GRACE: Alleviating Reconstruction Cost in Dynamic Graph Processing Systems

Hongru Gao, Shuhao Zhang*, Xiaofei Liao, Hai Jin
National Engineering Research Center for Big Data Technology and System,
Services Computing Technology and System Lab, Cluster and Grid Computing Lab,
School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China
{hrgao, shuhao_zhang, xfliao, hjin}@hust.edu.cn

***Abstract*—Efficient dynamic graph processing is critical for real-time applications. Recent systems utilize hybrid layouts combining *Packed Memory Array* (PMA) and *Compressed Sparse Row* (CSR) structures to balance updating and computing efficiency. However, these systems face key challenges, including costly global copying and traversal during reconstruction, which in turn induce excessive costly rebalancing processes during graph updating, limiting performance under intensive updates. To mitigate these, existing solutions either compromise cache efficiency by relaxing memory contiguity or apply OS-level techniques without exploiting graph structural properties, leaving large optimization space unexplored. In this paper, we propose *GRACE*, a lightweight extension for PMA-based CSR systems that leverages graph structural properties to improve reconstruction and support efficient updates without sacrificing layout contiguity. Specifically, *GRACE* incorporates 1) a *property-guided reservation* strategy that partitions the PMA into regions and applies tailored methods, minimizing copying and traversal during reconstruction while providing optimized layout for rebalancing, and 2) a *cousin-aware rebalancing* strategy that assesses the impact of the vertices and confines rebalancing to smaller ranges by exploiting cousin segments of PMA tree, reducing redundant relocation during insertion. We implement *GRACE* as a modular plugin atop representative dynamic graph processing systems, including PPCSR, Terrace, and VCSR. Experimental results show that *GRACE* effectively accelerates their reconstruction and achieves substantial improvements in graph updating efficiency while maintaining comparable computing performance.**

***Index Terms*—Dynamic graph processing, Graph updating, Packed memory array, Compressed sparse row**

I. Introduction

Dynamic graph processing plays a crucial role in modern data-intensive applications, such as social networks, e-commerce, traffic management, and fraud detection [1]–[6], where graph structures evolve continuously. The goal of dynamic graph processing is to achieve high performance in two critical tasks: (1) *graph updating*, which merges updates into the graph, and (2) *graph computing*, which runs various computing tasks on the updated graph [7]–[11]. Among existing dynamic graph processing systems, *Packed Memory Array* (PMA)-based *Compressed Sparse Row* (CSR) systems [12]–[17] stand out for combining high updating and computing efficiencies. These systems replace CSR's compact edge array with a PMA structure, reserving space between edges to mitigate structural changes during updates while maintaining CSR's edge indexing for efficient computations.

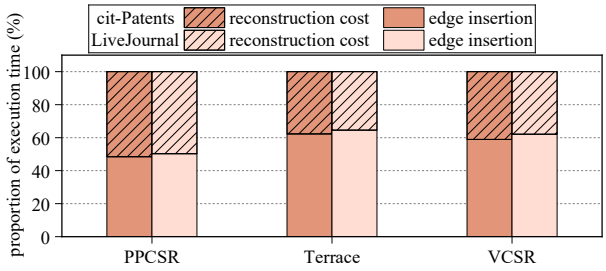


Fig. 1: Breakdown of overall updating times on three dynamic graph processing systems: PPCSR, Terrace, and VCSR.

Despite balancing update and computation performance, the efficient scaling of PMA-based CSR systems remains an open question. As new edges are inserted, the reserved space in PMA will be exhausted, and the system triggers a costly *reconstruction* process to reallocate more space for future updates. Figure 1 shows that in PMA-based CSR systems, including PPCSR, Terrace, and VCSR [13]–[15], reconstruction accounts for 35.4% to 51.5% of overall updating time under edge insertion workloads simulated by using real-world datasets `cit-Patents` and `LiveJournal` [18]. Such high reconstruction overhead leads to limited updating performance (handling 3.2 million to 7.8 million edges per second), and is particularly unacceptable for time-sensitive applications such as cybersecurity [19], where rapid updates (exceeding 10 million edges per second at peak times) and timely analysis results are crucial.

To address the performance degradation caused by reconstruction during updates, we observe that three critical challenges have to be overcome.

1) *Challenge 1: Massive global copying*. During the first phase of reconstruction, i.e., edge migration, physically copying all edges to a newly allocated array incurs high overhead due to PMA's contiguous structure.
2) *Challenge 2: Costly global traversal*. During the second phase of reconstruction, i.e., offset updating, updating vertex offsets requires full-array scans due to irregular neighbor list positions, resulting in high traversal costs.
3) *Challenge 3: Excessive rebalancing movement*. The

* Shuhao Zhang is the corresponding author.

reconstruction process may produce uniform layouts, which lead to extensive (and typically redundant) PMA rebalancing movements to insert edges, exacerbated by data distribution imbalances.

While existing works have proposed various methods to address these challenges, limitations remain. Zhang et al. [20] propose to solve *Challenge 1* by physically partitioning PMA into smaller arrays and limiting edge copying to localized array regions that exceed density thresholds. However, this method incurs high indexing overhead and a non-contiguous layout, degrading computing efficiency. Furthermore, it overlooks *Challenges 2 and 3*. Deleo et al. [21] try to tackle *Challenge 1* by pre-allocating large buffers and applying OS-level optimizations. Edges are first redistributed to the buffer and then remapped to the edge array, reducing edge copying overhead by eliminating the traditional gather stage. However, this approach suffers from poor memory utilization with high buffer maintenance overhead and leaves *Challenges 2 and 3* unresolved. To the best of our knowledge, no existing solution fully addresses all three challenges of reconstruction.

In this paper, we design *GRACE*, a novel method that addresses the aforementioned three challenges through two strategies: *property-guided reservation* and *cousin-aware rebalancing*. *GRACE* is a modular plugin that can be deployed atop existing PMA-based CSR systems to accelerate their reconstruction and improve updating efficiency. The design of *GRACE* is inspired by our analysis of dynamic graph processing workloads, which reveal significant variations in neighbor list expansion rates across vertices. This observation guides us to replace traditional indiscriminate reconstruction and rebalancing with a two-phase procedure: (1) *Property-driven classification* and (2) *Tailored methods*.

To address *Challenges 1 and 2*, *GRACE* uses the property-guided reservation strategy: (1) *Property-driven classification*. The strategy identifies *hot* and *cold* regions within the PMA based on their update frequencies. (2) *Tailored methods*. Hot regions apply fine-grained migration to redistribute edges one by one, while cold regions utilize coarse-grained migration via page remapping, efficiently relocating entire edge blocks without extensive redistribution. During offset updating, hot regions apply region-wide traversal to locate vertex offsets, whereas cold regions benefit from structural regularity, allowing simultaneous updates of multiple offsets through efficient localized traversals.

To address *Challenge 3*, *GRACE* uses the cousin-aware rebalancing strategy: (1) *Property-driven classification*. Vertices are classified as *low-impact* or *high-impact* based on their structural influence, where low-impact vertices cause minimal structural changes, while high-impact vertices play a critical role in space adjustment. (2) *Tailored methods*. Rebalancing for low-impact vertices incorporates nearby cousin segments to density check scopes, confining operations to smaller PMA segments and reducing large-scale movement. For high-impact vertices, original binary tree-guided rebalancing is performed to efficiently reorganize the reserved space and reduce rebalancing frequency.

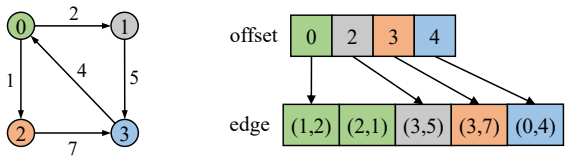


Fig. 2: An example graph and its CSR storage format.

We integrate *GRACE* into three representative dynamic graph processing systems, PPCSR, Terrace, and VCSR, for extensive evaluation. Experimental results show that *GRACE* significantly improves the performance of these systems across multiple aspects, with up to 10.34x faster reconstruction, 1.43x faster updates, and 1.15x higher graph computation efficiency, demonstrating the effectiveness and practicality of *GRACE* for real-world deployment. Our code is available at https://github.com/intellistream/GRACE.

In summary, the key contributions of this paper are:

- We propose a property-guided reservation strategy that adaptively reserves space using tailored techniques based on PMA region properties, reducing global copying and traversal costs during reconstruction.
- We develop a cousin-aware rebalancing strategy that uses cousin segments to confine data movement at lower PMA tree levels, mitigating redundant costs during insertion.
- We integrate the property-guided reservation strategy and the cousin-aware rebalancing strategy into the proposed system *GRACE*, and perform sufficient evaluations to demonstrate its effectiveness.

The rest of this paper is organized as follows. Section 2 presents the preliminaries and the motivation of this paper. Section 3 details *GRACE* with its property-guided reservation and cousin-aware rebalancing strategies. Section 4 provides the implementation details. Section 5 presents the evaluation of *GRACE*. Section 6 reviews related work. Section 7 concludes the paper and outlines future research directions.

II. Preliminaries and Motivation

This section presents the background and the motivation of our work. The related terminologies are listed in Table I.

