DCSR: A Fast Data Structure with Leaf-Oriented Locks for Streaming Graph Processing

Yue Shen
State Key Lab of Processors, Institute of Computing Technology, CAS
University of Chinese Academy of Sciences
Beijing, China
shenyue24b@ict.ac.cn

Jie Zhang
State Key Lab of Processors, Institute of Computing Technology, CAS
Beijing, China
zhangjie2023@ict.ac.cn

Huawei Cao
State Key Lab of Processors, Institute of Computing Technology, CAS
Beijing, China
caohuawei@ict.ac.cn

Yuan Zhang
State Key Lab of Processors, Institute of Computing Technology, CAS
Beijing, China
zhangyuan-ams@ict.ac.cn

Xuejun An
State Key Lab of Processors, Institute of Computing Technology, CAS
Beijing, China
axj@ict.ac.cn

Abstract

Streaming graph processing has been widely employed for real-time graph analytics across various domains. Its performance depends on both the graph update phase, which ingests edge updates, and the graph computation phase, which executes graph algorithms. The Packed Memory Array (PMA) maintains sorted elements in a gapped array, supporting efficient graph update and computation. However, existing PMA-based data structures either suffer from severe conflicts for the lock-based single-edge update or require an additional sorting phase for the lock-free batch update.

This paper presents a novel leaf-oriented dynamic data structure based on the PMA, called DCSR, designed to enable fast graph updates with varying batch sizes. The primary innovation of DCSR is a leaf-oriented parallel update strategy that comprises two phases of lock-based update and decoupled rebalancing to fully exploit parallelism with minimal conflicts. We implemented DCSR and compared its update performance with five state-of-the-art dynamic data structures. The experimental results show that DCSR enables fast graph updates, achieving around ten million updates per second across a range of batch sizes. For edge insertions, it achieves average speedups of 5.84×, 12.98×, 25.63×, 27.45×, and 11.14× over PPCSR, CPMA, PaC-tree, Terrace, and LSGraph, respectively.

Keywords

Data structure, Streaming graph processing, Packed memory array, Parallel graph update

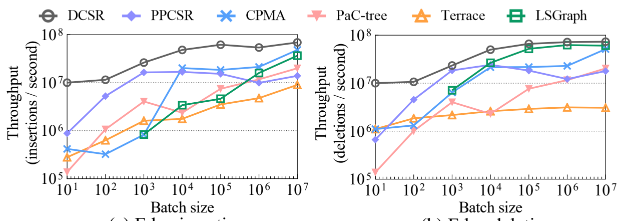
1 Introduction

Graphs are ubiquitous nowadays for representing vast amounts of information about individuals and their relationships. In real-world scenarios, graphs continuously evolve through an infinite stream of updates, known as streaming graphs. There has been growing interest in analyzing streaming graphs in real time to meet increasing social demands, including financial fraud detection [17], social network analysis [25], and content recommendation [40].

Streaming graph processing aims to provide up-to-date query results for the most recent graph data [12, 16, 20, 22, 24, 28, 31–33, 36, 45–47, 51]. In general, it comprises two phases: graph update and graph computation, both of which should be rapid to ensure a timely response. This places higher demands on the underlying data structure: it must not only support efficient graph computation but also accommodate high-throughput graph updates. Data structures commonly used in static graph processing, such as Compressed Sparse Row (CSR) [43], are unsuitable for streaming graph processing due to their poor graph update performance caused by excessive data movement overhead. Consequently, considerable effort has been devoted to developing dynamic data structures, including CPMA [47], LSGraph [36], PaC-tree [18], Terrace [35], PPCSR [50], GastCoco [31], DHB [44], Bubble [16], and so on.

Among the existing work, the Packed Memory Array (PMA) [6] is an ingenious dynamic data structure organized as an implicit complete binary tree. It maintains ordered edges within a single array for efficient graph computation and deliberately keeps gaps in each leaf (also called a segment) to facilitate graph updates. A classical method [6] for parallel graph updates via PMA assigns each thread an edge update and ensures data consistency using exclusive vertex locks (referred to as the lock-based single-edge update method in this paper). Although it exploits edge-level parallelism, the ***severe conflicts*** caused by dense PMA reads and writes not only incur costly atomic overheads but also reduce update parallelism. A recent study [47] employs a lock-free batch update strategy to process edge updates across multiple phases. As no locks are required, this approach avoids the significant overhead associated with atomic operations. However, it employs an ***additional sorting phase*** to make the lock-free batch update effective, which accounts for more than 80% of the update time for large batches. Moreover, the overhead associated with thread scheduling and synchronization in individual phases diminishes its benefits, especially for small batches with low workloads.

To address the above issues, we leverage the characteristics of the PMA and identify three opportunities to reduce access conflicts without requiring a sorting phase. *First*, since the elements within each leaf are *left-packed*, we can regulate the location search algorithm to minimize conflicts between the locating and updating steps. *Second*, because only the target leaf needs to be accessed during the updating step, we can use *fine-grained leaf*



(a) Edge insertions (b) Edge deletions

Figure 1: Update throughput of DCSR (this work) and the other five dynamic data structures (i.e., PPCSR [50], CPMA [47], PaC-tree [18], Terrace [35], and LSGraph [36]) with varying batch sizes on the Web-uk-2005 graph [30].

locks instead of coarse-grained vertex locks to maintain data consistency, thus enhancing update parallelism. *Third*, because the rebalancing step is more regular compared to the locating and updating steps, we decouple it from these two steps and perform it in a separate phase, further reducing access conflicts.

Based on the above three opportunities, this paper proposes ***DCSR***, a fast PMA-based data structure with leaf locks for streaming graph processing. We introduce the ***leaf-oriented parallel update*** strategy, which comprises two phases of *lock-based update* and *decoupled rebalancing*. Both phases are parallelized internally and use leaf locks to maintain data consistency. The lock-based update phase performs the locating and updating steps. For the locating step, we first search the first elements in the leaves without using any locks, and then we exclusively search the region within the target leaf. Moreover, for the updating step, we design two different algorithms for insertions and deletions. In the decoupled rebalancing phase, we utilize temp edge vectors to process the leaves that need to be rebalanced in parallel and employ a flag array to eliminate redundant rebalancing operations.

We thoroughly evaluate DCSR, comparing it with five cutting-edge data structures of various types, including two PMA-based structures, PPCSR [50] and CPMA [47], a tree-based structure, PaC-tree [18], a hybrid-based structure, Terrace [35], and an Adjacency List (AL)-based structure, LSGraph [36]. The experimental results demonstrate that DCSR achieves average speedups of 5.84×, 12.98×, 25.63×, 27.45×, and 11.14× over PPCSR, CPMA, PaC-tree, Terrace, and LSGraph, respectively. As shown in Figure 1, DCSR achieves *the best update performance* among the six data structures on the Web-uk-2005 graph in all cases, supporting an update throughput exceeding ten million updates per second across *a range of batch sizes*. Our ***contributions*** can be summarized as follows.

- We summarize the limitations of existing PMA-based solutions and identify three opportunities to reduce access conflicts without an additional sorting phase.
- We propose, *for the first time*, a leaf-oriented parallel update strategy that includes two phases of lock-based update and decoupled rebalancing to efficiently update the PMA.
- We implement the approaches and develop DCSR based on the BYO framework [48], which provides a true apples-to-apples comparison of graph containers.
- We compare DCSR with five state-of-the-art dynamic data structures, and the results demonstrate the efficacy of DCSR.

2 Related Work

A ***streaming graph*** is a dynamic graph whose structure continuously changes through an infinite stream of edge updates (i.e., insertions and deletions). Streaming graph processing primarily consists of two phases: ***graph update*** and ***graph computation***, with the goal of processing the *most recent* graph data to obtain the latest query results [12, 22, 24, 32, 33, 36, 45, 46, 51]. Unlike streaming graph frameworks, graph databases typically provide transactional capabilities that comply with ACID properties and specialize in handling simple graph queries or updates on the rich data (i.e., properties or labels) attached to edges or vertices [7, 8]. Dynamic graph structures for graph databases [4, 15, 21, 23, 41, 52] maintain edge timestamps to track modifications and employ multi-version concurrency control to enable the concurrent execution of graph updates alongside queries. These works were not originally designed to manage streaming graphs and therefore *fall outside* the primary focus of this paper.

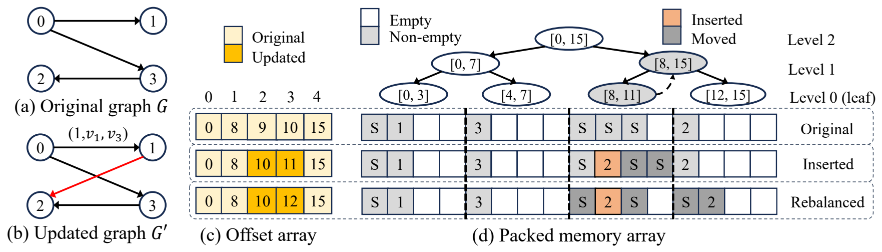
For streaming graph processing, the high-performance requirements of the graph update and graph computation pose significant challenges for data structure design. On the one hand, edge insertions and deletions demand locating target addresses with low complexity while minimizing the overhead of data movements. On the other hand, the performance of iterative graph algorithms relies on good data locality to reduce irregular memory accesses. Many related works propose dynamic data structures to address these challenges, which can generally be divided into the following four categories.

PMA-based structures [26, 27, 39, 47, 49, 50, 53] primarily consist of a PMA and an offset array. The PMA is used to store all the sorted edges in a graph, while the offset array is used to store the starting indices of each vertex's neighbors in the PMA. Notable examples include PPCSR [50] and CPMA [47]. Benefiting from the good data locality and ordered edge maintenance provided by the PMA, the graph computation performance of PPCSR and CPMA is only slightly lower than that of CSR [48]. Additionally, the empty entries in the PMA facilitate data movements for the graph update.

AL-based structures [20, 22, 29, 31, 36, 44] maintain a dynamic edge array for *each vertex*, while vertex-related data, such as degrees, are stored in a separate vertex array. The edges in a dynamic edge array are usually unordered, and locating a specific edge of a vertex with degree d has a time complexity of $O(d)$. This results in performance bottlenecks for hub vertices. Recent approaches [22, 44] use hash tables for hub vertices, reducing the complexity of a hub vertex to $O(1)$. However, hash tables incur significant memory overhead, which limits their scalability to large graphs.

