High-Performance Graph Storage and Mutation for Graph Processing and Streaming: A Review

Review Paper

Soukaina Firmli*
Mohammed V University in Rabat
Ecole Mohammadia d'Ingénieurs, SIP Research Team
Rabat, Morocco
soukaina.firmli@gmail.com

Dalila Chiadmi
Mohammed V University in Rabat
Ecole Mohammadia d'Ingénieurs, SIP Research Team
Rabat, Morocco
chiadmi@emi.ac.ma

*Corresponding author

Abstract *– The growing need for managing extensive dynamic datasets has propelled graph processing and streaming to the forefront of the data processing community. Given the irregularity of graph workloads and the large scale of real-world graphs, researchers face numerous challenges when designing high-performance graph processing and streaming systems, due to the sheer volume, intricacy, and continual evolution of graph data. In this paper, we highlight the challenges related to two vital aspects within Graph Processing Systems that significantly impact the overall system performance: 1) the graph storage, encompassing the data structures storing vertices and edges, and 2) graph mutation protocols, referring to the ingestion and storage of new graph updates, such as additions of edges and vertices. Our paper provides a practical taxonomy of techniques designed to improve the efficiency of graph storage and mutation, by reviewing state-of-the-art systems and highlighting the challenges they face in offering a good performance tradeoff for read, write, and memory consumption. Consequently, this enables us to highlight overlooked aspects of performance, that are essential for real-world applications, such as the lack of mutation protocols for graph properties and auxiliary graph data, lack of configurability and cross-platform evaluation of solutions for graph processing and streaming.*

Keywords: *Graph Processing; Graph Streaming; Data Structures; Mutation; Performance*

Received: May 9, 2024; Received in revised form: September 13, 2024; Accepted: September 16, 2024

1. INTRODUCTION

Graph processing technology is continually advancing and thriving due to the distinctive capacity of graphs to model intricate relationships and dependencies within data, making them ideal for various contemporary applications. Notable graph applications include Knowledge Graphs (KG) [1] used in search engines, personal assistants, and recommendation systems; Graph Neural Networks (GNNs) [2] employed in AI tasks such as node classification, link prediction, and graph classification; and real-time graph analysis for streaming data from platforms like Twitter and financial transactions.

Given the extensive use of graphs, there is an increasing demand for efficient management and analysis. This demand has spurred the creation of graph processing systems and databases like Neo4j [3], which are adept at storing, analyzing, and streaming large graph datasets due to their scalability, real-time processing capabilities, and proficiency in handling complex relationships.

An ideal graph processing system should excel in analytics performance, provide fast mutations, and exhibit low memory consumption, regardless of mutation operations. However, these systems encounter several challenges inherent to graph processing and streaming. These challenges include the varying graph characteristics, the memory-intensive graph algorithms, the access patterns that cause latency, and the continuous evolution of graph topology and properties [4].

To understand these challenges, it is important to recognize that among the crucial software design elements are the graph data structure, which is responsible for storing vertices and edges, and the graph mutation protocols, such as additions or deletions of vertices and edges. While most of the optimization techniques in the literature stem from the data structures research community, there is a lack of research on their practical applicability within graph workloads.

Our research contributes by providing a practical taxonomy of techniques aimed at enhancing the efficiency of graph storage and mutation. These techniques include the optimization of the memory layout of graph data structures, compression, partitioning, batching techniques, changeset-based updates with delta maps and multi-versioning to improve the update-friendliness of classic data structures such as the Compressed Sparse Row (CSR) [5]. Furthermore, our analysis delves into the performance claims of existing literature on high performance, unveiling persistent challenges in read, write, and memory performance. This allows us to shed light on research gaps and overlooked aspects of performance crucial for real-world scenarios.

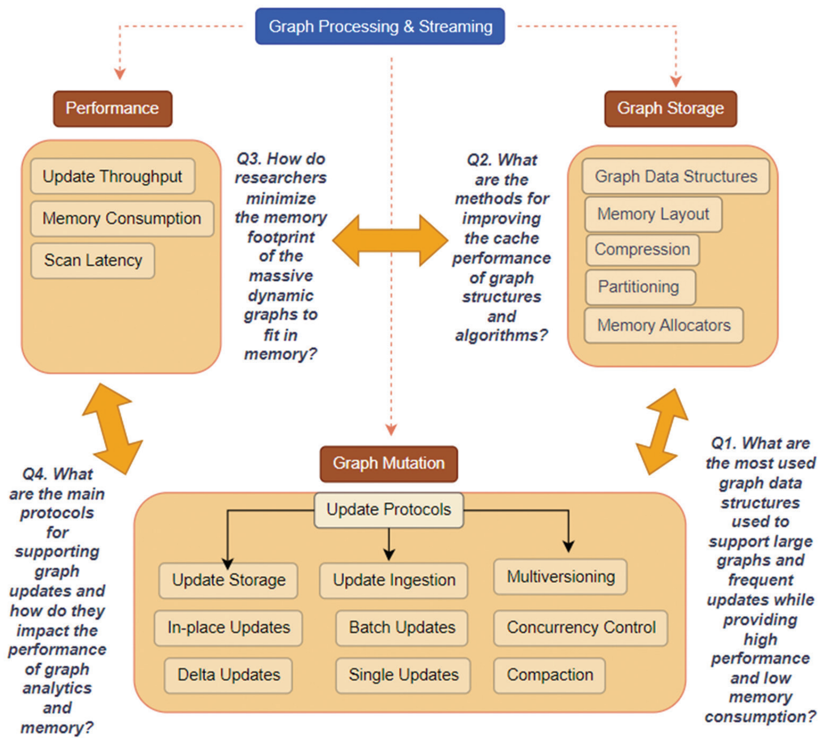
Graph Processing & Streaming
Performance
Update Throughput
Memory Consumption
Scan Latency
Q3. How do researchers minimize the memory footprint of the massive dynamic graphs to fit in memory?
Q2. What are the methods for improving the cache performance of graph structures and algorithms?
Graph Storage
Graph Data Structures
Memory Layout
Compression
Partitioning
Memory Allocators
Q4. What are the main protocols for supporting graph updates and how do they impact the performance of graph analytics and memory?
Graph Mutation
Update Protocols
Update Storage
Update Ingestion
Multiversioning
In-place Updates
Batch Updates
Concurrency Control
Delta Updates
Single Updates
Compaction
Q1. What are the most used graph data structures used to support large graphs and frequent updates while providing high performance and low memory consumption?

Fig. 1. Conceptual framework of our review

We present the background in Section 2 of our study. Second, we discuss our research methodology and research questions in Section 3 before presenting our review and analysis in sections 4 and 5. Finally, we discuss our findings in section 6, present the related work in section 7, and then present our future work and conclude in section 8.