A. Dynamic Graph Processing and Key Data Structures

We begin by presenting the definition of *dynamic graph processing*, derived from [22]–[24]. In this paper, we follow this sequential processing workflow:

Definition 1. *A dynamic graph $G_t = (V_t, E_t)$ consisting of a set of vertices V_t and edges E_t at time t, merges updates ΔG accumulated during the period $[t, t+1]$ and forms a new version $G_{t+1} = (V_{t+1}, E_{t+1})$. Then, a specific algorithm is conducted on this new version to provide analysis results for user requests. These processes are termed graph updating and graph computing, respectively.*

Compressed Sparse Row (CSR) is a widely-used data structure in dynamic graph processing [22], which stores edges contiguously and orderly in an edge array and maps vertices

TABLE I: Summary of terminologies

Symbol	Description
$G = (V, E)$	A graph G, its vertex set V and edge set E
$\|V\|$, $\|E\|$	The number of vertices and edges in the graph G
T_{updating}	Overall graph updating execution time
$T_{\text{reconstruction}}$	Reconstruction execution time
$T_{\text{insertion/deletion}}$	Edge insertion or deletion execution time
$T_{\text{computing}}$	Graph computing execution time
$f_{\text{Trad\_Recon}}$	The traditional reconstruction method
$f_{\text{Trad\_Rebal}}$	The traditional rebalancing method
T_{gc}	Global copying time in $f_{\text{Trad\_Recon}}$
T_{gt}	Global traversal time in $f_{\text{Trad\_Recon}}$
T_{rm}	Rebalancing movement time in $f_{\text{Trad\_Rebal}}$
D_{recon}	Data distribution generated by $f_{\text{Trad\_Recon}}$
M	Memory layout of the data structure

to neighbor lists via an offset array (Figure 2). CSR's sorted, compact format supports efficient computations but struggles with updates, as edge insertions require resizing the edge array, shifting massive edges, and modifying the offset array.

In recent years, *Packed Memory Array* (PMA)-based CSR [12]–[17] has emerged to enhance CSR's updating efficiency. PMA [25] achieves this goal by reserving spaces between edges and dynamically adjusting them as needed. Edges in PMA are organized into *leaf segments* controlled by predefined density thresholds. When these thresholds are exceeded, PMA rebalances the segments using an implicit binary tree, where non-leaf nodes denote larger segments formed by smaller ones. As a result, PMA-based CSR (Figure 3) replaces the original compact edge array with PMA to combine the strength of PMA and CSR. Neighbor lists are separated by *sentinels*, whose positions are stored in the offset array for efficient access. This data structure supports efficient updates by utilizing the reserved space, while preserving CSR's efficient indexing for high-performance computations.

B. Motivation

PMA-based CSR systems provide efficient solutions for updates and computations. However, when reserved space runs out as the graph grows (or remains excessive as the graph shrinks), a costly *reconstruction* process is triggered, interrupting continuous edge insertion (or deletion) to reallocate storage space for future updates, expressed as:

$$T_{\text{updating}} = T_{\text{reconstruction}} + T_{\text{insertion/deletion}} \tag{1}$$

While reconstruction is essential to maintain the system's functionality, it incurs significant overhead that limits overall updating performance (Figure 1). In this paper, we take insertion cases as examples for analysis and optimization, while deletion cases are discussed in our technical report [26].

A deep analysis on PPCSR [15], one of the state-of-the-art PMA-based CSR systems optimized for parallel updates, reveals three key challenges associated with reconstruction.

Challenge 1: Costly global copying, and Challenge 2: Global traversal overhead. Traditional PMA reconstruction involves two steps: *reallocation*, which doubles the array size while preserving the offsets of existing edges, and

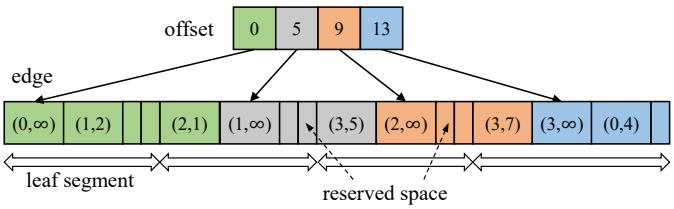
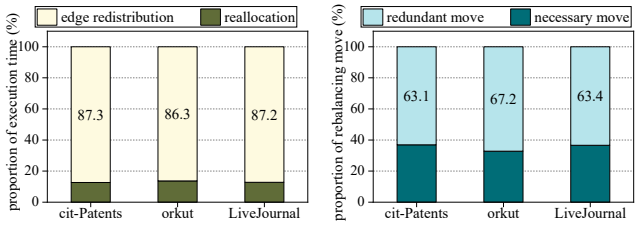


Fig. 3: An example PMA-based CSR representing the graph in Figure 2, where (i, ∞), $(i = 0, 1, 2, 3)$ denotes a sentinel used to separate the neighbor lists in the edge array.



(a) Reconstruction breakdown (b) Proportion of rebalancing

Fig. 4: Breakdown of reconstruction time, and proportion of rebalancing operations on PPCSR [15] in three test datasets: `cit-Patents`, `orkut`, and `LiveJournal` [18].

edge redistribution, which compacts existing edges (*Gather*), redistributes them with fixed intervals (*Scatter*), and updates the offset array through full-array traversals.

Our preliminary tests on reconstruction breakdown (Figure 4a) reveal that edge redistribution accounts for 86.3% to 87.3% of total reconstruction time, making it the dominant bottleneck. This is attributed to the **global copying time** (T_{gc}) for transferring edge data across arrays and the **global traversal time** (T_{gt}) for sentinel identifications. Both execution times are proportional to the edge count $|E|$, such that:

$$T_{\text{reconstruction}} = f_{\text{Trad\_Recon}}(T_{\text{gc}}, T_{\text{gt}}), \quad T_{\text{gc}}, T_{\text{gt}} \propto |E| \tag{2}$$

Challenge 3: Excessive rebalancing movement. Traditional reconstruction methods enforce uniform data layouts through indiscriminate fixed-interval space reservation policies, leading to excessive high-level rebalancing along the PMA tree and introducing large-scale data movement, while often ignoring nearby density-satisfying segments at the examined tree level.

Our preliminary tests on the rebalancing operations indicate that 63.1% to 67.2% of rebalancing operations are *redundant* (Figure 4b), as they could have been performed on smaller regions with fewer data movements by exploiting nearby eligible segments. Further analysis reveals that these redundant movements are predominantly caused by low-degree vertices, with over 80% of such vertices having degrees less than 5% of the maximum degree of the graph. This evidence highlights the limitations of *reconstruction's uniform layout and rigid rebalancing mechanisms*, which fail to adapt to imbalances in data distribution. Consequently, the suboptimal reconstruction significantly amplifies **rebalancing movement time** T_{rm}, making it a dominant contributor to $T_{\text{insertion}}$:

$$T_{\text{insertion}} = f_{\text{Trad\_Rebal}}(T_{\text{rm}}), \quad T_{\text{rm}} \propto \Phi(D_{\text{recon}}, M) \tag{3}$$

where $D_{\text{recon}} \sim U[0, E]$ denotes the uniform data distribution imposed by reconstruction, M represents the memory layout of data structure, and $\Phi(\cdot)$ is an abstract function capturing the combined influence of D_{recon} and M on T_{rm}.

III. *GRACE* System

This section begins with problem formulation, followed by a design overview. Then, it details the two proposed strategies of *GRACE* and concludes with a theoretical comparison with existing methods to highlight the effectiveness of *GRACE*.

A. Problem Formulation

The goal of *GRACE* is to minimize the total updating time of the underlying dynamic graph processing system when reconstruction occurs, which can be expressed by combining Equations (2) and (3):

$$T_{\text{updating}} = T_{\text{reconstruction}}(T_{\text{gc}}, T_{\text{gt}}) + T_{\text{insertion}}(T_{\text{rm}}) \quad (4)$$

We aim to achieve this goal by reducing T_{gc}, T_{gt}, and T_{rm}, while ensuring that $T_{\text{computing}}$, also influenced by $\Phi(D_{\text{recon}}, M)$, remains within acceptable bounds compared to the original systems without *GRACE* integration $T^{\text{original}}_{\text{computing}}$. The problem can be mathematically formulated as follows:

$$\min_{T_{\text{gc}}, T_{\text{gt}}, T_{\text{rm}}} T_{\text{updating}} \quad \text{s.t.} \quad T_{\text{computing}} \leq \delta \cdot T^{\text{original}}_{\text{computing}} \quad (5)$$

where δ (typically close to 1) defines the tolerance for $T_{\text{computing}}$ relative to the original systems. We set δ to 1 in our experiments, as *GRACE* can consistently improve computing performance of the original systems. More workloads may be evaluated in the future.

B. Design Overview

To address the challenges discussed above, we propose a system *GRACE*, which improves Graph updating performance by leveraging *property-guided Reservation* And *Cousin-aware rebalancing* stratEgies. Notably, *GRACE* can be deployed as a plugin on top of existing PMA-based CSR dynamic graph processing systems to enhance their updating efficiencies.

First, *GRACE* uses a property-guided reservation strategy to **reduce global copying and traversal overhead** through property-driven partitioning and adaptive processing. The PMA is divided into hot (frequently updated) and cold (infrequently updated) regions based on update frequency. Hot regions apply fine-grained migration to redistribute edges for efficient updates. Conversely, cold regions adopt coarse-grained, page remapping-based migration, processing on demand without premature redistribution. This method replaces global copying time T_{gc} with the combination of *region-wide copying* time and cheaper *remapping* time. Furthermore, using the structural regularity of remapping, the global traversal time T_{gt} is optimized to *region-wide traversal* time for hot regions and faster *localized traversal* time for cold regions to update multiple sentinels simultaneously.

Second, *GRACE* adopts a cousin-aware rebalancing strategy to **minimize rebalancing movement**. By exploiting reserved space in cousin segments, this strategy relaxes density constraints and significantly reduces data relocation for PMA rebalancing. *GRACE* introduces a decision model that classifies vertices as *high-impact* or *low-impact* based on their propensity of massive updates. High-impact vertices, prone to large-scale updates, retain binary tree-checking paths to handle updates efficiently and reduce rebalancing frequency. Conversely, low-impact vertices relax density checks to cover cousin segments, limiting rebalancing to lower tree levels with smaller movement. Thus, T_{rm} is reduced by shifting low-impact rebalancing from higher to lower levels, while the rebalancing for high-impact vertices is preserved.

