how many vertices are in these sections and their degrees by checking the *vertex array*. We add these degrees together to $total_{degree}$. Then, we can calculate the adaptive number of gaps per vertex as:

$$gaps_v = degree_v * \frac{gaps_{avail}}{total_{degree}}$$

Based on this per-vertex gaps, we need to move all the edges inside these sections to make sure there is correct number of gaps for each vertex. We do such movement in an in-place method, which reduces the memory usage and reduces the total memory accesses during re-balancing.

The benefit of such adaptive re-balance is two-fold. First, it leaves more gaps for the high-degree vertices, which often indicate a higher chance of getting more insertions. Second, placing smaller gaps after the low-degree vertices makes the edge list sections of low-degree vertices more compact, which increases the cache locality. We will show not only the write performance but also the read performance improvements over PCSR in our evaluation section.

V. EXPERIMENTAL RESULTS

In this section, to show the advantage of VCSR, we compare it with other graph storage formats on real-world graphs using different real-world temporal graphs and other synthetic graph insertion patterns.

A. *Evaluation Setup*

Evaluation Platform. We conducted all the evaluations on a Dell R740 rack server with two sockets. Each socket installs the 2nd generation Intel Xeon Scalable Processor (Gold 6254 @ 3.10G) with 18 physical (36 virtual) cores and 6 DRAM DIMMS with 32GB each. The machine is running Ubuntu 18.04 with a Linux kernel version 4.15.0.

Graph Algorithm Kernels. To show the performance of different graph storage formats on running graph analytic algorithms, we used the GAP Benchmark Suite (GAPBS) [31], the state-of-the-art graph benchmark suite, to evaluate their performance. GAPBS itself also provides an optimized CSR implementation, which serves as our baseline to compare with. In our evaluation, we used four representative graph kernels from different domains implemented in the GAPBS: PageRank (PR), Breadth-First Search (BFS), Connected Components (CC), and Single-Source Shortest Paths (SSSP). Table. II gives a brief overview of the algorithms.

Input Graphs. We used several real-world SNAP graphs from various domains [23] in our evaluations. We used weighted versions of all graphs for a fair comparison with the publicly available version of PCSR [25], which has a required field (e.g., value/weight) in its edge structure. We generated weighted graphs from unweighted graphs by assigning random integer weights in the range [0,256). Table. III lists the graphs used in the evaluation and their key properties. Among these selected graphs, the top six graphs are static graphs. When using them, we will manually generate the insertion workloads by reordering their edges. We created two different workloads ('Random' and 'Hammer') to show the performance in different cases. In addition to them, we also select four temporal graphs (the bottom four) to show the performance of running real-world temporal graphs.

System Implementation Details. In addition to the CSR implementation provided in GAPBS, we further implemented Adjacency-List (AL) and Blocked Adjacency-List (BAL) as baselines to compare with VCSR. These two formats are commonly used in many dynamic graph storage systems. They indicate one extreme case where the storage format is optimized for graph mutations, not graph analysis. Since Blocked Adjacency-List (BAL) stores a predefined number of edges (e.g., 512 edges in our case) in a continuous block, it can produce better cache behavior hence leading to better performance than the plain Adjacency-List (with the cost of higher memory consumption). So, in most of the results, we just report the performance of BAL instead of both.

As the state-of-the-art PMA-based CSR extension, PCSR is another graph format we compare. We modified and improved PCSR based on its open-source implementation for a fair comparison [25]. Specifically, we remove the binary search part in PCSR as all the new edges will be inserted at the very end of its source vertex' neighbors. We confirmed that this optimization produces better results compared with the default code. And we always reported the improved performance of PCSR in later sections.

Random/Hammer Workload. To show that VCSR works

Graph kernel	Kernel Type	Input	Output	Notes
PageRank (PR)	Link Analysis	-	$|V|$-sized array of ranks	Fixed number (20) of iterations
Breadth-First Search (BFS)	Graph Traversal	Source vertex	$|V|$-sized array of parent IDs	Direction-Optimizing approach [27]
Single-Source Shortest Paths (SSSP)	Shortest Path	Source vertex	$|V|$-sized array of distances	δ-stepping [28]
Connected Components (CC)	Connectivity	-	$|V|$-sized array of component labels	Afforest subgraph sampling [29, 30]

TABLE II: A list of graph kernels and inputs and outputs used to evaluate graph data-structures.