2. BACKGROUND & CONTEXTUALIZATION

A graph is defined as a mathematical representation comprising vertices (nodes) and edges, which represent entities and the relationships between them, respectively. The volume, velocity, and variety of big data [6-8] that come from storing and processing large graph data, pose unique challenges in the field of computer science and data processing.

Graph technology, including graph processing systems and graph databases like Neo4j [3], emerges as a suitable approach for storing, analyzing and streaming big graph data, due to its scalability, real-time processing capabilities, and ability to handle complex relationships.

These GPS store graph data (i.e., graph topology and properties) in containers using data structures, such as adjacency lists, edge lists, or matrices [5]. They also provide algorithms for running analytic workloads and queries as well as updating graphs. These systems use combinations of high-performance data structures and update protocols to achieve their target performance.

They take advantage of hardware resources such as parallelism and Distributed machines to address the scalability challenge of processing and streaming large graphs efficiently.

From our research, it became evident that graph processing and streaming systems encounter three major challenges in efficiently storing and updating graphs which are as follows.

Challenge 1. Many real-world situations can be understood via the lens of scale-free networks. These graphs have a power-law distribution of degrees and a low density [9], with many vertices having very few or even zero degrees. The Internet and other social networks are two common examples of such graphs.

When designing systems, however, it might be difficult to account for the skew (degree variation) of these graphs. In reality, additional memory and processing power (RAM, CPU) may be needed to process vertices with higher degrees.

Challenge 2. The large size of graph data and access patterns of graph algorithms are important factors in the practical applicability of graph systems in production environments. Essentially, the memory-intensive nature of graph algorithms and their access patterns is one of the primary causes of the latency in graph computations. For instance, the PageRank algorithm [9] requires a large amount of input/output (I/O) operations to main memory and random accesses when iterating over vertices and edges in the graph, causing a lot of cache misses [10] and more slowdown

Challenge 3. The velocity characteristic of big data makes processing data more challenging where large amounts of graph data are generated rapidly and need to be added to graphs as new relationships in real-time. This is handled by stream processing systems, closely associated with real-time processing, involving processing data as it is created [11].

In summary, big data's volume, velocity, and variety pose challenges for traditional data processing methods including the storage, mutation and processing of graph data sets.

3. METHODOLOGY

There is a growing body of literature in the context of processing and streaming big dynamic graphs. In our paper, we aim to give a global overview of the different techniques used by researchers to improve the performance of graph processing systems and streaming.

In an attempt to give this overview, we narrowed the scope of this review to cover literature published over the past 16 years. We define a set of strings derived from keywords related to our research. The initial keywords are Graph, Analytics, Processing, Storage, Streaming and Mutation. We then use combinations to form strings to search for relevant papers on IEEE Xplore, ACM Digital Library and Google Scholar. We compile our database of around 97 prominent publications, we exclude some papers as they are out of our scope (e.g., incremental computation and graph database systems [12]).

With the collected papers, we note the following techniques that are used generally in the literature, which include:

- The optimization of graph data structures for the storage of dynamic graphs
- The design of parallel algorithms to execute graph analytics and queries efficiently on dynamic graphs
- The design of high-level graph languages to express and execute graph queries
- The implementation of algorithms for Distributed processing of graphs
- The design of efficient graph mutation protocols for fast graph updates

We focus on the graph representation in memory and the algorithms for graph updates as shown in Fig. 1 and Table 1.

Table 1. Analysis dimensions and their corresponding sections in this paper

Dimension	Description	RQ	RQ Section
Graph Representation	The data structures to store the graphs	RQ1	Sec. 4
Memory Consumption	The memory footprint of the graph represented in physical memory	RQ2, RQ3	Sec. 4 & 5
Graph Mutation	The implementation of graph mutations: in-place, delta maps, snapshots	RQ4	Sec. 5
Performance Optimization	The process of modifying a system to improve its functionality, thus making it more efficient in read and updating workloads	RQ2, RQ3,	Sec. 4 & 5
Parallelism & Hardware Resources	The underlying architecture of systems such as multi-core, CPU Cache and Distributed Systems	RQ2, RQ3,	Sec. 4 & 5

4. TAXONOMY OF TECHNIQUES FOR EFFICIENT GRAPH REPRESENTATION

In this section, we give an overview of techniques for optimizing classical graph data structures for high analytic and update performance as well as minimizing the memory footprint. We organize this section as the following. First, we discuss the performance of classical data structures, to identify their limitations. Then we analyze the different techniques available for researchers to optimize them, namely: optimizing the memory layout of the data structures (Section 4.2), compression (Section 4.3), using memory allocator software (Section 4.4) and partitioning (Section 4.5). Finally, we present a summary in Table 2.

4.1. REPRESENTATIVE GRAPH CONTAINER

In the following, We provide descriptions of classical graph data structures and we discuss the costs of performing graph mutations on each structure.

Adjacency Matrix

It holds a square matrix M with dimensions VxV, where V stands for the graph's vertex count. To indicate a directed edge from a source vertex vs to a destination vertex vd, the cell M[vs][vd] must be assigned a non-zero value. While this method simplifies edge manipulation, it's inefficient for sparse graphs due to high memory usage and suboptimal analytics performance. Moreover, adding or removing vertices requires completely recreating the matrix.

Adjacency Lists

This structure stores vertex information within a node list, with each element pointing to a list of its neighbours. It consumes less memory compared to an adjacency matrix because it only stores existing edges. The typical approach involves using linked lists for these connections, yet there are more efficient alternatives designed for better caching. For instance, variants like Blocked Adjacency Lists use simpler arrays for representing adjacencies [13] or utilize linked lists with fixed-size edge-containing buckets.

CSR (Compressed Sparse Row)

This representation, widely used for sparse graphs, condenses adjacencies into primarily two arrays: an edge array holding indices of destination vertices from the node array. The latter contains offsets to identify the beginning and end of the neighbours' list. To find the degree of node i, we compute Node_Array[i+1] – Node_ Array[i]. However, while this format is efficient in many cases, it still faces limitations, particularly in contexts where frequent updates occur. Several variations of CSR (Compressed Sparse Row) have been suggested—such as CSR++ [14-16] —aiming to enhance support for quicker structural updates. Further details on this topic are discussed in subsequent sections.

4.2. MEMORY LAYOUT AND CACHE AWARENESS

A good memory layout for a graph representation refers to the optimal way to arrange the graph data in memory to enhance its performance [10], by using hardware optimizations such as caching and prefetching. Moreover, knowing the access patterns of graph algorithms and storing graph entities in a specific layout can guarantee both spatial and temporal locality for graph data structures and algorithms [14, 17, 18], hence better performance.