C. Property-guided Reservation Strategy

The property-guided reservation strategy consists of four steps: *PMA partitioning*, *space assignment*, *edge migration*, and *sentinel identification*. Figure 5 visually illustrates this strategy, highlighting the current PMA (in green) and the newly allocated array (in orange). The two differently shaded blocks represent existing edges and sentinels, respectively.

A) PMA Partitioning: Most real-world graphs exhibit power-law distributions, where a few vertices' neighbor lists grow disproportionately compared to others [27]. These high-growth vertices require more reserved space to store future updates efficiently. Differentiating between frequently and infrequently updated regions and applying customized strategies creates opportunities to mitigate global copying and traversal overhead while optimizing reconstruction layout.

Our strategy combines *vertex degree* and *dynamic updates of vertices* to identify *hot* vertices. High-degree vertices tend to experience frequent updates, and incorporating dynamic updates provides a better prediction of graph evolution. Specifically, we compute a *hotness score* for each vertex u:

$$score(u) = 0.5 \times \hat{Degree}(u) + 0.5 \times \Delta\hat{Update}(u) \quad (6)$$

where $\hat{Degree}(u)$ is the normalized degree of u relative to the maximum degree, and $\Delta\hat{Update}(u)$ denotes the normalized change in u's degree between two reconstructions relative to the maximum change. As $score(u) \in [0, 1]$, the vertex u is classified as *hot* if its score exceeds a tunable threshold η, where $\eta \in (0, 1)$ controls the classification sensitivity. Vertices with hotness scores below this threshold are classified as *cold*. Furthermore, using a user-defined parameter *block size* (bs, denoting the number of *pages* in a block) and the vertex offsets accessible through the offset array, we identify the starting indices of *hot regions*, i.e., the ranges of neighbor lists corresponding to hot vertices in the PMA. For example, Figure 5 shows the current edge array divided into four blocks based on the *block size*, and the 2^{nd} block is marked as hot.

B) Space Assignment: Based on PMA partitioning results, we design an *adaptive* space reservation scheme tailored to the update frequency of each region. Firstly, we assign a *base space* of length bs to every two successive regions, no matter whether they are hot or cold. These base spaces act as *buffers*, ensuring stability when regions undergo hotness transitions. Then, the remaining new space is assigned among all identified hot regions based on their *region scores*, which

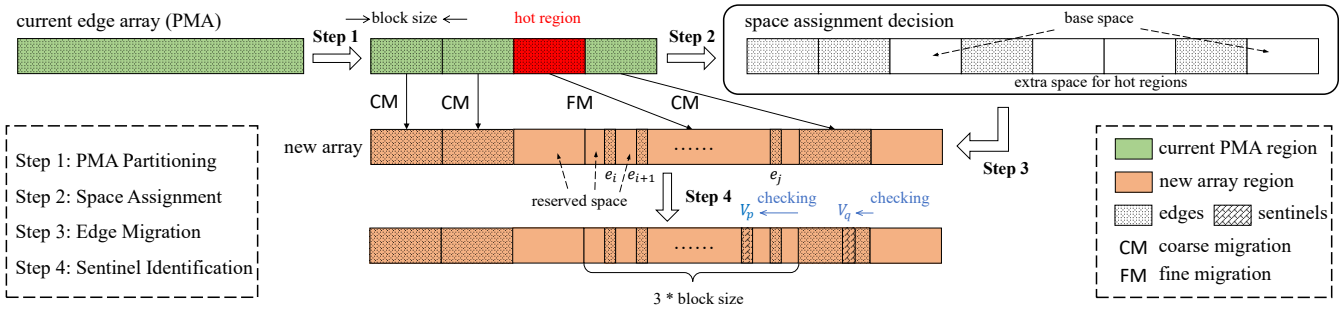


Fig. 5: Procedures of the property-guided reservation strategy.

are the sum of vertex hotness scores within each region. The space assignment for hot regions is based on a sub-linear function (i.e., the square root of the region score), providing a smoother, less aggressive distribution compared to a linear method. In contrast, with the base space provided, cold regions are no longer assigned additional space, ensuring that memory resources are concentrated in regions with higher anticipated update frequencies. Finally, the relocation destinations for both hot and cold regions in the new array are computed using a prefix-sum calculation based on the above space assignment.

For example, in Figure 5, two newly allocated blocks are reserved as base spaces. Since only one hot region (2^{nd} block) is identified, the remaining two blocks are assigned as its extra space. Notably, this step defines an abstract assignment plan, without any actual data migration.

C) Edge Migration: According to the space assignment, to efficiently migrate edge data across arrays while minimizing copying overhead, we adopt tailored strategies for the distinct characteristics of hot and cold regions.

For hot regions, we use a *fine-grained migration* approach, redistributing their edges to their reserved space in the new array at fixed intervals. The interval is calculated by dividing the total reserved space for a hot region by the number of edges it contains. This precise redistribution evenly distributes the reserved space among edges, allowing the new layout to handle future updates with minimal structural adjustments.

In contrast, cold regions are handled using an on-demand design paradigm to avoid premature redistribution, which can lead to costly "ping-pong" effects [21]. These effects arise when data are unnecessarily moved during reconstruction but later require further adjustments during insertion. To address this, cold regions migrate in *coarse* block units, deferring fine-grained movement until PMA rebalancing is required. To efficiently execute such block migrations, we employ OS-supported *page remapping* techniques, which directly relocate the page frames of cold regions to the targeted virtual pages of the new array, eliminating costly physical copying and improving migration efficiency.

In Figure 5, hot region (2^{nd} block) is handled with fine-grained migration, redistributing its edges e_i to e_j evenly across its reserved space in the new array (3^{rd} to 5^{th} blocks). Cold regions apply coarse-grained migration, where the 0^{th}, 1^{st}, and 3^{rd} blocks of the PMA are remapped directly to the 0^{th}, 1^{st}, and 6^{th} blocks of the new array, respectively.

D) Sentinel Identification: Once all existing edges have been migrated to the new array, the offset array needs to be updated by identifying sentinels within the array. The method of sentinel identification varies depending on the migration strategy. Hot regions require *region-wide traversals* to locate sentinels, due to their irregular redistribution. Conversely, cold regions leverage the structural regularity introduced by block migrations to apply more efficient *localized traversals*. As edges within a block are migrated simultaneously, their relative positions are preserved during the process. This allows the vertex range within a block to be determined through localized checks, so that the offsets of these vertices can be updated simultaneously by adding the same interval value.

As illustrated in Figure 5, sentinel identification on the hot region requires a traversal from e_j to e_i to locate the sentinels, while cold regions employ an optimized method. The sentinels within the 0^{th} and 1^{st} blocks are unchanged, as they retain their original offsets after migrations. To handle the 6^{th} block, the sentinel nearest to its preceding block and the corresponding vertex ID V_p are pre-identified when traversing in the hot region. Then a localized check is conducted, starting from the end of 6^{th} block and moving left, until a sentinel and its vertex ID V_q are found. This process establishes the vertex range V_k ($k \in [p+1, q]$) within the 6^{th} block. Using this range, the offsets of these vertices are updated simultaneously by adding the same value (i.e., $3\times$ *block size*, from the 3^{rd} to 6^{th} blocks) to their original offsets.

Configuration: After describing the four primary steps, we present the configurations underpinning the proposed strategy. The new array is allocated with 4KB memory alignment to enable page-based optimization techniques. Moreover, unlike traditional methods that extend the current edge array and initialize the additional space with zeros, our approach allocates the new edge array without initialization, facilitating region-tailored edge migration and sentinel identification.

D. Cousin-aware Rebalancing Strategy

Before detailing our strategy, we first illustrate the redundant movement caused by PMA rebalancing. As shown in Figure 6, inserting a new edge e_6 causes the leaf segment [4,7] to exceed its threshold, triggering a density check along the PMA tree. This check also fails at the parent segment [0,7] and ultimately triggers a root-level full rebalancing of all 10 edges. As e_6 resides in a cold region, such excessive rebalancing is

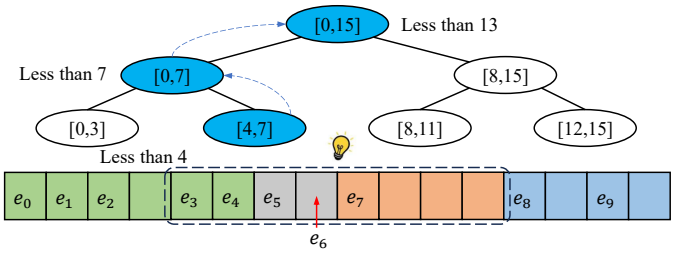


Fig. 6: Redundant movements in PMA rebalancing. Each tree node label "[a,b]" indicates the index range of the corresponding PMA segment in the edge array. Different colors in the array represent the neighbor lists of vertices. The blue path on the PMA tree represents the density check triggered by the insertion of e_6.

unnecessary. This phenomenon arises from the strict density checking path on the PMA tree, overlooking potentially available nearby segments. As shown in Figure 6, we find that the filled leaf segment [4,7] could instead be rebalanced with its cousin segment [8,11], effectively maintaining the density threshold with less movement.

Inspired by this insight, we propose a cousin-aware rebalancing strategy that uses *cousin segment* assessments to reduce redundant movements during PMA rebalancing. This strategy is built on an adaptive processing principle, customizing rebalancing approaches for vertex characteristics. At its core, the strategy dynamically classifies the vertices to be processed as *low-impact* or *high-impact*, selecting the most efficient rebalancing method for each case.

Classification Rationale: The terms *low-impact* and *high-impact* reflect whether it is worthwhile for vertices to perform higher density checks and balancing operations.