Tree-based structures [18, 19] utilize a vertex tree to manage vertices, and the neighbors of a vertex are usually stored in an edge block to improve data locality. Due to the vertex tree, the insertion or deletion of a *vertex* is more efficient than in PMA-based and AL-based structures. However, the extensive use of pointers not only leads to numerous cache misses when accessing vertices or edges but also incurs additional memory overhead.

Hybrid-based structures [2, 35] utilize multiple fundamental data structures to leverage their respective strengths. Terrace [35] is a representative example featuring a three-level structure. At the first level, a few edges of each vertex are stored contiguously in the vertex array. The second level employs a PMA to store some edges of vertices continuously. For each high-degree vertex, its remaining edges are maintained in an individual B-tree,



	Level 0	Level 1	Level 2
Node size (# of entries)	4	8	16
Lower density bound ρ_l	0.1250	0.1875	0.2500
Upper density bound τ_l	1.0000	0.8750	0.7500
Min # of elements	1	2	5
Max # of elements	3	6	11

(e) Node information

Figure 2: PMA insertion example based on CSR.

constituting the third level. Terrace delivers better graph computation performance and comparable graph update performance compared to a tree-based solution [19]. However, it suffers from poor scalability to large batches due to inefficient search and numerous data movements in the PMA [36].

3 Preliminaries and Motivations

This section first presents the fundamental theory of the PMA, followed by an introduction to the PMA based on CSR for dynamic graph storage. With these preliminaries, we discuss the limitations of existing PMA-based structures, which motivate us to design our proposed DCSR with the leaf-oriented parallel update strategy.

3.1 Packed Memory Array

A ***PMA*** [6] (shown in Figure 2(d)) is a dynamic array that stores n ordered ***elements*** with ***gaps*** between them to facilitate the insertion or deletion of elements. The entries in the dynamic array that store elements are called ***non-empty*** entries, whereas the entries that indicate gaps are called ***empty*** entries. The total number of entries N in the PMA is chosen to be a power of 2 and satisfies $N > n$. The PMA is logically divided into $N/\log N$ segments, each of size $\log N$. The elements stored in each segment are ***left-packed***. Treating segments as ***leaves***, the PMA can be managed by an implicit ***complete binary tree*** with height $h = \log(N/\log N)$. A ***node*** n_l at ***level*** l covers $2^l \log N$ entries. Let m denote the number of *elements* stored in the entries, and the ***density*** of the node n_l is defined as $\text{Density}(n_l) = m/(2^l \log N)$.

The density of the node n_l is constrained by the ***lower and upper density bounds***, ρ_l and τ_l, such that $\rho_l < \text{Density}(n_l) < \tau_l$. As the level l grows, the lower density bound ρ_l becomes larger, whereas the upper density bound τ_l becomes smaller. Formally, the lower and upper density bounds at a level l are defined as $\rho_l = \rho_h - (\rho_h - \rho_0)(h-l)/h$ and $\tau_l = \tau_h + (\tau_0 - \tau_h)(h-l)/h$, , where ρ_0, ρ_h, τ_0, and τ_h are predefined. As the PMA becomes too dense (or too sparse) to satisfy the upper density bound τ_h (or the lower density bound ρ_h) of the root, a PMA ***expansion*** (or ***contraction***) is required. Moreover, to ensure that the density of the new PMA remains within the specified density bounds when we ***double*** (or ***halve***) the PMA, the requirement is that $2\rho_h < \tau_h$.

In the example shown in Figure 2(d), the PMA contains sixteen entries, which are logically divided into four segments, each consisting of four entries. The corresponding complete binary tree has four leaves and a height of two. The node information of the complete binary tree is shown in Figure 2(e). The lower density bounds of a leaf and the root, denoted as ρ_0 and ρ_2, are set to 0.125 and 0.25, respectively, while their upper density bounds, τ_0 and τ_2, are set to 1.00 and 0.75, respectively. The minimum or maximum number of elements in a node can be calculated as described above. For example, the maximum number of elements the root can have is 11, which is less than $16 \times 0.75 = 12$.

3.2 PMA Based on CSR

Since the PMA exhibits good data locality and maintains elements in order, it is well-suited for graph computation. Furthermore, the gaps in each leaf facilitate data insertion. Therefore, the PMA is a popular data structure used in existing streaming graph processing systems and has demonstrated excellent performance [27, 39, 47, 49, 50, 53].

In general, the PMA used to store a dynamic graph is based on the CSR format. Figure 2 presents an example. The original graph G contains four vertices and three edges (shown in Figure 2(a)). The ***PMA based on CSR*** stores the original graph G in the ***offset array*** with five entries (shown in Figure 2(c)) and the PMA with sixteen entries (shown in Figure 2(d)).

Each vertex has a specific element stored in the PMA, called a ***sentinel***, marking the *beginning* of the vertex's neighbors. A sentinel is depicted as "S" in the PMA (shown in Figure 2(d)), and the *index* of it is stored in the corresponding entry in the offset array. Specifically, the last entry of the offset array stores the index of the last entry of the PMA. For the example shown in Figures 2(c) and 2(d), since v_0's sentinel is stored in the first entry of the original PMA, the index stored in the first entry of the offset array is 0. In the implementation, the *actual number* stored in an empty entry is `MAX_NUM`[1], while that of a sentinel is $\texttt{MAX\_NUM} - 1$. The edges of a vertex may be distributed across multiple leaves. As shown in Figure 2(d), the first edge (v_0, v_1) of the vertex v_0 is stored in leaf 0, while its second edge (v_0, v_3) is stored in leaf 1.

Updating the PMA based on CSR. An ***edge update*** from vertex v_a to vertex v_b is represented as (f, v_a, v_b), where $f = 1$ indicates an insertion while $f = 0$ indicates a deletion. As the edge update is digested into the PMA based on CSR, the PMA is adjusted, and the offset array should be updated to align with the moved sentinels. The PMA is adjusted in *three steps*: 1) a locating step, 2) an updating step, and 3) a rebalancing step. Inserting or deleting an edge (v_a, v_b) in the PMA requires $O(\log n + (\log^2 n)/B)$ amortized memory transfers in the external memory model [1], where B is the cache-line size [6, 47, 49, 50].

In the ***locating step***, the entry where the edge update (f, v_a, v_b) is to be inserted into or deleted from the PMA is determined using a search algorithm (e.g., binary search). We refer to the entry as the ***located entry***, and denote the index of it as $loc(f, v_a, v_b)$. The *located entry* of the edge update (f, v_a, v_b) is searched from the beginning of vertex v_a's neighbors to the beginning of vertex

[1]For example, `MAX_NUM` is the maximum value of an n-bit unsigned integer if the elements in the PMA are stored using n-bit unsigned integers.

v_{a+1}'s neighbors. Let P_o be ***the pointer to the offset array***. We have $loc(f, v_a, v_b) \in (P_o[a], P_o[a+1]]$.

In the ***updating step***, the data adjustment is confined to a single leaf. We refer to the leaf as the ***target leaf*** and denote the leaf index of the target leaf as $leaf_{\text{tgt}}(f, v_a, v_b)$. The updating steps of an insertion or a deletion are different. *For an insertion*, if the located entry is empty, we can directly insert the element v_b into it. Otherwise, we should rearrange the other entries around the located entry to make room for v_b (e.g., by shifting elements larger than v_b one position to the right) and then insert the element v_b into the newly vacated entry. *For a deletion*, once the located entry of the element v_b is identified, all elements in the target leaf larger than v_b are shifted one position to the left to remove v_b.

After the updating step, a ***rebalancing step*** is triggered if the target leaf is full (or empty). We locate the first ***ancestor*** of the target leaf, finding from bottom to top, whose density falls within the specified lower and upper density bounds. Then, the elements within the ancestor's region are rebalanced and evenly distributed among its inclusive leaves. There is an extreme case in which the root of the PMA tree exceeds its upper density bound τ_h (or lower density bound ρ_h), requiring a PMA expansion (or contraction).

Insertion example. As illustrated in Figure 2(b), the edge from vertex v_1 to v_2 is inserted into the original graph G, forming the updated graph G'. The three steps for updating the PMA based on CSR with the edge update $(1, v_1, v_2)$ are as follows:

❶ Locating step. $P_o[1]$ and $P_o[2]$ in the offset array are read, and the *located entry* of the edge update $(1, v_1, v_2)$ is identified within the region $(8, 9]$. Using a search algorithm, we can find that the first element in the region $(8, 9]$ that is *not smaller* than the destination vertex ID 2 is "S", the index of which is 9.

❷ Updating step. The two sentinels of vertices v_2 and v_3, which are larger than the destination vertex ID 2 in the leaf 2 (i.e., the *target leaf*), are shifted one position to the right to make room for the new element 2 (shown in Figure 2(d)). Since the two sentinels of vertices v_2 and v_3 are moved, $P_o[2]$ and $P_o[3]$ in the offset array are updated to 10 and 11, respectively (shown in Figure 2(c)).

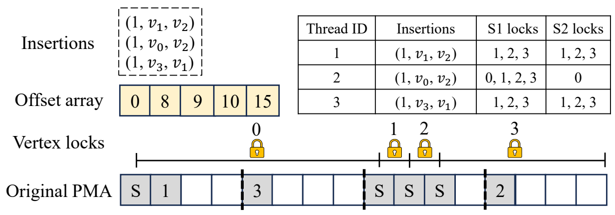
❸ Rebalancing step. After the updating step, the third leaf becomes full. Checking from bottom to top, we find that the number of elements in its *father*'s region $[8, 15]$ is 5, and the density of the father is $5/(2^1 \log 16) = 0.625$ which satisfies the density bounds. Hence, the five elements in the father's region $[8, 15]$ are rebalanced (shown in Figure 2(d)). To align with the sentinel of vertex v_3, $P_o[3]$ is updated to 12 (shown in Figure 2(c)).

3.3 Limitations of Existing Solutions

To enhance update performance, existing PMA-based solutions perform edge updates in parallel. Generally, there are two approaches to update edges for the PMA in parallel: (1) lock-based single-edge update (LSU for short) and (2) lock-free batch update (BU for short). We first discuss LSU, followed by BU. Since deletions are considered symmetric to insertions in existing solutions, we omit the discussion of deletions.

3.3.1 ***Lock-based single-edge update.***

LSU assigns a single edge insertion $(1, v_a, v_b)$ per thread, and threads carry out edge insertions based on the three steps described in Section 3.2 in parallel in a single loop. Since the complicated read and write operations each thread applies for, LSU produces a great amount of conflicts. First of all, the thread needs to search the entries within the region $(P_o[a], P_o[a+1]]$ in the locating step. Subsequently, the thread probably moves elements in the target leaf in the updating step. Finally, if the leaf is full, the elements in some leaves should be redistributed in the rebalancing step. Hence, locks are required to maintain the data consistency. However, it is not trivial to design a high-performance lock-based parallel update strategy with severe conflicts.