By considering these factors, researchers in the graph community propose new variants of data structures by changing their memory layouts [9, 14, 16, 18, 19] to optimize the performance of graph analytics, queries and streaming, and improve their overall efficiency.

Essentially, to remediate the poor cache locality of AL, many researchers [13, 20] use bucketing technique where buckets are used to group edges from the same source vertex together, or use linked lists to group edges from different source vertices together (we elaborate this on Section 5. As we note from the evaluation results of works in the literature, there are only a few works that provide a thorough sensitivity analysis of different variations of their solutions like [19, 21]. There is a consensus on the size of the buckets which shouldn't be too huge, as it would slow down update performance, or too small, as that would cause cache misses. Apart from [16] as a pioneer solution for dynamic graph storage, we think there is a lack of configurability in systems in the literature and therefore a lot of opportunities to tweak existing designs in favour of new ones and for different types of workloads.

Furthermore, since CSR is known for its high cache performance and slow update performance, many researchers [15, 16, 19] opt for it as a main data structure for graph analytics and queries, then use an extra data structure to store the updates. For instance, to support fast updates, LLAMA stores multiple versions of the graph in CSR structures.

Essentially, it implements two variants of CSR, namely performance-optimized (PO) and space-optimized (SO), where the former keeps a complete list of edges of the same vertex in each version of the graph, and the latter only stores fragments of the edge lists for every vertex in different snapshots. As the names suggest, the PO provides high performance since all the edges are stored contiguously but the memory suffers from multiple copies of the edge list.

The SO saves on memory; however, it is slower since the edge lists are not stored contiguously, and the system needs to reconstruct the full adjacency of a vertex for read queries.

Finally, while memory layout techniques predominantly focus on graph topology, we emphasise the criticality of aligning memory layout with storing graph properties and auxiliary graph data (e.g., user keys), especially given the prevalence of graph algorithms dealing with weighted graphs.

4.3. COMPRESSION FOR DYNAMIC GRAPHS

Compression allows for a reduction of the memory needed to store data while maintaining its essential properties and functionality [22]. It is more efficient in graph representation since it allows for better cache performance since more data can be loaded in the cache and accessed at once by the CPU.

However, despite being a classic technique, there appears to be a scarcity of research exploring the application of compression techniques within the realm of graph updates. Notably, we observed a prevailing reliance on Ligra+ [23] within existing systems, attributable to its user-friendly interfaces. Among the notable systems claiming high performance in this context are Aspen [18], and SSTGraph [24], suggesting a potential avenue for further investigation into optimizing graph update procedures.

Aspen stores graph data in compressed purely functional trees, a form of persistent data structure for storing large graphs. By compressing trees, Aspen solves the issue of storing massive graphs (up to 200B edges) in machines with just 1TB of RAM by using compression. Given that Aspen stores edges as integers, it uses difference encoding to reduce the size of the edge arrays. However, while this method can significantly reduce memory consumption, it still increases the price of encoding and decoding processes, and a penalty may be incurred when executing queries or updating the graph.

To remediate this cost, SSTGraph a parallel framework designed for the storage and analysis of dynamic graphs, is based on the tinyset parallel dynamic set data structure, which implements set membership using sorted packed memory arrays. This allows for logarithmic time access and updates, as well as optimal linear time scanning. Compared to systems that use data compression, tinyset achieves comparable space efficiency without the computational and serialization overhead.

Finally, there is a notable absence of exploration into advanced compression algorithms within this domain. We propose that the graph processing community delve deeper into the challenges associated with graph workloads to assess the feasibility and potential benefits of implementing sophisticated compression algorithms.

4.4. MEMORY ALLOCATORS IN GRAPH STREAMING

A memory allocator is a software component that manages the allocation and deallocation of memory in a computer program [25]. GPS either develop their memory allocators or use out-of-the-box allocator libraries to manage their memory allocations and reduce memory fragmentation [26].

The first approach is used by some systems [20, 21, 27] where they develop built-in memory managers that facilitate the speedy allocation of memory needed for applying mutations. For instance, to efficiently perform memory reclamation and manage space, Hornet's [27] internal memory management uses a B+ tree for insertions and deletions to keep track of the available blocks of edges. Moreover, when data is duplicated, the system uses a load-balancing mechanism to locate the freed memory for later usage.

However, our findings underscore both the lack and the potential for smart predictive allocation techniques [28], particularly concerning updates within graph processing systems. Notably, while reallocation of edges commonly employs a predefined factor in existing systems, this often results in unnecessary allocation of extra space. We posit the feasibility of implementing smarter allocators leveraging machine learning methodologies to predict the optimal reallocation factor, thus enhancing memory utilization efficiency within graph processing frameworks.

The second approach uses memory allocators such as Jemalloc [29] and TCMalloc [30], which are widely used for their parallel support of memory allocation which helps with providing high update throughput. Moreover, these allocators use highly efficient algorithms to limit memory fragmentation, which leads to better cache locality and lower memory footprint.

We note a particular system called Metall [31] which is a persistent memory allocator that uses the copy-on-write technique for graph workloads, stores and manipulates large graphs at the exascale (billions of billions of operations per second), by employing smart allocation algorithms like those found in TCMalloc [30]. Essentially, Metall employs the use of mmap system calls to create memory-mapped files. With mmap, one may essentially access the files as if they were RAM, since it redirects the data to a virtual memory region. Therefore, to offer lightweight multi-versioning, Metall makes use of copy-on-write by taking snapshots of the graph after ingesting a batch of updates and employing a file copy method in the filesystems called reflink, which permits copy-on-write of data.

4.5. DYNAMIC DATA STRUCTURE PARTITIONING

Graph data partitioning is the process of dividing a large graph into smaller subgraphs [7], called partitions, to enable parallel processing of the graph on multiple machines or processors [8].

Some GPS systems [32, 33] use partitioning to optimize their performance by introducing several novel techniques to scale graph processing on a distributed cluster, including partitioning for sequential storage access, random distribution of data across the cluster, and work stealing for load balancing. These techniques enable GPS to handle graphs with trillions of edges, representing up to 16 TB of input data. However, the research on graph mutations using partitioning in a distributed system is still premature, and a very small number of articles address the challenges that come with it.

Finally, a single previous study provided an overview of graph update types, dynamic graph partitioning, and associated challenges [34]. Additionally, [35] did not address the performance consequences of partitioning on updates, memory, and read operations. Hence, it could be beneficial to investigate this aspect further.