- *High-impact Vertices*: These vertices, characterized by high degrees or significant influence in their local regions, often receive frequent updates and may extensively expand their neighbor lists. To minimize rebalancing on these vertices, it is beneficial to reserve sufficient space for their future growth proactively.
- *Low-impact Vertices*: These vertices have relatively low degrees and minimal influence on their local regions. Large-scale data adjustments for low-impact vertices are often counterproductive, as their neighbor lists may compress again after expansion due to the dominant influence of nearby high-impact vertices. Thus, rebalancing for low-impact vertices should be confined to small, localized ranges, reducing data movement caused by repeated cycles of expansion and compression.

A Lightweight Decision Model for Classification: As shown in Figure 7, we design a lightweight decision model to classify vertices efficiently. The decision model combines the global information (maximum degree $d_{\max}$ of the graph) and local information (total degree d within a vertex window).

- *Degree Threshold*: A vertex is initially classified as high-impact if its degree exceeds a global threshold, defined as $d_{\max} \cdot \eta$, where η is a user-defined parameter consistent with that used in the property-guided reservation strategy.
- *Localized Influence Evaluation*: For vertices below the global threshold, the model evaluates their localized influence. A vertex window $[v-i, v+i]$, centered on the vertex v being processed, is constructed by incrementally accumulating nearby vertices with consecutive i IDs, until the total degree d within the window exceeds the segment length of the current inspected level. A vertex is classified as high-impact if its degree exceeds half of d; otherwise, it is classified as low-impact.

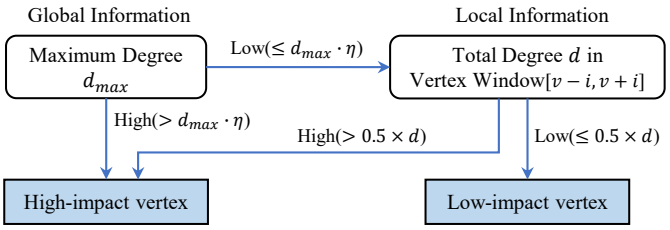


Fig. 7: Lightweight decision model for classifying high-impact and low-impact vertices.

Adaptive Rebalancing Method: After classifying the vertex, the adaptive rebalancing strategy is triggered when a density violation persists over two levels (i.e., from level l to level $l+2$). The strategy intervenes at level $l+1$, applying distinct approaches to high-impact and low-impact vertices.

For high-impact vertices, the strategy doubles the segment length through the PMA tree hierarchy. The density check then continues upward (from level $l+1$ to level $l+2$) without additional processing. This method provides sufficient space for accommodating frequent updates at once, reducing rebalancing frequency. For low-impact vertices, the strategy introduces *cousin segments* into the density check to minimize rebalancing movements caused by these vertices. Specifically, when a density violation occurs at level $l+1$, the cousin segment is dynamically identified and added to the check scope. If the segment at level l is the right child of its parent, its cousin is located contiguously to its right and merged into the *cousin scope*. Similarly, if the segment is the left child, its cousin is merged from its left. If this cousin scope satisfies density, rebalancing is performed within this scope, avoiding further processing at level $l+2$. Otherwise, the strategy escalates the density checks to higher levels of the binary tree until a valid segment is identified.

For example, in Figure 6, the density check initially fails for segment [4,7], and then for segment [0,7]. Under our strategy, the cousin scope [4,11], combining segment [4,7] with its cousin [8,11], is evaluated. As this cousin scope meets level 1's density requirement (less than 7 edges), the rebalancing is confined to [4,11], avoiding the original full-array rebalancing at the root segment.

E. Theoretical Analysis and Comparison

We analyze the advantages of *GRACE* over traditional methods in reducing *global copying*, *global traversal*,

and *rebalancing movement*, which improve reconstruction efficiency and support faster updates.

1) Reduction in Global Copying: The traditional method processes the entire edge array in two steps: (1) gathering all edges and (2) scattering them into the new array, each step requiring $|E|$ time. Therefore, the global copying time T_{gc} is:

$$T_{\text{gc}} = \Theta(|E|) \tag{7}$$

GRACE partitions the PMA into $\frac{|E|}{bs}$ blocks, where bs is the block size. Let $\alpha \in (0,1)$ denote the proportion of hot blocks. The hot blocks, containing $\alpha|E|$ edges, apply fine-grained migrations and take $\alpha|E|$ time. On cold blocks, remapping is performed $(1-\alpha)\frac{|E|}{bs}$ times for coarsely migrating edge blocks, each involving a constant overhead of page remapping. Hence, the edge migration time of *GRACE* $T_{\text{GRACE\_migration}}$ is:

$$T_{\text{GRACE\_migration}} = \Theta\left(\alpha|E| + (1-\alpha)\frac{|E|}{bs}\right) \tag{8}$$

The reduction in global copying time ΔT_{gc} is therefore:

$$\Delta T_{\text{gc}} = \Theta\left((1-\alpha)|E| - (1-\alpha)\frac{|E|}{bs}\right) \tag{9}$$

The reduction ΔT_{gc} increases with smaller α (fewer hot blocks) and larger bs (fewer remapping operations).

2) Reduction in Global Traversal: In traditional methods, identifying sentinels requires traversing the entire edge array, costing $|E|$ time. Hence, the global traversal time T_{gt} is:

$$T_{\text{gt}} = \Theta(|E|) \tag{10}$$

GRACE optimizes global traversal by distinguishing hot and cold regions. For hot blocks, $\alpha|E|$ edges are scanned, taking $\Theta(\alpha|E|)$ time. For $(1-\alpha)\frac{|E|}{bs}$ cold blocks, localized traversal processes $\bar{k}$ edges on average per block, where $\bar{k} \in [1, bs]$. Thus, the total traversal time of *GRACE* $T_{\text{GRACE\_traversal}}$ is:

$$T_{\text{GRACE\_traversal}} = \Theta\left(\alpha|E| + (1-\alpha)\frac{|E|}{bs}\cdot\bar{k}\right) \tag{11}$$

The reduction in global traversal time ΔT_{gt} is:

$$\Delta T_{\text{gt}} = \Theta\left((1-\alpha)|E|\cdot\left(1-\frac{\bar{k}}{bs}\right)\right) \tag{12}$$

In most practical scenarios, $\bar{k}$ is much smaller than bs, leading to a significant ΔT_{gt}.

3) Reduction in Rebalancing Movement: In the traditional method, density checks propagate along the PMA tree from leaf to root. At each level $h \in [0, H]$ (with H the tree height), let M_h denote the edges moved during rebalancing. The total rebalancing movement time T_{rm} is therefore $\sum_{h=0}^{H} M_h$. Given the inherent complexity and unpredictability of the rebalancing process, we approximate T_{rm} using upper bounds. As M_h is bounded by $O(|E|)$, the expression simplifies to:

$$T_{\text{rm}} \le \sum_{h=0}^{H} O(|E|) = O(|E|\cdot H) \tag{13}$$

In *GRACE*, let β represent the proportion of high-impact vertices for which traditional rebalancing is retained. The rebalancing time for these high-impact vertices is given by $\beta\cdot\sum_{h=0}^{H} M_h$. For the remaining $(1-\beta)$ low-impact vertices, *GRACE* shifts the rebalancing process at each level h down by Δh levels, reducing the movement per level to $\frac{M_h}{2^{\Delta h}}$. Thus, their rebalancing time is $(1-\beta)\sum_{h=0}^{H-\Delta h}\frac{M_h}{2^{\Delta h}}$. Using the upper bound $M_h = O(|E|)$, the total rebalancing time in *GRACE* $T_{\text{GRACE\_rebalancing}}$ is expressed as:

$$T_{\text{GRACE\_rebalancing}} = O\left(\beta|E|\cdot H + (1-\beta)|E|\cdot\frac{H-\Delta h}{2^{\Delta h}}\right) \tag{14}$$

The reduction in rebalancing movement time ΔT_{rm} is:

$$\Delta T_{\text{rm}} = O\left((1-\beta)|E|\cdot\left(H - \frac{H-\Delta h}{2^{\Delta h}}\right)\right) \tag{15}$$

This shows that ΔT_{rm} increases as Δh grows, due to the exponential decrease in the impact of higher-level rebalancing.

Algorithm 1: Property-guided reservation strategy

Input: current edge array PMA, current offset array $offset\_array$, predefined block size value bs, predefined selected threshold η
Output: resized edge array PMA, updated offset array $offset\_array$

1 $len \leftarrow$ Get_length(PMA);
2 $new\_PMA \leftarrow$ Aligned_alloc(2×len);
3 $vertex\_score[] \leftarrow$ Compute_score($offset\_array$);
4 $regions[] \leftarrow$ Identify_region(PMA, $vertex\_score[]$, bs, η);
5 $space\_assign[] \leftarrow$ Assign_space($regions[]$, len);
6 $migration\_des[] \leftarrow$ Prefix_sum($space\_assign[]$);
7 $total\_blocks \leftarrow \frac{len}{bs}$;
8 $block\_id \leftarrow 0$;
9 **while** $block\_id < total\_blocks$ **do**
10 **if** $regions[block\_id]$ is hot **then**
11 Fine_migration(PMA, new_PMA, $block\_id$, $migration\_des$, bs);
12 **else if** $regions[block\_id]$ is cold **then**
13 Coarse_migration(PMA, new_PMA, $block\_id$, $migration\_des$, bs);
14 $block\_id \leftarrow block\_id + 1$;
15 $PMA \leftarrow new\_PMA$;
16 Update_offset($offset\_array$, $regions[]$, bs);
17 **return** PMA, $offset\_array$;

IV. Implementation Details

This section gives the algorithm and implementation details of *GRACE*.