Thread ID	Insertions	S1 locks	S2 locks
1	$(1, v_1, v_2)$	1, 2, 3	1, 2, 3
2	$(1, v_0, v_2)$	0, 1, 2, 3	0
3	$(1, v_3, v_1)$	1, 2, 3	1, 2, 3

Figure 3: Lock-based single-edge update for the PPCSR.

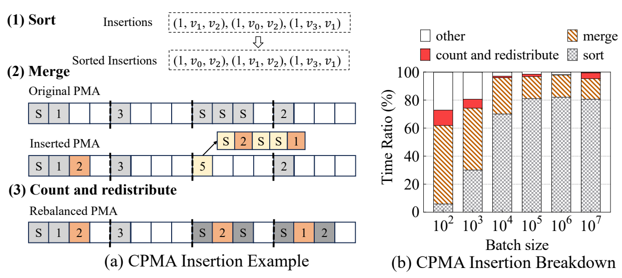


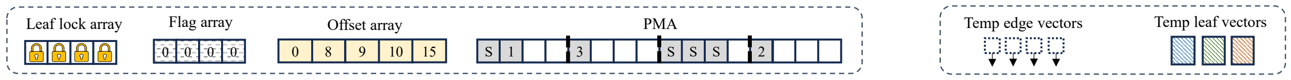
Figure 4: Lock-free batch update for the CPMA.

PPCSR [50] achieves LSU by utilizing exclusive vertex locks. Figure 3 illustrates an example of the lock-based single-edge update using PPCSR. For clarity, the description of the rebalancing step is omitted. As shown in this figure, the three insertions, $(1, v_1, v_2)$, $(1, v_0, v_2)$, and $(1, v_3, v_1)$, are assigned to threads 1, 2, and 3, respectively. Accessing the same leaf is *exclusive*. During the *locating step* (illustrated as "S1"), each thread attempts to acquire the necessary vertex locks and searches for the located entry. In the *updating step* (illustrated as "S2"), each thread attempts to acquire the vertex locks that control the region of the target leaf. For instance, for the insertion $(1, v_0, v_2)$, thread 2 must acquire the four vertex locks of vertices v_0, v_1, v_2, and v_3 in the locating step to secure leaves 0, 1, and 2, which are involved in the region $(0, 8]$. After the locating step, thread 2 identifies the target leaf of the $(1, v_1, v_2)$ is leaf 0. In the updating step, since leaf 0 can be locked by the vertex lock of vertex v_0, thread 2 tries to grab the vertex lock.

Discussion. The disadvantage of LSU is that it causes severe conflicts reducing update parallelism. For example, as shown in Figure 3, all three threads need to acquire vertex locks of vertices v_1, v_2, and v_3 during the locating steps, and both threads 1 and 3 acquire vertex locks of vertices v_1, v_2, and v_3 during the updating steps. As a result, the three threads almost insert the three edges sequentially. Considering the overhead of atomic operations, the performance of LSU may be even worse than that of serial execution.

3.3.2 ***Lock-free batch update.***

The key to BU is to eliminate thread conflicts. With the help of the complete binary tree, a natural approach is to insert edges with a leaf-centric strategy. We can group the edge insertions whose target leaves are the



(a) Primary structures (b) Auxiliary structures

Figure 5: DCSR format.

same and then attempt to insert these grouped edges into their corresponding leaves with leaf-level parallelism. For the remaining edges that cannot be inserted due to the density limits, we retry and merge them into the regions of their ancestors.

CPMA [47] is a recent work employing BU. Figure 4(a) provides an example of BU using CPMA. The process consists of three core phases. *First*, the three insertions, $(1, v_1, v_2)$, $(1, v_0, v_2)$, and $(1, v_3, v_1)$, are *sorted* by their source vertices and then by their destination vertices. After sorting, the edge insertions located in the same leaf are adjacent within the sorted batch. *Second*, the target leaf of each insertion is identified, and the insertions belonging to the same leaf are *merged* with its original edges. The insertion $(1, v_0, v_2)$ is merged into leaf 0, while the insertions $(1, v_1, v_2)$ and $(1, v_3, v_1)$ are merged into leaf 2. Since leaf 2 already contains three elements, all five merged elements are temporarily stored out-of-place due to the upper density limit, and the number of the elements is recorded in leaf 2. *Third*, the densities of leaf 2's ancestors are *counted* from bottom to top, revealing that its father is the first node that satisfies the upper density bound. Subsequently, the six elements in leaves 2 and 3 are *redistributed*.

Discussion. The advantage of BU is that it does not require locks for parallel updates, thereby avoiding the significant overhead associated with atomic operations. However, it incurs two notable costs. *First*, because BU divides the PMA update procedure into several individual phases, it introduces substantial thread scheduling and synchronization overhead for each phase. This overhead is particularly non-negligible for small batches with low workloads. *Second*, the additional sorting phase incurs $O(k \log k)$ work for sorting a batch of k edges [47], leading to considerable overhead. Figure 4(b) illustrates the time ratio of edge insertions with CPMA across varying batch sizes. As shown in this figure, the proportion of the sorting phase increases with batch size. For instance, the sorting time accounts for over 60% of the total time with a batch size of 10^4, rising to as much as 80% for larger batch sizes.

3.3.3 ***Insight***. We find that LSU and BU come to two extremes. In LSU, a single thread handles all three steps of an edge update, leading to significant thread contention due to the complex PMA update principles. In contrast, BU achieves parallel updates without locks by dividing the graph update process into multiple phases, which introduces additional sorting overhead. This observation motivates us to design a leaf-oriented parallel update strategy that reduces access conflicts without requiring a sorting phase. The characteristics of PMA provide three opportunities to reduce access conflicts:

- The elements in each leaf are packed to the left, which allows us to ***regulate*** the search algorithm to reduce conflicts between locating and updating steps.
- During the updating step, only the target leaf is accessed. This encourages the use of ***leaf locks*** rather than vertex locks to enhance parallelism while maintaining data consistency.
- Compared to the rebalancing step, the locating and updating steps are much more regular. This motivates us to ***decouple*** the rebalancing step from the other two steps, further reducing conflicts.

4 DCSR Design

For the graph update, we update the insertions ΔG_+ and deletions ΔG_- in two separate procedures, as in previous works [18, 31, 35, 36, 47, 50]. Overall, the leaf-oriented parallel update strategy of DCSR comprises two phases: 1) ***lock-based update*** and 2) ***decoupled rebalancing***. The first phase performs the locating and updating steps within a single loop, while the second phase carries out the rebalancing step in another loop. The two phases are executed *in sequence*, but each phase is *parallelized internally* with leaf locks to ensure data consistency. We first introduce the DCSR format in Section 4.1, followed by a detailed presentation of the lock-based update in Section 4.2 and the decoupled rebalancing in Section 4.3. Finally, we analyze the update complexity of DCSR in Section 4.4.

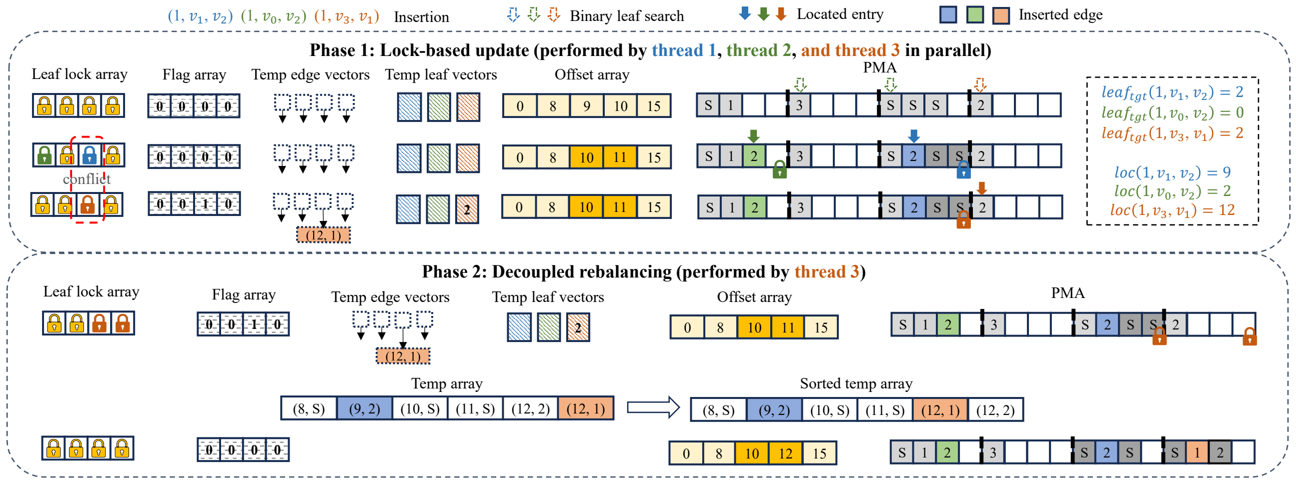
4.1 DCSR Format

Figure 5 illustrates the DCSR format, which is divided into primary and auxiliary structures. The ***primary structures*** are maintained throughout the streaming graph processing. The ***auxiliary structures*** are allocated at the start of the graph update phase and released once this phase has finished.

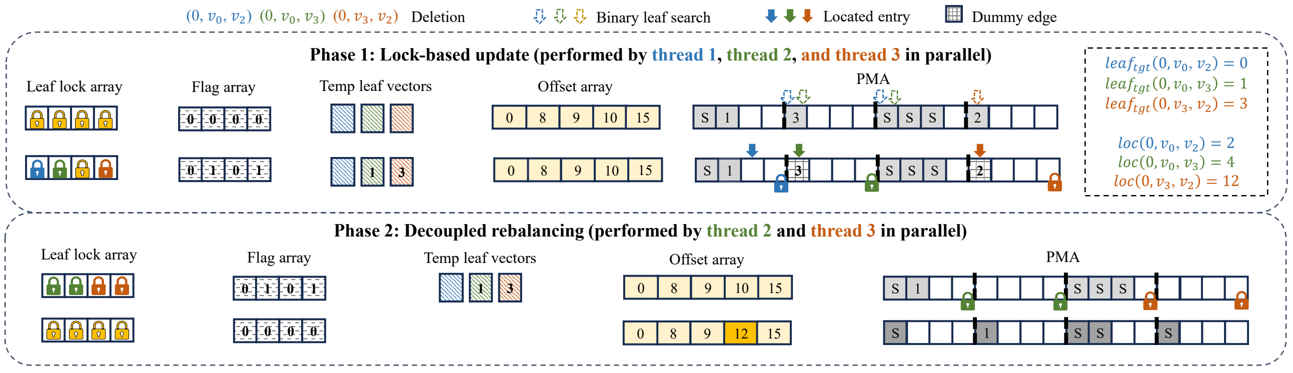
As shown in Figure 5(a), the primary structures store the original graph G depicted in Figure 2(a). Compared to the PMA based on CSR shown in Figure 2, a leaf lock array and a flag array are additionally employed. The ***leaf lock array*** maintains a lock for each leaf, with each lock occupying 4 bytes of memory. The ***flag array*** stores a ***rebalancing flag*** for each leaf, indicating whether that leaf requires rebalancing.