Table 2. Summary of the characterization of systems included in this study based on their techniques for efficient graph storage

Dimension	Impact on Perf.	Implementations	Graph Processing & Streaming Systems
Memory Layout	Cache-friendly data structures	Blocking, CSR, PMA	[15, 17, 19, 27, 36]
Data Compression	Small memory footprint and better cache locality	Difference Encoding, Bit Indexing	[18, 24, 33]
Partitioning	Distributed processing and load balancing	Edge-cut, Vertex-cut	[32, 37]
Memory Allocators	Parallel allocation, low memory fragmentation, high cache performance	Jemalloc, TCMalloc, B+ Trees	[14, 16, 20, 27, 31]

5. TAXONOMY OF UPDATE PROTOCOLS FOR EFFICIENT GRAPH MUTATION

In addition to the representation of graphs in memory for graph queries and analytics, a wide range of GPS [16, 18, 20, 38] support graph mutation by allowing modifications to the graph's topology by adding or removing edges and vertices, as well as modification to the graph properties.

Essentially, to achieve that, we identify emerging patterns in the literature where GPS implement different techniques that we refer to as update protocols, which are different approaches for the ingestion and storage of new incoming graph data.

In the following, and in an attempt to answer our research question, we define the update protocols and analyze the different techniques in the literature for implementing graph updates and discuss their performance implications and limitations. Finally, we present a summary in Table 3.

5.1. OVERVIEW OF UPDATE PROTOCOLS

a) Update Ingestion

We categorize the update ingestions depending on two criteria: i) How a stream of incoming updates is handled right before storing them in the system? (Section 5.2), and ii) how a stream of incoming updates is handled in the presence of analytical queries (e.g., algorithms, pattern matching, scans ...) (Section 5.3).

First, there are two approaches to ingesting the updates in the GPS: i) Single update queries refer to the insertion or removal of a single edge or vertex at a time, while ii) batch updates refer to the grouping of the updates in a batch before applying them all at once. Moreover, we extrapolated two modes researchers are exploring for ingesting updates depending on how the graph analytics and queries are executed: in bulk or concurrently. Essentially, in the bulk mode, updates and graph algorithms are executed sequentially "in phases". On the other hand, in the concurrent mode, updates and graph analytics are processed simultaneously [13, 20, 27].

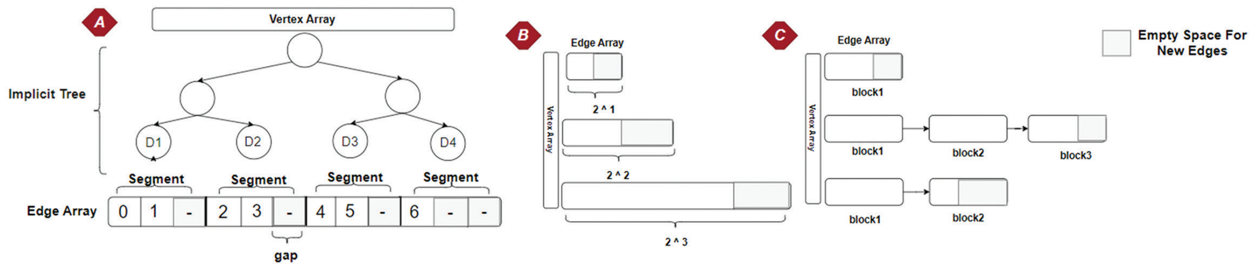

Fig. 2. Dynamic graph data structures. **A)** CSR representation of a graph using PMA to store the edge array. **B)** Adjacencies of a graph are stored in growable arrays with factor x2. **C)** Adjacencies of a graph are stored in a linked list of blocks

b) Update Storage

When applying the mutations, changes can be applied 1) in-place (i.e., incorporated into the main structure) or 2) stored in additional data structures called deltas [16]. In-place update is a technique where systems augment the traditional graph data structures, by permitting in-place storage of updates without the costly rebuilding of the whole graph data structure. As for the Delta approach, GPS use update-friendly data structures such as Adjacency List or Edge List to quickly store the updates, with the additional cost of merging these updates into the main read-friendly structure such as CSR. We elaborate on these techniques in Sections 5.4 and 5.5.

5.2. UPDATE INGESTION: SINGLE VS BATCH UPDATES

First, single updates are challenging to support for two main reasons. First, in most cases [14, 19, 20] and especially in deletion workloads, the system needs to perform a search over the neighbours of a vertex upon every edge insertion, which is not possible to do in parallel. This makes the system's performance very slow.

Second, depending on the availability of memory, systems need to allocate more memory to store the new edges [14]. Consequently, the frequent checks for memory availability and reallocations cause a large overhead, making the single updates very slow.

To remediate the slow single update performance, systems [14, 16, 18, 38] opt for batch updates where the batch of edge updates is pre-processed.

For instance, the sorting allows grouping all the edge updates of a specific vertex, separating deletions from insertions, which allows running edge updates in parallel for separate vertices. This improves the rate of update ingestions. This finding is supported by prior research [9, 17, 20] highlighting the effectiveness of sorting in optimizing the processing of edge updates.

Moreover, another technique used in batch updates is partitioning, which refers to splitting the updates into partitions that can be handled in parallel by multiple threads simultaneously [32]. This allows for better load balancing between parallel threads, especially for skewed graphs.

Unfortunately, the techniques mentioned above still incur large latency overhead as measured by GPS in literature [14, 16, 19], and most systems do not offer both single updates and batch updates, which is necessary for some real-world scenarios where updates are not frequent.

5.3. UPDATE INGESTION & MULTI-VERSIONING: BULK VS CONCURRENT UPDATES

The approach used by systems [13, 20, 27], to implement updates using the bulk mode, is a sequential approach where updates are held back until queries are completed, allowing updates to modify the graph while keeping data consistency [39], which ensures that the returned results accurately represent the current state of the data.

On the other hand, in the concurrent mode, systems may process updates and queries simultaneously. In this case, maintaining query consistency can be challenging.

Systems [14, 17, 20] in the bulk mode, mostly focus on supporting high update rates since updates don't have to be delayed by the queries. For instance, STINGER achieves an update rate of over 1.8 million updates per second on single multi-core machines, by executing updates in batches and running them in parallel without being concerned about concurrent reads.

Despite the high update throughput, research [18] shows that systems that employ the bulk mode have limited usage, since in real-world scenarios, graph users are constantly updating and running analytic queries concurrently. In the case where update/read happens in phases, this can introduce delays and decrease the system's overall performance.

On the other hand, in the concurrent mode, systems may process updates and queries simultaneously. In this case, maintaining query consistency can be challenging . In the following, we review different approaches used in practice, to allow concurrent updates and queries and discuss the challenges researchers face and potential research areas for the future.