Datasets	Domain	$|V|$	$|E|$	$|E|/|V|$
Amazon	purchase	403393	4886816	12
Orkut	social	3072626	234370166	76
Live-journal	social	4847570	85702474	18
Cit-Patents	citation	6009554	33037894	6
Road	geo	1971280	5533214	3
as-Skitter	network	1696414	22190596	13
sx-stackoverflow	temporal	6024270	57724802	10
enron	temporal	87273	594912	7
sx-mathoverflow	temporal	88580	375972	4
fb-wall	temporal	63891	366824	6

TABLE III: Graph inputs and their key properties.

(a) Random Workload (b) Temporal Graphs

Fig. 6: Dynamic graph insertion performance in seconds for 10% pre-initialization.

(a) Random Workload (b) Temporal Graphs

Fig. 7: Dynamic graph insertion performance in seconds for 30% pre-initialization.

well in different scenarios, when using the static graphs, we manually generate two different types of insertion patterns. First, we randomly shuffle all the edge insertions. We name this *Random Workload*. Second, we aggregate all edges of the same vertex and insert them all together. We name this *Hammer Workload* as an extreme case discussed earlier.

B. Graph Insertion Performance

We first compare the graph insertion performance. In this experiment, we first insert edges from the first $X\%$ base graph. Then, we insert the rest of the graphs into the existing graph. All the edges were ordered based on either the manually created insertion patterns (randomly shuffled) or their inherent order from the temporal graphs. For each case, we measured the time of dynamic graph insertions. We evaluated cases with X values from 10% to 50%. Due to limited space, we show the results with 10% and 30% in Fig.6 and Fig. 7. Here, the y-axis shows the execution time for building the graph. So, smaller values indicate better performance. The left sub-figure shows the performance of different static graphs with random workloads. The right sub-figure shows the performance of different temporal graphs. Note that, the results for VCSR are based on the default VCSR (pure degree-based strategy). We did not show the results of VCSR with the 'tracking insertion history' option because 1) it is not the default VCSR setting; 2) it constantly introduces around 10% overhead compared with the default VCSR. We will discuss more about it in later evaluations on hammer workloads.

These results clearly show VCSR gains better performance compared with PCSR across all cases. VCSR has achieved 1.41x-3.81x times better performance than PCSR in static graphs with Random Workloads. It achieved more than 2.0x better performance on all the temporal graphs as well.

Inserting an edge in BAL is essentially appending an edge to the per-vertex edge list. Hence it is expected to have good performance. Our evaluation results confirm that. Interestingly,
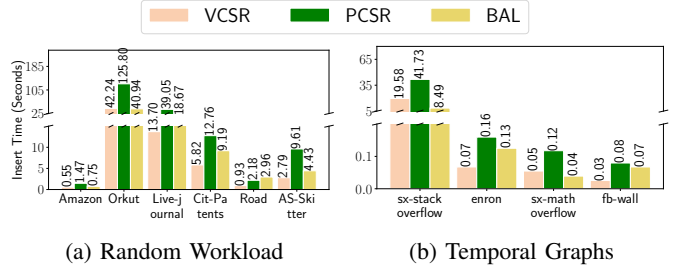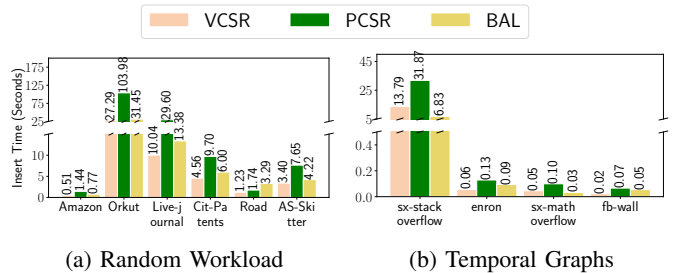
the performance of VCSR is very close to BAL in many cases. Note that, the design goal of VCSR is not to beat BAL in dynamic graph insertions. Instead, the comparable graph mutation performance is already impressive considering the memory layout of VCSR will be more compact and can deliver much better performance for graph analytic workloads.