Auxiliary structures are depicted in Figure 5(b), which includes ***temp edge vectors*** and ***temp leaf vectors***. For insertions, both the temp edge vectors and temp leaf vectors are used. At the beginning of the insertion procedure, temp edge vectors are created for *each leaf*, and temp leaf vectors are created for *each thread*. When a thread attempts to insert an edge (v_a, v_b) into a *full leaf*, the index of the located entry and the destination vertex of the edge are stored as a pair (loc, v_b) in the leaf's temp edge vector. Then, the thread records the index of the full leaf in its temp leaf vector. For deletions, only the temp leaf vectors are used. The temp leaf vectors are created at the beginning of the deletion procedure for *each thread*. If a leaf becomes *empty* after a deletion performed by a thread, the index of the leaf is recorded in the thread's temp leaf vector. The details are provided in Section 4.2.

Notation. We denote the offset array, the PMA, the leaf lock array, and the flag array as P_o, P_e, P_l, and P_f, respectively. We denote the temp edge vectors as P_t, and the temp edge vector of the leaf i is represented as $P_t[i]$. The temp leaf vector of a thread is denoted as Q. The primary structures of DCSR are denoted collectively by $\mathscr{D}$.



(a) DCSR insertion example



(b) DCSR deletion example

Figure 6: Examples of the leaf-oriented parallel update strategy of DCSR.

Algorithm 1: Regulated location search

```
    Data: the edge $(v_a, v_b)$, the primary structures $\mathscr{D}$.
    Result: the pair $(leaf_{\text{tgt}}, loc)$.
1  def RegulatedLocSearch($v_a, v_b, P_o, P_e, P_l$):
2      $leaf_{\text{begin}}$ = FindLeaf($P_o[a]$);
3      $leaf_{\text{end}}$ = FindLeaf($P_o[a+1]$);
4      $leaf_{\text{tgt}}$ = BinaryLeaf($leaf_{begin}, leaf_{end}, v_b, P_e$);
5      $P_l[leaf_{\text{tgt}}]$.lock(); // lock the target leaf
6      if $leaf_{begin} == leaf_{tgt}$ then
7          $left = P_o[a] + 1$;
8      else
9          $left = leaf_{\text{tgt}} \times \log N$;
10     end
11     if $leaf_{end} == leaf_{tgt}$ then
12         $right = P_o[a+1]$;
13     else
14         $right = (leaf_{\text{tgt}} + 1) \times \log N$;
15     end
16     $loc$ = BinaryEntry($left, right, v_b, P_e$);
17 return $(leaf_{\text{tgt}}, loc)$;
```

4.2 Lock-Based Update

We novelly propose a regulated location search method for the locating step that leverages the left-packed property of the PMA to minimize conflicts. We have designed different parallel update algorithms for insertions and deletions. To ensure data consistency while enhancing parallelism, we use leaf locks instead of vertex locks during this phase. We first introduce the regulated location search method, followed by a detailed description of the lock-based update phase for insertions and deletions.

4.2.1 *Regulated location search*.

In existing PMA-based solutions [27, 47, 50, 53], the located entry of edge update (f, v_a, v_b) is identified using a binary search algorithm. However, there are three major drawbacks to using the binary search algorithm in a straightforward manner. *First*, since the region $(P_o[a], P_o[a+1]]$ may contain *empty entries*, the binary search algorithm must be able to handle this special case, which increases the algorithm complexity. *Second*, the accesses to the region are *irregular*, making conflicts between the locating and updating steps unpredictable. *Third*, these conflicts are addressed using *coarse-grained vertex locks*, which limit update parallelism (as described in Section 3.3.1). To address these issues, we propose the function RegulatedLocSearch, as outlined in Algorithm 1.

Algorithm 1. First, the beginning and ending leaf indices of the region $(P_o[a], P_o[a+1]]$, $leaf_{\text{begin}}$ and $leaf_{\text{end}}$, are calculated by the function FindLeaf (lines 2 and 3, respectively), which returns the *leaf index* corresponding to a given entry index. Then, the target leaf is identified using the function BinaryLeaf (line 4). This function performs a binary search on the *first elements* of the leaves whose indices lie within the region $(leaf_{\text{begin}}, leaf_{\text{end}}]$, and tries to find the *first leaf* whose first element is *larger* than v_b. Let $leaf_{\text{larger}}$ be the leaf index of the leaf. If the leaf is found,

Algorithm 2: Lock-based update phase for insertions

```
Data: the edge insertions $\Delta G_+$, the primary structures $\mathscr{D}$.
Result: the DCSR format updated by $\Delta G_+$ after phase 1.
parallel for each $(1, v_a, v_b) \in \Delta G_+$:
    $(leaf_{tgt}, loc)$ = RegulatedLocSearch($v_a$, $v_b$, $\mathscr{D}$);
    def Insert($leaf_{tgt}$, $loc$, $v_a$, $v_b$, $\mathscr{D}$):
        if CheckElement($loc$, $v_b$, $P_e$, $P_t[leaf_{tgt}]$) == 1 then
            return;
        end
        if CheckLeafFull($leaf_{tgt}$, $P_e$) == 1 then
            $P_t[leaf_{tgt}]$.PushInEdgeVector($loc$, $v_b$);
            if $P_f[leaf_{tgt}]$ == 0 then
                $Q$.PushInLeafVector($leaf_{tgt}$);
                $P_f[leaf_{tgt}]$ = 1;
            end
        else
            AddElement($loc$, $v_b$, $P_e$, $P_o$);
        end
        AtomicAddDegree($v_a$, 1);
        return;
    $P_l[leaf_{tgt}]$.unlock(); // unlock the target leaf
end
```

$leaf_{tgt} = leaf_{larger} - 1$. Otherwise, $leaf_{tgt} = leaf_{end}$. Subsequently, the *target leaf* is locked by the assigned thread with the leaf lock array (line 5). To deal with the boundary situation, the left and the right indices of the search region *within* the target leaf are calculated (lines 6–10 and lines 11–15, respectively), ensuring that $[left, right) \subset (P_o[a], P_o[a+1]] \cap [leaf_{tgt} \times \log N, (leaf_{tgt}+1) \times \log N)$. Afterward, the function BinaryEntry performs a binary search on the region $[left, right)$ to find the *first entry* whose element is *not less* than v_b (i.e., the located entry) and returns its index (line 16). Finally, the pair ($leaf_{tgt}$, loc) is returned (line 17).

Example. Take the insertion $(1, v_0, v_2)$ in Figure 6(a) as an example, which is assigned to thread 2 (depicted with the color green). Since its located entry is within the region $(0, 8]$, we have $leaf_{begin} = 0$ and $leaf_{end} = 2$. The first elements of leaf 1 and leaf 2 are accessed. Since the first element of leaf 1 is 3, which is larger than 1, we have $leaf_{larger} = 1$ and $leaf_{tgt} = 0$. Then, thread 2 successfully acquires the lock of leaf 0. Since $leaf_{begin} = leaf_{tgt}$ and $leaf_{end} \neq leaf_{tgt}$, we have $left = 1$ and $right = 4$. After a binary search in the region $[1, 4)$, the located entry is identified, and we have $loc = 2$. We omit the description of other edge updates due to space limitations; their target leaves and located entries are shown in Figure 6.

4.2.2 *Lock-based update phase for insertions*.

Algorithm 2. Each insertion $(1, v_a, v_b)$ is ingested into DCSR by a thread (line 1). First, the function RegulatedLocSearch (shown in Algorithm 1) is used to identify the pair ($leaf_{tgt}$, loc) (line 2). Note that, afterwards, the lock of the target leaf is already grabbed by the thread. Then, we use the function insert to carry out the updating step. First, we determine whether the edge (v_a, v_b) is already stored in the PMA or in the temp edge vector of the target leaf (i.e., $P_t[leaf_{tgt}]$) using the function CheckElement (line 4). If it exists, we return immediately. Otherwise, we insert it. There are two cases. ❶ If the target leaf is full, we use the function PushInEdgeVector to push the pair (loc, v_b) into the temp edge vector $P_t[leaf_{tgt}]$ (line 8). If the rebalancing flag of the target leaf

Algorithm 3: Lock-based update phase for deletions

```
Data: edge deletions $\Delta G_-$, the primary structures $\mathscr{D}$.
Result: The DCSR format updated by $\Delta G_-$ after phase 1.
parallel for each $(0, v_a, v_b) \in \Delta G_-$:
    $(leaf_{tgt}, loc)$ = RegulatedLocSearch($v_a$, $v_b$, $\mathscr{D}$);
    def Delete($leaf_{tgt}$, $loc$, $v_a$, $v_b$, $\mathscr{D}$):
        if CheckElement($loc$, $v_b$, $P_e$, $null$)== 0 then
            return;
        end
        if CheckLeafWillEmpty($leaf_{tgt}$, $P_e$) == 1 then
            SetDummyEdge($loc$, $P_e$);
            $Q$.PushInLeafVector($leaf_{tgt}$);
            $P_f[leaf_{tgt}]$ = 1;
        else
            RemoveElement($loc$, $P_e$, $P_o$);
        end
        AtomicAddDegree($v_a$, −1);
        return;
    $P_l[leaf_{tgt}]$.unlock(); // unlock the target leaf
end
```

has not been marked (line 9), we push its leaf index into the temp leaf vector using the function PushInLeafVector (line 10) and set the rebalancing flag $P_f[leaf_{tgt}]$ to 1 (line 11). ❷ Otherwise, if the target leaf still contains empty entries, we use the function AddElement to insert the element v_b into the entry $P_e[loc]$, and to update the offset array to align with the moved sentinels (line 14). After the insertion, we atomically increment the degree of the source vertex v_a by one using the function AtomicAddDegree (line 16). At the end of the algorithm, the thread unlocks the target leaf (line 18).