A. Algorithm for GRACE

1) Property-guided Reservation Strategy: Its pseudo-code is outlined in Algorithm 1. The process begins with allocating a memory-aligned array new_PMA with double capacity (Lines 1-2). Then, the hotness scores $vertex\_score[]$ for vertices are computed, and the edge array is partitioned and classified into $regions[]$ based on the block size bs and a predefined threshold η (Lines 3-4). Subsequently, a space assignment scheme $space\_assign[]$ is derived from $regions[]$, and migration destinations $migration\_des[]$ for

Algorithm 2: Cousin-aware rebalancing strategy

Input: a new edge $e = (u, v)$, the edge array PMA, predefined block size value bs, pedefined threshold η, maximum degree $d_{\max}$
Output: updated edge array PMA

```
1  d_u ← degree(u);
2  impact_type ← Decision_model(u, d_u, d_max);
3  current_scope ← Determine_initial_scope(e, PMA);
4  while !Is_scope_satisfied(current_scope) do
5      current_scope ← Expand(current_scope);
6      if current_scope == Get_length(PMA) then
7          Reconstruct(PMA);
8          break;
9      if Is_scope_satisfied(current_scope) then
10         Rebalancing(current_scope, PMA);
11         break;
12     else if impact_type is low-impact then
13         cousin_scope ← Identify_cousins(current_scope,
             PMA);
14         if Is_scope_satisfied(cousin_scope) then
15             Rebalancing(cousin_scope, PMA);
16             break;
17 return PMA;
```

these regions in the new array new_PMA are computed using a prefix sum (Lines 5-6). The algorithm then iterates over each block, migrating edges from PMA to new_PMA using region-specific strategies: fine-grained migration for hot regions and coarse-grained migration for cold regions (Lines 9–14). Finally, new_PMA replaces the original PMA as the edge array (Line 15), and the offset array is updated in an adaptive manner consistent with the migration strategy (Line 16).

2) Cousin-aware Rebalancing Strategy: Its pseudo-code is given in Algorithm 2. Initially, the degree d_u of the source vertex u is computed (Line 1). Based on d_u and the maximum degree $d_{\max}$, the $impact\_type$ of u is determined using our proposed decision model (Line 2). After classifying the impact type, the algorithm identifies the initial segment $current\_scope$ for the density check (Line 3). If this segment does not meet the density threshold, it is expanded by doubling its length (Lines 4-5). Once the expansion reaches the root, a reconstruction is executed (Lines 6-8). If, at this stage, $current\_scope$ satisfies the density condition, rebalancing is performed (Lines 9-11). Otherwise, if $impact\_type$ is low-impact, the cousin assessment is triggered (Lines 12-16). Specifically, the $cousin\_scope$ is identified and tested against the density threshold. If the threshold is met, rebalancing is performed on this $cousin\_scope$; otherwise, the while-loop proceeds to the next iteration.

B. Implementation of GRACE

The key implementation of *GRACE* is the page remapping technique adopted in the property-guided reservation strategy. This technique utilizes the Linux `mremap` system call with the `MREMAP_FIXED` option and a configurable parameter bs, allowing for precise remapping of selected page frames to fixed virtual addresses. As the newly allocated array is uninitialized, its virtual-to-physical mapping is not pre-established, thus avoiding additional unmapping overhead and improving `mremap` efficiency. Additionally, empty pages within the array are sourced from the operating system's memory pool and automatically zeroed, obviating the need for manual initialization. This implementation offers several key benefits: (1) Multi-system Compatibility: By leveraging `mremap`, *GRACE* can be seamlessly integrated into different systems without requiring system kernel modifications; (2) Error Mitigation: The use of `mremap` avoids errors associated with direct page table manipulation, such as segmentation faults and memory corruptions. We implement *GRACE* and deploy it on existing PMA-based CSR dynamic graph processing systems PPCSR, Terrace, and VCSR for evaluation.

TABLE II: Graphs used for evaluations (M means million, Ave. D means the average degree of a graph, α means the exponent of power-law distribution)

Graphs	Types	Vertices	Edges	Ave. D	α
cit-Patents	citation	3.7M	33.0M	9	6.75
LiveJournal	social	4.8M	85.7M	18	58.99
orkut	social	3.1M	234.3M	76	50.79
rmat422	synthetic	2.1M	200.0M	95	17.39
rmat511	synthetic	8.3M	406.8M	49	4.48

C. Limitation

While the page remapping technique offers benefits, our implementation relies on the Linux-specific `mremap` syscall, limiting portability. Other widely-used OS, like Windows, lack a comparable efficient primitive for page remapping [28], requiring alternative APIs or costly emulation. From hardware aspects, GPUs manage memory through vendor-specific unified virtual memory frameworks, where page migrations may introduce significant latencies [29]. Finally, in resource-constrained environments, such as embedded systems, many devices employ lightweight Linux variants with limited support for advanced page operations [30]. Addressing these cross-platform portability challenges is left for future work.

V. Evaluation

This section first presents the experimental setups and then evaluates *GRACE* on three PMA-based CSR systems: PPCSR, Terrace, and VCSR from the following aspects:

- Does *GRACE* outperform existing methods in reconstruction performance? (Section 5.2)
- How does *GRACE* improve overall graph updating and computing efficiencies? (Section 5.3)
- How much memory usage and system-level cost does *GRACE* introduce? (Section 5.4)
- What is the impact of each strategy employed in *GRACE* on the key performance metrics? (Section 5.5)
- How do the parameters bs and η influence performance outcomes? (Sections 5.6 and 5.7)
- What is the effect of varying the number of threads on the performance of *GRACE*? (Section 5.8)

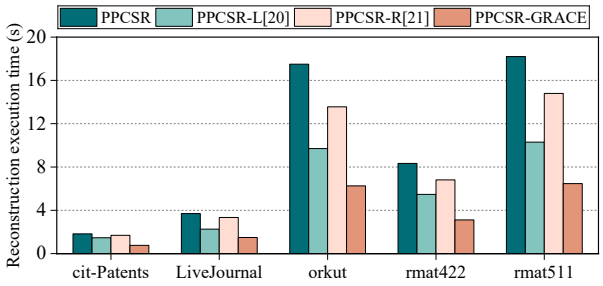
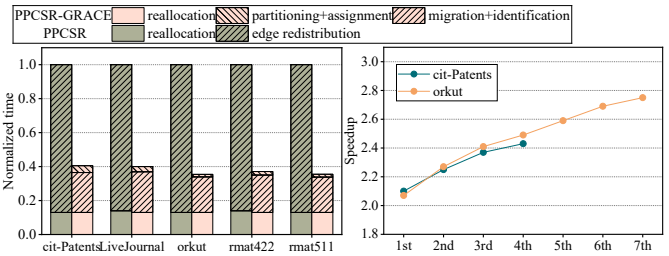


Fig. 8: Comparison of *GRACE* with traditional and state-of-the-art reconstruction methods on PPCSR: PPCSR-L and PPCSR-R respectively integrating methods from [20] and [21].



(a) Breakdown of reconstruction time (b) Speedups of each reconstruction

Fig. 9: Further analysis of PPCSR-*GRACE*'s reconstruction.

A. Experiment Setting

Baseline: We choose three representative PMA-based CSR dynamic graph processing systems as baselines. PPCSR [31] serves as the *basic baseline*, which employs traditional PMA-based CSR representation and update logic. Terrace [13] and VCSR [14] are selected as *extended baseline* to evaluate the compatibility and effectiveness of *GRACE* when integrated with specific structural and algorithmic designs. For clarity, we use PPCSR-*GRACE*, Terrace-*GRACE*, and VCSR-*GRACE* to represent the revised systems that deploy *GRACE*.

Test-bed: Experiments are conducted on a Linux Server (Ubuntu 20.04) with two Intel Xeon Gold 5117 CPUs (each with 14 cores, 2.00GHz) and 512GB memory.

Dataset: Table II lists the graph datasets, including three real-world graphs (`cit-Patents`, `LiveJournal`, and `orkut`) from SNAP [18], converted to directed graphs, and two synthetic directed graphs (`rmat422` and `rmat511`) generated by the RMAT graph generator [32]. Specifically, `rmat422` uses parameters a=0.4, b=c=0.2, d=0.2, while `rmat511` uses a=0.5, b=c=0.1, d=0.3, where a, b, c, and d define the edge sampling probabilities. All graphs are unweighted; for weighted graph evaluation, edges are assigned random integer weights in (0, 100).

Evaluation Method: We design tasks to assess reconstruction, overall updating, and computing improvements.

1) *Updating Tasks*: Our evaluation primarily focuses on edge insertions, the most dominant operations in dynamic graph processing, while also assessing deletions to ensure completeness for full dynamic workloads. Specifically, since the datasets in Table II are static graphs whose edges are sorted, we randomly shuffle these edges to simulate streaming updates. These edges are batched (10 million per batch) and inserted into initially empty graph representations of both baselines and revised systems. After all insertions are completed, the same set of edges is then removed from the graph to assess deletion performance.

2) *Computing Tasks*: We evaluate four representative graph algorithms, *Breadth-First Search* (BFS), *Single-Source Shortest Path* (SSSP), *PageRank* (PR), and *Triangle Counting* (TC), covering *partial-active graph traversal*, *all-active graph traversal*, and *graph mining* workloads [33], [34]:

- *Partial-active Graph Traversal (BFS, SSSP)*: Both expand frontiers of active vertices, leading to sparse and irregular accesses. BFS uses a queue-based method, while SSSP follows an optimized Dijkstra's algorithm [35].
- *All-active Graph Traversal (PR)*: PR iteratively updates all vertices in each round, representing relatively dense and regular accesses. We run PR for 15 iterations, initializing each rank to $\frac{1}{|V|}$.
- *Graph Mining (TC)*: TC is dominated by intensive set-intersection operations. We implement TC using a two-pointer adjacency list intersection approach, emphasizing compute-intensive local structure discovery.