To further reveal the reason why VCSR performs better than PCSR, we measured the detailed number of re-balancing operations triggered by edge insertions. The re-balancing needs to move edges in multiple sections, hence contributing the most overheads in both VCSR and PCSR. The results are shown in Table IV. Here, we report the number of re-balancing operations triggered from different tree levels (higher tree levels indicate more overhead during re-balancing). Since the sections in VCSR have variable sizes but sections in PCSR have the same size, we also report the actual memory accesses that occurred during these re-balancing for a fair comparison. Due to the space limit, we only show results of randomly shuffled *Amazon* and *sx-Stackoverflow* graphs. Other graphs show similar results. From these results, we can easily observe VCSR significantly reduce the number of re-balancing as well as the memory accesses needed to perform them. This again confirms that the performance improvement of VCSR

mainly comes from the lower number of re-balances due to its adaptive gaps redistribution.

Dataset	Tree Levels	VCSR (Mem. Acc.)	PCSR (Mem. Acc.)
Amazon (shuffled)	[1-3]	710 (0.84 M)	251077 (35.50 M)
	[3-7]	4 (0.02 M)	12964 (8.49 M)
	≥ 7	0 (0.00 M)	45 (0.41 M)
sx-Stack overflow	[1-3]	904363 (457.23 M)	13876752 (2262.72 M)
	[3-7]	357575 (1056.19 M)	3053307 (2720.56 M)
	[7-15]	26530 (1594.56 M)	70307 (1704.70 M)
	≥ 15	21 (438.37 M)	288 (2531.68 M)

TABLE IV: Number of re-balancing operations triggered by edge insertions in VCSR and PCSR on two graphs.

C. VCSR Performance on Hammer Workload

In the previous evaluations, we show the performance advantages of default VCSR (pure degree-based strategy). As discussed earlier, such a strategy may have issues with extreme workloads; and the optional 'tracking insertion history' feature supported by VCSR can help. In this section, we will show how VCSR and VCSR+Tracking perform on Hammer insertion patterns. Specifically, for each graph, we inserted the first 10% of it as base graph. Then, we inserted the rest of the graph and counted how long it took. The results are reported in Fig. 8. From these results, we can observe that for such an extreme insertion pattern, VCSR basically performs the same as PCSR. The previously observed big performance improvements disappear. This is because the degree-based distribution is as inaccurate as the even distribution now. With history tracking enabled, VCSR+Tracing is able to perform better than PCSR again, because the recent insertion history captures the intensive insertion patterns. However, it is important to note that such a result does not mean VCSR+Tracking is better than default VCSR. As described earlier, for real-world temporal graphs or insertion patterns that are not so extreme, VCSR still performs better than VCSR+Tracking. We demonstrate this in Fig. 9, where we show the normalized insertion performance of VCSR and VCSR+Tracking on real-world temporal graphs and shuffled static graphs. We show the same 10% pre-initialization case as Fig. 8. The results show that compared with VCSR, VCSR+Tracking takes a longer time (up to 1.35x) to finish insertion.
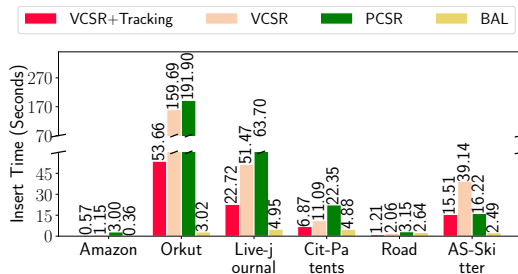

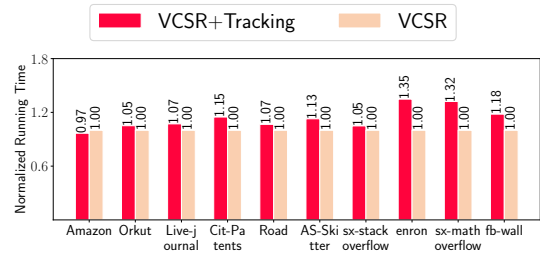Fig. 8: Insertion performance (in seconds) for hammer workload in 10% Pre-initialization case.