Example. Take the insertions $(1, v_1, v_2)$ and $(1, v_3, v_1)$ in Figure 6(a) as two examples. The located entry of the insertion $(1, v_1, v_2)$ is $P_e[9]$. To make room for the edge (v_1, v_2), the two sentinels of vertices v_2 and v_3 in leaf 2 are shifted one position to the right. Then the element v_2 is successfully inserted in the entry $P_e[9]$. To align with the moved sentinels, $P_o[2]$ and $P_o[3]$ in the offset array are updated to 10 and 11, respectively. However, for the insertion $(1, v_3, v_1)$ whose target leaf 2 is full, the pair $(12, 1)$ has to be pushed into the temp edge vector $P_t[2]$. In addition, the index of the target leaf is pushed into the temp leaf vector of thread 3, and the rebalancing flag $P_f[2]$ is set to 1.

4.2.3 *Lock-based update phase for deletions*.

We employ the ***dummy edge*** for deletions, which is marked as *non-existent* but can still be *read*. All the dummy edges are removed at the beginning of the decoupling rebalancing phase.

Algorithm 3. Similar to Algorithm 2, each deletion $(0, v_a, v_b)$ is assigned to a thread (line 1), and the pair ($leaf_{tgt}$, loc) is calculated with the function RegulatedLocSearch first of all (line 2). Then, we update the deletion using the function Delete. First, we check whether the element v_b is stored in the located entry $P_e[loc]$ using the function CheckElement (line 4). If it does not exist, we return immediately. Otherwise, we remove it. We use the function CheckLeafWillEmpty to identify if the target leaf will become empty (i.e., only its first entry is non-empty) (line 7). There are two cases. ❶ If the element v_b is the only element in the target leaf, we do not truly remove it. We convert it into a dummy edge by using the function SetDummyEdge (line 8). Then, we push the index of the empty leaf (i.e., $leaf_{tgt}$) into the temp leaf vector of the thread using the function `PushInLeafVector` (line 9) and set the rebalancing flag $P_f[leaf_{\text{tgt}}]$ to 1 (line 10). ❷ Otherwise, if the target leaf will not be empty, we remove the element v_b from the entry $P_e[loc]$ and update the offset array with the function `RemoveElement` (line 12). After the deletion, the degree of the source vertex v_a is atomically decremented by one using the function `AtomicAddDegree` (line 14). At the end of the algorithm, the target leaf is unlocked (line 16).

Example. Take the deletions $(0, v_0, v_2)$ and $(0, v_0, v_3)$ in Figure 6(b) as an example. For the deletion $(0, v_0, v_2)$, its located entry is $P_e[2]$. Since the element v_2 is not stored in the entry $P_e[2]$, thread 1 returns immediately. While for the deletion $(0, v_0, v_3)$, its target leaf is leaf 1. Since element 3 is the only element in leaf 1, the located entry $P_e[4]$ is set to a dummy edge. Then, the index of leaf 1 is pushed into the temp leaf vector of thread 2, and the rebalancing flag of leaf 1 is set to 1.

4.2.4 ***Discussion****. Note that during the function* `BinaryLeaf` *(Algorithm 1, line 4), since only the first entries in the leaves are accessed, no locks are required.* We discuss the situations in which the located entry is the first entry in a leaf. For insertions, there are two cases. ❶ If the leaf containing the located entry is not the target leaf, the insertion will not be performed in that leaf (e.g., the insertion $(1, v_3, v_1)$ shown in Figure 6(a)). ❷ Otherwise, the element to be inserted must be *equal* to the element stored in the located entry. In both of the two cases, the first entry will not be modified. For deletions, removing the element in the first entry does not affect the output of the function `BinaryLeaf`. There are also two cases. ❶ If the first element is the only element in the target leaf, it can still be read during the function `BinaryLeaf` since it is changed to a dummy edge. ❷ Otherwise, the elements larger than the first element are moved *left* by one position. Since the second element must be larger than the first element, the output of the function `BinaryLeaf` is correct.

The leaf indices stored in the temp leaf vectors are not duplicated. For insertions, since the rebalancing flag of a full leaf is checked before a thread pushes the index of the full leaf into its temp leaf vector (Algorithm 2, lines 9–11), the index of the full leaf is added to a temp leaf vector only once. For deletions, since the empty leaf, whose first element has already been converted to a dummy edge, causes the algorithm to return immediately (Algorithm 3, lines 4–6), the index of the empty leaf is added to a temp leaf vector only once.

4.3 Decoupled Rebalancing

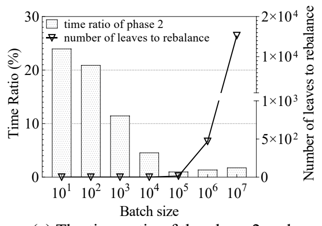
The decoupled rebalancing phase performs the rebalancing step in an *individual* loop using leaf locks to ensure data consistency. In this phase, *each temp leaf vector* is processed by *a thread.* Since the number of temp leaf vectors is equal to that of threads, parallelism is fully exploited. The leaves within the same temp leaf vector are rebalanced sequentially by a thread. After a thread completes the rebalancing of a leaf, the rebalancing flag of the leaf is set to 0. *To avoid redundant rebalancing of a leaf, the thread checks the leaf's rebalancing flag before rebalancing it.* If the rebalancing flag is 0, indicating that the leaf has already been rebalanced by another thread, we immediately return without performing any rebalancing.

There are some differences between the decoupled rebalancing phases of insertions and deletions. For insertions, the edges exceeding the capacity of a leaf, which are stored as (loc, v_b) pairs in the leaf's temp edge vector, need to be redistributed with the elements stored in the PMA together. For deletions, the dummy edges stored in the PMA are removed at the beginning of the phase. Next, we introduce two examples for them, respectively.

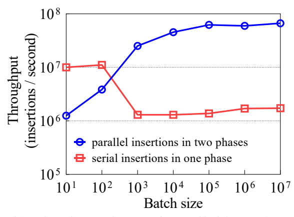
4.3.1 ***Rebalancing example of insertions****.* Figure 6(a) provides an example of the decoupled rebalancing phase of insertions. Since only the temp leaf vector of thread 3 is non-empty, thread 3 performs the rebalancing of leaf 2. First, thread 3 exclusively calculates the density of leaf 2's ancestors from bottom to top. *In addition to the corresponding elements in the PMA, the pairs in the corresponding temp edge vector should also be counted.* Thread 3 requires the locks of leaves 2 and 3, and calculates the density of leaf 2's father. The father of leaf 2 contains five elements in the PMA and a pair in leaf 2's temp edge vector, so its density is $(5 + 1)/8 = 0.75$. Since the density of the father satisfies the upper density bound (i.e., 0.875), the region $[8, 15]$ ought to be rebalanced. Then, we convert each element e stored in the region $[8, 15]$ of the PMA into a pair $(loc(e), e)$, where $loc(e)$ is the index of e. A ***temp array*** is used to store the five pairs converted from the PMA, along with the pair $(12, 1)$ stored in the leaf 2's temp edge vector. Next, the pairs stored in the temp array are sorted *first by $loc(e)$ and then by e*. Afterwards, the sorted elements in the temp array are evenly distributed into the region $[8, 15]$ of the PMA. The offset array is updated to align with the moved sentinels, and the rebalancing flag of leaf 2 is set to 0. At last, thread 3 releases the leaf locks of leaves 2 and 3. At the end of the phase, the temp edge vectors and temp leaf vectors are deallocated.

4.3.2 ***Rebalancing example of deletions****.* Figure 6(b) provides an example of the decoupled rebalancing phase of deletions. *First of all, the dummy edges in the PMA are removed.* Then, the temp leaf vectors of threads 2 and 3 are processed *in parallel.* Threads 2 and 3 exclusively calculate the density of the ancestors of leaves 1 and 3 from bottom to top and perform the rebalancing steps, respectively. Thread 2 successfully requires the locks of leaves 0 and 1 and calculates that the density of leaf 1's father is $2/8 = 0.25$, which satisfies its lower density bound (i.e., 0.1875). Hence, thread 2 evenly redistributes the two elements in the region $[0, 7]$ and sets the rebalancing flag of leaf 1 to 0. Then, thread 2 unlocks the leaf locks of leaves 0 and 1. Concurrently, thread 3 successfully requires the locks of leaves 2 and 3 and calculates that the density of leaf 3's father is $3/8 = 0.375$, which also satisfies its lower density bound. Thus, thread 3 evenly redistributes the three elements in the region $[8, 15]$ and sets the rebalancing flag of leaf 3 to 0. In addition, thread 3 updates the offset array to align with the moved sentinels. Afterwards, thread 3 unlocks the leaf locks of leaves 2 and 3. After all threads finish, the temp leaf vectors are deallocated.

4.3.3 ***Discussion****. Utilizing the temp leaf vectors to process each vector in parallel is more efficient than using the flag array to process each leaf in parallel.* The probability that many leaves in the PMA are either empty or full during the update procedure is low. Figure 7(a) shows the number of leaves that need to be rebalanced when inserting batches of varying sizes into the Web-uk-2005 graph [37], whose PMA contains 67,108,864 leaves. In this example, no leaves require rebalancing when the batch size is no more than 10^4, and for batches of size 10^7, the number of leaves that need rebalancing accounts for only 0.023% of the total leaves. Therefore, the flag array is always too sparse to be traversed efficiently. Figure 8 illustrates an example of the flag array and the temp leaf vectors. Compared to the temp leaf vectors, using the flag array to process each leaf in parallel results in numerous

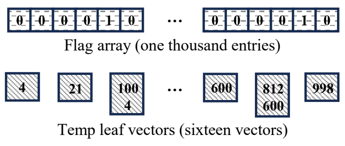


(a) The time ratio of the phase 2 and the number of leaves to rebalance



(b) The throughput of parallel insertions and serial insertions (without locks)

Figure 7: An insertion example of DCSR on the Web-uk-2005 graph [37] (with 67,108,864 leaves) with varying batch sizes.



Flag array (one thousand entries)
Temp leaf vectors (sixteen vectors)

Figure 8: An example of the flag array and the temp leaf vectors, in which only twenty leaves require rebalancing.

unnecessary accesses, thereby reducing update performance. For example, DCSR's performance decreases by 52.6% when processing batches of size 10^3 using the flag array on the Web-uk-2005 graph.