Hybrid Store

A lot of systems [16, 18, 19, 40] implement protocols to maintain data coherence and consistency between multiple readers and writers through different isolation levels, which is similar to traditional database systems.

One way to achieve concurrent analytic workloads and update workloads is by creating a hybrid graph representation that uses separate data structures [16], [19]: one usually referred to as the write-optimised (WO Storage) data structure for the incoming updates and another read-optimised (RO Storage) structure for storing the main graph and can be accessed concurrently. This way, updates and read workloads would operate in parallel on different structures.

In this category, we cite a notable system LLAMA [16], which creates a new delta, a.k.a., snapshot, every time the user runs a batch of updates as shown in Fig. 4 B). This technique enables readers to have parallel access to the previously created snapshots and run analytics and queries on the RO store without interfering with the newer update queries performed on the WO store. Another example is GraphOne [40], which implements a hybrid store for snapshots using an adjacency lists (AL) store and an edge list (EL) as shown in Fig. 3 A). The AL keeps track of a linked list of vertex degrees at various points in time using timestamps.

Concurrency Control

Concurrency Control (CC) is a notable approach we extracted from the literature for processing concurrent reads and writes in GPS literature. For instance, a popular model is the Multiversion Concurrency Control (MVCC) [19] used by transactional systems such as Teseo [19] to achieve Snapshot Isolation. Teseo uses timestamps and a reversed chain of images to store the original copies of data, showing the items as they were before any changes were made (from newest to oldest). These versions are temporarily kept in the transaction's undo buffers whilst they are being rolled back, and then they are garbage collected as soon as the transaction is no longer valid (i.e. version pruning).

However, Teseo and many other systems face challenges with garbage collection, necessitating its execution without disrupting ongoing queries, thereby imposing additional costs on performance. We observed the lack of systems addressing the performance implications of executing compaction. We deem this a critical performance concern as both the execution of version pruning and the required resources may be hindered by the granularity and frequency of updates.

5.4. UPDATE STORAGE: IN-PLACE UPDATES

Systems [9, 13, 14, 19, 20] implement in-place updates by designing data structures that are suitable for graph updates, where the new entities (vertices or edges) can be directly stored in the data structures without requiring to reallocate the main data structure or to store them in extra data structures.

The main idea is to leave some space in the data structure for the new incoming entities, and in case there is not enough space, the data structure should allocate extra space suitable for that new entity as shown in Fig. 2. In the following, we discuss the popular approaches to achieve in-place updates.

a) Dynamic Arrays

A lot of systems use growable dynamic [9, 13, 14] where edge insertions are performed by directly storing the new edges in dynamic growable edge arrays. Systems employing Adjacency Lists are more prone to use this technique as shown in Figs. 2 B) and C).

NetworKit [13] performs edge insertions by directly storing the new edges in dynamic growable edge arrays and reallocating twice the initial array size when there is no memory space available. The same method is employed by Madduri et. al. [9] except that the size of the new edge array is defined in terms of a customizable factor rather than a fixed factor of 2.

Subsequently, when employing dynamic arrays, the amortised cost for updates is O(1) for insertions and O(deg V) for deletions. However, the memory footprint can be quite substantial as the reallocations leave unused space, when there are no updates. As mentioned above, using smart memory managers can help keep track of unused space and perform a better strategy for pre-allocation while maintaining a memory footprint comparable to static graph data structures. We also highlight the importance of knowing the pattern of the streams, to estimate the size of the reallocation. We predict that machine learning techniques are a possible venue for research that the community of graph processing and streaming should explore to propose innovative mechanisms to tackle these challenges in real-world graphs. For instance, by learning the patterns of the creation of relationships in social networks, e.g., a post that goes viral on social media might bring many followers in a short amount of time.

b) PMAs

Packed Memory Arrays (PMAs) are another classic technique in data structures that allows in-place updates by maintaining dynamic sets of sorted elements. It is based on storing an array of sizes larger than the number of elements N and leaving gaps between the elements to allow for new insertions with a moderate cost of $O(\lg^2(N))$. However, PMA relies on extra computation using an implicit balanced tree that keeps track of gaps within regions of the array as shown in Fig. 2 A). Essentially, when the number of gaps is too small or too large, systems are required to perform a re-balancing of the tree to rearrange the gaps in the array. This may slow down the update performance and delay the analytic workloads.

For instance, [17] developed a variant of CSR based on PMAs called PCSR, which provides efficient single-threaded mutations by leaving space at the end of each adjacency list. Moreover, multi-versioning systems like Teseo use PMAs to store multiple versions of the graph. By using PMAs, Teseo updates graph data in place, hence preserving the contiguous memory placement of the data. Ultimately, PMAs have gained traction within dynamic graph representations [19, 41, 42]. However, the expected improvement in read performance associated with PMAs does not justify the overhead incurred by tree rebalancing. Systems employing dynamic arrays [21] outperform PMAs in terms of update performance while supporting in-place updates.

c) Deletions

Regarding the deletions of graph entities, there are two main approaches: physical deletions and logical deletions. In the latter, systems [14, 16] enhance vertex and edge data with flags that can be set if they are removed, which allows for logical deletions of entities. The disadvantage of this method is that it requires changing the graph algorithms to account for deleted entities during traversals, which can make them slower, due to extra branches in the algorithm.

On the other hand, when entities are deleted physically from memory, the corresponding slots in the data structure are left unfilled, which results in a higher storage cost than logical deletion. Systems [16, 19, 40] employ compaction, which means reconstructing the graph without assigning space for the removed entities, to decrease unnecessary space after numerous physical deletions. This operation is very costly as it requires a whole rebuild of the graph, however, it helps reduce memory fragmentation and, therefore better read performance.

5.5. UPDATE STORAGE: DELTA UPDATES

One way to extend CSR to support fast updates is by employing deltas, which are separate structures to store only the new changes in vertex/edge logs. For instance, edge lists [16, 40] are particularly well suited for storing updates as deltas, as we can append new edges in O(1) time and space. They also help to maintain the temporality/history of updates, since the edges are stored in the order they arrive.

In practice, to support deltas, systems [16, 18, 19, 40] tend to design separate structures as read-optimized stores (usually CSR-based) and write-optimized stores which practically refer to a delta. However, the downside of storing updates in delta maps or edge lists is two-fold. First, maintaining separate structures for each batch of updates increases the system's memory requirements, as a frequent stream of graph updates results in many deltas. Second, analytics performance is degraded because they need to read from both the original structure and the deltas and reconcile them.