Parameters: In our experiments with *GRACE*, the bs value is set to 256, and the selected threshold η is chosen as 0.5. These parameter choices are carefully optimized to maximize the performance improvements of overall updating.

B. Reconstruction Improvement

To demonstrate the superior reconstruction performance achieved by *GRACE*, we first compare it against both traditional and state-of-the-art optimized reconstruction methods. Using PPCSR as the baseline system, we integrate the methods used in LPMA [20] and RewiredPMA [21] into PPCSR, resulting in the comparative systems PPCSR-L and PPCSR-R, respectively. The experimental results, shown in Figure 8, indicate that *GRACE* consistently outperforms existing methods among all test graphs, with improvements ranging from 2.43x to 2.88x compared to the baseline.

Secondly, since *GRACE* optimizes the edge redistribution process, we analyze the breakdown of reconstruction time in Figure 9a for further analysis. For the baseline, reconstruction time is divided into *reallocation* and *edge redistribution*, while PPCSR-*GRACE* further separates it into *reallocation*, *partitioning+assignment*, and *migration+identification*[1]. From this figure, we observe that the reallocation time remains nearly constant, as no targeted modifications are applied to this stage. However, *GRACE* substantially reduces the edge redistribution time of baselines, achieving speedups ranging from 3.10x to 4.25x with only minimal PMA partitioning and space assignment costs. These results highlight the effectiveness of *GRACE* in minimizing global copying and traversal overhead during the reconstruction process.

[1]For brevity, we refer to PMA partitioning, space assignment, edge migration, and sentinel identification (four steps of property-guided reservation) as partitioning, assignment, migration, and identification.

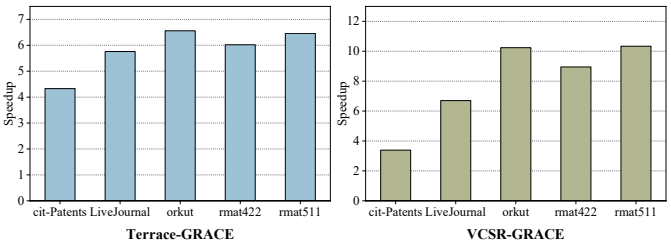


Fig. 10: Reconstruction speedups on Terrace-*GRACE* and VCSR-*GRACE*, compared to the baselines.

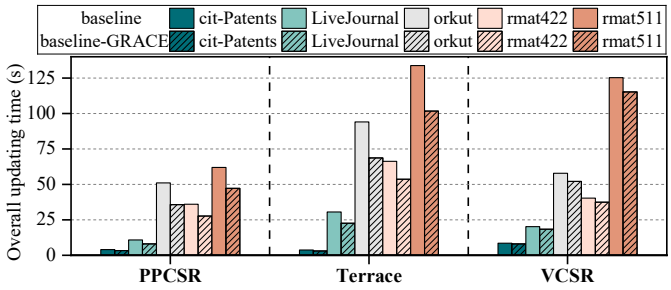


Fig. 11: Overall updating performance under insertion cases.

Next, we further analyze the impact of edge scales on reconstruction efficiency, as the speedups vary across datasets. Using `orkut` (highest speedup) and `cit-Patents` (lowest speedup) as case studies, both datasets initialize edge array capacities to 32MB, adhering to the default PMA configuration. Figure 9b shows the speedup of each reconstruction, where `cit-Patents` undergoes four reconstructions and `orkut` undergoes seven. During the first four reconstructions, both datasets expand their edge arrays from 32MB to 256MB with similar reconstruction improvements ranging from 2.10x to 2.41x. However, during the fifth to seventh reconstructions in `orkut`, further improvements are observed, contributing most to its overall speedup. These results suggest that larger edge arrays benefit more from our method, highlighting *GRACE*'s particular effectiveness in reconstructing large-scale graph data.

Finally, we evaluate the reconstruction improvements on the extended baselines: Terrace and VCSR, with the results shown in Figure 10. For Terrace-*GRACE*, the reconstruction is accelerated by 4.33x to 6.56x, while VCSR-*GRACE* achieves speedups from 3.39x to 10.34x. These results demonstrate the compatibility and practicality of *GRACE* in improving the reconstruction performance across different systems.

C. Updating and Computing Improvement

Insertion Updating Performance: Figure 11 shows the overall updating performance improvements after integrating *GRACE* across the three systems. The results indicate that *GRACE* enhances the updating efficiency of PPCSR, Terrace, and VCSR by 1.22x to 1.43x, 1.25x to 1.40x, and 1.03x to 1.11x, respectively. The highest update throughput on PPCSR, Terrace, and VCSR reaches 10.85, 10.69, and 5.34 million edges per second, respectively. These improvements underscore the effectiveness of *GRACE* in significantly accelerating updates. Notably, the performance gains for

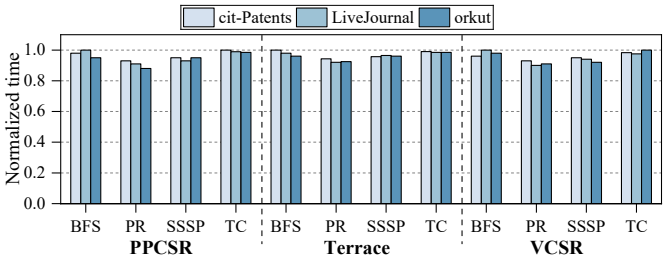


Fig. 12: Execution times of BFS, PR, SSSP, and TC algorithms on the revised systems, normalized to those of the baselines.

VCSR are more modest compared to PPCSR and Terrace. This is due to VCSR's specialized insertion logic, which limits the potential benefits of the cousin-aware rebalancing strategy. In contrast, PPCSR and Terrace follow the traditional PMA insertion logic, allowing them to fully leverage the proposed strategies and exhibit more substantial performance gains. The highest improvements for both PPCSR and Terrace are observed on `orkut`, which can be attributed to its large edge scale and high power-law exponent, amplifying the effectiveness of *GRACE*.

Deletion Updating Performance: Due to page limitations, we omit the deletion performance figures here and provide them in our technical report [26]. In short, the deletion performance of PPCSR, Terrace, and VCSR improves by 1.07x to 1.19x, 1.07x to 1.14x, and 1.03x to 1.07x, respectively. These gains arise from the cousin-aware rebalancing strategy, which remains effective during deletions by restricting density checks to the lower density threshold compared to insertion cases. Nevertheless, property-guided reservation strategy is less applicable, as the space is reclaimed and cannot be reassigned during structure shrinkage.

Computing Performance: We assess computing efficiency using four algorithms (BFS, SSSP, PR, TC) after edge insertions. As shown in Figure 12, the normalized results show slight but consistent improvements across all workloads: BFS (1.00x to 1.05x), SSSP (1.03x to 1.09x), PR (1.07x to 1.15x), and TC (1.00x to 1.03x). These gains stem from the page-aligned allocation of the edge array during initialization and reconstruction, which reduces page boundary-crossing overhead and thereby improves performance.

Improvements vary by workloads. Partial-active traversal (BFS and SSSP) achieves modest gains, as each iteration processes only active vertices. Memory alignment benefits are more noticeable when the frontier is large, but diminish as it shrinks. Graph mining (TC) benefits the least, as its adjacency-list intersections are compute-intensive, with memory access playing a relatively smaller role. By comparison, all-active traversal (PR) shows the greatest improvement. Since each iteration processes the entire graph, the algorithm incurs significant memory access. Moreover, multiple iterations further amplify the advantages of memory alignment. Overall, these results confirm that *GRACE* not only preserves but can modestly enhance computing efficiency, especially for bandwidth-sensitive workloads such as all-active traversal.

TABLE III: Memory usage (in GB). Each entry shows *baseline* → *baseline+GRACE*, denoting the variation in memory usage after deploying *GRACE*.

Graphs	PPCSR-*GRACE*	Terrace-*GRACE*	VCSR-*GRACE*
cit-Patents	1.52 → 1.61	1.60 → 1.64	2.32 → 2.43
LiveJournal	3.32 → 3.50	4.53 → 4.71	5.73 → 5.99
orkut	10.62 → 11.02	9.69 → 10.03	15.10 → 15.77
rmat422	7.24 → 7.44	8.34 → 8.65	12.83 → 13.49
rmat511	15.31 → 16.13	16.92 → 17.51	26.39 → 27.78

TABLE IV: Time overhead (in ms) of `mremap` under stress testing. Integers in parentheses mean the number of syscalls.

Graphs	PPCSR-*GRACE*	Terrace-*GRACE*	VCSR-*GRACE*
LiveJournal	31ms (1920 calls)	57ms (1792 calls)	32ms (2016 calls)
orkut	178ms (8128 calls)	113ms (3968 calls)	89ms (3968 calls)

D. Memory Usage and System-level Cost

Memory Overhead: As *GRACE* employs auxiliary data structures to record migration flags and destinations, hot region IDs, hotness scores, and other metadata, we evaluate its memory overhead across the three systems. As reported in Table III, *GRACE* incurs additional memory consumption of up to 5.9%, 4.1%, and 5.2% on PPCSR, Terrace, and VCSR, respectively. These overheads are modest relative to the overall memory footprint, indicating that *GRACE* is practical even for large-scale graphs or memory-constrained environments.

Overhead of `mremap`: As *GRACE* relies on the `mremap` syscall, we evaluate the potential overhead introduced by frequent remapping. To address this concern, we perform a stress test that simulates the worst case where all blocks are classified as "cold". Although such a case does not occur in practice, it provides an upper bound on the remapping overhead. Table IV shows that on `LiveJournal` and `orkut`, the syscall requires only up to 57 ms (for 1792 calls) and 178 ms (for 8128 calls), respectively. These time costs account for less than 0.7% of the total update time, indicating that even under extreme remapping conditions, the overhead of frequent `mremap` calls is marginal and even negligible.