4.3.4 ***Optimization****. For small batch sizes (batch size $\leq 10^2$ in our implementation), we update the edges sequentially in a single phase instead of updating them in parallel in the two phases of lock-based update and decoupled rebalancing.* As discussed above, hardly any leaves require rebalancing for small batch sizes. Therefore, the decoupled rebalancing phase is almost *unnecessary* in these cases, but it introduces a *non-negligible* overhead to the two-phase parallel update procedure due to thread scheduling and synchronization. As shown in Figure 7(a), the ratio of the decoupled rebalancing phase execution time to the overall two-phase parallel update execution time exceeds 20% when the batch size is no greater than 10^2. Therefore, for small batches, it is better not to separate the rebalancing step from the locating and updating steps, and instead perform the three steps together in a single phase using one loop. Moreover, because the workload of a small batch is low, the overhead associated with leaf locks and atomic operations used in the two-phase parallel update procedure *outweighs* the benefits of parallelism. Figure 7(b) shows an example of insertion throughput on the Web-uk-2005 graph [37] using both parallel insertions in two phases and serial insertions in one phase (without locks). When the batch size is no more than 10^2, the average speedup of the one-phase serial insertions compared to the two-phase parallel insertions ranges from 2.89× to 8.00×. Therefore, to further improve the update performance, we update the edges in serial without locks in one phase for small batches with batch size $\leq 10^2$.

4.4 Complexity Analysis

We use the work-span model [13] and the external memory model [1] to analyze the bounds for the leaf-oriented parallel update strategy of DCSR. The ***work*** is the total running time of the algorithm on a single processor. The ***span*** is the longest sequence of dependent steps in the computation.

Theorem 1. *For a PMA containing n elements and cache-line size B, updating a batch ΔG of k edge updates to the DCSR incurs an amortized work of $O(k \log n + k(\log^2 n)/B)$ and a worst-case span of $O(\log^2 n)$.*

Proof. The lock-based update phase includes the locating and updating steps. For the locating step, each edge update performs a PMA search to find its target leaf and located entry, incurring $O(\log n)$ cache-line transfers per update and thus $O(k \log n)$ total work across the batch. For the updating step, an insertion or a deletion is ingested into its target leaf of size $O(\log n)$, which costs $O((\log n)/B)$ cache-line transfers per update, and therefore $O((k \log n)/B)$ total work for the batch. In the decoupled rebalancing phase, we leverage the flag array to eliminate unnecessary rebalancing operations. As in a standard PMA, DCSR performs $O(\log^2 n)$ amortized element moves per update, which corresponds to $O((\log^2 n)/B)$ cache-line transfers, giving $O((k \log^2 n)/B)$ amortized work over the batch. The critical path is dominated by the decoupled rebalancing phase, which may traverse $O(\log n)$ levels. At each level, the redistribution operations use parallel primitives with polylogarithmic depth, leading to a worst-case span of $O(\log^2 n)$. Combining the two phases, the total work is $O(k \log n + (k \log^2 n)/B)$ and the worst-case span is $O(\log^2 n)$. □

5 Evaluation

We first introduce the experimental setup, then assess the performance of the graph update and graph computation. Finally, we evaluate scalability and memory footprint.

5.1 Experimental Setup

Competitors. We compare DCSR with five state-of-the-art dynamic data structures: PPCSR [50], CPMA [47], PaC-tree [18], Terrace [35], and LSGraph [36]. PPCSR is a popular PMA-based data structure with the lock-based single-edge update (see Section 3.3.1). CPMA is a *compressed* PMA-based data structure proposed recently via the lock-free batch update (see Section 3.3.2). PaC-tree is a representative tree-based data structure that supports parallel and *compressed* purely functional collections by storing sorted elements in edge blocks to improve data locality. Terrace is a hybrid-based data structure that employs a local vertex array, a PMA, and B-trees to store the edges of vertices with various degrees. LSGraph is an AL-based data structure that uses a hierarchical indexed graph representation to achieve efficient data search and movement.

Environment. All experiments were conducted on a machine equipped with four Intel Xeon Gold 6148 processors, providing a total of 80 cores (with two-way hyper-threading) running at 2.7 GHz and 256 GB of memory. The operating system on this machine is CentOS 7.9. Our proposed DCSR is implemented in C++ and developed using the BYO framework [48], which maintains the core framework and all other infrastructure consistently across each graph container. LSGraph was compiled with clang++ (version 10.0.1) and employed the *OpenCilk* (version 1.0) parallel programming platform [38], following the configuration in the paper [36]. DCSR, PPCSR, CPMA, PaC-tree, and Terrace were integrated into the *BYO* framework and compiled with the GNU GCC compiler (version 11.2.1) using *OpenMP* [14] for parallelism.

Datasets. Table 1 lists five real-world graphs used in the evaluation, along with their abbreviated names and sizes. The five graphs are undirected and unweighted to meet the requirements

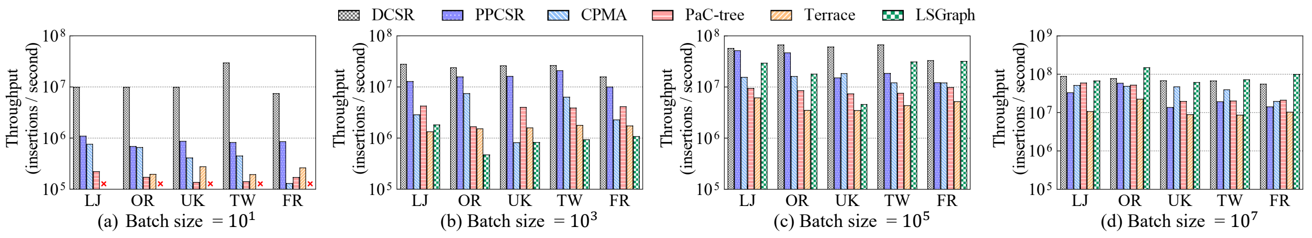


Figure 9: Throughput for edge insertions of DCSR, PPCSR, CPMA, PaC-tree, Terrace, and LSGraph on the graphs listed in Table 1 with batch sizes of 10, 10^3, 10^5 and 10^7.

Table 1: Statistics of the real-world graph datasets.

Graph	Abbr.	Vertices	Edges
LiveJournal [30]	LJ	3,997,962	34,681,189
Orkut [30]	OR	3,072,441	117,185,083
Web-uk-2005 [37]	UK	39,459,925	936,364,282
Twitter [30]	TW	41,652,230	1,468,365,182
Friendster [30]	FR	65,608,366	1,806,067,135

of all of the evaluated data structures. The edge updates are generated using an RMAT [11] generator with parameters $a = 0.5$, $b = 0.1$, and $c = 0.1$, as in the prior work [36, 47, 48].

Workloads. To assess the performance of graph computation, we select four representative graph algorithms: *Breadth-First Search (BFS)* [5], *PageRank* [10], *betweenness centrality* [9], and *triangle counting* [3]. Note that we use *static* graph algorithms to process each streaming graph *from scratch*, rather than employing *incremental graph algorithms* that build upon historical results, as done in previous work on dynamic data structures [16, 31, 36, 47]. How to carry out the graph computation along with the graph update efficiently is a challenging issue and is tackled by the other work on incremental computation [24, 33, 34, 45, 46], *falling outside* the main focus of this paper.

5.2 Graph Update Performance

We convert the execution time of the graph update to *throughput* by dividing the batch size by the execution time. The results reported are the averages of five trials, with each trial inserting a different set of edges. Since the source code of LSGraph [36] does not support graph updates when the batch size is smaller than the number of threads, we do not report its performance results for these scenarios and exclude them from the calculation of speedups.

Figure 9 illustrates the throughput for edge insertions of DCSR, PPCSR, CPMA, PaC-tree, Terrace, and LSGraph with batch sizes of 10, 10^3, 10^5, and 10^7. DCSR achieves speedups of 1.10×-36.45× over PPCSR, 1.44×-66.87× over CPMA, 1.48×-210.01× over PaC-tree, 3.43×-153.32× over Terrace, and 0.52×-50.76× over LSGraph. *As shown in Figure 9, as the batch size is no more than* 10^5*, DCSR outperforms all five competitors for all cases.* For edge deletions, DCSR achieves average speedups of 4.97× (up to 22.06×) over PPCSR, 4.72× (up to 17.96×) over CPMA, 26.51× (up to 208.33×) over PaC-tree, 18.79× (up to 51.02×) over Terrace, and 2.19× (up to 5.93×) over LSGraph. Figure 1 presents the insertion and deletion throughput on the Web-uk-2005 graph [37] with batch sizes ranging from 10 to 10^7. For edge insertions shown in Figure 1(a), DCSR achieves an average speedup of 4.64× over PPCSR, 14.41× over CPMA, 18.05× over PaC-tree, 19.14× over Terrace, and 12.84× over LSGraph. For edge deletions shown in Figure 1(b), DCSR achieves an average speedup of 4.73× over PPCSR, 4.27× over CPMA, 18.01× over PaC-tree, 15.39× over Terrace, and 1.90× over LSGraph.

For batch sizes of 10, DCSR achieves the largest speedups over PPCSR, CPMA, PaC-tree, and Terrace, respectively. There are three main reasons. *First*, compared with PaC-tree and Terrace, which store edges in various edge structures, DCSR maintains all the edges in a single PMA, achieving good locality for location search. *Second*, compared with CPMA using several complicated operations (i.e., merge, count, and redistribute) for edge updates, DCSR employs more simple and effective functions (e.g., `RegulatedLocSearch`, `CheckElement`, `CheckLeafFull`). *Third*, we optimize DCSR to update the edges in a small batch sequentially rather than in parallel as PPCSR (discussed in Section 4.3). DCSR avoids the overhead of leaf locks and atomic operations and thus achieves better update performance (shown in Figure 7(b)).

When the batch size exceeds 10^2, DCSR updates edges in parallel in two phases: lock-based update and decoupled rebalancing. As shown in Figures 9(b), 9(c), and 9(d), DCSR outperforms PPCSR and CPMA in all cases for batch sizes of 10^3, 10^5, and 10^7. This is due to the innovative design of the DCSR. *First*, because the function `BinaryLeaf` is executed without any locks during the regulated location search, the conflicts between the locating and updating steps are minimized. *Second*, DCSR performs the rebalancing steps in a separate phase using an independent loop, further reducing the conflicts among the locating, updating, and rebalancing steps. *Third*, DCSR employs leaf locks to maintain data consistency, avoiding the irregular and coarse-grained locking requirements associated with vertex locks in PPCSR, thereby enhancing update parallelism. By managing conflicts with leaf locks, DCSR achieves superior parallelism without requiring an additional sorting phase like CPMA.