Fig. 9: Normalized insertion performance for 10% pre-initialization.

D. Graph Analytic Algorithms Performance

The ultimate goal of extending mutable CSR is to leverage its good performance on graph analysis. In this evaluation, we systematically compared the performance of CSR, VCSR, PCSR, and BAL layouts running different graph algorithms on all graph inputs in both single-thread and multi-thread (32 threads) settings. All the results are reported in Fig. 10 and Table V. Due to the space limitation, we can only show the plots of two graphs: Orkut and Road in Fig. 10. We present the performance of the other four graphs in Table V. All the reported results are normalized using CSR as the baseline. Since CSR is fully compacted, it provides the best graph analysis performance. Other storage layouts should present a value larger than 1. Larger values indicate worse performance.

From these results, we can observe that in all the graphs, VCSR performs very close to CSR in both single thread and 32-thread settings. For instance, it only introduces 5% overhead to run PageRank on Orkut graph in 32-thread compared with CSR. While, the state-of-the-art PMA-based CSR extension, PCSR introduced 87% overhead upon CSR in the same setting. In general, we observe 1.22x-2.05x performance overhead from PCSR compared with our VCSR. The blocked adjacency list (BAL), designed for easy graph mutations, performs the worst in most of the settings. For instance, it runs PageRank 10x longer on Road graph using 32 threads. These results clearly show the advantage of VCSR in supporting graph analysis algorithms. The key behind its performance is its adaptive gap distribution that can maximally preserve the good cache behaviors.

To summarize, from these experiments, we show that VCSR is able to achieve both good graph insertion performance compared with storage formats that are designed for graph mutations such as AL and BAL and similar graph analysis performance compared with immutable compacted storage formats such as CSR, showing it as a promising storage format for in-memory dynamic graphs.

VI. RELATED WORK

There have been a enormous number of studies on in-memory sparse graph or matrix storage formats for efficiently running graph analytic algorithms, such as diagonal format (DIA) [7], compressed sparse row (CSR) [9], Ellpack (ELL) [6], and many others [9]. However, these formats are not designed for dynamic or temporal graphs and hence often have limited performance in graph mutations.
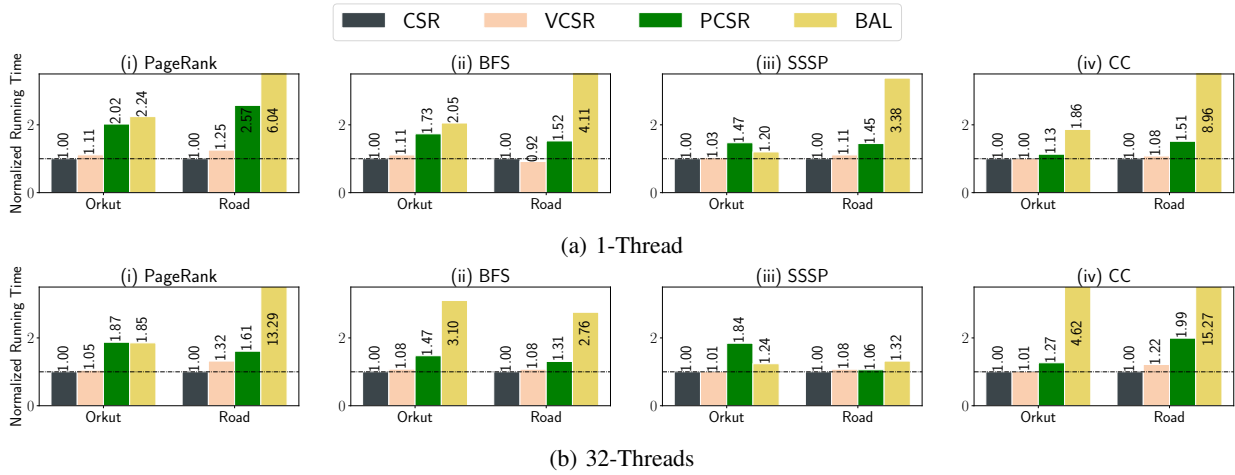
Fig. 10: Time to run Graph Analysis normalized to CSR.