For example, LLAMA creates a new delta (i.e., snapshot) once the write-optimized store has been "flushed" into the read-optimized store. This might be practical for multi-versioning, but creating new deltas too frequently often results in out-of-memory errors as shown in [21] which can be fatal for real-time systems.

Finally, to remediate this problem, these systems perform compaction on the frequently created data structure into one main structure. For this purpose, several authors [16, 19, 40] have attempted to design their version of compaction. Throughout our research, we came across a multitude of references to the same operation, such as "archiving," "merging," and "building."

GraphOne [40] stores newly added edges in a circular log and uses adjacency lists as the main read-optimized store, which provides a coarse-grained snapshot method as shown in Fig. 3 A). First, GraphOne establishes an edge log cutoff for the "transfer" of the edges from the buffer to the base structure to preserve the separate store for the update operations and achieve good analytic performance. Second, in some cases, some data may be kept in both stores for a predetermined amount of time to maximize efficiency, this is known as data overlap. However, when there is a lot of duplication, it may use more RAM than usual.

It is shown in [19] that GraphOne's iterator architecture is to blame for its subpar performance in reading workloads. Point lookups are not possible because of the high space and time costs associated with analytic performance, which are incurred whenever the system iterates over vertices or edges and copies neighbours into an intermediate buffer.

In summary, compaction is computationally demanding, often zeroing the mutability performance benefits of these structures. For users who wish to operate on the most up-to-date version of the graph data, we believe that designing a system for in-place graph mutations is a better option to achieve higher analytics and update performance with lower memory requirements.

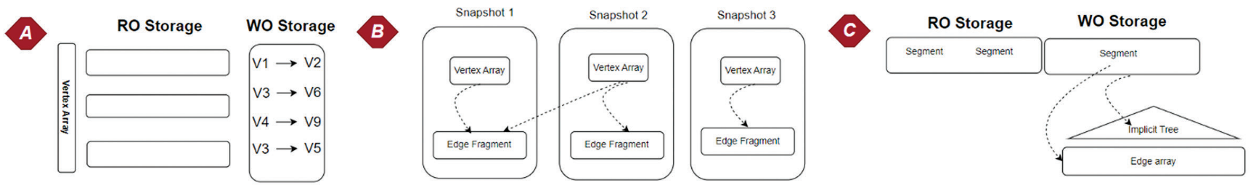

Fig. 3. Delta Updates, **A)** Hybrid storage using AL main storage and Edge List as delta. **B)** Snapshot mechanism as seen in LLAMA to access multiple versions of the graph. **C)** Delta structures stored in PMAs used in Teseo to support concurrent read and writes

Table 3. Summary of the characterization of systems included in this study based on their techniques for efficient graph updates

Technique	Category	Advantages	Systems
Single Upd.	Ingestion	Fine granularity of updates & Requires more CPU and memory	[14, 17, 20]
Batch Upd.	Ingestion	Fast update, load balancing Requires extra pre-processing	[14, 16, 18, 40]
Upd. in Bulk	Ingestion	Fast update throughput, fast analytics & No concurrency	[9, 14, 17, 20, 27]
Concurrent Upd.	Ingestion	Parallel updates and reads& High memory footprint, slow updates	[16, 18, 19, 40]
In-place Upd.	Storage	Low memory footprint, fast analytics & Slow updates	[9, 13, 14, 17, 19, 42]
Delta Upd.	Storage	Multi-versioning, fewer resources & Slow analytics, high memory footprint slow updates	[16, 18, 40]

6. RESULTS & DISCUSSION

In the following, we summarize key insights about the 50 research on high-efficiency graph representation and 51 update protocols. Table 4 presents a classification of the most popular graph systems reviewed in our paper, based on the dimensions discussed previously.

a) Configurations and Hybrid Representations

Based on the findings presented in Table 4, it is evident that numerous systems heavily rely on the Compressed Sparse Row (CSR) format due to its cache-friendly nature and reduced memory footprint. However, these systems often incorporate optimizations such as hardware enhancements or distributed computing frameworks [37], and supplementary data structures to manage updates efficiently, but there are limited designs for inherently fast data structures that support fast in-place and single updates while maintaining low memory footprint and high read performance.

Researchers can take advantage of the power of machine learning, to have more adaptable configurations depending on the workload. For instance, in the case of social networks, we can use learning algorithms to estimate the factor by which we can grow our dynamic arrays. This means that for a celebrity account, there are higher chances of gaining more followers than a normal account, therefore we can choose a higher factor (x4, x5) to pre-allocate the edge lists while keeping an x2 factor for normal accounts to maintain a low memory footprint of our dynamic graph.

b) Cross-platform implementation and evaluation

We highlight the prevalence of shared-memory systems, underscoring the ongoing necessity for advancements in this domain to attain comparable performance levels to distributed systems. Moreover, within the realm of distributed systems, there exists a notable lack of research on graph streaming, particularly concerning protocols tailored for update ingestion and storage in distributed infrastructures, or at least evaluation of the data structures in Distributed environments to showcase the advances in that area.

This observation implies that while the examined systems may ensure performance within specific hardware configurations, they lack assurance regarding the portability of their data structures across different infrastructures. Thus, there arises a demand for solutions which can seamlessly transition between single-machine and distributed environments.

Table 4. Summary of reviewed systems. Infra.: the supported architecture either Single Machine (SM) or Distributed (Dist); DS: data structures (CSR, Adjacency List, Tree); SU: support for single updates; Batch: support for batch updates; MV: support for multi-versioning; Compact: support for compaction; Up. Store: the update storage, either In-place (IP) or Delta (D); Scans: performance in read workloads; Mem: memory consumption; Up. Perf.: performance in mutation

Systems	Infra.	Data Structure	SU	Batch	MV	Compact.	Up. Store	Scans	Mem	Up.Perf.
BGL[43]	SM	AL	+	-	-	+	IP	-	-	+
PGX.SM[36]	SM	CSR	-	+	+	-	D	+	+	-
Ligra[44]	SM	CSR	-	-	-	-	X	+	+	-
GraphLab[45]	Dist	CSR	-	-	-	-	X	-	+	-
PGX.D[37]	Dist	CSR	-	+	+	-	D	+	+	-
STINGER[20]	SM Dist	AL	+	+	-	-	IP	-	-	+
Hornet[27]	GPU	AL	+	+	-	-	IP	-	+	+
Madduri et. Al. [9]	SM	AL	-	+	-	+	IP	-	-	+
PCSR[42]	SM	CSR	+	-	-	+	IP	+	-	+
LLAMA[16]	SM	CSR	-	-	+	+	D	+	-	-
Metall[31]	SM	AL	-	+	+	+	D	-	+	+
GraphOne[40]	SM	AL+EL	+	+	+	+	D	-	+	+
Aspen[18]	SM	Tree	-	+	+	+	IP	-	+	+
Teseo[19]	SM	Tree	+	+	+	+	IP	+	-	+

c) Achieving optimal performance trade-offs

We have shown that the choice of the optimization techniques, in either graph representation or updates depends on the specific characteristics of the graphs, the types of workloads, and the constraints of the application environment. Most systems tend to improve on an aspect of performance, either read performance, updates throughput or memory consumption, and to barely achieve the best trade-off between all three of these aspects. For instance, the majority of the systems that we reviewed struggle to support several versions of the graph because of high memory cost, and graph compaction is necessary to reduce the amount of space needed for changes, however, this operation requires expensive computation and slows down the performance of the system.