As shown in Figure 9(d), for batch sizes of 10^7, the update throughput of DCSR is 0.52×, 0.93×, and 0.56× smaller than that of LSGraph on OR, TW, and FR, respectively. There are mainly four reasons. *First*, LSGraph is an AL-based data structure in which the edge structure for each vertex can be adjusted separately without conflicts. *Second*, the hierarchical indices used in LSGraph facilitate data search, and the HITree employed for each high-degree vertex reduces data movement. *Third*, OpenCilk [38] introduces additional optimizations and yields better performance compared with OpenMP [44]. *Fourth*, due to strict PMA update principles, large batches (e.g., batch size $\geq 10^6$) leads to numerous conflicts and data movements for lock-based methods. However, DCSR achieves average speedups of 28.09× and 4.43× over LSGraph for batch sizes of 10^3 and 10^5, respectively (shown in Figures 9(b) and 9(c)). This is because DCSR has better

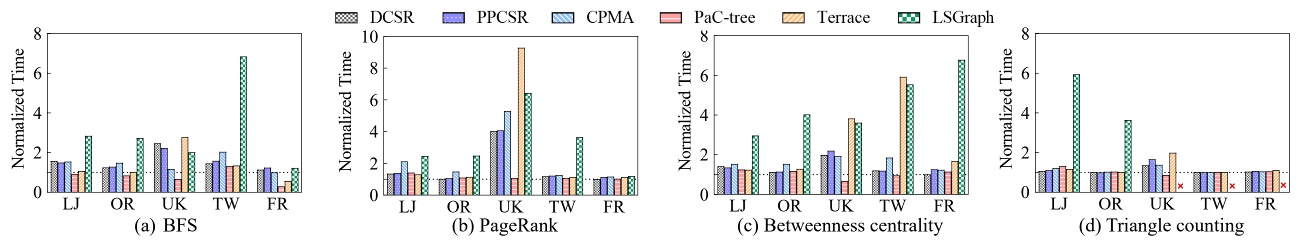


Figure 10: Time to run BFS, PageRank, betweenness centrality, and triangle counting of DCSR, PPCSR, CPMA, PaC-tree, Terrace, and LSGraph normalized to CSR on the graphs listed in Table 1.

data locality, which also results in its better graph computation performance (discussed in Section 5.3).

5.3 Graph Computation Performance

We evaluate the graph computation performance of CSR, DCSR, PPCSR, CPMA, PaC-tree, Terrace, and LSGraph on the five graphs listed in Table 1 for BFS, PageRank, betweenness centrality, and triangle counting. We report the running time of DCSR and five competitors normalized to CSR. Since the triangle counting algorithm of LSGraph did not finish in 5 hours on the UK, TW, and FR graphs, we do not report its results for these graphs.

Figure 10 illustrates the normalized running time of the six data structures. DCSR, PPCSR, CPMA, PaC-tree, Terrace, and LSGraph incur slowdowns of 1.00×-4.03×, 1.00×-4.08×, 1.00×-5.30×, 0.29×-1.42×, 0.57×-9.28×, and 1.19×-6.84×, compared to CSR. Moreover, DCSR achieves average speedups of 1.06× (up to 1.71×) over PPCSR, 1.32× (up to 4.87×) over CPMA, 0.83× (up to 1.42×) over PaC-tree, 1.37× (up to 4.93×) over Terrace, and 4.97× (up to 6.78×) over LSGraph. Although DCSR consumes more time to execute graph algorithms than PaC-tree, it achieves an average speedup of 25.63× over PaC-tree for the graph update. We find that PaC-tree is even more efficient than CSR in some cases. For example, it achieves an average speedup of 1.58× over CSR for BFS. The main reason is that PaC-tree uses compressed cache-friendly blocks to store edges, which possibly leads to better performance with an efficient parallel decoding algorithm [42]. Among the six structures, LSGraph presents the worst graph computation performance and cannot scale to large graphs for triangle counting. This is because it stores edges of a vertex in hierarchical structures, which reduces data locality and brings more cache misses.

Although DCSR, PPCSR, and CPMA are all PMA-based data structures, there are a few differences in their structures that lead to variations in graph computation performance. DCSR stores the leaf locks in an individual leaf lock array, whereas PPCSR stores the vertex locks in the offset array. Since graph computation does not require any locks, DCSR provides better data locality compared with PPCSR, hence enhancing computation performance. CPMA compresses the elements stored in the PMA, which introduces additional decompression overhead during graph computation and results in longer running time compared to DCSR.

5.4 Scalability

Figure 11 illustrates the scalability of DCSR and five competitors on the Web-uk-2005 graph with varying thread counts. *As shown in this figure, DCSR achieves the best update performance in all cases.* This is because the two-phase leaf-oriented parallel update

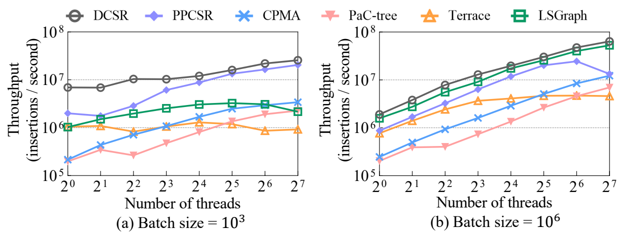


Figure 11: Scalability of edge insertions using multi-threading on the Web-uk-2005 graph with batch sizes of 10^3 and 10^6.

of DCSR is well parallelized. On the one hand, DCSR assigns each thread an edge update to process during the lock-based update phase, thereby fully utilizing multi-threading resources. On the other hand, because the number of the temp leaf vectors equals the number of threads, all threads can be utilized during the decoupled rebalancing phase.

Among the six data structures, only DCSR, CPMA, and PaC-tree scale well for both batch sizes of 10^3 and 10^7. As shown in Figure 11(a), when the batch size is 10^3, the insertion throughput of LSGraph decreases as the thread count reaches 64. The primary reason may be that LSGraph assigns edge updates of a vertex to a single thread and performs the graph update in vertex-level parallelism, rather than the edge-level parallelism employed by DCSR and PPCSR. When the batch size is 10^7, PPCSR's update performance decreases when using 128 threads. This issue primarily arises because PPCSR brings significant access conflicts with large batches, and the overhead associated with vertex locks and atomic operations outweighs the benefits of parallelism when the number of threads is excessive. Moreover, Terrace does not scale well for batch sizes of both 10^3 and 10^6 due to the data search and movement bottlenecks in its PMA structure.

5.5 Memory Footprint

Table 2 shows the memory usage of each data structure for *storing* the five graphs listed in Table 1. For clarity, we additionally evaluate the memory footprint of DCSR's auxiliary structures for insertions with a batch size of 10^7. As shown in this table, the memory consumption of the auxiliary structures accounts for 8.90%-17.67% of that of DCSR. Because the ratio of the number of leaves requiring rebalancing to the total number of leaves is extremely low (discussed in Section 4.3), the overriding auxiliary structures consuming memory are the temp edge vectors, which have a space complexity of $O(\log N)$. Although the temp edge vectors introduce some additional memory overhead, we use them only during the update procedure for insertions.

Table 2: Memory footprint (GB) of various data structures for storing the graphs in Table 1. D-A represents the auxiliary structures of DCSR.

Graph	DCSR	D-A	PPCSR	CPMA	PaC-tree	Terrace	LSGraph
LJ	1.13	0.19	1.15	0.20	0.31	1.51	0.66
OR	2.15	0.38	2.19	0.61	0.95	2.51	1.89
UK	17.19	1.53	17.47	2.19	6.59	23.20	13.12
TW	33.36	3.10	33.57	5.32	10.25	44.71	18.72
FR	35.42	3.22	36.65	13.21	15.17	51.73	25.85

Terrace's memory usage is 1.17×-1.46× higher than that of DCSR. This is because Terrace additionally uses B-trees to store the edges of high-degree vertices, and the pointers associated with these trees result in more memory consumption. The memory usage of DCSR is slightly lower than that of PPCSR. This is because DCSR uses 4 bytes for a leaf lock, whereas PPCSR uses 16 bytes for a vertex lock. Since both CPMA and PaC-tree compress the elements stored in their data structures, DCSR consumes more memory than they do because it stores the elements uncompressed. LSGraph's memory usage is lower than that of DCSR and PPCSR because it keeps far fewer gaps in the edge structures and stores edges more compactly.

6 Future Work

We propose the following four potential extensions for DCSR: ❶ Compressed PMA. We can naturally adopt delta encoding to compress the sorted elements, reducing memory overhead, as demonstrated in CPMA [47]. ❷ Hierarchical edge structures. We can utilize hierarchical edge structures for array extensions to reduce data movement for large batches, as demonstrated by LSGraph [36] and LPMA [53]. ❸ Skewed updates. When a few hot vertices are heavily updated, a hybrid parallel update strategy combining the lock-free batch update and the lock-based methods can be employed to avoid severe conflicts. ❹ Distributed settings. An adaptive PMA update principle can be employed to manage imbalanced updates and computation workloads across different machines.

7 Conclusion

This paper proposes DCSR, a fast data structure that employs a leaf-oriented parallel update strategy to enable efficient streaming graph processing. We identify and summarize the limitations of the lock-based single-edge update and lock-free batch update. To reduce access conflicts without requiring a sorting phase, we designed a leaf-oriented parallel update strategy consisting of the lock-based update and decoupled rebalancing phases. Our evaluation results indicate that DCSR is able to support both high-throughput graph updates and high-performance graph computation. Specifically, for the graph update, DCSR outperforms state-of-the-art dynamic data structures across a range of batch sizes.

Acknowledgments

We would like to thank all anonymous reviewers for their constructive suggestions. Huawei Cao is the corresponding author. This paper is supported by the National Science and Technology Major Project (TC240A9ED-74) and the Beijing Natural Science Foundation (4232036).

Artifacts

We develop DCSR based on BYO [48]. The source code and other artifacts for DCSR have been made available at https://github.com/IamwhatIamSY/DCSR.

References

[1] Alok Aggarwal and S Vitter, Jeffrey. 1988. The input/output complexity of sorting and related problems. *Commun. ACM* 31, 9 (1988), 1116–1127.

[2] Alif Ahmed, Farzana Ahmed Siddique, and Kevin Skadron. 2024. GraphTango: A Hybrid Representation Format for Efficient Streaming Graph Updates and Analysis. *International Journal of Parallel Programming* 52, 3 (2024), 147–170.

[3] Mohammad Al Hasan and Vachik S Dave. 2018. Triangle counting in large networks: a review. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 8, 2 (2018), e1226.

[4] Angelos Christos Anadiotis, Muhammad Ghufran Khan, and Ioana Manolescu. 2025. Dynamic graph databases with out-of-order updates. *Proceedings of the VLDB Endowment (PVLDB)* 17, 13 (2025), 4799–4812.