(a) 1-Thread

(b) 32-Threads

	PageRank								BFS							
	CSR		VCSR		PCSR		BAL		CSR		VCSR		PCSR		BAL	
Graph	T_1	T_{32}	T_1	T_{32}	T_1	T_{32}	T_1	T_{32}	T_1	T_{32}	T_1	T_{32}	T_1	T_{32}	T_1	T_{32}
Amazon	1(0.27)	1(0.02)	1.40	1.17	2.27	1.96	4.55	7.71	1(0.03)	1(0.00)	1.15	1.26	1.42	1.69	3.70	5.81
Live-journal	1(6.54)	1(0.41)	1.10	1.14	1.75	1.61	3.06	4.59	1(0.22)	1(0.02)	1.13	1.12	1.74	1.48	2.88	6.97
Cit-Patents	1(4.26)	1(0.27)	1.11	1.14	1.82	1.76	3.03	5.14	1(0.29)	1(0.02)	1.12	1.16	1.92	1.77	3.00	6.42
AS-Skitter	1(1.21)	1(0.09)	1.23	1.19	1.95	1.57	4.89	6.74	1(0.06)	1(0.00)	1.35	1.24	2.02	1.66	7.06	12.02
Stackoverflow	1(4.55)	1(0.29)	1.17	1.15	1.91	1.72	2.15	2.69	1(0.18)	1(0.01)	1.22	1.29	1.66	1.58	1.66	5.32
Enron	1(0.02)	1(0.00)	1.43	1.43	2.26	1.58	5.08	14.03	1(0.00)	1(0.00)	1.19	1.53	1.84	1.88	3.22	7.47
Mathoverflow	1(0.01)	1(0.00)	1.19	1.35	2.55	1.71	3.26	3.69	1(0.00)	1(0.00)	1.03	0.91	1.66	1.71	1.72	1.79
FB-Wall	1(0.01)	1(0.00)	1.27	1.35	2.99	1.87	5.01	9.32	1(0.00)	1(0.00)	1.11	0.73	1.97	1.01	2.99	2.41
	SSSP								CC							
	CSR		VCSR		PCSR		BAL		CSR		VCSR		PCSR		BAL	
Graph	T_1	T_{32}	T_1	T_{32}	T_1	T_{32}	T_1	T_{32}	T_1	T_{32}	T_1	T_{32}	T_1	T_{32}	T_1	T_{32}
Amazon	1(0.08)	1(0.02)	1.12	1.07	1.33	1.26	1.92	1.23	1(0.05)	1(0.00)	1.14	1.23	0.92	1.53	2.46	5.01
Live-journal	1(1.53)	1(0.10)	1.02	0.97	1.24	1.42	1.31	1.44	1(0.50)	1(0.02)	1.07	1.07	1.27	1.26	3.11	6.32
Cit-Patents	1(0.97)	1(0.07)	1.01	1.03	1.28	1.52	1.34	1.50	1(0.69)	1(0.03)	1.14	1.11	1.20	1.24	2.44	3.84
AS-Skitter	1(0.35)	1(0.04)	1.13	1.05	1.28	1.39	1.50	1.45	1(0.11)	1(0.01)	1.31	1.14	1.35	1.40	6.04	8.62
Stackoverflow	1(0.94)	1(0.08)	1.04	1.02	1.47	1.36	1.32	1.42	1(0.35)	1(0.02)	1.28	1.05	1.48	1.23	2.04	4.39
Enron	1(0.00)	1(0.01)	1.19	1.09	1.34	1.15	2.23	1.10	1(0.00)	1(0.00)	1.23	1.39	1.37	1.51	5.86	3.55
Mathoverflow	1(0.00)	1(0.01)	1.12	0.97	1.60	1.02	1.94	1.05	1(0.00)	1(0.00)	1.18	1.17	1.53	1.38	2.18	1.73
FB-Wall	1(0.00)	1(0.02)	1.13	1.06	1.59	1.11	2.05	1.10	1(0.00)	1(0.00)	1.21	1.10	1.52	1.21	3.45	2.30