d) Overlooked metrics and benchmarks

While memory layout techniques typically concentrate on graph topology, we stress the importance of aligning memory layout with the storage of additional graph data, such as user keys and properties. This alignment becomes particularly critical due to the widespread use of graph algorithms dealing with weighted graphs.

Furthermore, there is a notable emphasis on protocols for updating graph topology. However, it is infrequent to find comprehensive implementations and evaluations of performance when considering other graph data elements, such as graph properties, reverse edges, or intermediary results in graph algorithms.

7. RELATED WORK

The prevalence of graph processing and streaming in various domains has prompted researchers to conduct reviews to understand how graphs are used in practice [5], [35].

Graph Systems and Databases. Angles et al. [46] provide a comprehensive survey of graph database models, focusing on data structures, query languages, and integrity constraints. [5] review graph systems and classifies them based on their infrastructure. Furthermore, Besta et. al. [35] provide descriptions and analysis of different approaches for representing graphs in a streaming context. However, they do not analyze the performance trade-offs of different approaches for graph storage and mutation.

High-Performance Graph Representations. [47] propose theoretical frameworks for the study of data structure designs and the generation of new structures to better serve specific types of workloads. They highlight the applicability of their abstractions for key-value stores, however, they do not study the compositions for the graph model.

Moreover, Wheatman et al. [42] review existing graph representations such as CSR and adjacency lists. They provide a theoretical study of the time complexity of graph access operations as well as update operations on such data structures. However, they do not review how systems in the literature aim to optimize the performance of these graph data structures.

8. CONCLUSION

In this paper, we discuss the increasing focus within the literature on optimizing classical graph data structures to achieve high analytical and update performance while minimising memory usage. We provide an overview of techniques available to researchers and developers, highlighting their roles in enhancing graph processing and streaming. Additionally, we examine update protocols and the performance implications of different designs. Our main findings reveal challenges faced by systems offering multi-versioning in terms of compaction costs, the inadequate evaluation of auxiliary data structures like reverse edges and multiple graph properties, and the need for more advanced partitioning and compression algorithms for dynamic graph data structures. Furthermore, our survey leads us to suggest that systems cannot provide high performance for scans, updates and memory consumption simultaneously. As a future scope, we aim to develop a framework for graph processing and streaming that would address the gaps in the research literature, especially tackling the expensive cost of compaction for multi-versioning. Furthermore, we aim to extend our study by delving into algorithmic details of compression and partitioning for graph mutations and, performing quantitative analysis and benchmarks.

9. REFERENCES:

[1] C. Peng, F. Xia, M. Naseriparsa, F. Osborne, "Knowledge graphs: Opportunities and challenges", Artificial Intelligence Review, Vol. 56, No. 11, 2023, pp. 13071-13102.

[2] C. Gao et al., "A survey of graph neural networks for recommender systems: Challenges, methods, and directions", ACM Transactions on Recommender Systems, Vol. 1, No. 1, 2023, pp. 1-51.

[3] M. Saad, Y. Zhang, J. Tian, J. Jia, "A graph database for life cycle inventory using Neo4j", Journal of Cleaner Production, Vol. 393, 2023, p. 136344.

[4] S. Firmli et al. "CSR++: a Fast, Scalable, Update-Friendly Graph Data Structure", Proceedings of the 24th International Conference on Principles of Distributed Systems, Leibniz, Germany, 2021, pp. 1-16.

[5] S. Firmli, D. Chiadmi, "A Review of Engines for Graph Storage and Mutations", Innovation in Information Systems and Technologies to Support Learning Research: Proceedings of EMENA-ISTL 2019, 2020, pp. 214-223.

[6] Y. Cui, S. Kara, K. C. Chan, "Manufacturing big data ecosystem: A systematic literature review", Robotics and Computer-Integrated Manufacturing, Vol. 62, 2020, p. 101861.

[7] S. Phansalkar and S. Ahirrao, "Survey of data partitioning algorithms for big data stores", Proceedings of the Fourth International Conference on Parallel, Distributed and Grid Computing, Waknaghat, India, 22-24 December 2016, pp. 163-168.

[8] R. Dindokar, N. Choudhury, Y. Simmhan, "A meta-graph approach to analyze subgraph-centric distributed programming models", Proceedings of the IEEE International Conference on Big Data, Washington, DC, USA, 5-8 December 2016, pp. 37-47.

[9] K. Madduri, D. A. Bader, "Compact Graph Representations And Parallel Connectivity Algorithms For Massive Dynamic Network Analysis", Proceedings of the IEEE International Symposium on Parallel & Distributed Processing, Rome, Italy, 23-29 May 2009.

[10] U. Drepper, "What every programmer should know about memory", Red Hat, Inc, Vol. 11, No. 2007, 2007.

[11] A. Bifet, R. Gavalda, G. Holmes, B. Pfahringer, "Machine learning for data streams: with practical examples in MOA", MIT press, 2023.

[12] W. Ju, J. Li, W. Yu, R. Zhang, "iGraph: an incremental data processing system for dynamic graph", Frontiers of Computer Science, Vol. 10, No. 3, 2016, pp. 462-476.

[13] C. L. Staudt, A. Sazonovs, H. Meyerhenke, "NetworKit: a Tool Suite For Large-Scale Complex Network Analysis", Network Science, Vol. 4, No. 4, 2016, pp. 508-530.

[14] S. Firmli et al. "CSR++: A Fast, Scalable, Update-Friendly Graph Data Structure", Proceedings of the 24th International Conference on Principles of Distributed Systems, 2020.

[15] R. Raman, O. van Rest, S. Hong, Z. Wu, H. Chafi, J. Banerjee, "PGX.ISO: parallel And Efficient In-memory Engine For Subgraph Isomorphism", Proceedings of Workshop on GRAph Data management Experiences and Systems, Snowbird, UT, USA, 22-27 June 2014.