[5] Scott Beamer, Krste Asanovic, and David Patterson. 2012. Direction-optimizing breadth-first search. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–10.

[6] Michael A Bender and Haodong Hu. 2007. An adaptive packed-memory array. *ACM Transactions on Database Systems (TODS)* 32, 4 (2007), 26–es.

[7] Maciej Besta, Marc Fischer, Vasiliki Kalavri, Michael Kapralov, and Torsten Hoefler. 2023. Practice of Streaming Processing of Dynamic Graphs: Concepts, Models, and Systems. *IEEE Transactions on Parallel & Distributed Systems* 34, 06 (2023), 1860–1876.

[8] Maciej Besta, Robert Gerstenberger, Emanuel Peter, Marc Fischer, Michał Podstawski, Claude Barthels, Gustavo Alonso, and Torsten Hoefler. 2023. Demystifying graph databases: Analysis and taxonomy of data organization, system designs, and graph queries. *Comput. Surveys* 56, 2 (2023), 1–40.

[9] Ulrik Brandes. 2001. A faster algorithm for betweenness centrality. *Journal of mathematical sociology* 25, 2 (2001), 163–177.

[10] Sergey Brin. 1998. The PageRank citation ranking: bringing order to the web. *Proceedings of ASIS, 1998* 98 (1998), 161–172.

[11] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*. SIAM, 442–446.

[12] Dan Chen, Chuangyi Gui, Yi Zhang, Hai Jin, Long Zheng, Yu Huang, and Xiaofei Liao. 2022. GraphFly: efficient asynchronous streaming graphs processing via dependency-flow. In *2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Computer Society, 632–645.

[13] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2022. *Introduction to algorithms*. MIT press.

[14] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE computational science and engineering* 5, 1 (1998), 46–55.

[15] Dean De Leo and Peter Boncz. 2021. Teseo and the analysis of structural dynamic graphs. *Proceedings of the VLDB Endowment* 14, 6 (2021), 1053–1066.

[16] Long Deng, Yongkun Li, Zaigui Zhang, Yinlong Xu, and John CS Lui. 2025. Bubble: Towards Scalable Evolving Graph Processing via Mini-Batch Sorting. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1553–1571.

[17] Sahil Dhankhad, Emad Mohammed, and Behrouz Far. 2018. Supervised machine learning algorithms for credit card fraudulent transaction detection: a comparative study. In *2018 IEEE international conference on information reuse and integration (IRI)*. IEEE, 122–125.

[18] Laxman Dhulipala, Guy E Blelloch, Yan Gu, and Yihan Sun. 2022. Pac-trees: Supporting parallel and compressed purely-functional collections. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 108–121.

[19] Laxman Dhulipala, Guy E Blelloch, and Julian Shun. 2019. Low-latency graph streaming using compressed purely-functional trees. In *Proceedings of the 40th ACM SIGPLAN conference on programming language design and implementation*. 918–934.

[20] David Ediger, Rob McColl, Jason Riedy, and David A Bader. 2012. Stinger: High performance data structure for streaming graphs. In *2012 IEEE Conference on High Performance Extreme Computing*. IEEE, 1–5.

[21] Zhuochen Fan, Yalun Cai, Zirui Liu, Jiarui Guo, Xin Fan, Tong Yang, and Bin Cui. 2025. Cuckoograph: A scalable and space-time efficient data structure for large-scale dynamic graphs. In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*. IEEE, 1265–1277.

[22] Guanyu Feng, Zixuan Ma, Daixuan Li, Shengqi Chen, Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2021. Risgraph: A real-time streaming system for evolving graphs to support sub-millisecond per-update analysis at millions ops/s. In *Proceedings of the 2021 International Conference on Management of Data*. 513–527.

[23] Per Fuchs, Domagoj Margan, and Jana Giceva. 2023. Sortledton: A universal graph data structure. *ACM SIGMOD Record* 52, 1 (2023), 17–25.

[24] Shufeng Gong, Chao Tian, Qiang Yin, Wenyuan Yu, Yanfeng Zhang, Liang Geng, Song Yu, Ge Yu, and Jingren Zhou. 2021. Automating incremental graph processing with flexible memoization. *Proceedings of the VLDB Endowment*

14, 9 (2021), 1613–1625.

[25] Haruna Isah, Paul Trundle, and Daniel Neagu. 2014. Social media analysis for product safety using text mining and sentiment analysis. In *2014 14th UK workshop on computational intelligence (UKCI)*. IEEE, 1–7.

[26] Abdullah Al Raqibul Islam and Dong Dai. 2023. DGAP: Efficient dynamic graph analysis on persistent memory. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–13.

[27] Abdullah Al Raqibul Islam, Dong Dai, and Dazhao Cheng. 2022. Vcsr: Mutable csr graph format using vertex-centric packed memory array. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 71–80.

[28] Anand Padmanabha Iyer, Qifan Pu, Kishan Patel, Joseph E Gonzalez, and Ion Stoica. 2021. TEGRA: Efficient Ad-Hoc Analytics on Evolving Graphs.. In *NSDI*. 337–355.

[29] Pradeep Kumar and H Howie Huang. 2020. Graphone: A data store for real-time analytics on evolving graphs. *ACM Transactions on Storage (TOS)* 15, 4 (2020), 1–40.

[30] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[31] Hongfu Li, Qian Tao, Song Yu, Shufeng Gong, Yanfeng Zhang, Feng Yao, Wenyuan Yu, Ge Yu, and Jingren Zhou. 2024. GastCoCo: Graph Storage and Coroutine-Based Prefetch Co-Design for Dynamic Graph Processing. *Proceedings of the VLDB Endowment* 17, 13 (2024), 4827–4839.

[32] Yinnian Lin, Lei Zou, and Xunbin Su. 2024. Towards Sufficient GPU-accelerated Dynamic Graph Management: Survey and Experiment. *Proceedings of the VLDB Endowment* 18, 3 (2024), 599–612.

[33] Mugilan Mariappan, Joanna Che, and Keval Vora. 2021. Dzig: sparsity-aware incremental processing of streaming graphs. In *Proceedings of the Sixteenth European Conference on Computer Systems*. 83–98.

[34] Mugilan Mariappan and Keval Vora. 2019. Graphbolt: Dependency-driven synchronous processing of streaming graphs. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–16.

[35] Prashant Pandey, Brian Wheatman, Helen Xu, and Aydin Buluc. 2021. Terrace: A hierarchical graph container for skewed dynamic graphs. In *Proceedings of the 2021 international conference on management of data*. 1372–1385.

[36] Hao Qi, Yiyang Wu, Ligang He, Yu Zhang, Kang Luo, Minzhi Cai, Hai Jin, Zhan Zhang, and Jin Zhao. 2024. LSGraph: a locality-centric high-performance streaming graph engine. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 33–49.

[37] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *AAAI*. https://networkrepository.com

[38] Tao B Schardl and I-Ting Angelina Lee. 2023. Opencilk: A modular and extensible software infrastructure for fast task-parallel code. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 189–203.

[39] Mo Sha, Yuchen Li, Bingsheng He, and Kian-Lee Tan. 2017. Accelerating Dynamic Graph Analytics on GPUs. *Proceedings of the VLDB Endowment* 11, 1 (2017).

[40] Aneesh Sharma, Jerry Jiang, Praveen Bommannavar, Brian Larson, and Jimmy Lin. 2016. GraphJet: Real-time content recommendations at Twitter. *Proceedings of the VLDB Endowment* 9, 13 (2016), 1281–1292.

[41] Jifan Shi, Biao Wang, and Yun Xu. 2024. Spruce: a fast yet space-saving structure for dynamic graph storage. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 1–26.

[42] Julian Shun, Laxman Dhulipala, and Guy E Blelloch. 2015. Smaller and faster: Parallel processing of compressed graphs with Ligra+. In *2015 Data Compression Conference*. IEEE, 403–412.

[43] William F Tinney and John W Walker. 1967. Direct solutions of sparse network equations by optimally ordered triangular factorization. *Proc. IEEE* 55, 11 (1967), 1801–1809.

[44] Alexander van der Grinten, Maria Predari, and Florian Willich. 2022. A fast data structure for dynamic graphs based on hash-indexed adjacency blocks. In *20th International Symposium on Experimental Algorithms (SEA 2022)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 11–1.

[45] Keval Vora, Rajiv Gupta, and Guoqing Xu. 2017. Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations. In *Proceedings of the twenty-second international conference on architectural support for programming languages and operating systems*. 237–251.

[46] Qiange Wang, Yongze Yan, Hongshi Tan, Cheng Chen, Cheng Zhao, Jiaming Tian, Jiaxin Jiang, Xiaoliang Cong, Yanfeng Zhang, Ge Yu, et al. 2025. Efficient Graph Data Access for Out-of-Memory GPU Streaming Graph Processing. *Proceedings of the VLDB Endowment* 18, 11 (2025), 3854–3867.

[47] Brian Wheatman, Randal Burns, Aydin Buluc, and Helen Xu. 2024. CPMA: An efficient batch-parallel compressed set without pointers. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 348–363.

[48] Brian Wheatman, Xiaojun Dong, Zheqi Shen, Laxman Dhulipala, Jakub Łącki, Prashant Pandey, and Helen Xu. 2024. BYO: A Unified Framework for Benchmarking Large-Scale Graph Containers. *Proceedings of the VLDB Endowment* 17, 9 (2024), 2307–2320.

[49] Brian Wheatman and Helen Xu. 2018. Packed compressed sparse row: A dynamic graph representation. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 1–7.

[50] Brian Wheatman and Helen Xu. 2021. A parallel packed memory array to store dynamic graphs. In *2021 Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 31–45.

[51] Jin Zhao, Qian Wang, Ligang He, Yu Zhang, Sheng Di, Bingsheng He, Xinlei Wang, Hui Yu, Hao Qi, Longlong Lin, et al. 2025. TempGraph: An Efficient Chain-driven Temporal Graph Computing Framework on the GPU. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 230–246.

[52] Xiaowei Zhu, Guanyu Feng, Marco Serafini, Xiaosong Ma, Jiping Yu, Lei Xie, Ashraf Aboulnaga, and Wenguang Chen. 2020. LiveGraph: a transactional graph storage system with purely sequential adjacency list scans. *Proceedings of the VLDB Endowment* 13, 7 (2020), 1020–1034.

[53] Lei Zou, Fan Zhang, Yinnian Lin, and Yanpeng Yu. 2023. An efficient data structure for dynamic graph on GPUs. *IEEE Transactions on Knowledge and Data Engineering* 35, 11 (2023), 11051–11066.