E. Ablation Analysis

In this subsection, we conduct an ablation study to assess the impacts of the two strategies in *GRACE*. In our test, PPCSR-reservation integrates only the property-guided reservation strategy, while PPCSR-*GRACE* applies the full *GRACE* implementation. Our goal is to compare the overall updating efficiency of these two configurations.

Figure 13 shows the updating improvements achieved by PPCSR-*GRACE* and PPCSR-reservation, compared to the original PPCSR. PPCSR-*GRACE* consistently outperforms PPCSR-reservation with speedups ranging from 1.22x to 1.43x, while PPCSR-reservation accelerates updates by 1.15x to 1.27x. These results indicate the effectiveness of the cousin-aware rebalancing, which significantly contributes to overall performance gains by reducing redundant rebalancing movements. When combined with the property-guided reservation, it can further enhance update efficiency. Datasets

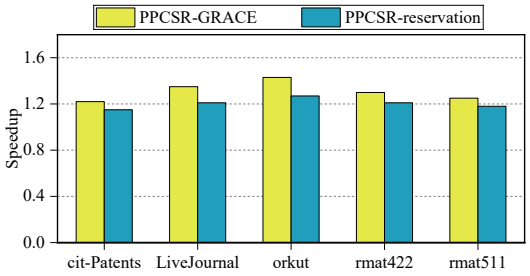
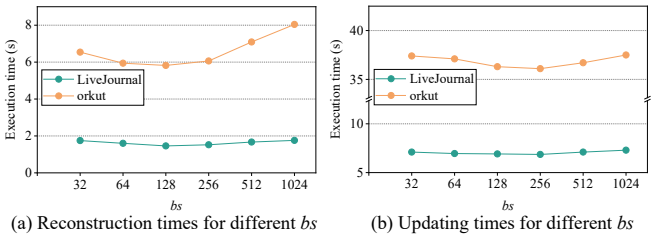


Fig. 13: Ablation analysis of the property-guided reservation and cousin-aware rebalancing strategies of *GRACE* on PPCSR.



(a) Reconstruction times for different *bs* (b) Updating times for different *bs*

Fig. 14: Effects of different bs values on PPCSR-*GRACE*.

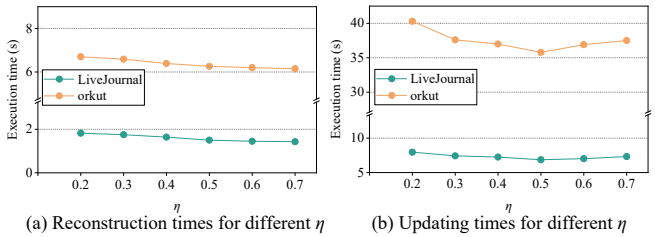
such as `orkut` and `LiveJournal` exhibit more pronounced benefits from the cousin-aware rebalancing strategy due to their distinct power-law degree distributions (Table II). This characteristic allows the strategy to confine more ping-pong effects by leveraging cousin scopes for low-impact vertices. In contrast, the improvement on `rmat422` is relatively modest, as its vertex degree distribution is more uniform.

F. Parameter Effect of bs

We examine the effects of bs on reconstruction and update efficiency by varying bs from 32 to 1024 on `LiveJournal` and `orkut`. The tests are conducted on PPCSR-*GRACE* with reconstruction and update performance shown in Figures 14a and 14b. In Figure 14a, reconstruction time first decreases and then increases as bs grows. This trend can be attributed to two competing factors. At lower bs values (32 to 128), increasing bs reduces remapping frequency, leading to performance gains. However, these gains gradually plateau due to the increasing dominance of the underlying kernel operating costs. With a very high bs value (e.g., 1024), the overheads of hot region-wide copying and traversal outweigh the benefits of fewer remapping operations, degrading reconstruction performance. As illustrated in Figure 14b, the fastest update time occurs at $bs = 256$ for both datasets, consistent with the reconstruction performance trend observed in Figure 14a. However, the differences between results are relatively modest (ranging from 6.86s to 7.31s for `LiveJournal` and 36.1s to 37.5s for `orkut`). This is because bs mainly affects reconstruction and data layout, and varying the bs value plays a relatively smaller role in determining the overall updating time.

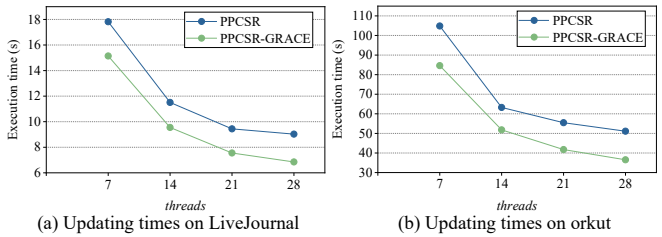
G. Parameter Effect of η

We explore the impact of η on the reconstruction and update performance of PPCSR-*GRACE* by changing its value from 0.2 to 0.7 on `LiveJournal` and `orkut` datasets. The results shown in Figure 15, reveal the following findings:



(a) Reconstruction times for different η (b) Updating times for different η

Fig. 15: Effects of different η on PPCSR-*GRACE*.



(a) Updating times on LiveJournal (b) Updating times on orkut

Fig. 16: Scaling-up evaluation on PPCSR-*GRACE*.

TABLE V: Total counts (in million) of rebalancing operations under different η.

Graphs\η	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**
LiveJournal	1.39	1.35	1.09	1.03	1.07	1.15
orkut	7.45	7.41	7.24	7.22	7.28	7.37

1) Reconstruction time decreases with higher η (Figure 15a). For `LiveJournal`, it reduces from 1.82s to 1.43s, while for `orkut`, from 6.70s to 6.15s. This is because larger η values identify fewer hot regions, reducing fine-grained copying and traversals. As a result, the benefits of page remapping and localized traversals become more pronounced.

2) Overall updating time varies significantly as η changes, with optimal performance achieved at $\eta = 0.5$ for both datasets. From Figure 15b, higher η values reserve excessive space for only a small portion of the array, resulting in memory under-utilization. Conversely, low η values hinder the benefits of prioritizing hot regions. Additionally, these result in fewer low-impact vertices being classified, which reduces the effectiveness of minimizing rebalancing movements.

Furthermore, we calculate the total rebalancing operations for different η, as shown in Table V. On `LiveJournal`, the count ranges from 1.03 million to 1.39 million, while on `orkut`, it ranges from 7.22 million to 7.45 million. The lowest counts occur at $\eta = 0.5$, aligning with the update time shown in Figure 15b, which confirms that reduced rebalancing operations drive performance improvements.

H. Scaling-up Experiments

We vary the number of threads from 7 to 28 and present the overall updating times for `LiveJournal` and `orkut` in Figure 16. As shown in this figure, across all tested thread configurations, *GRACE* consistently improves updating performance. Specifically, `LiveJournal` achieves an improvement from 1.18x to 1.33x, while `orkut` improves from 1.24x to 1.40x, with the highest speedups for both datasets observed at 28 threads. This is because increasing the number of threads accelerates the insertion process, making reconstruction overhead a more significant factor, and therefore the benefits of *GRACE* become more pronounced.

VI. Related Work

In recent years, multiple PMA-based dynamic graph processing systems have emerged, each introducing strategies to enhance performance. PCSR [12], the sequential version of PPCSR, is the first system to combine CSR with PMA, thereby improving its updating efficiency. RewiredPMA [21] identifies excessive movement overhead during data adjustments and leverages RUMA's page remapping technique [36] to reduce data movement by half. Terrace [13] adopts a hybrid graph representation of arrays, PMAs, and B-trees to store neighbor lists according to the vertex degree. Teseo [17] uses a trie tree variant to index PMA segments and a hash table to index each vertex's neighbor list. SPMA [37] collects the heads of leaf segments in a separate array and sorts them in specific orders to improve search performance. VCSR [14] builds a vertex-centric PMA tree that adaptively balances PMA segments based on the current vertex degree, thus enhancing updating efficiency. Finally, GPMA [16] extends PMA-based CSR to GPUs with specialized designs for efficient parallel updates.

Some other data structures are also employed in dynamic graph processing systems. STINGER [38] uses a blocked adjacency list for efficient updates but suffers from limited query performance due to its non-contiguous storage format. GraphTinker [39] improves STINGER's query efficiency with a new hashing scheme. cuSTINGER [40] and Hornet [41] are GPU extensions of STINGER. Graphone [42] also uses a blocked adjacency list, but buffers updates in an edge list. When the storage threshold is reached, the edges in the edge list are applied to the adjacency list to provide a snapshot for computing. Aspen [43] represents graphs by using tree-of-trees structures, where vertices are stored in a purely-functional tree and the edges of each vertex are stored in a separate C-tree.

VII. Conclusion and Future Work

This work identifies three key overheads in PMA-based CSR dynamic graph processing systems: global copying, full-array traversal, and massive rebalancing movement, which limit update performance. To address them, we propose *GRACE*, a plugin system that accelerates reconstruction and improves overall update efficiency through property-guided reservation and cousin-aware rebalancing strategies. Experimental evaluations on three representative systems demonstrate the effectiveness of *GRACE*. Future work includes: (1) extending our strategies to other array-based systems, such as those based on adjacency arrays [40], [41] and blocked hash tables [44], and (2) exploring concurrency control mechanisms to coordinate updates with computations, further improving system responsiveness and scalability.

Acknowledgment

This work is supported by National Key Research and Development Program of China (No. 2023YFB4502300) and NSFC-RGC under Grant No.62461160333.