[16] P. Macko, V. J. Marathe, D. W. Margo, M. I. Seltzer, "Llama: Efficient graph analytics using large multiversioned arrays", Proceedings of the IEEE 31st International Conference on Data Engineering, Seoul, Korea, 13-17 April 2015, pp. 363-374.

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

[18] L. Dhulipala, G. E. Blelloch, J. Shun, "Low-Latency Graph Streaming Using Compressed Purely-Functional Trees", Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, New York, NY, USA, 2019, pp. 918-934.

[19] D. De Leo, P. Boncz, "Teseo and the Analysis of Structural Dynamic Graphs", Proceedings of the VLDB Endowment, Vol. 14, No. 6, 2021, pp. 1053-1066.

[20] D. Ediger, R. McColl, J. Riedy, D. A. Bader, "Stinger: High performance data structure for streaming graphs", Proceedings of the IEEE Conference on High Performance Extreme Computing, Waltham, MA, USA, 10-12 September 2012, pp. 1-5.

[21] S. Firmli, D. Chiadmi, "A Scalable Data Structure for Efficient Graph Analytics and In-Place Mutations", Data, Vol. 8, No. 11, 2023.

[22] P. Boldi, S. Vigna, "The Webgraph Framework I: Compression Techniques", Proceedings of the 13th International Conference on World Wide Web, New York, NY, USA, 2004, pp. 595-602.

[23] J. Shun, L. Dhulipala, G. E. Blelloch, "Smaller and Faster: Parallel Processing of Compressed Graphs with Ligra+", Proceedings of the Data Compression Conference, 2015, pp. 403-412.

[24] B. Wheatman, R. Burns, "Streaming sparse graphs using efficient dynamic sets", Proceedings of the IEEE Interna-

tional Conference on Big Data, Orlando, FL, USA, 15-18 December 2021, pp. 284-294.

[25] E. D. Berger, K. S. McKinley, R. D. Blumofe, P. R. Wilson, “Hoard: A scalable memory allocator for multithreaded applications”, ACM Sigplan Notices, Vol. 35, No. 11, 2000, pp. 117-128.

[26] M. S. Johnstone, P. R. Wilson, “The memory fragmentation problem: Solved?”, ACM Sigplan Notices, Vol. 34, No. 3, 1998, pp. 26-36.

[27] F. Busato, O. Green, N. Bombieri, D. A. Bader, “Hornet: An Efficient Data Structure for Dynamic Sparse Graphs and Matrices on GPUs”, Proceedings of hte IEEE High Performance extreme Computing Conference, Waltham, MA, USA, 25-27 September 2018, pp. 1-7.

[28] I. U. Akgun, A. S. Aydin, A. Shaikh, L. Velikov, E. Zadok, “A machine learning framework to improve storage system performance”, Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems, 2021, pp. 94-102.

[29] D. Durner, V. Leis, T. Neumann, “On the Impact of Memory Allocation on High-Performance Query Processing”, Proceedings of the 15th International Workshop on Data Management on New Hardware, New York, NY, USA, July 2019, pp. 1-3.

[30] S. Lee, T. Johnson, E. Raman, “Feedback directed optimization of TCMalloc”, Proceedings of the workshop on Memory Systems Performance and Correctness, New York, NY, USA, June 2014, pp. 1-8.

[31] K. Iwabuchi, K. Youssef, K. Velusamy, M. Gokhale, R. Pearce, “Metall: A persistent memory allocator for datacentric analytics”, Parallel Computing, Vol. 111, 2022, p. 102905.

[32] A. Kyrola, G. Blelloch, C. Guestrin, “GraphChi: large-Scale Graph Computation on Just a PC”, Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation, 2012.

[33] A. Roy, L. Bindschaedler, J. Malicevic, W. Zwaenepoel, “Chaos: Scale-out graph processing from secondary storage”, Proceedings of the 25th Symposium on Operating Systems Principles, 2015, pp. 410-424.

[34] L. M. Vaquero, F. Cuadrado, M. Ripeanu, “Systems for near real-time analysis of large-scale dynamic graphs”, arXiv:1410.1903, 2014.

[35] M. Besta, M. Fischer, V. Kalavri, M. Kapralov, T. Hoefler, “Practice of streaming and dynamic graphs: Concepts, models, systems, and parallelism”, arXiv:1912.12740, 2020.

[36] S. Hong, H. Chafi, E. Sedlar, K. Olukotun, “Green-Marl: a DSL For Easy And Efficient Graph Analysis”, ACM SIGPLAN Notices, Vol. 47, No. 4, 2012, pp. 349-362.

[37] N. P. Roth et al. “PGX.D/Async: a Scalable Distributed Graph Pattern Matching Engine”, Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems, Chicago, IL, USA, 19 May 2017.

[38] M. Haubenschild, M. Then, S. Hong, H. Chafi, “Asgraph: a mutable multi-versioned graph container with high analytical performance”, Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems, Redwood Shores, CA, USA, 24 June 2016, pp. 1-6.

[39] A. Adya, B. Liskov, P. O’Neil, “Generalized isolation level definitions”, Proceedings of 16th International Conference on Data Engineering, San Diego, CA, USA, 29 February - 3 March 2000, pp. 67-78.

[40] P. Kumar, H. H. Huang, “GraphOne: a Data Store for Real-Time Analytics on Evolving Graphs”, ACM Transactions on Storage, Vol. 15, No. 4, 2020.

[41] M. A. Bender, H. Hu, “An Adaptive Packed-Memory Array”, ACM Transactions on Database Systems, Vol. 32, No. 4, 2007, pp. 26-es.

[42] B. Wheatman, H. Xu, “Packed Compressed Sparse Row: a Dynamic Graph Representation”, Proceedings of the IEEE High Performance extreme Computing Conference, Waltham, MA, USA, 25-27 September 2018.

[43] “The Boost Graph Library - 1.75.0.”, https://www.boost.org/doc/libs/1_75_0/libs/graph/doc/index.html (accessed: 2024)

[44] J. Shun, G. E. Blelloch, “Ligra: a lightweight graph processing framework for shared memory”, Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming, 2013, pp. 135-146.

[45] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, J. Hellerstein, “Graphlab: A new framework for parallel machine learning”, arXiv:1408.2041, 2014.

[46] R. Angles, C. Gutierrez, “An Introduction to Graph Data Management”, Data-Centric Systems and Applications, Springer International Publishing, 2018, pp. 1-32.

[47] S. Idreos et al. “The Periodic Table of Data Structures”, Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, Vol. 41, No. 3, 2018, pp. 64-75.