TABLE V: The normalized running time of different algorithms (PageRank, BFS, SSSP, CC) on CSR, VCSR, PCSR, and BAL. CSR format is the baseline for all cases. We also add the actual runtime of CSR (in seconds). T_1 denotes the time of one thread and T_{32} denotes the time of 32 threads.

To support temporal graphs, researchers proposed several CSR extensions, including GPMA [13] and PCSR [12]. Both of them leverage the packed memory array data structure, and share the same limitation as they consider graph edges as the array elements and ignore the imbalanced graph structures. VCSR is able to significantly outperform them by introducing a new vertex-centric strategy to use the PMA.

Other data structures such as adjacency list, edge list, or tree are often used to build graph databases that can efficiently support graph mutations. Compared with VCSR, they often suffer in graph analysis performance due to poor cache usage. For example, Aspen [32] is a compressed purely-functional search tree designed for streaming graphs. Its graph analysis suffers due to the high cache miss of tree structure. STINGER [33] is a data structure for streaming graphs based on the linked lists of blocks, so it suffers due to the pointer chasing while accessing the graph. VCSR is designed to efficiently support graph mutations without scarifying its graph analysis performance.

VII. CONCLUSION AND FUTURE WORK

In this study, we present VCSR, a new mutable CSR graph storage format, to support both high-performance graph analytic tasks and graph mutations. VCSR introduces a novel vertex-centric strategy to use the packed memory array (PMA) structure to achieve our goal. Throughout extensive evaluations, we confirm that VCSR achieves better performance than the state-of-the-art mutable CSR extensions on both graph insertion and graph analysis. In the future, we plan to further investigate other re-balancing strategies to better leverage the degree information.

References

[1] H. Kwak, C. Lee, H. Park, and S. Moon, "What is twitter, a social network or a news media?" in *Proceedings of the 19th international conference on World wide web*, 2010.

[2] Y. Zhou, L. Liu, S. Seshadri, and L. Chiu, "Analyzing enterprise storage workloads with graph modeling and clustering," *IEEE Journal on Selected Areas in Communications*, vol. 34, no. 3, pp. 551–574, 2016.

[3] R. Albert, H. Jeong, and A.-L. Barabási, "Diameter of the world-wide web," *nature*, vol. 401, no. 6749, 1999.

[4] H. Jeong, B. Tombor, R. Albert, Z. N. Oltvai, and A.-L. Barabási, "The large-scale organization of metabolic networks," *Nature*, vol. 407, no. 6804, pp. 651–654, 2000.

[5] D. Dai, R. B. Ross, P. Carns, D. Kimpe, and Y. Chen, "Using property graphs for rich metadata management in hpc systems," in *Parallel Data Storage Workshop (PDSW), 2014 9th*. IEEE, 2014, pp. 7–12.

[6] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey, "Efficient sparse matrix-vector multiplication on x86-based many-core processors," in *Proceedings of the 27th international ACM conference on International conference on supercomputing*, 2013, pp. 273–282.

[7] Y. Saad, *Iterative methods for sparse linear systems*. SIAM, 2003.

[8] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks," in *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, 2009, pp. 233–244.

[9] J. L. Greathouse and M. Daga, "Efficient sparse matrix-vector multiplication on gpus using the csr storage format," in *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2014, pp. 769–780.

[10] "Twitter statistics," http://www.statisticbrain.com/twitter-statistics/.

[11] P. Kumar and H. H. Huang, "Graphone: A data store for real-time analytics on evolving graphs," in *17th {USENIX} Conference on File and Storage Technologies ({FAST} 19)*, 2019, pp. 249–263.

[12] B. Wheatman and H. Xu, "Packed compressed sparse row: A dynamic graph representation," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 2018, pp. 1–7.