AI-Generated Content Acknowledgment

We did not use any AI tools in this work.

References

[1] M. Alkhamees, S. Alsaleem, M. Al-Qurishi, M. Al-Rubaian, and A. Hussain, "User Trustworthiness in Online Social Networks: A Systematic Review," *Applied Soft Computing*, vol. 103, no. 1, pp. 107 159–107 182, 2021.

[2] Z. Tian, L. Jia, H. Dong, F. Su, and Z. Zhang, "Analysis of Urban Road Traffic Network based on Complex Network," *Procedia Engineering*, vol. 137, no. 1, pp. 537–546, 2016.

[3] C. Ye, Y. Li, B. He, Z. Li, and J. Sun, "GPU-Accelerated Graph Label Propagation for Real-Time Fraud Detection," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 2348–2356.

[4] A. D. Kent, L. M. Liebrock, and J. C. Neil, "Authentication Graphs: Analyzing User Behavior within an Enterprise Network," *Computers & Security*, vol. 48, no. 1, pp. 150–166, 2015.

[5] J. Brailovskaia and J. Margraf, "The Relationship between Active and Passive Facebook Use, Facebook Flow, Depression Symptoms and Facebook Addiction: A Three-month Investigation," *Journal of Affective Disorders Reports*, vol. 10, no. 1, pp. 100 374–100 379, 2022.

[6] X. Qiu, W. Cen, Z. Qian, Y. Peng, Y. Zhang, X. Lin, and J. Zhou, "Real-time Constrained Cycle Detection in Large Dynamic Graphs," *Proceedings of the VLDB Endowment*, vol. 11, no. 12, pp. 1876–1888, 2018.

[7] M. Besta, M. Fischer, V. Kalavri, M. Kapralov, and T. Hoefler, "Practice of Streaming Processing of Dynamic Graphs: Concepts, Models, and Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 6, pp. 1860–1876, 2023.

[8] X. Zhu, G. Feng, M. Serafini, X. Ma, J. Yu, L. Xie, A. Aboulnaga, and W. Chen, "LiveGraph: A Transactional Graph Storage System with Purely Sequential Adjacency List Scans," *Proceedings of the VLDB Endowment*, vol. 13, no. 7, pp. 1020–1034, 2020.

[9] S. Karamati, J. Young, and R. Vuduc, "An Energy-Efficient Single-Source Shortest Path Algorithm," in *Proceedings of 2018 IEEE International Parallel and Distributed Processing Symposium*, 2018, pp. 1080–1089.

[10] X. Shi, Z. Zheng, Y. Zhou, H. Jin, L. He, B. Liu, and Q. Hua, "Graph Processing on GPUs: A Survey," *ACM Computing Surveys*, vol. 50, no. 6, pp. 1–35, 2018.

[11] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen, "Kineograph: Taking the Pulse of a Fast-Changing and Connected World," in *Proceedings of the 7th ACM European Conference on Computer Systems*, 2012, pp. 85–98.

[12] B. Wheatman and H. Xu, "Packed Compressed Sparse Row: A Dynamic Graph Representation," in *Proceedings of 2018 IEEE High Performance Extreme Computing Conference*, 2018, pp. 1–7.

[13] P. Pandey, B. Wheatman, H. Xu, and A. Buluc, "Terrace: A Hierarchical Graph Container for Skewed Dynamic Graphs," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 1372–1385.

[14] A. Al Raqibul Islam, D. Dai, and D. Cheng, "VCSR: Mutable CSR Graph Format Using Vertex-Centric Packed Memory Array," in *Proceedings of 2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing*, 2022, pp. 71–80.

[15] B. Wheatman and H. Xu, "A Parallel Packed Memory Array to Store Dynamic Graphs," in *Proceedings of the 2021 Symposium on Algorithm Engineering and Experiments*, 2021, pp. 31–45.

[16] M. Sha, Y. Li, B. He, and K. Tan, "Accelerating Dynamic Graph Analytics on GPUs," *Proceedings of the VLDB Endowment*, vol. 11, no. 1, pp. 107–120, 2017.

[17] D. D. Leo and P. Boncz, "Teseo and the Analysis of Structural Dynamic Graphs," *Proceedings of the VLDB Endowment*, vol. 14, no. 6, pp. 1053–1066, 2021.

[18] "SNAP Dataset," 2014, http://snap.stanford.edu/data.

[19] J. Berry, M. Oster, C. A. Phillips, S. Plimpton, and T. M. Shead, "Maintaining Connected Components for Infinite Graph Streams," in *Proceedings of the 2nd International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications*, 2013, pp. 95–102.

[20] F. Zhang, L. Zou, and Y. Yu, "LPMA: An Efficient Data Structure for Dynamic Graph on GPUs," in *Proceedings of 2021 22nd International Conference on Web Information Systems Engineering*, 2021, pp. 469–484.

[21] D. De Leo and P. Boncz, "Packed Memory Arrays - Rewired," in *Proceedings of 2019 IEEE 35th International Conference on Data Engineering (ICDE)*, 2019, pp. 830–841.

[22] Q. Wang, L. Zheng, Y. Huang, P. Yao, C. Gui, X. Liao, H. Jin, W. Jiang, and F. Mao, "GraSU: A Fast Graph Update Library for FPGA-based Dynamic Graph Processing," in *Proceedings of the 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2021, pp. 149–159.

[23] H. Gao, X. Liao, Z. Shao, K. Li, J. Chen, and H. Jin, "A Survey on Dynamic Graph Processing on GPUs: Concepts, Terminologies and Systems," *Frontiers of Computer Science*, vol. 18, no. 4, pp. 184 106–184 131, 2024.

[24] J. Su, C. Hao, S. Sun, H. Zhang, S. Gao, J. Jiang, Y. Chen, C. Zhang, B. He, and M. Guo, "Revisiting the Design of In-Memory Dynamic Graph Storage," in *Proceedings of the 2025 International Conference on Management of Data*, 2025, pp. 1–27.

[25] M. A. Bender and H. Hu, "An Adaptive Packed Memory Array," *ACM Transactions on Database Systems*, vol. 32, no. 4, pp. 1–43, 2007.

[26] H. Gao, S. Zhang, X. Liao, and H. Jin, "GRACE: Alleviating Reconstruction Cost in Dynamic Graph Processing Systems - Technical Report," 2025.

[27] 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*, 2005, pp. 177–187.

[28] "Windows Memory Management," 2025, https://learn.microsoft.com/zh-cn/windows/win32/memory/memory-management.

[29] N. Nazaraliyev, E. Sadredini, and N. Abu-Ghazaleh, "DREAM: Device-Driven Efficient Access to Virtual Memory," in *Proceedings of the 39th ACM International Conference on Supercomputing*, 2025, pp. 1190–1205.

[30] M. E. Gerber, L. Gerhorst, I. Mudraje, K. Vogelgesang, T. Herfet, and P. Wägemann, "VNV-Heap: An Ownership-Based Virtually Non-Volatile Heap for Embedded Systems," in *Proceedings of the 26th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, 2025, pp. 97–108.

[31] "PPCSR," 2022, https://github.com/domargan/parallel-packed-csr.

[32] F. Khorasani, R. Gupta, and L. N. Bhuyan, "Scalable SIMD-Efficient Graph Processing on GPUs," in *Proceedings of the 24th International Conference on Parallel Architectures and Compilation Techniques*, 2015, pp. 39–50.

[33] L. Hu, L. Zou, and M. T. Özsu, "GAMMA: A Graph Pattern Mining Framework for Large Graphs on GPU," in *Proceedings of 2023 IEEE 39th International Conference on Data Engineering (ICDE)*, 2023, pp. 273–286.

[34] X. Chen, "GraphCage: Cache Aware Graph Processing on GPUs," *arXiv preprint*, 2019.

[35] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. MIT press, 2009.

[36] F. M. Schuhknecht, J. Dittrich, and A. Sharma, "RUMA has it: Rewired User-Space Memory Access is Possible!" *Proceedings of the VLDB Endowment*, vol. 9, no. 10, pp. 768–779, 2016.

[37] B. Wheatman, R. Burns, A. Buluç, and H. Xu, "Optimizing Search Layouts in Packed Memory Arrays," in *Proceedings of the 2023 Symposium on Algorithm Engineering and Experiments (ALENEX)*, 2023, pp. 148–161.

[38] D. Ediger, R. McColl, J. Riedy, and D. A. Bader, "STINGER: High Performance Data Structure for Streaming Graphs," in *Proceedings of 2012 IEEE Conference on High Performance Extreme Computing*, 2012, pp. 1–5.

[39] W. Jaiyeoba and K. Skadron, "Graphtinker: A High Performance Data Structure for Dynamic Graph Processing," in *Proceedings of 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2019, pp. 1030–1041.

[40] O. Green and D. A. Bader, "CuSTINGER: Supporting Dynamic Graph Algorithms for GPUs," in *Proceedings of 2016 IEEE High Performance Extreme Computing Conference*, 2016, pp. 1–6.

[41] F. Busato, O. Green, N. Bombieri, and D. A. Bader, "Hornet: An efficient data structure for dynamic sparse graphs and matrices on gpus,"

in *Proceedings of 2018 IEEE High Performance Extreme Computing Conference*, 2018, pp. 1–7.

[42] P. Kumar and H. H. Huang, "Graphone: A Data Store for Real-time Analytics on Evolving Graphs," *ACM Transactions on Storage*, vol. 15, no. 4, pp. 1–40, 2020.

[43] 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*, 2019, pp. 918–934.

[44] M. A. Awad, S. Ashkiani, S. D. Porumbescu, and J. D. Owens, "Dynamic Graphs on the GPU," in *Proceedings of 2020 IEEE International Parallel and Distributed Processing Symposium*, 2020, pp. 739–748.