[13] M. Sha, Y. Li, B. He, and K.-L. Tan, "Technical report: Accelerating dynamic graph analytics on gpus," *arXiv preprint arXiv:1709.05061*, 2017.

[14] D. De Leo and P. Boncz, "Packed memory arrays-rewired," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 2019, pp. 830–841.

[15] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graphs over time: Densification laws, shrinking diameters and possible explanations," in *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, ser. KDD '05. New York, NY, USA: Association for Computing Machinery, 2005. [Online]. Available: https://doi.org/10.1145/1081870.1081893

[16] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-centric graph processing using streaming partitions," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013.

[17] A. Pinar and M. T. Heath, "Improving performance of sparse matrix-vector multiplication," in *SC'99: Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*. IEEE, 1999.

[18] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer, "Llama: Efficient graph analytics using large multiversioned arrays," in *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 2015, pp. 363–374.

[19] M. A. Bender, E. D. Demaine, and M. Farach-Colton, "Cache-oblivious b-trees," in *Proceedings 41st Annual Symposium on Foundations of Computer Science*. IEEE, 2000, pp. 399–409.

[20] A. Paranjape, A. R. Benson, and J. Leskovec, "Motifs in temporal networks," in *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, ser. WSDM '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 601–610. [Online]. Available: https://doi.org/10.1145/3018661.3018731

[21] B. Klimt and Y. Yang, "The enron corpus: A new dataset for email classification research." Springer Berlin / Heidelberg, 2004, vol. Volume 3201/2004, pp. 217–226. [Online]. Available: http://www.springerlink.com/content/q8g7blqvqyxrpvap/

[22] M. Beladev, L. Rokach, G. Katz, I. Guy, and K. Radinsky, "Tdgraphembed: Temporal dynamic graph-level embedding," in *Proceedings of the 29th ACM International Conference on Information amp; Knowledge Management*, ser. CIKM '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 55–64. [Online]. Available: https://doi.org/10.1145/3340531.3411953

[23] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, 2014.

[24] M. A. Bender and H. Hu, "An adaptive packed-memory array," *ACM Transactions on Database Systems (TODS)*, vol. 32, 2007.

[25] "Dynamic data structure for sparse graphs," https://github.com/wheatman/Packed-Compressed-Sparse-Row, accessed July, 2021.

[26] S. Firmli, V. Trigonakis, J.-P. Lozi, I. Psaroudakis, A. Weld, D. Chiadmi, S. Hong, and H. Chafi, "CSR++: A Fast, Scalable, Update-Friendly Graph Data Structure," in *24th International Conference on Principles of Distributed Systems (OPODIS '20)*, Strasbourg (on line), France, Dec. 2020. [Online]. Available: https://hal.archives-ouvertes.fr/hal-03060095

[27] S. Beamer, K. Asanovic, and D. Patterson, "Direction-optimizing breadth-first search," in *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012, pp. 1–10.

[28] U. Meyer and P. Sanders, "δ-stepping: A parallelizable shortest path algorithm," *J. Algorithms*, vol. 49, no. 1, p. 114–152, Oct. 2003. [Online]. Available: https://doi.org/10.1016/S0196-6774(03)00076-2

[29] M. Sutton, T. Ben-Nun, and A. Barak, "Optimizing parallel graph connectivity computation via subgraph sampling," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018, pp. 12–21.

[30] Y. Shiloach and U. Vishkin, "An o(logn) parallel connectivity algorithm," *Journal of Algorithms*, vol. 3, no. 1, 1982.

[31] "Gap benchmark suite," https://github.com/sbeamer/gapbs, accessed July. 30, 2021.

[32] L. Dhulipala, G. E. Blelloch, and J. Shun, "Low-latency graph streaming using compressed purely-functional trees," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 918–934. [Online]. Available: https://doi.org/10.1145/3314221.3314598

[33] D. Ediger, R. McColl, J. Riedy, and D. A. Bader, "Stinger: High performance data structure for streaming graphs," in *2012 IEEE Conference on High Performance Extreme Computing*, 2012, pp. 1–5.